Advanced Bash-Scripting Guide

Углубленное изучение искусства написания сценариев командной оболочки

Mendel Cooper

<thegrendel.abs@gmail.com>

Посильный перевод: Быков О.В. *aka Oleg65*

10 Марта 2014

История проверок

Revision 6.5 'TUNGSTENBERRY' release	05 Апреля 2012	Проверено: mc
Revision 6.6 'YTTERBIUMBERRY' release	27 Ноября 2012	Проверено: mc
Revision 10 'PUBLICDOMAIN' release	10 Марта 2014	Проверено: mc

Этот учебник не предполагает никаких предварительных знаний о написании сценариев или программировании, но позволит быстро продвинуться до промежуточного/продвинутого уровня знаний... *постоянно находя их в небольших самородках мудрости и знаний UNIX®.* Она может служить в качестве учебного пособия, пособия для самостоятельного изучения и в качестве эталона и источника знаний о способах написания сценариев оболочки. Упражнения и примеры достаточно полно прокомментированны и предполагают активное участие читателя, убеждая, ***что единственный способ по настоящему научиться написанию сценариев это писать сценарии***.

Эта книга может быть использована, как общее введение в основные понятия программирования.

Посвящение

Аните, не иссякаемому источнику волшебства

Содержание

Список таблиц

Список Примеров

15-2. ***printf*** в действии
15-3. Присваивание переменной с помощью ***read***
15-4. Что случится, если у ***read*** не будет переменной
15-5. Многострочный ввод в ***read***
15-6. Обнаружение нажатия клавиш со стрелками
15-7. Применение ***read*** с *перенаправлением файла*
15-8. Проблемы при чтении из конвейера
15-9. Изменение текущей рабочей директории
15-10. Давайте посчитаем с ***let***.
15-11. Демонстрация эффекта ***eval***
15-12. Выбор между переменными с помощью ***eval***
15-13. Вывод параметров командной строки
15-14. Принудительный выход
15-15. *Версия rot13*
15-16. Использование ***set*** позиционными параметрами
15-17. Реверс позиционных параметров
15-18. Переназначение позиционных параметров
15-19. "Сброс" переменной
15-20. Применение ***export*** для передачи переменной во встроенный сценарий **awk**
15-21. Чтение опций/аргументов передаваемых в сценарий с помощью ***getopts***.
15-22. "Включение" файла данных
15-23. Пример (бесполезный) сценария, который сам является источником
15-24. Эффект ***exec***
15-25. Сценарий который порождает сам себя
15-26. Ожидание завершения перед продолжением
15-27. Сценарий, который убивает сам себя
16-1. Создание оглавления для записи CDR диска с помощью ***ls***
16-2. Hello или Good-bye
16-3. ***Badname***, исключение имен файлов в текущей директории содержащих плохие символы и пробелы.
16-4. Удаление файла по номеру его ***inode***
16-5. Logfile: Использование ***xargs*** для мониторинга системного журнала
16-6. Копирование файлов текущей директории в другую
16-7. Уничтожение процесса по имени
16-8. Частотный анализ слов с помощью ***xargs***
16-9. Использование ***expr***
16-10. Использование ***date***
16-11. Расчет *дат*
16-12. Анализ частоты встречаемости слова
16-13. Какие файлы являются сценариями?
16-14. Генерация случайных 10-значных чисел
16-15. Проверка системного журнала с помощью ***tail***
16-16. Вывод строк *From* из сохраненных сообщений e-mail
16-17. Эмуляция ***grep*** в сценарии
16-18. Решатель кроссворда
16-19. Поиск определений в *Словаре Webster 1913*
16-20. Проверка на правильность слов в списке
16-21. ***toupper***: Преобразование файла в верхний регистр.
16-22. ***lowercase****:* Изменение всех имен файлов в рабочей директории в нижний регистр.
16-23. ***du***: преобразование текстового файла DOS в UNIX.

16-24. ***rot13***: сверхслабое шифрование.
16-25. Создание головоломки "Crypto-Quote"
16-26. Форматированный вывод файла.
16-27. Использование ***column*** для форматирования листинга директории
16-28. ***nl***: Самонумерующийся сценарий
16-29. ***manview***: Просмотр форматированных справочных страниц
16-30. Перемещение дерева директории с помощью ***cpio***
16-31. Распаковка архива ***rpm***
16-32. Удаление комментариев из файлов программы Си
16-33. Знакомство с `/usr/X11R6/bin`
16-34. "Улучшения" команды ***strings***
16-35. Сравнение двух файлов сценария с помощью ***cmp***.
16-36. ***basename*** и ***dirname***
16-37. Сценарий копирующий себя частями
16-38. Проверка целостности файлов
16-39. Шифрование файлов **Uudecoding**
16-40. Узнаем откуда спамер
16-41. Анализ спам домена
16-42. Получение котировок акций
16-43. Обновление FC4
16-44. Использование ***ssh***
16-45. Сценарий, самостоятельно рассылающий почту
16-46. Создание простых чисел
16-47. Ежемесячный платеж по ипотечному кредиту
16-48. Преобразование основания
16-49. Вызов ***bc*** с помощью ***here document***
16-50. Подсчет числа Пи
16-51. Преобразование десятичного числа в шестнадцатеричное
16-52. Факторинг
16-53. Вычисление гипотенузы треугольника
16-54. Создание аргументов цикла с помощью ***seq***
16-55. Подсчет букв
16-56. Анализ опций командной строки с помощью ***getopt***
16-57. Сценарий, который копирует себя
16-58. Работа ***dd***
16-59. Захват нажатий клавиш
16-60. Подготовка загрузочной SD карты для *Raspberry Pi*
16-61. Безопасное удаление файла
16-62. Генератор имен файлов
16-63. Преобразование метров в мили
16-64. Использование ***m4***
17-1. Установка нового пароля
17-2. Настройка ***удаления*** символа
17-3. ***Безопасность пароля***: отключение вывода в терминале
17-4. Определение нажатой клавиши
17-5. Проверка ***identd*** удаленного сервера
17-6. ***pidof*** помогает убить процесс
17-7. Проверка образа на CD
17-8. Создание файловой системы в файле
17-9. Добавление нового жесткого диска

17-10. Укрытие от посторонних глаз выходного файла с помощью ***umask***
17-11. ***Backlight***: Изменение яркости подсветки (ноутбук)
17-12. ***killall*** из /etc/rc.d/init.d
19-1. ***broadcast***: Отправка сообщения всем зарегистрированным пользователям
19-2. ***dummyfile***: Создание пустого 2-строчного файла
19-3. Многострочные сообщения с помощью ***cat***
19-4. Многострочные сообщения с подавлением табуляций
19-5. ***Here document*** с изменяемыми параметрами
19-6. Загрузка пары файлов *Sunsite* находящихся в директории
19-7. Выключение подстановки параметров
19-8. Сценарий, который создает другой сценарий
19-9. ***Here document*** и функции
19-10. "Анонимный" ***Here Document***
19-11. Закомментирование блока кода
19-12. Самодокументируемый сценарий
19-13. Добавление строки в файл
19-14. Разборка почтового ящика
20-1. Перенаправление stdin используя ***exec***
20-2. Перенаправление stdout используя ***exec***
20-3. Перенаправление вместе stdin и stdout в тот же сценарий *с* ***exec***
20-4. Избегаем subshell
20-5. Перенаправление цикла ***while***
20-6. Другая форма перенаправления цикла ***while***
20-7. Перенаправление цикла ***until***
20-8. Перенаправление цикла ***for***
20-9. Перенаправление цикла ***for*** (перенаправление обоих stdin и stdout)
20-10. Перенаправление сравнения ***if/then***
20-11. Данные файла *names.data* для примеров выше
20-12. Регистрация событий
21-1. Область видимости переменной в подоболочке
21-2. Список профилей пользователей
21-3. Запуск параллельных процессов в ***subshells***
22-1. Запуск сценария в ограниченном режиме
23-1. Перенаправление блока кода без порождения потомка (форка)
23-2. Перенаправление вывода ***подстановки процесса*** в цикл.
24-1. Простые функции
24-2. Функция принимающая параметры
24-3. Функции и аргументы командной строки передающиеся в сценарий
24-4. Передача косвенной ссылки функции
24-5. Разыменование параметра переданного функции
24-6. Снова, разыменование параметра переданного функции
24-7. Максимальное из двух чисел
24-8. Преобразование чисел в римские цифры
24-9. Тестирование наибольшего возвращаемого значения функции
24-10. Сравнение двух больших целых чисел
24-11. Настоящее имя из имени пользователя
24-12. Видимость локальной переменной
24-13. Демонстрация простой рекурсивной функции
24-14. Еще одна простая демонстрация
24-15. Рекурсия, использующая локальную переменную

A-17. ***tree2***: Альтернативный сценарий *дерева* директории
A-18. ***string functions***: Функции обработки строк в стиле Си
A-19. Информация о директории
A-20. Библиотека хэш функций
A-21. Раскрашивание текста с помощью хэш-функции
A-22. Большие возможности хеш-функций
A-23. Монтирование USB оборудования хранения данных
A-24. Преобразование в HTML
A-25. Сохранение веблогов
A-26. Защита буквенных строк
A-27. Снятие защиты буквальности строки
A-28. Определение спамеров
A-29. Spammer Hunt
A-30. Делаем ***wget*** легким в использовании
A-31. Сценарий ***podcasting***
A-32. Ночной бэкап firewire HD
A-33. Расширение команды ***cd***
A-34. Сценарий настройки звуковой карты
A-35. Положение отдельных абзацев в текстовом файле
A-36. Вставка сортировки
A-37. Общее отклонение
A-38. Файловый генератор ***pad*** для условно-бесплатных продуктов
A-39. Редактор ***man page***
A-40. Лепестки розы
A-41. ***Quacky***: игра слов ***Perquackey***
A-42. ***Nim***
A-43. Секундомер в командной строке
A-44. Универсальные оболочки сценариев для решений домашних заданий
A-45. Рыцарский турнир
A-46. Волшебные квадраты
A-47. «Пятнашки»
A-48. ***Ханойские пагоды***, графическая версия
A-49. ***Ханойские пагоды***, другая графическая версия
A-50. Альтернативная версия сценария ***getopt-simple.sh***
A-51. Версия примера ***UseGetOpt.sh***, использующего приложение *Табличные расширения*
A-52. Перебор всех возможных цветов фона
A-53. Упражнение в Азбуке Морзе
A-54. Кодирование/декодирование ***Base64***
A-55. Вставка текста в файл с помощью *sed*
A-56. Шифр ***Gronsfeld***
A-57. Генератор чисел ***Bingo***
A-58. Рассмотренные основы
A-59. Проверка времени выполнения различных команд
A-60. Ассоциативные массивы против обычных массивов (время выполнения)
C-1. Подсчет вхождений буквы
J-1. Завершающий сценарий для *UseGetOpt.sh*
M-1. Образец файла `.bashrc`
M-2. Файл `.bash_profile`
N-1. VIEWDATA.BAT: Пакетный файл DOS
N-2. *viewdata.sh*: Конвертация VIEWDATA.BAT в Shell Script

Часть 1. Введение

Сценарий: Написанное; Написанный документ. [Obs.]

--Webster's Dictionary, 1913 ed.

Оболочка это интерпретатор команд. Больше, чем просто изолирующий слой между ядром операционной системы и пользователем, это довольно мощный язык программирования. Программная оболочка, которая называется ***сценарий***, является, простым в использовании, инструментом для создания приложений с помощью «склеивания» системных вызовов, инструментов, утилит и скомпилированных двоичных файлов. Практически весь набор команд, утилит и инструментов UNIX доступен для вызова с помощью сценария оболочки. Если этого не достаточно, то внутренние команды оболочки, такие как проверка и конструкции циклов, придают сценариям дополнительную мощь и гибкость. Сценарии оболочки особенно хорошо подходят для решения задач управления системой и других рутинных, повторяющихся, задач, не требующих колокольчиков и свистков полномасштабных, хорошо структурированных, языков программирования.

Содержание

Глава 1. Программирование в оболочке!

Ни один язык программирования не является совершенным. Нет лучшего языка; есть только языки хорошо или плохо подходящие для конкретных целей.

--Herbert Mayer

Практическое знание сценариев оболочки важно для тех, кто хочет стать достаточно сильным системным администратором, даже если они не собираются в будущем писать сценарии. Если посмотреть, как загружается Linux машина, то мы увидим, что она выполняет сценарии `/etc/rc.d`, восстанавливая конфигурацию системы и настраивая сервисы. Детальное понимание этих стартовых сценариев важно для анализа поведения системы, и, возможно, его изменения.

Ремесло сценариев не трудно освоить, потому что сценарии могут создаваться частями и содержать только достаточно небольшой набор операторов и опций, [1] известных конкретной оболочке. Синтаксис прост - даже строг - похож на вызов и объединение вместе утилит командной строки, и существует всего несколько «правил», регулирующих их использование. Большинство коротких сценариев правильно работают с первого раза, а отладка даже более объемных - проста.

> В первые дни персональных компьютеров, язык BASIC был установлен на компьютеры для того, чтобы писать программы для первого поколения микрокомпьютеров. Десятилетия спустя язык сценариев Bash позволяет любому, знающему Linux или UNIX, делать то же самое для современных машин.
>
> Теперь у нас есть миниатюрные одноплатные компьютеры, такие как Raspberry Pi, с удивительными возможностями. Сценарии Bash дают возможность исследовать возможности этих увлекательных устройств.

Сценарий является быстрым и черновым способом создания прототипов сложных приложений. Иметь даже ограниченное подмножество функциональности сценария, зачастую достаточно на первом этапе в развитии проекта. Таким образом, структура приложения может быть проверена, исправлена и найдены основные подводные камни до

начала окончательного кодирования на C, C, Java, Perl или Python .

Написание сценариев оболочки соотносится с классической философией UNIX разделять сложные проекты на более простые подзадачи, объединяя вместе компоненты и утилиты. Многие считают это лучшим, или, по крайней мере, очень эстетическим, подходом к решению проблем, в отличии от использования одного из мощных, все-в-одном, языков нового поколения, таких как Perl, которые пытаются быть заменой для всех, но мыслительная деятельность вашей головы должна соответствовать этому инструменту.

По Herbert Mayer, "полезный язык нуждается в массивах, указателях и общем механизме для создания структур данных". По этим критериям, сценарии оболочки менее «полезны». Или, может быть, нет...

Где не используются сценарии оболочки

- Ресурсоемкие задачи, особенно там, где скорость является основным фактором (сортировка, хэширование, рекурсии [2] ...)
- Процедуры, связанные с тяжелыми математическими операциями, особенно при вычислениях с плавающей точкой, вычислениях произвольной точности или комплексными числами (использование вместо Си или FORTRAN)
- Если требуется кросс-платформенная переносимость (использование вместо Си или Java)
- Сложные приложения, где необходимо структурное программирование (проверки типов переменных, прототипы функций, и т.д.)
- Критически важные приложения, на которые Вы ставите будущее компании
- Ситуации, где важна безопасность, когда необходимо гарантировать целостность системы и защиту от вторжения, взлома и вандализма
- Если проект состоит из нескольких компонентов с блокировкой зависимостей
- Если требуются обширные файловые операции (в Bash ограничен последовательный доступ, в нем только неуклюжий и не эффективный построчный стиль.)
- Необходима встроенная поддержка многомерных массивов
- Нужны структуры данных, такие как связанные списки и деревья
- Нужно создавать/ изменять графику или GUI
- Необходим прямой доступ к оборудованию системы или внешним периферийным устройствам

- Нужны порты или сокеты I/O
- Необходимо использовать библиотеки или интерфейсы с устаревшим кодом
- Проприетарность, приложения с закрытым исходным кодом (сценарии оболочки содержат исходный код в открытом виде и доступным для всех.)

Для любого из вышеуказанного, рассмотрите более мощный язык сценариев - возможно, Perl, Tcl, Python, Ruby - или, возможно, компилируемый язык, такой как Си, C++ или Java. Даже тогда, прототипирование приложений сценарием оболочки, может быть полезным шагом в развитии.

Мы будем использовать Bash, акроним [3] для «***Bourne-Again shell***» и игру слов, теперь классического Bourne shell, написанного Stephen Bourne. Bash является стандартом для сценариев оболочки на большинстве UNIX систем. Эта книга охватывает, в равной степени хорошо, большинство принципов сценариев других оболочек, таких как Korn Shell, C Shell и их вариантов, из которых Bash наследует некоторые из их особенностей [4]. (Обратите внимание, что программирование на C Shell не рекомендуется из-за присущих ему определенных проблем, указанных Tom Christiansen в октябре 1993г.).

Ниже приводится руководство по написанию сценариев оболочки. Оно опирается на иллюстрации примеров различных особенностей оболочки. Все примеры сценариев рабочие - они были проверены, насколько это возможно - а некоторые из них даже полезны в реальной жизни. Читатель может найти исходные коды работающих примеров в архиве (`scriptname.sh` или `scriptname.bash`), [5] дать им права на выполнение (**`chmod u+rx имя_сценария`**), а затем запустить их, чтобы увидеть, что произойдет. Если архив исходных кодов не доступен, то вырежьте-и-вставьте в исполняемые версии HTML или pdf. Имейте в виду, что некоторые из сценариев, представленных здесь, содержат функции до их разъяснения, а это может потребовать от читателя временно пропустить этот материал до изучения этих функций.

Если не указано иное, примеры сценариев написаны автором этой книги.

Его лицо было дерзким и не битым.

--Edmund Spenser

Примечания

[1] Называются *встроенными*, особенность внутренней оболочки.

[2] Хотя рекурсия возможна в сценарии оболочки, она, как правило, медленна и ее осуществление, зачастую, это уродливые ляп.

[3] Акроним — эрзац слово, формируемое путем объединения вместе начальных букв слов фразы.

[4] Многие из особенностей ***ksh88***, и даже несколько из обновленного ***ksh93***, были объединены в Bash.

[5] По соглашению, Bourne shell совместимые сценарии, написанные пользователем, принимают имя с расширением .sh. Системные сценарии, такие как в `/etc/rc.d`, не всегда соответствуют этой номенклатуре.

Глава 2. Начнем с Sha-Bang

Шел-программирование — это музыкальный автомат 1950-х . . .

--Larry Wall

Содержание

2.1. Вызов сценария
2.2. Предварительные упражнения

В простейшем случае, сценарий является не более, чем списком системных команд, хранящихся в файле. Каждый раз, когда он вызывается, то экономит усилия при вводе определенной последовательности команд.

Пример 2-1. *cleanup*: Сценарий очищающий лог-файлы в /var/log

```
# Очистка
# Конечно запускается из-под root.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Лог-файлы очищены."
```

Ничего необычного здесь нет, только набор команд, которые могли бы так же легко вызываться по одной, из командной строки в консоли или в окне терминала. Преимущества размещения команд в сценарии велики, не надо вводить их каждый раз заново. Сценарий становится программой - инструментом - и он может быть легко модифицирован или изменен для конкретного приложения.

Пример 2-2. *cleanup*: Улучшенный сценарий очистки

```
#!/bin/bash
# Правильный заголовок для сценария Bash.

# Очистка, версия 2

# Конечно запускается из-под root.
# Вставляем код здесь, чтобы вывести сообщение об ошибке и выйти, если не root.

LOG_DIR=/var/log
# Переменные лучше, чем жестко закодированные значения.
cd $LOG_DIR

cat /dev/null > messages
cat /dev/null > wtmp


echo "Логи очищены."

exit #  Правильный и надлежащий способ «выхода» из сценария.
     #  Голый "exit" (без параметров) возвращает статус выхода
     #+ предыдущей команды.
```

Теперь это начинает выглядеть как настоящий сценарий. Но мы пойдем еще дальше...

Пример 2-3. *cleanup*: Расширенный и обобщенный вариант сценариев выше.

```
#!/bin/bash
# Cleanup, version 3

#  Внимание:
#  -------
#  Этот сценарий использует ряд особенностей, которые будут объяснены позже
#  К тому времени, как вы закончите первую половину книги,
#+ в нем не будет ничего загадочного.


LOG_DIR=/var/log
ROOT_UID=0     # Только пользователь с $UID 0 имеет права root.
LINES=50       # Число сохраняемых строк по умолчанию.
E_XCD=86       # Невозможно изменить директорию?
E_NOTROOT=87   # Выход с ошибкой не root.


# Конечно запускаем из-под root.
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Запускать этот сценарий может только root."
  exit $E_NOTROOT
fi

if [ -n "$1" ]
# Проверка наличия аргумента командной строки (наличия).
then
  lines=$1
else
  lines=$LINES # По умолчанию, если не задано в командной строке.
fi


#  Stephane Chazelas предлагает следующее, как улучшенный
#+ способ проверки аргументов командной строки, но это все еще
#+ немного шире для этой стадии изучения.
#
#    E_WRONGARGS=85  # Не числовой аргумент (вывод ошибки при не правильном
#                    #  формате аргумента).
#
#    case "$1" in
#    ""      ) lines=50;;
#    *[!0-9]*) echo "Используйте: `basename $0` удаляемые_строки";
#     exit $E_WRONGARGS;;
#    *       ) lines=$1;;
#    esac
#
#* Пропустите и перейдите к главе «Циклы», что бы все это расшифровать.


cd $LOG_DIR

if [ `pwd` != "$LOG_DIR" ]  # или   if [ "$PWD" != "$LOG_DIR" ]
                            # Не /var/log?
then
  echo "Нельзя изменить на $LOG_DIR."
  exit $E_XCD
fi  #  Двойная проверка if правильности директории, прежде чем
    #+ заниматься с лог-файлами.
```

```
# Гораздо более эффективным будет:
#
# cd /var/log || {
#   echo "Невозможно перейти в необходимую директорию." >&2
#   exit $E_XCD;
# }


tail -n $lines messages > mesg.temp #  Сохраняем последнюю часть
                                    #+ сообщений лог-файла.
mv mesg.temp messages               # Переименовываем его как
                                    #+ системный лог-файл.

#  cat /dev/null > messages
#* Больше не нужно, поскольку данный метод является более безопасным.

cat /dev/null > wtmp  #  ': > wtmp' и '> wtmp'  имеют различный эффект.
echo "Лог-файлы очищены."
#   Обратите внимание, что другие лог-файлы, кроме /var/log, не зависят
#+ от этого сценария.

exit 0
#  Возвращаемое из сценария значение ноль, после выхода, указывает на успех
оболочки.
```

Поскольку вы не можете очистить весь системный журнал, эта версия сценария сохраняет последний раздел журнала сообщений нетронутыми. Далее вы откроете для себя способы тонкой настройки ранее написанных сценариев, для повышения их эффективности.

* * *

Заголовок сценария *sha-bang* (**#!**) [1] говорит системе, что этот файл является набором команд, которые будут подаваться на *указанный* командный интерпретатор. **#!** это, на самом деле, двух байтовое [2] *магическое число*, специальный маркер, обозначающий тип файла, или, в данном случае, что это исполняемый сценарий оболочки (распечатайте *мануал* ***magic*** для получения более подробной информации по этой увлекательной теме). Сразу же после *sha-bang* - идет *имя пути*. Это путь к программе, которая интерпретирует команды сценария, будь то оболочка, язык программирования или утилита. Затем этот командный интерпретатор выполняет команды сценария, начиная с верхней части (строка после строки *sha-bang*), и игнорирует комментарии.[3]

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/bin/awk -f
```

Каждая из строк, заголовка выше, вызывает свой командный интерпретатор сценария, по умолчанию оболочку `/bin/sh` (Баш в системе Linux), или другую. [4] Использование

`#!/bin/sh`, оболочки Bourne в большинстве коммерческих вариантов UNIX по умолчанию, делает сценарий переносимым на не Linux машины, хотя бы пожертвовав и Bash-специфическими чертами. Сценарий, однако, будет соответствовать ***sh*** стандарту POSIX [5].

Обратите внимание, что путь задаваемый «***sha-bang***» должен быть правильным, в противном случае выводится сообщение об ошибке - обычно "команда не найдена " - будет результатом выполнения сценария. [6]

`#!` может быть опущен, если сценарий состоит только из набора общих системных команд, не использующих никаких директив внутренней оболочки. Второй пример, выше, должен начинаться с `#!`, т.к. строка присваивания переменной, *строка 50*, использует конструкцию конкретной оболочки. [7] Обратите внимание, что `#!/bin/sh` вызывает интерпретатор оболочки по умолчанию, которая на машине Linux, по умолчанию, `/bin/bash`

Этот учебник призывает к модульному подходу при построении сценария. Принимать к сведению и собирать "шаблонные" фрагменты кода, которые могут быть полезны в будущих сценариях. В конце концов вы соберете довольно обширную библиотеку изящных процедур. В качестве примера, следующие сценарий проверки вызова сценария с правильным количеством параметров.

```
E_WRONG_ARGS=85
параметры_сценария="-a -h -m -z"
#                  -a = все, -h = помощь, и т.д.

if [ $# -ne $Число_ожидаемых_аргументов ]
then
  echo "Используйте: `basename $0` $параметры_сценария"
  # `basename $0` это имя сценария.
  exit $E_WRONG_ARGS
fi
```

Много раз, вы будете писать сценарии, которые выполняют одну конкретную задачу. Пример - первый сценарий в этой главе. Позже вы сможете обобщить сценарии, чтобы реализовать другие, подобные задачи. Замена буквальных ("вшитых") констант переменных, это шаг в этом направлении, как и замена повторяющихся блоков кода функциями.

Примечания

[1] Более часто встречается в литературе, как *she-bang* или *sh-bang*. Это вытекает из объединения лексем sharp (**#**) и bang (**!**).

[2] Некоторые системы UNIX (основанные на BSD 4.2), якобы берут четырех байтное магическое число, оставляя пустое место после `!` - **`#! /bin/sh`**. Но Sven Mascheck поясняет, что это, вероятно, миф.

[3] Строка **#!** в сценарии оболочки будет первой, которую увидит интерпретатор команд (**sh** или **bash**). Так как эта строка начинается с **#**, то она будет правильно интерпретироваться, как комментарий, когда интерпретатор команд запустит исполнение сценария. Строка уже выполнила свою задачу - вызвала командный интерпретатор.

Если сценарий включает в себя дополнительную строку **#!**, то **Bash** будет интерпретировать ее как комментарий.

```
#!/bin/bash

echo "Первая часть сценария."
a=1

#!/bin/bash
# Она не запускает новый сценарий.

echo "Вторая часть сценария."
echo $a  # Значение $a осталось 1.
```

[4] Позволяет делать некоторые симпатичные трюки.

```
#!/bin/rm
# Самоудаляющийся сценарий.

#   Кажется ничего не происходит при запуске ... кроме того, что файл
#+  исчезает.

WHATEVER=85

echo "Эта строка ничего не выводит (betcha!)."

exit $WHATEVER  # Не имеет значения. Этот сценарий здесь не завершается.
                # Попробуйте echo $?, после завершения сценария.
                # Вы получите значение 0, не 85.
```

Попробуйте запустить файл README с **#!/bin/more** сделав его исполняемым. В результате получите списочный файл документации. (***here docu*ment** использующий **cat**, возможно, будет лучшей альтернативой - см. Пример 19-3).

[5] **P**ortable **O**perating **S**ystem **I**nterface, попытка стандартизировать UNI**X**-подобные операционные системы. Характеристики POSIX, перечислены на сайте Open Group.

[6] Чтобы избежать этого, сценарий может начинаться со строки *sha-bang* `#!/bin/env bash` . Это может быть полезно на UNIX-машинах, где bash находится не в `/bin`

[7] Если Bash используется оболочкой по умолчанию, то в начале сценария нет необходимости в **#!**. Тем не менее, если сценарий запускается в другой оболочке, такой как ***tcsh***, то будет нужен **#!**.

2.1. Вызов сценария

Написав сценарий, вы можете вызвать его набрав **`sh scriptname`**, [1] или, альтернативно, **`bash scriptname`** (где **`scriptname`** - это название сценария). (Не рекомендуется использовать `sh <scriptname`, так как это отключает чтение из **`stdin`** сценария.) Но гораздо удобнее сделать сам сценарий исполняемым, при помощи **`chmod`**.

Например:

> **`chmod 555 scriptname`** (Дает право каждому на чтение/выполнение) [2]

или

> **`chmod +rx scriptname`** (Дает право каждому на чтение/выполнение)
>
> **`chmod u+rx scriptname`** (дает право только владельцу на чтение/выполнение сценария)

Сделав сценарий исполняемым, вы можете проверить его работоспособность

`./scriptname`. [3] Если он начинается со строки «*sha-bang*», вызов сценария вызовет для его запуска правильный командный интерпретатор.

В качестве последнего шага, после проверки и отладки, вы вероятно захотели бы переместить его в `/usr/local/bin` (как root, конечно), чтобы сделать сценарий исполняемым общесистемно и доступным для себя и всех других пользователей. После этого сценарий можно будет вызывать, просто набрав в командной строке `scriptname` [**ENTER**] .

Примечания

[1] Предупреждение: при вызове сценария *Bash* **`sh scriptname`** выключается специфичность Bash, и поэтому сценарий может не выполняться.

[2] Сценарию необходим *read*, также как и право исполнения на запуск, ведь оболочка должна прочитать его (сценарий).

[3] Почему бы просто не вызывать сценарий `scriptname`? Если ваша директория (**$ PWD**), где находится `scriptname`, почему он не работает? Это не получится по соображениям безопасности, текущая директория (`./`) по умолчанию не включена в пользовательский **$PATH**. Поэтому в текущей директории необходимо явно вызывать сценарий с **`./scriptname`**.

2.2. Предварительные упражнения

1. Системные администраторы часто пишут сценарии для автоматизации общих задач. Приведите несколько случаев, когда такие сценарии полезны.
2. Напишите сценарий, который при вызове показывает время и дату, перечисляет всех зарегистрированных пользователей и дает системный **uptime**. Сценарий должен сохранять эту информацию в лог-файл.

Часть 2. Основы

Содержание

Глава 3. Специальные символы

Что делает *специальный* символ? Если символ имеет значение вне пределов его *буквального смысла*, ***метасмысл***, то мы ссылаемся на него как на *специальный символ*. Наряду с командами и ключевыми словами (***keywords***), *специальные символы* - это строительные блоки сценариев Bash.

Специальные символы находящиеся в сценариях и других местах

#

Комментарии. Строки начинающиеся с **#** (за исключением **#!**) это комментарии и *не будут* выполняться в сценарии.

```
# Эта строка комментарий.
```

Комментарии могут находиться после окончания команды.

```
echo "Комментарий будет следом." # Здесь какой-то комментарий.
#                                  ^ Обратите внимание, перед # пробел
```

Комментарии могут также следовать за пробелами в начале строки.

```
     # Табуляция перед комментарием.
```

Комментарии даже могут быть вставлены в *конвейер*.

```
initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
# Удаление строк содержащих '#' символ комментария.
          sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# Выдержка из сценария life.sh
```

После комментария команда не может находиться с ним на одной строке. Не существует способа прекращения действия комментария, что бы «живой код» начал работать на той же строке. Следующую команду начинайте с новой строки.

Конечно, помещение в *кавычки* или *экранирование*, объявленного в **echo** символа **#,** не начинает комментарий. Кроме того, символ **#** появляется в некоторых конструкциях подстановки параметров и численных констант.

```
echo "# здесь не начинается комментарий."
echo '# здесь не начинается комментарий'
echo \# здесь не начинается комментарий.
echo # А вот здесь — начало комментария.

echo ${PATH#*:}       # Подстановка параметра, не комментарий.
echo $(( 2#101011 ))  #  Преобразование основания,
                      #+ не комментарий.

# Спасибо S.C.
```

Обычные символы кавычек и управления (escape) (**" ' **) экранируют **#**.

Некоторые *шаблоны совпадения операций* также используют **#**.

;

Разделитель команд [точка с запятой]. Разрешает поместить две или более команд на одной строке.

```
echo hello; echo there


if [ -x "$filename" ]; then    #  Обратите внимание на пробел после
#+                   ^^           точки с запятой
```

```
  echo "File $filename exists."; cp $filename $filename.bak
else   #                       ^^
  echo "File $filename not found."; touch $filename
fi; echo "File test complete."
```

Обратите внимание, что «**;**» иногда необходимо ***экранировать***.

;;

[двойная точка с запятой] Прекращение действия опции в ***case*** .

```
case "$variable" in
  abc)  echo "\$variable = abc" ;;
  xyz)  echo "\$variable = xyz" ;;
esac
```

;;&, **;&**

Прекращение действия опции в ***case*** **(версия Bash 4+).**

.

команда "точка" [точка]. Эквивалентна *source* (см. Пример 15-22). Внутренняя команда bash.

.

"точка", как элемент имени файла. В именах файлов, точка вначале имени файла является префиксом «скрытого» файла, который обычно не выводится *ls*.

```
bash$ touch .скрытый-файл
bash$ ls -l
total 10
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook


bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.addressbook
 -rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .скрытый-файл
```

При рассмотрении имен директорий, ***одна*** точка представляет текущую рабочую директорию, а ***две*** точки обозначают родительскую директорию.

```
bash$ pwd
/home/bozo/projects

bash$ cd .
bash$ pwd
/home/bozo/projects

bash$ cd ..
bash$ pwd
/home/bozo/
```

Точка часто является файлом назначения (директорией) действия команды, в этом контексте подразумевается *текущая* директория.

```
bash$ cp /home/bozo/current_work/junk/* .
```
Копирование всех файлов "junk" в $PWD.

.

"точка" соответствующий символ. Где *соответствующий* символ это часть регулярного выражения (***regexp***), а "***точка***" соответствует *одному любому символу*.

"

мягкие кавычки [двойные кавычки]. *"СТРОКА"* сохраняют (от интерпретации) большинство специальных символов в строке. См. Главу 5.

'

жесткие кавычки [одинарные кавычки]. *'СТРОКА'* сохраняют *все* специальные символы в строке. Это более строгая форма, чем *"СТРОКА"*. См. Главу 5.

,

оператор запятая *Запятая* — оператор, [1] связывающий вместе серии вычислительных операций. Все оцениваются, но возвращает только последняя.

```
let "t2 = ((a = 9, 15 / 3))"
# Присваиваем  "a = 9" и "t2 = 15 / 3"
```

Оператор запятая может объединять строки.

```
for file in /{,usr/}bin/*calc
#             ^    Ищет все исполняемые файлы, оканчивающиеся на «calc»
#+                 в директории /bin and /usr/bin.
do
        if [ -x "$file" ]
        then
          echo $file
        fi
```

```
done

# /bin/ipcalc
# /usr/bin/kcalc
# /usr/bin/oidcalc
# /usr/bin/oocalc


# Благодарю за пояснение Rory Winston.
```

,, ,

Преобразование в нижний регистр, при подстановке параметров (Добавлено в 4 версии Bash).

****

экранирование [обратный слэш]. Механизм кавычек для отдельных символов.

\X *экранирует* символ *X*. Т.е. эффект "помещения в кавычки" *X*, равнозначно **'***X***'**.

**** может применяться для кавычек **"** и **'**, так что они (кавычки) будут интерпретироваться в буквальном смысле.

См. Главу 5 где подробно объясняется действие экранирующих символов.

/

разделитель пути к файлу [прямой слэш]. Разделяет составляющие файлового имени (как `/home/bozo/projects/Makefile`).

А также арифметический *оператор деления*.

`

подстановка команды Конструкция **`***команда***`** делает доступными выходные данные *команда* для назначения переменной. Также называются *обратными* кавычками или обратными галочками.

:

команда null [двоеточие]. Это эквивалент оболочки "NOP" (*no op*, ничего не делающая операция). Она может считаться синонимом для встроенного **`true`** оболочки. Команда **':'** сама *встраивается* в Bash, а *статус* ее *выхода* равен **`true`** (`0`).

```
:
echo $?   # 0
```

Бесконечный цикл

```
while :
do
   операция-1
   операция-2
   ...
   операция-n
done

# То же, что и:
#    while true
#    do
#      ...
#    done
```

Заполнитель в проверке ***if/then***:

```
if условие
then :   # Ничего не делает и передает управление далее
else     # Или иначе ...
   производятся-какие-то-действия
fi
```

Заполнитель, где ожидается бинарная операция, см. Пример 8-2 и параметры по умолчанию.

```
: ${username=`whoami`}
# ${username=`whoami`}   Без : в начале, выдаст ошибку
#                        'имя пользователя' это команда или builtin...

: ${1?"Usage: $0 ARGUMENT"}     # Из сценария "usage-message.sh.
```

Заполнитель, где команда это ожидаемый ***here document***. См. Пример 19-10.

Оценка строковых переменных с помощью *параметра подстановки* (Пример 10-7).

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
#  Выведет сообщение об ошибке,
#+ если одна или более, из основных переменных окружения, не

#+ установлена.
```

Расширение переменной/подстановка содержимого строки (substring).

В сочетании с оператором перенаправления **>**, усекает заданный файл до нулевой длины, без изменения его прав. Если файл ранее не существовал, создает его.

```
: > data.xxx   # Файл "data.xxx" теперь пуст.

# Такой же эффект, как   cat /dev/null >data.xxx
# Однако, это не форк нового процесса, поскольку ':'  встроено.
```

См. также Пример 16-15.

В сочетании с оператором перенаправления **>>**, не влияет на существующий заданный файл (**: >> target_file**). Если файл ранее не существовал, создает его.

Это относится к обычным файлам, а не к конвейерам, символическим ссылкам и некоторым специальным файлам.

Может использоваться для начала строки комментариев, но это не рекомендуется. Использование **#** для комментариев, отключается проверку на ошибки оставшейся части строки, потому что в комментарии почти ничего может появиться. Тем не менее, это не случай с **:**.

```
: Этот комментарий выдаст ошибку, ( if [ $x -eq 3] ).
```

Служит в качестве разделителя *полей* в `/etc/passwd` и в переменной `$PATH`.

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

Двоеточие приемлемо в качестве имени функции.

```
:()
{
  echo "Имя этой функции "$FUNCNAME" "
  # Зачем использовать двоеточие в качестве имени функции?
  # Для запутывания кода.
}

:

# Имя этой функции - :
```

Это не переносимое поведение и поэтому не рекомендуется. В самом деле, более поздние релизы Bash не разрешают его использовать. Хотя *подчеркивания* _ работают.

Двоеточие может служить в качестве заполнителя в случае пустой функции.

```
not_empty ()
{
  :
} # Содержит : (команду null), а поэтому не является пустой.
```

!

противоположность (или отрицание) смысла проверки или статус выхода [напрямую]. Оператор **!** инвертирует *статус выхода* команды, с которой он применяется (см. Пример 6-2). Кроме того, инвертирует значение оператора проверки. Например, изменяет смысл *равно* (**=**) на *не равно* (**!=**). Оператор **!** является ключевым словом (***keyword***) Bash.

В другом контексте, **!** обозначает *косвенную ссылку на переменную*.

В еще одном контексте, **!**, из командной строки, вызывает *механизм Истории* Bash (см. Приложение L). Обратите внимание, что в сценарии механизм истории отключен.

*****

подстановка, замена (wild card) [звездочка]. Символ ***** служит «wild card», подстановкой, в расширении имени файла. Само по себе - это каждое соответствие в имени файла в данной директории.

```
bash$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

Также ***** представляет *любое количество (или ноль) символов в регулярном выражении*.

*****

арифметический оператор. В контексте арифметических операций, ***** обозначает ***умножение***.

****** Две звездочки можно представить оператором ***возведения в степень*** или расширенной подстановкой файловых соответствий.

?

оператор проверки. В рамках некоторых выражений **?** указывает на проверку соответствия условию.

В конструкции ***двойных скобок*** **?** может служить в качестве одного из элементов тройного оператора Си-типа. [2]

```
условие?результат-если-true:результат-если-false
```

```
(( var0 = var1<98?9:21 ))
#                ^ ^

# if [ "$var1" -lt 98 ]
# then
```

```
#   var0=9
# else
#   var0=21
# fi
```

В выражении подстановки параметра, **?** проверяет, *была ли установлена переменная*.

?

подстановка (wild card). Символ **?** служит ***односимвольной*** «wild card», подстановкой, в расширении имени файла, а также представляет *один* символ в *расширенном регулярном выражении*.

$

подстановка переменных (содержимого переменной).

```
var1=5
var2=23skidoo

echo $var1     # 5
echo $var2     # 23skidoo
```

Префикс **$** в имени переменной указывает на то, что переменная содержит *значение*.

$

конец строки. В регулярном выражении, **$** указывает *на окончание строки текста*.

${}

подстановка параметра

$'...'

расширяемая строка в кавычках. Эта конструкция экранирует расширение одного или нескольких восьмеричных или шестнадцатеричных значений в ASCII [3] или символов Unicode.

$*, $@

позиционные параметры

$?

переменная *статуса выхода*. Переменная **$?** содержит статус выхода *команды*, *функции* или самого *сценария*.

$$

ID переменной процесса. Переменная **$$** содержит *ID процесса* [4] сценария, в котором она находится.

()

группа команд

```
(a=hello; echo $a)
```

Список команд в *скобках* запускает ***subshell*** (дочернюю оболочку, **подоболочку**).

Переменные в скобках, в *subshell*, не видны для остальной части сценария. Родительскому процессу, сценарию, ***не удается прочитать переменные, создаваемые в дочернем процессе, subshell.***

```
a=123
( a=321; )
 ^      ^
echo "a = $a"   # a = 123
# "a" в скобках подобна локальной переменной.
```

Инициализация массива

```
Array=(элемент1 элемент2 элемент3)
```

{xxx,yyy,zzz,...}

расширение в фигурных скобках

```
echo \"{These,words,are,quoted}\"   # " префикс и суффикс
# "These" "words" "are" "quoted"


cat {file1,file2,file3} > combined_file
# Объединение файлов file1, file2, и file3 в combined_file.

cp file22.{txt,backup}
# Копирование "file22.txt" в "file22.backup"
```

Команда может работать со списком файлов разделенными запятыми находящимся в *фигурных скобках*. [5] Расширение имени файла (подстановка) относится к файлам находящимся между скобками.

Пробелы внутри фигурных скобок ***не допускаются***, пробелы должны заключаться в кавычки или экранироваться.

```
echo {file1,file2}\ :{\ A," B",' C'}

file1 : A file1 : B file1 : C file2 : A file2 : B
file2 : C
```

{a..z}

Расширенные расширения в фигурных скобках.

```
echo {a..z} # a b c d e f g h i j k l m n o p q r s t u v w x y z
# выводятся символы между a и z.

echo {0..3} # 0 1 2 3
# выводятся символы между 0 и 3.


base64_charset=( {A..Z} {a..z} {0..9} + / = )
#  Инициализация массива с помощью расширенных расширений в фигурных
#+ скобках.
# Из примера сценария "base64.sh" написанного vladz.
```

Конструкция расширенных расширений в фигурных скобках *{a..z}* - это функция, появившаяся в версии 3 Bash.

{}

Блок кода [фигурные скобки]. Также упоминается, как встроенная группа, эта конструкция, по сути, создает *анонимную функцию* (функцию без имени). Однако в отличие от «обычной» ***функции***, переменные, внутри блока кода, остаются видимыми для оставшейся части сценария.

```
bash$ { local a;
              a=123; }
bash: local: can only be used in a
function
```

```
a=123
{ a=321; }
echo "a = $a"   # a = 321   (значение внутри блока кода)

# Спасибо S.C.
```

Блок кода, заключенный в фигурные скобки может быть *перенаправлен в I/O и из него*.

Пример 3-1. Блок кода и перенаправление I/O

```
#!/bin/bash
# Чтение строк из /etc/fstab.

File=/etc/fstab

{
read line1
read line2
} < $File

echo "Первая строка $File это:"
echo "$line1"
echo
echo "Вторая строка $File это:"
echo "$line2"

exit 0
```

```
# Теперь, как вы разделите отдельные поля каждой строки?
# Подсказка: с помощью awk, или ...
# ... Hans-Joerg Diers предполагает использовать «set» встроенное в

#+ Bash.
```

Пример 3-2. Сохранение вывода блока кода в файл

```
#!/bin/bash
# rpm-check.sh

#  Запрос описания rpm файла, список, и можно ли файл установить.
#  Сохраняет вывод файла.
#
#  Сценарий иллюстрирует применение блока кода.

SUCCESS=0
E_NOARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` rpm-file"
  exit $E_NOARGS
fi

{ # Начало блока кода.
  echo
  echo "Описание архива:"
  rpm -qpi $1       # Запрос описания.
  echo
  echo "Листинг архива:"
  rpm -qpl $1       # Запрос списка файлов.
  echo
  rpm -i --test $1  # Запрос, может ли быть установлен файл rpm.
  if [ "$?" -eq $SUCCESS ]
  then
    echo "$1 может быть установлен."
  else
    echo "$1 не может быть установлен."
  fi
  echo              # Окончание блока кода.
} > "$1.test"       # Перенаправление полного вывода блока в файл.

echo "Результаты проверки rpm в файле $1.test"

# См. мануал rpm для объяснения опций.

exit 0
```

В отличии от вставленной **(** в круглые скобки **)**группы команд, заключенный блок кода **{**в фигурные скобки**}**, как выше, обычно не

запускает подоболочку. [6]

Можно повторять блок кода с помощью не обычного цикла ***for***.

{}

заполнитель в тексте. Используется после **`xargs -i`** (*опция перемещения строк*). Двойные фигурные скобки **{}** являются заполнителем вывода текста.

```
ls . | xargs -i -t cp ./{} $1
#            ^^         ^^

# Из примера "ex42.sh" (copydir.sh).
```

{} \;

путь к имени (pathname). В основном используется в конструкции ***find***. Она не встроена в оболочку.

> **Определение**: Pathname это имя файла, включающее в себя полный путь. Например `/home/bozo/Notes/Thursday/schedule.txt`. Иногда называется ***абсолютный*** путь.

"**;**"после опции `-exec` равнозначна команде ***find***. Она должна быть экранирована для защиты от интерпретации оболочкой.

[ ]

проверка.

Проверка выражения между **[ ]**. Обратите внимание, что **[** является частью *встроенной* в оболочку конструкции ***test*** (и синоним для нее), а не ссылкой на внешнюю команду `/usr/bin/test`.

[[ ]]

проверка.

Проверка выражения между **[[ ]]**. Более гибкая, чем проверка в одинарных скобках

`[ ]`, это ключевое слово (***keyword***) оболочки.

См. обсуждение конструкции **`[[...]]`**.

`[ ]`

элемент массива

В контексте массива, скобки определяют *нумерацию* каждого элемента этого массива.

```
Array[1]=slot_1
echo ${Array[1]}
```

`[ ]`

диапазон символов

Как часть регулярного выражения, скобки определяют соответствующий *диапазон символов*.

`$[ ... ]`

Целочисленное расширение.

Выражение вычисления целого числа между **`$[]`**.

```
a=3
b=7

echo $[$a+$b]   # 10
echo $[$a*$b]   # 21
```

Обратите внимание, что это устарело и заменено конструкцией **`((...))`**.

`(( ))`

Целочисленное расширение.

Расширение и оценка выражения целых чисел между **`(( ))`**.

См. обсуждение конструкции **`((...))`**.

`> &> >& >> < <>`

перенаправление.

`scriptname >filename` перенаправляет вывод `scriptname` в файл `filename`. Перезаписывает `filename`, если он уже существует.

команда &>filename перенаправляет оба `stdout` и `stderr` команды в `filename`.

Это полезно для подавления вывода при проверке условия. Например давайте проверим существование некоторой команды.

```
bash$ type bogus_command &>/dev/null



bash$ echo $?
1
```

Или в сценарии

```
command_test () { type "$1" &>/dev/null; }
#                                       ^

cmd=rmdir            # Законная команда.
command_test $cmd; echo $?   # 0


cmd=bogus_command    # Не законная команда
command_test $cmd; echo $?   # 1
```

команда >&2 перенаправляет `stdout` команды в `stderr`.

scriptname >>filename *добавляет* вывод `scriptname` в файл `filename`. Если `filename` не существует, он будет создан.

[i]<>filename открывает `filename` для чтения и записи, и присваивает ему файловый дескриптор **i**. Если `filename` не существует, он будет создан.

Процес подстановки (замены)

(команда)>

<(команда)

В другом контексте,символы **<** и **>** могут выступать в качестве операторов *сравнения строк*.

В еще одном контексте символы **<** и **>** могут выступать в качестве операторов *сравнения целых чисел*. См. Пример 16-9.

<<

Перенаправление используемое *here document.*

<<<

Перенаправление используемое *here string*.

<, >

Сравнение ASCII.

```
veg1=carrots
veg2=tomatoes

if [[ "$veg1" < "$veg2" ]]
then
  echo "Хотя $veg1 предшествует в словаре $veg2,"
  echo -n "это не обязательно говорит "
  echo "о моих гастрономических предпочтениях."
else
  echo "Какие словари вы используете?"
fi
```

\<, \>

граница слова в регулярном выражении.

```
bash$ grep '\<the\>' textfile
```

|

конвейер (туннель). Передает вывод (`stdout`) предыдущей команды на вход (`stdin`) следующей, или оболочке. Это способ связывания вместе команд.

```
echo ls -l | sh
#  передает вывод "echo ls -l" оболочке,
#+ с тем же результатом, как просто "ls -l".

cat *.lst | sort | uniq
#  Объединяет и сортирует все файлы ".lst", затем удаляет

#+ повторяющиеся строки.
```

Конвейер, как классический способ межпроцессного взаимодействия, направляет `stdout` одного процесса в `stdin` другого. В типичном случае команды, такие как **`cat`** или **`echo`**, передают поток данных в фильтр, команду, которая преобразует входные данные для обработки. [7]

```
cat $filename1 $filename2 | grep $search_word
```

Вывод команды или команд может быть передан конвейером в сценарий.

```
#!/bin/bash
# uppercase.sh : Изменение ввода на верхний регистр.

tr 'a-z' 'A-Z'
#  Диапазон букв должен быть помещен в кавычки
#+ для предотвращения создания файла из однобуквенных файлов.

exit 0
```

Теперь давайте туннелируем вывод ***ls -l*** в этот сценарий.

```
bash$ ls -l | ./uppercase.sh
-RW-RW-R--    1 BOZO  BOZO       109 APR  7 19:49 1.TXT
 -RW-RW-R--    1 BOZO  BOZO       109 APR 14 16:48 2.TXT
 -RW-R--R--    1 BOZO  BOZO       725 APR 20 20:56 DATA-FILE
```

`Stdout` каждого процесса в конвейере должен читаться, как `stdin` следующего. Если этого не происходит, то поток данных будет блокироваться, и конвейер не будет вести себя как ожидается.

```
cat file1 file2 | ls -l | sort
# Вывод из "cat file1 file2" исчезает.
```

Конвейер запускается как ***дочерний процесс*** и поэтому невозможно изменять переменные сценария.

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"     # variable = initial_value
```

Если одна из команд в конвейере будет прервана, то это преждевременно прекратит выполнение туннелирования. Это состояние называется *сломанным* конвейером, и посылает *сигнал* `SIGPIPE`.

>|

принудительное перенаправление (даже если установлена опция noclobber). Будет принудительно перезаписан существующий файл.

||

логический оператор ИЛИ В конструкции проверки, оператор **||** вызывает возвращение 0 (успешное завершение), если *одно из* связанных *условий* проверки верно.

&

запуск задания в фоновом режиме. Команда, за которой следует **&**, будет запущена в

фоновом режиме.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10


```

В сценарии, команды и даже циклы могут запускаться в фоновом режиме.

Пример 3-3. Запуск цикла в фоновом режиме

```
#!/bin/bash
# background-loop.sh

for i in 1 2 3 4 5 6 7 8 9 10            # Первый цикл.
do
  echo -n "$i "
done & # Этот цикл запускается в фоновом режиме.
       # Иногда будет выполняться после второго цикла.

echo   # Это 'echo' иногда не будет выводиться на экран.

for i in 11 12 13 14 15 16 17 18 19 20   # Второй цикл.
do
  echo -n "$i "
done

echo   # Это 'echo' иногда не будет выводиться на экран.

# ================================================

# Ожидаемый вывод сценария:
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20

# Иногда, однако, получите:
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (Второе 'echo' не выполнилось. Почему?)

# Иногда так:
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (Первое 'echo' не выполнилось. Почему?)

# Очень редко, что-то вроде этого:
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# Цикл в основном режиме  вытесняет цикл в фоновом режиме.

exit 0

#  Для полного веселья Nasimuddin Ansari предлагает
#+ после echo -n "$i" в строках 6 и 14,
#+ добавить  sleep 1.
```

Команда запущенная в сценарии в фоновом режиме может вызовом сценария повесить его, ожидая нажатия клавиши. К счастью, от этого есть средство.

&&

логический оператор И. В конструкции проверки, оператор **&&** вызывает возвращение `0` (успех), только если *оба* связанные **условия** проверки являются верны.

-

опция, префикс. Флаг опции для команды или фильтра. Префикс оператора. Префикс для *параметра по умолчанию* в *подстановке параметров*.

```
КОМАНДА -[Опция1][Опция2][...]
```

```
ls -al
```

```
sort -dfu $filename
```

```
if [ $file1 -ot $file2 ]
then #      ^
  echo "Файл $file1 старше $file2."
fi

if [ "$a" -eq "$b" ]
then #    ^
  echo "$a равно $b."
fi

if [ "$c" -eq 24 -a "$d" -eq 47 ]
then #    ^              ^
  echo "$c равно 24 а $d равно 47."
fi


param2=${param1:-$DEFAULTVAL}
#              ^
```

--

Двойное тире – префикс длинной (подробной) опции команды.

```
sort --ignore-leading-blanks
```

Используется Bash builtin, означает окончание опций для этой конкретной команды.

Это удобное средство удаления файлов, имена которых *начинаются с тире.*

```
bash$ ls -l
-rw-r--r-- 1 bozo bozo 0 Nov 25 12:29 -badname
```

```
bash$ rm -- -badname

bash$ ls -l
total 0
```

Двойное тире также используется в сочетании с ***set***.

set -- $переменная (как в Примере 15-18)

-

перенаправление из/в `stdin` или `stdout` [тире].

```
bash$ cat -
abc
abc

...

Ctl-D
```

Как и ожидалось, **`cat -`** выводит `stdin`, в случае пользовательского ввода, в `stdout`. А использовать перенаправляемый I/O - реальные приложения умеют?

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
# Перемещает все файловое дерево из одной директории в другую
# [любезность Alan Cox <a.cox@swansea.ac.uk>, с маленькими изменениями]

# 1) cd /source/directory
#    Переходим в директорию источника с файлами для перемещения.
# 2) &&
#   "оператор И": если операция 'cd' удачна,
#    то управление передается следующей команде.
# 3) tar cf - .
#    Опция 'c', команды архивирования 'tar', создает новый архив,
#    опция 'f' (файл), следующая '-' обозначает целевым файлом
#    stdout, и происходит это в текущем дереве директории ('.').
# 4) |
#    Конвейер в ...
# 5) ( ... )
#    дочернюю оболочку
# 6) cd /dest/directory
#    Переходим в директорию назначения.
# 7) &&
#   "оператор И", как и выше
# 8) tar xpvf -
#    Разархивация ('x'),с сохранением прав и владения файлов ('p'),
#    и отсылает подробное сообщение в stdout ('v'),
#    считывает данные из stdin (за 'f' следует '-').
#
#    Обратите внимание, что 'x' это команда, а 'p', 'v', 'f' - опции.
#
# Вот так!
```

```
# Равнозначно, но более элегантно:
#   cd source/directory
#   tar cf - . | (cd ../dest/directory; tar xpvf -)
#
#     Имеют такой же эффект:
# cp -a /source/directory/* /dest/directory
#     Или:
# cp -a /source/directory/* /source/directory/.[^.]* /dest/directory
#     Если существуют скрытые файлы в /source/directory.

bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -
#     --извлекаем tar файл--    | --затем передаем его "tar"--
#  Если «tar» не пропатчена для обработки «bunzip2», то нужно
#+ сделать в два дискретных шага, используя конвейер.
#  Цель заключается в разархивировании, сжатых bzip, исходных текстов #+
ядра.
```

Обратите внимание, что в этом контексте само «**-**» не является оператором Bash, а скорее вариантом, признанным некоторыми утилитами UNIX, которые пишут в `stdout` командами **`tar`**, **`cat`** и т.д.

```
bash$ echo "whatever" | cat -
whatever
```

Команда **`file`** ожидает, пока "**-**" перенаправит вывод в `stdout` (иногда можно увидеть **`tar cf`**), или примет ввод от `stdin`, а уж потом **`file`**. Это способ использования *файло-ориентированной* утилиты - ***конвейера фильтров***.

```
bash$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

В самой командной строке, **`file`** не выдает сообщение об ошибке.

Добавление "**-**" дает лучший результат. Оно заставляет оболочку ждать пользовательского ввода.

```
bash$ file -
abc
обычный ввод:              текст ASCII



bash$ file -
#!/bin/bash
standard input:              Bourne-Again shell script text executable
```

Теперь команда принимает ввод из `stdin` и анализирует его.

«**-**» может использоваться для туннелирования `stdout` в другие команды. Это позволяет делать такие трюки, как *добавление строк в файл*.

Сравнение файла с другим с помощью **diff**:

```
grep Linux file1 | diff file2 -
```

Наконец, реальный пример использования **-** с **tar**.

Пример 3-4. Резервное копирование всех файлов, измененных в прошедший день

```
#!/bin/bash

#  Резервное копирование всех файлов в текущей директории,
#+ измененных в течение последних 24 часов в «тарболе»
#+ (архивированных и сжатых файлов).

BACKUPFILE=backup-$(date +%m-%d-%Y)
#                  Встраивание даты в имя файла резервной копии.
#                  Спасибо за идею, Joshua Tschida.
archive=${1:-$BACKUPFILE}
#  Если  название архива резервного копирования файла не указано в
#+ командной строке, то по умолчанию будет "backup-MM-DD-YYYY.tar.gz."

tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Резервное копирование директории $PWD произведено в архивный
файл \"$archive.tar.gz\"."


#  Stephane Chazelas указывает, что приведенный выше
#+ код будет ошибочен, если будет слишком много обнаруживаемых
#+ файлов или какие-либо из имен файлов будут содержать пробелы.

# Он предложил следующую альтернативу:
# ------------------------------------------------------------------
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
#      с помощью GNU весии "find".


#   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
#      переносимая на другие UNIX подобные, но гораздо медленнее.
# ------------------------------------------------------------------


exit 0
```

Имена файлов, начинающиеся с " **-** " могут вызвать проблемы при сочетании с оператором перенаправления " **-** ". Сценарий должен это проверить и добавить соответствующий префикс таким именам файлов, например `./-FILENAME`, `$PWD/-FILENAME` или `$PATHNAME/-FILENAME`.

Если значение переменной начинается с **-**, это также может создать проблемы.

```
var="-n"
echo $var
```

```
# Имеет эффект "echo -n" и ничего не выведет.
```

-

предыдущая рабочая директория. Команда **cd-** меняет на предыдущую рабочую директорию. Она использует переменную среды $OLDPWD.

Не путайте " **-** " с оператором перенаправления " **-** ", который мы только что обсудили. Интерпретация " **-** "зависит от контекста, в котором оно появляется.

-

минус. Знак арифметической операции.

=

равно. Оператор присваивания

```
a=28
echo $a   # 28
```

В другом контексте " **=** " это *оператор сравнения строк*.

+

плюс. Арифметический оператор сложения.

В другом контексте, **+** это *оператор регулярного выражения*.

+

опция. Флаг опции для команды или фильтра.

Некоторые команды и ***builtins*** используют **+** для включения определенных параметров и их отключения. В подстановке параметров, **+** это префикс альтернативного *значения расширяемой переменной*.

%

модуль. Модуль (остаток от деления) арифметической операции.

```
let "z = 5 % 3"
```

```
echo $z  # 2
```

В другом контексте **%** это оператор ***соответствия шаблона.***

~

домашняя директория [тильда]. Соответствует внутренней переменной `$HOME`. `~bozo` это домашняя директория bozo, а **`ls ~bozo`** выводит ее содержимое. **`~/`** это домашняя директория текущего пользователя, а **`ls ~/`** выводит ее содержимое.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

bash$ echo ~nonexistent-user
~nonexistent-user
```

~+

текущая рабочая директория. Соответствует внутренней переменной `$PWD`.

~-

предыдущая рабочая директория. Соответствует внутренней переменной `$OLDPWD`.

=~

регулярное выражение соответствия. Этот оператор введен с 3 версии Bash.

^

начало строки. В *регулярном выражении*, " **^** "является местом ***начала строки*** *текста*.

^, ^^

Преобразование в верхний регистр в *подстановке параметров* (Добавлено в 4 версии Bash).

Управляющие символы

Изменяют поведение терминала или вывода текста. Управляющие символы это комбинации **CONTROL + клавиша** (нажатые одновременно). Символы управления могут быть написаны в восьмеричной и шестнадцатеричной форме и предварены *управляющим* символом (\).

Управляющие символы обычно не приносят пользы внутри сценария.

- **Ctl-A** Перемещает курсор в начало строки текста (в командной строке).
- **Ctl-B** Backspace **- возврат** (не разрушающий).
-
- **Ctl-C** Break **- Прерывание**. Прерывание выполнения задачи в фоновом режиме.
- **Ctl-D** *Выход* из оболочки (аналогично **exit**).

  EOF (end-of-file — окончание файла). Прекращает ввод из stdin.

  При вводе текста в консоли или в окне *Xterm*, **CTL-D** удаляет символ под курсором. Когда нет символов, **CTL-D** производит выход из сессии, как и ожидалось. В окне *XTerm*, это эффект закрытия окна.
- **Ctl-E** Перемещает курсор в конец строки (в командной строке).
- **Ctl-F** Перемещает курсор вперед на один символ (в командной строке).
- **Ctl-G** ЗВОНОЧЕК. На некоторых старых телетайпах на самом деле зазвонит звоночек. В *Xterm* - это подача звукового сигнала.
- **Ctl-H** Rubout (разрушающий возврат). Стирает символы, находящиеся позади курсора.

```
#!/bin/bash
# Встраивание Ctl-H в строку.

a="^H^H"                  # Два Ctl-H — возврат на две позиции
                          # ctl-V ctl-H, с помощью vi/vim
echo "abcdef"             # abcdef
echo
echo -n "abcdef$a "       # abcd f
# Пропуск в конце^             ^  Возврат на вторую позицию.
echo
echo -n "abcdef$a"        # abcdef
#  Без пропуска в конце         ^ Не возвращается (почему?)
                          # Результаты могут оказаться не
                          #+ ожидаемыми.
echo; echo

# Constantin Hagemeier:
# a=$'\010\010'
```

```
# a=$'\b\b'
# a=$'\x08\x08'
# Но это ничего не меняет.

########################################

# Теперь попробуем это.

rubout="^H^H^H^H^H"       # 5 x Ctl-H.

echo -n "12345678"
sleep 2
echo -n "$rubout"
sleep 2
```

- **Ctl-I** Горизонтальная табуляция
- **Ctl-J** НоваяСтрока (перевод строки). В сценарии может быть выражена в восьмеричном – «\012» или в шестнадцатеричном – «\x0a» формате.
- **Ctl-K** Вертикальная табуляция

  При вводе текста в консоле или в окне *xterm*, **Ctl-K** стирает все от символа позиции курсора до конца строки. Внутри сценария **Ctl-K** может вести себя по-разному, как в примере Lee Maschmeyer, ниже.
- **Ctl-L** Formfeed - Перевод страницы (очистка экрана терминала). В терминале имеет тот же эффект, что и команда **clear**. При отправке на принтер, **Ctl-L** вызывает исполнение до конца листа бумаги.
- **Ctl-M** Перевод каретки

```
#!/bin/bash
# Благодарность за этот пример Lee Maschmeyer.

read -n 1 -s -p \
$'Control-M отведет курсор в начало строки. Нажмите Enter. \x0d'
          # '0d' это шестнадцатеричный эквивалент Control-M.
          # '-s', когда что-нибудь набрано, то она необходима для
          #+ явного перехода к новой строке.
echo >&2

read -n 1 -s -p $'Control-J устанавливает курсор в новую строку. \x0a'
          #  '0a' шестнадцатеричный эквивалент Control-J,
          #+ перевода строки.
echo >&2

###

read -n 1 -s -p $'Control-K \x0b перемещает курсор прямо вниз.'
echo >&2   #  Control-K это вертикальная табуляция.

# Более лучший пример вертикальной табуляции:
```

```
var=$'\x0aЭто нижняя строка\x0bЭто верхняя строка\x0a'
echo "$var"
#  Это работает так же, как в приведенном выше примере. Но:
echo "$var" | col
#  Правое окончание строки получается выше, чем  левое окончание.
#  Это также объясняет, почему мы начали и закончили
#+ переводом строки – чтобы избежать искажения на экране.

# Как объяснил Lee Maschmeyer:
# -------------------------
#  В [примере первой вертикальной табуляции] ...
#+ вертикальная табуляция
#+ делает вывод на экран идущим прямо вниз без возврата каретки.
#  Это работает только в таких устройствах, как консоль Linux,
#+ которая не возвращает 'назад'.
#  Реальная задача вер.таб.- это двигаться прямо вверх, а не вниз.
#  Может использоваться для печати на принтере  надстрочных знаков.
#  Утилита col может использоваться для имитации поведения вер.таб.

exit 0
```

- **Ctl-N** Удаляет строку текста из *буфера истории* [8] (в командной строке).
- **Ctl-O** Символ *новой* строки (в командной строке).
- **Ctl-P** Вызывает снова последнюю команду из буфера истории (в командной строке).
- **Ctl-Q** Возобновление (**XON**).

  Возобновление `stdin` в терминале.
- **Ctl-R** Обратный поиск текста в *буфере истории* (в командной строке).
- **Ctl-S** *Приостановка* (**XOFF**).

  Замораживает `stdin` в терминале. (С помощью **Ctl-Q** восстанавливается input.)
- **Ctl-T** Изменяет позицию символа, где находится курсор на предыдущий символ (в командной строке).
- **Ctl-U** Удаляет введенную строку, от курсора до начала строки. В некоторых ситуациях **Ctl-U** удаляет всю введенную строку, *независимо от позиции курсора*.
- **Ctl-V** При вводе текста, **Ctl-V** позволяет вставлять управляющие символы. Например, оба следующих являются эквивалентами:

```
echo -e '\x0a'
echo <Ctl-V><Ctl-J>
```

CTL-V главным образом полезна в текстовом редакторе.

- **Ctl-W** При вводе текста в консоли или в окне *xterm*, **Ctl-W** стирает символ под курсором и сзади до первого ***пробела***. В некоторых ситуациях **Ctl-W** стирает сзади первый не алфавитно-цифровой символ.

- **Ctl-X** В некоторых программах обработки текстов *вырезает* подчеркнутый текст и копирует в *буфер* обмена.

- **Ctl-Y** Вставляет обратно, ранее стертый, текст (с **Ctl-U** или **Ctl-W**).

- **Ctl-Z** *Пауза* выполнения задачи в режиме переднего плана.

Операции *подстановки* в некоторых приложениях обработки текстов.

Символ **EOF** (конец файла) в MSDOS .

Пробелы (пропуски, пустое место)

функции разделения между командами или переменными. Пробелы состоят из *пробелов*, табуляций, *пустых строк* или любых их комбинаций. [9] В некоторых контекстах, таких как **присваивание переменных**, пробелы не разрешаются и приводят к ошибке синтаксиса.

Пустые строки не влияют на действие сценария и поэтому полезны для визуального разделения функциональных частей.

$IFS, специальная переменная, вводящая разделительные поля в определенных командах. В пробелах — по умолчанию.

> **Определение**: Поле — дискретная часть данных, выраженных, как строка последовательных символов. Разделителем каждого поля от смежных полей является пробел или другой назначенный символ, (часто определяющийся **$IFS**). В некоторых контекстах, поле может называться *record*.

Чтобы сохранить *пробелы* в строке или в переменной - заключайте их в ***кавычки***.

Фильтры UNIX могут ориентироваться и оперировать пробелами с помощью класса символов POSIX **[:space:]**.

Примечания

[1] Оператор — это агент, который выполняет операцию. Примерами являются обычные ***арифметические операторы*** **+ - * /**. В Bash существует некоторое дублирование

между понятиями *оператор* и *ключевое слово* (***keyword***).

[2] Более известно, как *троичный* оператор. К сожалению, *троичный* уродливое слово. Оно не скатывается с языка и он не проливает свет. Оно запутывает. Гораздо более элегантнее - ***тройной***.

[3] **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. Это система для кодирования текстовых символов (букв, чисел и ограниченного набора символов), как 7-битных чисел, которые могут храниться и управляться компьютерами. Многие из символов ASCII представлены на стандартной клавиатуре.

[4] *PID* или *идентификатор процесса* - это число назначаемое выполняющемуся процессу. С помощью команды **ps** может просмотреть PID запущенных процессов.

Определение: Процесс это выполняемая в настоящее время команда (или программа), иногда упоминается как *задача*.

[5] Оболочка производится в *скобках расширения*. Сама команда воздействует на *результат* расширения.

[6] Исключение: блок кода в фигурных скобках, как часть конвейера *может* выполняться как дочерняя оболочка (subshell).

```
ls | { read firstline; read secondline; }
#  Ошибка. Блок кода в скобках запускается как subshell,
#+ поэтому вывод "ls" не будет передан переменной в блоке.
echo "Первая строка $firstline; вторая строка $secondline"  # Не работает.

# Спасибо S.C.
```

[7] Как и в былые времена, philtre обозначает зелье обладающее волшебной превращающей силой, вот и UNIX фильтры преобразуют (примерно) аналогичным образом. (Кодер, имеющий«любимый philtre», работающий на Linux машине, имеет все шансы выиграть награды и почести).

[8] Bash сохраняет список ранее запущенных команд командной строки в *буфере*, или пространстве памяти, для запоминания во **встроенной** *истории команд*.

[9] Перевод строки (*новая строка*) является также пробелом. Это объясняет, почему *пустая строка*, состоящая только из перевода строки, является пробелом.

Глава 4. Введение в переменные и параметры

Содержание

Переменные, как в программировании, так и в языках сценариев, являются данными. Переменная не более чем ярлык, имя, назначенное месту или совокупности мест в памяти

компьютера, где находятся элементы данных.

Переменные появляются в арифметических операциях, и при изменении количества, и при разборе строк.

4.1. Подстановка переменных

Имя переменной является местозаполнителем для его значения, данных, которые переменная содержит. Ссылка (извлечение) на ее значение называется *подстановкой переменой*.

$

Давайте внимательно разберем отличие *имени* переменной от ее *значения*. Если **variable1** это *имя* переменной, то **$variable1** это *ссылка* на ее значение, элемент данных, которые она содержит. [1]

```
bash$ variable1=23


bash$ echo variable1
variable1

bash$ echo $variable1
23
```

Единственный раз переменная появляется «*голой*» -- без префикса **$** -- это когда объявление или назначение отключено, при экспорте, в арифметическом выражении в **двойных скобках ((...))**, или в случае, когда переменная представляет ***сигнал*** (см. Пример 32-5). Присваивается символом **=** (*var1=27*), оператором **read**, и в заголовке цикла (*for var2 in 1 2 3*).

Заключение значения ссылки в двойные кавычки (**"**...**"**) не мешает подстановке переменных. Они называются частичными кавычками, иногда называемыми "слабыми кавычками". С помощью одинарных кавычек (**'**...**'**) вызываемое имя переменной будет пониматься в буквальном смысле и подстановка не состоится. Это полные кавычки, иногда называемые «сильными кавычками". См. Главу 5 где они детально обсуждаются.

Обратите внимание, что **$variable** это упрощенная форма **${variable}**. В контекстах, где синтаксис **$variable** приводит к ошибке полная форма может работать (см. Раздел 10.2).

Пример 4-1. Присваивание переменных и подстановка

```
#!/bin/bash
# ex9.sh

# Переменные: присваивание и подстановка
```

```
a=375
hello=$a   # Нет пробелов!
#   ^ ^

#-----------------------------------------------------------------------
#  При инициализации переменных не допускаются пробелы по обе стороны от
#+ знака =.
# Что случиться, если будут пробелы?

#  "VARIABLE =value"
#           ^
#% Сценарий попытается запустить команду "VARIABLE" с одним аргументом,
#+ "=value".

#  "VARIABLE= value"
#            ^
#% Сценарий попытается запустить команду "value" с
#+установкой переменной окружения "VARIABLE" "".
#-----------------------------------------------------------------------


echo hello    # hello
# Это не ссылка на переменную, просто строка "hello" ...

echo $hello   # 375
#    ^           *Это* ссылка на переменную.
echo ${hello} # 375
#               Такая же ссылка на переменную, как выше.

# Заключим в кавычки ...
echo "$hello"    # 375
echo "${hello}"  # 375

echo

hello="A B  C   D"
echo $hello   # A B C D
echo "$hello" # A B  C   D
# Как видим, echo $hello и echo "$hello" выдают различные результаты.
# ======================================
#  Переменная заключенная в кавычки сохраняет пробелы.
# ======================================

echo

echo '$hello'  # $hello
#    ^      ^
#  Одинарные кавычки отключают (экранируют) ссылки на переменную,
#+ интерпретируя «$» буквально.

# Обратите внимание на различие эффекта различных типов кавычек.

hello=    # Устанавливаем в нулевое значение.
echo "\$hello (null value) = $hello"      # $hello (null value) =
#  Обратите внимание, что присвоение переменной нулевого значения это
#+ не то же самое, что не присвоение ей ничего, хотя конечный результат
#+ тот же (см. ниже).

# ----------------------------------------------------------------
```

```
#  Допустимо задание нескольких переменных в одной строке, если они
#+ разделяются пробелами.
#  Осторожно, это может ухудшить читабельность и не может быть
#+ портировано.

var1=21  var2=22  var3=$V3
echo
echo "var1=$var1   var2=$var2   var3=$var3"

# Могут быть проблемы с устаревшими версиями "sh" ...

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

echo; echo

numbers="one two three"
#           ^   ^
other_numbers="1 2 3"
#               ^ ^
#  Если переменная содержит пробелы,
#+ то заключение в кавычки - обязательно.
#  other_numbers=1 2 3                  # Выдаст сообщение об ошибке.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"   # other_numbers = 1 2 3
#  Экранирование пробелов так же сработает.
mixed_bag=2\ ---\ Whatever
#           ^    ^ Пробел после управляющего символа (\).

echo "$mixed_bag"         # 2 --- Whatever

echo; echo

echo "uninitialized_variable = $uninitialized_variable"
#  Не инициализированная переменная имеет значение null (нет значения
#+ вообще!).
uninitialized_variable=   #  Объявляется, но не инициализируется --
                          #+ то же, что и установить ее в значение null,
                          #+ как выше.
echo "uninitialized_variable = $uninitialized_variable"
                          # Все еще имеет значение null.

uninitialized_variable=23       # Присваиваем ее.
unset uninitialized_variable    # Сбрасываем ее значение.
echo "uninitialized_variable = $uninitialized_variable"
                                # uninitialized_variable =
                                # Все еще имеет значение null.
echo

exit 0
```

Не инициализированная переменная имеет значение "**null**" - т.е. не присвоено вообще никакого значения (не *ноль*!).

```
if [ -z "$unassigned" ]
then
  echo "\$unassigned это NULL."
fi     # $unassigned это NULL.
```

Использование переменной до присвоения ей значения может вызвать проблемы. Тем не менее, над не инициализированной переменной можно выполнять арифметические операции.

```
echo "$uninitialized"                   # (пустая строка)
let "uninitialized += 5"                # Увеличиваем ее на 5.
echo "$uninitialized"                   # 5

#  Объяснение:
#  Не инициализированная переменная не имеет значения, однако
#+ ее выражение в арифметической операции оценивается как 0.
```

См. Пример 15-23.

Примечания

[1] Технически, *имя* переменной вызывается ***lvalue***, означающее, что оно (имя) находится в *левой* части оператора присваивания, как в **VARIABLE=23**. *Значение* переменной это ***rvalue***, означающее, что оно (значение) находится *справа* от оператора присваивания, как в **VAR2=$VARIABLE**.

Имя переменной является, по сути, *ссылкой*, указателем на место в памяти, где хранятся *фактические данные*, связанные с этой переменной.

4.2. Присваивание переменной

=

оператор присваивания (***без пробелов до*** *него и* ***после*** *него*)

Не путайте его с **=** и **-eq** , которые *проверяют*, а не присваивают!

Обратите внимание, что **=** , в зависимости от контекста, может быть либо оператором *присваивания*, либо оператором *проверки*.

Пример 4-2. Обычное присваивание переменной

```
#!/bin/bash
# Голые переменные
```

```
echo

# Когда переменная «голая», когда спереди отсутствует «$»?
# Когда она назначается, но не ссылается.

# Присваивание
a=879
echo "Значение \"a\" это $a."

# Присваивание с помощью 'let'
let a=16+5
echo "Значение \"a\" теперь $a."

echo

# Циклом 'for' (на самом деле замаскированное присваивание):
echo -n "Значение \"a\" в цикле: "
for a in 7 8 9 11
do
  echo -n "$a "
done

echo
echo

# Оператором 'read' (тат же тип присваивания):
echo -n "Введите \"a\" "
read a
echo "Значение \"a\" теперь $a."

echo

exit 0
```

Пример 4-3. Присваивание переменной, обычное и не обычное

```
#!/bin/bash

a=23              # Простой случай
echo $a
b=$a
echo $b

# Теперь узнаем немного побольше (подстановка команды).

a=`echo Hello!`   # Присвоение результата команды 'echo' в 'a' ...
echo $a
#  Обратите внимание, что восклицательный знак (!)  находящийся в пределах
#+ конструкции подстановки команды не будет работать из командной строки,
#+ так как он вызывает «механизм истории» Bash.
#  Внутри сценария функция истории, по умолчанию, отключена.

a=`ls -l`         # Присвоение результата команды 'ls -l' в 'a'
echo $a           # Без кавычек - удалит всю табуляцию и пробелы.
echo
echo "$a"         # Переменная заключенная в кавычки сохраняет пробелы.
                  # (См главу "Кавычки.")
```

```
exit 0
```

Присваивать переменные можно с помощью механизма **$(...)** (новый способ, в отличии от обратных кавычек). Это также форма подстановки команд.

```
# Из /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

4.3. Не типизированные переменные Bash

В отличие от многих других языков программирования, Bash не обособляет свои переменные «типами.» По существу, переменные Bash являются *символьными строками*, но, в зависимости от контекста, Bash допускает арифметических операции и сравнения переменных. Определяющим фактором является содержание значением переменной только цифр.

Пример 4-4. Целые числа или строки?

```
#!/bin/bash
# int-or-string.sh

a=2334                   # Целое число.
let "a += 1"
echo "a = $a "           # a = 2335
echo                     # Все еще целое число.


b=${a/23/BB}             # Подставляем "BB" вместо "23".
                         # Это превращает значение $b в строку.
echo "b = $b"            # b = BB35
declare -i b             # Объявление ее как целого числа не помогает.
echo "b = $b"            # b = BB35

let "b += 1"             # BB35 + 1
echo "b = $b"            # b = 1
echo                   # Bash устанавливает "значение целого числа" строки в 0.

c=BB34
echo "c = $c"            # c = BB34
d=${c/BB/23}             # Подставляем "23" вместо "BB".
                         # Это делает $d целым числом.
echo "d = $d"            # d = 2334
let "d += 1"             # 2334 + 1
```

```
echo "d = $d"           # d = 2335
echo


# А как с null переменными?
e=''                    # ... Или e="" ... Или e=
echo "e = $e"           # e =
let "e += 1"            # Возможны арифметические операции с null переменными?
echo "e = $e"           # e = 1
echo                    # Null переменная превращается в целое число.

# А как с не объявленными переменными?
echo "f = $f"           # f =
let "f += 1"            # Возможны арифметические операции?
echo "f = $f"           # f = 1
echo                    # Не объявленная переменная превращается в целое
                        #+ число.
#
# Однако ...
let "f /= $undecl_var"  # Делит на ноль?
#   let: f /= : ошибка синтаксиса: ожидаемый операнд (маркер ошибки " ")
# Синтаксическая ошибка! Переменной $undecl_var здесь не присваивается ноль!
#
# Но все еще ...
let "f /= 0"
#   let: f /= 0: деление на 0 (маркер ошибки "0")
# Ожидаемое поведение.

#  При выполнении арифметических операций, Bash (как правило) устанавливает
#+ "целочисленное значение" null в ноль.
#  Люди, не пытайтесь повторить это дома!
#  Это недокументированное и, вероятно, не переносимое поведение.


# Объяснение: Переменные в Bash не типизированы,
#+ со всеми вытекающими последствиями.

exit $?
```

Не типизированные переменные являются одновременно и благословением и проклятием. Они позволяют сценариям быть более гибкими и делают более легким оттачивание строк кода (и дают вам достаточно веревки, чтобы повесится!). Однако они также позволяют допускать трудноуловимые ошибки и поощряют привычку небрежного программирования.

Для облегчения бремени по отслеживанию типов переменных в сценарии, Bash разрешает ***объявлять*** переменные.

4.4. Специальные типы переменных

Локальные переменные

Переменные, *видимые* только в пределах *блока кода* или функции (см. Локальные

переменные в функциях)

Переменные окружения

Переменные, которые влияют на поведение оболочки и пользовательского интерфейса.

В более общем контексте, каждый ***процесс*** имеет «окружающую среду», то есть, группу переменных, которые могут ссылаться на этот процесс. В этом смысле оболочка ведет себя как любой другой процесс.

Каждый раз при запуске оболочки создаются переменные *среды*, которым соответствуют свои собственные переменные среды. Обновление или добавление новых переменных среды приводит к обновлению среды оболочки, и все оболочки дочерних процессов (команды, которые она выполняет) наследуют эту среду.

Пространство, отводимое для окружающей среды ограниченно. Создание большого количества переменных среды, использующих чрезмерное пространство, может вызвать проблемы.

```
bash$ eval "`seq 10000 | sed -e 's/.*/export
var&=ZZZZZZZZZZZZZZ/'`"

bash$ du
bash: /usr/bin/du: Список аргументов очень велик
```

Примечание: Эта ошибка была исправлена, начиная с ядра версии 2.6.23.

(Спасибо Stéphane Chazelas за разъяснения и за приведенный выше пример.)

Если сценарий задает переменные среды, то они должны быть «экспортируемы», то есть, сообщать сценарию локальную среду. Это функция команды **`export`**.

Сценарий может экспортировать переменные только в дочерние процессы, то есть, только в команды или процессы, которые создал конкретный сценарий. Сценарий, вызываемый из командной строки, *`не может`* экспортировать переменные обратно, в среду командной строки. *Дочерние процессы не могут экспортировать переменные обратно в родительские процессы, которые их породили.*

`Определение`: *Дочерний процесс* это подпроцесс запущенный другим процессом, его родителем.

Позиционные параметры

Аргументы, передаваемые в сценарий из командной строки [1] : $0, $1, $2, $3 . . .

$0 это название самого сценария, **$1** это первый аргумент, **$2** второй, **$3** третий, и так же четвертый и т.д. . [2] после аргумента **$9,** должны заключаться в фигурные скобки, например, **${10}**, **${11}**, **${12}**.

Специальные переменные **$*** и **$@** обозначают *все* позиционные параметры.

Пример 4-5. Позиционные параметры

```
#!/bin/bash

# Этот сценарий вызывается, по крайней мере, с 10 параметрами,
# например: ./scriptname 1 2 3 4 5 6 7 8 9 10
MINPARAMS=10

echo

echo "Название этого сценария \"$0\"."
# Добавляем ./ в текущей директории
echo "Название этого сценария \"`basename $0`\"."
# информация о пути имени (см 'basename')

echo

if [ -n "$1" ]              # Проверьте, переменные должны быть в
                            #+ кавычках.
then
 echo "Параметр #1 это $1"  # Нужны кавычки для экранирования #
fi

if [ -n "$2" ]
then
 echo "Параметр #2 это $2"
fi

if [ -n "$3" ]
then
 echo "Параметр #3 это $3"
fi

# ...


if [ -n "${10}" ]  # Параметры > $9 должны быть в {фигурных скобках}.
then
 echo "Параметр #10 это ${10}"
fi

echo "-----------------------------------"
echo "Все параметры командной строки: "$*""

if [ $# -lt "$MINPARAMS" ]
then
  echo
  echo "Этому сценарию нужно хотя бы $MINPARAMS аргументов командной
строки!"
fi
```

```
echo

exit 0
```

Нотация *скобок* для позиционных параметров указывает на довольно простой способ ссылки на последний аргумент, переданный в сценарий командной строкой. В них также нуждаются и косвенные ссылки.

```
args=$#           # Число передаваемых аргументов.
lastarg=${!args}
# Обратите внимание: это *косвенная ссылка*(!) на $args ...


# Или:       lastarg=${!#}             (Спасибо Chris Monson.)
# Это *косвенная ссылка* на переменную $#.
# Обратите внимание, что lastarg=${!$#} не будет работать.
```

Некоторые сценарии могут выполнять различные операции, в зависимости от того, с каким именем они вызываются. Для этого необходимо проверить **$0** сценария, имя с которым он был вызван. [3] Также должны существовать символические ссылки на все альтернативные имена сценария. См. Пример 16-2.

Если сценарий ожидает параметра командной строки, а вызывается без него, то это может привести к *присвоению переменной значения* ***null***, и,как правило, не желательному результату. Один из способов избежать этого - добавление дополнительных символов с обеих сторон оператора присваивания, с использованием ожидаемого позиционного параметра.

```
variable1_=$1_  # Вместо variable1=$1
# Это предотвращает ошибку, даже если позиционный параметр отсутствует.

critical_argument01=$variable1_

#  Дополнительный символ может быть снят позже, как здесь.
#+ variable1=${variable1_/_/}
#  Побочные эффекты могут быть, только если $variable1_ начинается с
#+ символа подчеркивания.
#  При этом используется один из шаблонов подстановки параметра,
#+ обсуждаемый позднее.
# (Игнорирование шаблона подстановки приводит к удалению.)

#  Наиболее простым способом борьбы с этим является простая проверка
#+ ожидаемых передаваемых позиционных параметров.
if [ -z $1 ]
then
  exit $E_MISSING_POS_PARAM
fi


#  Как поясняет Fabian Kreutz,
#+ способ выше, может иметь неожиданные побочные эффекты.
#  Лучший способ — это подстановка параметра:
```

```
#         ${1:-$DefaultVal}
#  См. раздел "Подстановка параметров"
#+ главе "Возвращения переменных".
```

Пример 4-6. Поиск доменного имени *wh, whois*

```
#!/bin/bash
# ex18.sh

# Производит поиск 'whois domain-name' на 3-х различных серверах:
#                    ripe.net, cw.net, radb.net

# Поместите этот сценарий -- переименовав 'wh' -- в /usr/local/bin

# Требуемые символические ссылки:
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-apnic
# ln -s /usr/local/bin/wh /usr/local/bin/wh-tucows

E_NOARGS=75


if [ -z "$1" ]
then
  echo "Usage: `basename $0` [доменное_имя]"
  exit $E_NOARGS
fi

# Проверка имени сценария и вызов правильного сервера.
case `basename $0` in    # Или:    case ${0##*/} in
    "wh"       ) whois $1@whois.tucows.com;;
    "wh-ripe"  ) whois $1@whois.ripe.net;;
    "wh-apnic" ) whois $1@whois.apnic.net;;
    "wh-cw"    ) whois $1@whois.cw.net;;
    *          ) echo "Usage: `basename $0` [доменное_имя]";;
esac

exit $?
```

Команда **shift** переназначает позиционные параметры, перемещая их влево на одно положение.

$1 <--- **$2, $2** <--- **$3, $3** <--- **$4**, и т.д..

Старый **$1** исчезает, а **$0** *(имя сценария) не изменяется*. Если вы используете в сценарии большое количество позиционных параметров, сдвиг позволяет получить доступ к **10** прошлым, нотация **{**скобок**}** также это позволяет.

Пример 4-7. Использование *shift*

```
#!/bin/bash
#  shft.sh: Использование 'shift' для сдвига всех позиционных
#+ параметров.

#  Назовите этот сценарий наподобие shft.sh,
#+ и вызовите его с какими-нибудь параметрами.
#+ Например:
#             sh shft.sh a b c def 83 barndoor

until [ -z "$1" ]  # Пока все параметры не будут использованы ...
do
  echo -n "$1 "
  shift
done

echo               # Дополнительный перевод строки.

# Но, что происходит с "уже использованными" параметрами?
echo "$2"
#  Ничего не выводится на экран!
#  Когда $2 смещается в $1 (а $3 не смещается в $2)
#+ то $2 остается пустым.
#  Таким образом, это параметр не *копирования*, а *перемещения*.

exit

#  См. также сценарий echo-params.sh  для альтернативного «без
#+ смещаемого» способа пошагового перемещения позиционных

#+ параметров.
```

Команда **shift** может принимать числовой параметр, указывающий, на сколько позиций перемещаться.

```
#!/bin/bash
# shift-past.sh

shift 3    # Перемещение на 3 позиции.
#  n=3; shift $n
#  Будет иметь какой-то эффект.

echo "$1"

exit 0

# ======================== #


$ sh shift-past.sh 1 2 3 4 5
4

#  Однако, как указывает Eleni Fragkiadak, попытка 'сдвига'
#+ вдоль ряда позиционных параметров ($#) возвращает статус
#+ выхода 1 и позиционные параметры сами по себе не изменяются.
#  Это означает возможность застрять в бесконечном цикле....
#  Например:
#      until [ -z "$1" ]
```

```
#       do
#          echo -n "$1 "
#          shift 20     #  Если менее 20 позиционных параметров,
#       done            #+ то цикл никогда не закончится!
#
# Если вы сомневаетесь, добавьте проверку вменяемости....
#            shift 20 || break
#                     ^^^^^^^^
```

Команда **shift** работает аналогично передаче параметров **function**. См. Пример 36-18.

Примечания

[1] Обратите внимание, что функции также принимают позиционные параметры.

[2] Процесс, вызывающий сценарий, устанавливает параметр **$0**. По соглашению, этот параметр является именем сценария. Обратитесь к справочной странице **execv**.

Из командной строки, `$0` это имя оболочки.

```
bash$ echo $0
bash

tcsh% echo $0
tcsh
```

[3] Если сценарий это источник или символьная ссылка, то не сработает. Безопаснее проверить `$BASH_Source`.

Глава 5. Кавычки

Содержание

Заключение в кавычки то же самое, что и строка в скобках заключенная в кавычки. Они обладают эффектом защиты ***специальных символов*** от интерпретации в строке или расширения оболочки или сценария оболочки. (Символ является "специальным", если он имеет интерпретацию, кроме его буквального смысла. Например, ***звездочка*** ***** обозначает символ подстановки в ***шаблоне*** в и ***регулярном выражении***).

```
bash$ ls -l [Vv]*
-rw-rw-r--    1 bozo  bozo       324 Apr  2 15:05 VIEWDATA.BAT
 -rw-rw-r--    1 bozo  bozo       507 May  4 14:25 vartrace.sh
 -rw-rw-r--    1 bozo  bozo       539 Apr 14 17:11 viewdata.sh

bash$ ls -l '[Vv]*'
ls: [Vv]*: Нет такого файла или директории
```

В повседневной речи или письменной форме, когда мы фразу заключаем в кавычки, мы выделяем эту часть и придаем ей особое значение. В сценарии Bash, когда мы заключаем

строку в кавычки, мы отделяем ее и защищаем ее *буквальное* значение.

Некоторые программы и утилиты интерпретируют или расширяют специальные символы в кавычках. Важное использование заключения в кавычки - это защита параметров командной строки из оболочки, которая все еще позволяет вызывающей программе расширять ее.

```
bash$ grep '[Ff]irst' *.txt
file1.txt:This is the first line of file1.txt.
 file2.txt:This is the First line of file2.txt.
```

Обратите внимание, что не заключенная в кавычки **`grep [Ff]irst *.txt`** , работает под оболочкой Bash. [1]

Заключение в кавычки может подавлять "аппетит" (развернутый вывод) новых строк **`echo`.**

```
bash$ echo $(ls -l)
total 8 -rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh -rw-rw-r-- 1 bo bo 78 Aug 21
12:57 u.sh


bash$ echo "$(ls -l)"
total 8
 -rw-rw-r--  1 bo bo  13 Aug 21 12:57 t.sh
 -rw-rw-r--  1 bo bo  78 Aug 21 12:57 u.sh
```

Примечания

[1] Если нет файла с именем ***first*** в текущей рабочей директории. Есть еще и другая причина, чтобы заключить в кавычки. (Спасибо за разъяснение Harald Koenig.)

5.1. Переменные, заключенные в кавычки

При обращении к ***переменной***, вообще целесообразно, заключать ее имя в ***двойные*** кавычки. Это предотвращает другое интерпритирование всех специальных символов в строке заключенной в кавычки--за исключением **`$`**, **`` ` ``** (надстрочного) и **`\`**(экранирование). [1] Сохранение **`$`**, как специального символа, в двойных кавычках разрешает ссылаться на переменную, заключенную в кавычки (**`"$variable"`**), то есть, заменять переменную ее значением (см. Пример 4-1, выше).

С помощью двойных кавычек предотвращается разделение слова. [2] Аргумент,

заключенный в двойные кавычки позиционирует себя как одно слово, даже если он содержит разделительные пробелы.

```
List="one two three"

for a in $List     # Переменная разделенная на части пробелами.
do
  echo "$a"
done
# one
# two
# three

echo "---"

for a in "$List"   # Сохраняют пробелы в переменной.
do #     ^     ^
  echo "$a"
done
# one two three
```

Более сложный пример:

```
variable1="a variable containing five words"
COMMAND Это $variable1    # Выполняет COMMAND с 7 аргументами:
# "This" "is" "a" "variable" "containing" "five" "words"

COMMAND "Это $variable1"  # Выполняет COMMAND с 1 аргументом:
# "This is a variable containing five words"


variable2=""    # Пустая.

COMMAND $variable2 $variable2 $variable2
                # Выполняет COMMAND без аргументов.
COMMAND "$variable2" "$variable2" "$variable2"
                # Выполняет COMMAND с 3 пустыми аргументами.
COMMAND "$variable2 $variable2 $variable2"
                # Выполняет COMMAND с 1 аргументом (2 пробела).

# Спасибо Stéphane Chazelas.
```

Заключать аргументы инструкции **echo** в двойные кавычки необходимо только тогда, когда слово может быть разделено или важно сохранение пробелов.

Пример 5-1. Вывод на экран необычных переменных

```
#!/bin/bash
# weirdvars.sh: Вывод на экран необычных переменных.

echo
```

```
var="'(]\\{}\$\""
echo $var         # '(]\{}$"
echo "$var"       # '(]\{}$"     Без разницы.

echo

IFS='\'
echo $var         # '(] {}$"     \ преобразуется в пробел. Почему?
echo "$var"       # '(]\{}$"

# Примеры выше предоставлены Stephane Chazelas.

echo

var2="\\\\\""
echo $var2        #   "
echo "$var2"      # \\"
echo
# Но ... var2="\\\\"" является правильным. Почему?
var3='\\\\'
echo "$var3"      # \\\\
# Хотя жесткие кавычки работают.


# ************************************************************ #
# В первом примере выше показано разрешаемое в кавычках.

echo "$(echo '"')"           # "
#    ^           ^


# Иногда это выходит хорошо.

var1="Two bits"
echo "\$var1 = "$var1""      # $var1 = Two bits
#    ^               ^

# Или, как пояснил Chris Hiestand ...

if [[ "$(du "$My_File1")" -gt "$(du "$My_File2")" ]]
#     ^     ^         ^ ^     ^     ^         ^ ^
then
  ...
fi
# ************************************************************ #
```

Одинарные кавычки (**' '**) работают аналогично двойным кавычкам, но не разрешают ссылаться на переменные, так как специальное значение **$** в них выключается. В одинарных кавычках, каждый специальный символ, кроме **'** , интерпретируется буквально. Одинарные кавычки ("полные кавычки") рассматриваются, как более строгий вариант кавычек, чем двойные кавычки ("частичные кавычки").

Поскольку даже управляющий символ (****) получает буквальное толкование в одинарных кавычках, то попытка заключить одинарную кавычку внутри одинарных кавычек не даст ожидаемого результата.

```
echo "Why can't I write 's between single quotes"

echo

# Окольный метод.
echo 'Why can'\''t I write '"'"'s between single quotes'
#    |-------|  |----------|   |----------------------|
#  Три одиночных строки в кавычках, с экранированием и кавычками
#+ между одинарными кавычками.

# Этот пример предоставлен Stéphane Chazelas.
```

Примечания

[1] Инкапсуляция '**!**' внутри двойных кавычек при использовании *в командной строке* выдаст ошибку. Он будет интерпретироваться как команда *истории*. Хотя внутри сценария эта проблема не возникает, поскольку механизм истории Bash там отключен.

Большее беспокойство вызывает явно *несовместимое* поведение **** внутри двойных кавычек и особенно после команды **echo -e**.

```
bash$ echo hello\!
hello!
bash$ echo "hello\!"
hello\!


bash$ echo \
>
bash$ echo "\"
>
bash$ echo \a
a
bash$ echo "\a"
\a


bash$ echo x\ty
xty
bash$ echo "x\ty"
x\ty

bash$ echo -e x\ty
xty
bash$ echo -e "x\ty"
x       y
```

Двойные кавычки, после ***echo,*** иногда экранируют ****. Кроме того, опция **-e** в echo, при повторении вызывает «**\t**», интерпретируемое как табуляция

(Спасибо, Wayne Pollock за указания, а Geoff Lee и Daniel Barclay за объяснения .)

[2] «Разделение слова» в данном контексте означает разделение символьной строки на отдельные и дискретные аргументы.

5.2. Экранирование

Экранирование — это способ заключения в кавычки одиночных символов. Предшествующий символ экранирования (\) сообщает оболочке, что интерпретировать данный символ нужно буквально.

В некоторых командах и утилитах, таких как ***echo*** и ***sed***, экранирование символов может иметь обратный эффект - может переключить на специальное значение для этого символа.

Специальные значения определенных экранированных символов

используемых **echo** и **sed**

\n значение символа новой строки

\r значение возвращения

\t **з**начение табуляции

\v значение вертикальной табуляции

\b начение забоя

\a значение оповещения (звуковой сигнал или вспышка)

\0xx перевод в восьмеричный эквивалент **ASCII** `0nn`, где `nn` строка из цифр

Заключенная в кавычки конструкция **$'...'** механизма расширения, которая использует экранированные восьмеричные или шестнадцатеричные значения переменных, назначает переменным символы ASCII , например, `quote=$'\ 042'`.

Пример 5-2. Экранированные символы

```
#!/bin/bash
```

```
# escaped.sh: экранированные символы

#############################################################
### Сначала покажем некоторые основы использования        ###
###          экранированных   символов.                   ###
#############################################################

Экранирование
# -----------------

echo ""

echo "Это выведется на экран
как две строки."
# Это выведется на экран
# как две строки.

echo "Это выведется на экран \
как одна строка."
# Это выведется на экран как одна строка.

echo; echo

echo "============="


echo "\v\v\v\v\v"      # Выводится буквально \v\v\v\v\v.
# С помощью опции -e в 'echo' выводятся экранированные символы.
echo "============="
echo "VERTICAL TABS"
echo -e "\v\v\v\v\v"   # Выведутся 4 вертикальные табуляции.
echo "=============="

echo "QUOTATION MARK"
echo -e "\042"        # Выведется " (кавычки, восьмеричный ASCII символ
                      #+42).
echo "=============="



# Конструкция $'\X' делает опцию -e не обязательной.

echo; echo "НОВАЯ_СТРОКА и (может быть) ЗВУКОВОЙ_СИГНАЛ"
echo $'\n'            # Новая строка.
echo $'\a'            # Звуковой сигнал.
                      # В зависимости от терминала, может быть вспышка.

# Мы видели расширение строки $'\nnn", и теперь ...

# ================================================================= #
# Конструкция расширения строки $'\nnn' была введена во 2 версии Bash.
# ================================================================= #

echo "Введите конструкцию \$\' ... \' ... "
echo "... с большим количеством кавычек."

echo $'\t \042 \t'   # Кавычки (") окружают табуляцию.
# Обратите внимание, что  '\nnn' это восьмеричное значение.

# Это так же сработает и с шестнадцатеричными конструкциями $'\xhhh'.
echo $'\t \x22 \t'  # Кавычки (") окружают табуляцию.
```

```
# Спасибо Greg Keraunen, за разъяснение.
# Ранние версии Bash допускали '\x022'.

echo


# Назначение переменной символов ASCII.
# --------------------------------------
quote=$'\042'        # " присваивается переменной.
echo "$quote  В кавычки заключена строка $quote, а это находится вне
кавычек."

echo

# Объединение символов ASCII в переменной.
triple_underline=$'\137\137\137'  # 137 это восьмеричный код ASCII для
                                  #+ '_'.
echo "$triple_underline ПОДЧЕРКНУТО $triple_underline"

echo

ABC=$'\101\102\103\010'           # 101, 102, 103 восьмеричные A, B, C.
echo $ABC

echo

escape=$'\033'                    # 033 восьмеричное escape.
echo "\"escape\" выводится на экран как $escape"
#                                 без видимого вывода.

echo

exit 0
```

Более сложный пример

Пример 5-3. Определение нажатой клавиши

```
#!/bin/bash
# Автор: Sigurd Solaas, 20 Apr 2011
# Используется в ABS Guide с разрешения.
# Используйте версию 4.2+ Bash.

key="еще не назначено"
while true; do
  clear
  echo "Bash Extra Keys Demo. Клавиши для определения:"
  echo
  echo "* Insert, Delete, Home, End, Page_Up and Page_Down"
  echo "* Четыре клавиши со стрелками"
  echo "* Клавиши Tab, enter, escape, и space"
  echo "* Клавиши букв и цифр, и т.д."
  echo
  echo "    d = показать дату/время"
  echo "    q = выход"
  echo "================================="
  echo
```

```
Преобразование отдельной клавиши home в home-key_num_7
if [ "$key" = $'\x1b\x4f\x48' ]; then
 key=$'\x1b\x5b\x31\x7e'
 #   Конструкция расширения строки в кавычках.
fi

# Преобразование отдельной клавиши end в end-key_num_1.
if [ "$key" = $'\x1b\x4f\x46' ]; then
 key=$'\x1b\x5b\x34\x7e'
fi

case "$key" in
 $'\x1b\x5b\x32\x7e')  # Клавиши Insert
  echo Клавиша Insert
 ;;
 $'\x1b\x5b\x33\x7e')  # Delete
  echo Клавиша Key
 ;;
 $'\x1b\x5b\x31\x7e')  # Home_key_num_7
  echo Клавиша Home
 ;;
 $'\x1b\x5b\x34\x7e')  # End_key_num_1
  echo Клавиша End
 ;;
 $'\x1b\x5b\x35\x7e')  # Page_Up
  echo Клавиша Page_Up
 ;;
 $'\x1b\x5b\x36\x7e')  # Page_Down
  echo Клавиша Page_Down
 ;;
 $'\x1b\x5b\x41')  # Up_arrow
  echo Up arrow
 ;;
 $'\x1b\x5b\x42')  # Down_arrow
  echo Клавиша Down arrow
 ;;
 $'\x1b\x5b\x43')  # Right_arrow
  echo Клавиша Right arrow
 ;;
 $'\x1b\x5b\x44')  # Left_arrow
  echo Клавиша Left arrow
 ;;
 $'\x09')  # Tab
  echo Tab Key
 ;;
 $'\x0a')  # Enter
  echo Клавиша Enter
 ;;
 $'\x1b')  # Escape
  echo Клавиша Escape
 ;;
 $'\x20')  # Space
  echo Клавиша Space
 ;;
 d)
  date
 ;;
 q)
 echo Время вышло...
 echo
```

```
  exit 0
  ;;
  *)
   echo Вы нажали: \'"$key"\'
  ;;
 esac

 echo
 echo "================================"

 unset K1 K2 K3
 read -s -N1 -p "Нажмите клавишу: "
 K1="$REPLY"
 read -s -N2 -t 0.001
 K2="$REPLY"
 read -s -N1 -t 0.001
 K3="$REPLY"
 key="$K1$K2$K3"

done

exit $?
```

См. Пример 37-1.

\" задает кавычкам буквальный их смысл, экранирует кавычки.

```
echo "Hello"                      # Hello
echo "\"Hello\" ... he said."     # "Hello" ... he said.
```

\$ задает символу доллара буквальный его смысл (следующее за \$ не будет являться ссылкой на имя переменной), экранирует символ доллара.

```
echo "\$variable01"           # $variable01
echo "The book cost \$7.98."  # The book cost $7.98.
```

**** задает обратному слэш буквальный его смысл, экранирование обратного слэш

```
echo "\\"  # В результате \

# В то время как ...

echo "\"   # Вызывает дополнительное приглашение командной строки.
           # В сценарии выдает сообщение об ошибке.

# Однако ...

echo '\'   # В результате \
```

Поведение `\` зависит от экранирования, сильных кавычек, слабых кавычек, или появления в подстановке команд или в ***here document***.

```
                      #  Простое экранирование и кавычки
echo \z               #  z
echo \\z              # \z
echo '\z'             # \z
echo '\\z'            # \\z
echo "\z"             # \z
echo "\\z"            # \z

                      #  Подстановка команд
echo `echo \z`        #  z
echo `echo \\z`       #  z
echo `echo \\\z`      # \z
echo `echo \\\\z`     # \z
echo `echo \\\\\\z`   # \z
echo `echo \\\\\\\z`  # \\z
echo `echo "\z"`      # \z
echo `echo "\\z"`     # \z

                      # Here document
cat <<EOF
\z
EOF                   # \z

cat <<EOF
\\z
EOF                   # \z

# Этот пример предоставлен Stéphane Chazelas.
```

Элементы строки присвоения переменной могут быть экранированы, но символ экранирования не может быть присвоен переменной.

```
variable=\
echo "$variable"
# Не будет работать — выдаст сообщение об ошибке:
# test.sh: : команда не найдена
# «Голый» символ экранирования не может быть безопасно назначен
# переменной.
#  Что происходит здесь на самом деле, это экранирование '\'  новой
#+ строки и эффект от этого        variable=echo "$variable"
#+                            не правильное присвоение переменной

variable=\
23skidoo
echo "$variable"        #  23skidoo
                        #  Это работает, поскольку вторая строка
                        #+ является допустимой для присваивания
                        #+ переменным.

variable=\
#        \^    Экранирование следующего за ним пробела
echo "$variable"        # пробел

variable=\\
```

```
echo "$variable"        # \

variable=\\\
echo "$variable"
# Не будет работать, выдаст сообщение об ошибке:
# test.sh: \: команда не найдена
#
#  Первый escape экранирует второй, а третий "пустоту",
#+ одинаковый результат как в первом примере, выше.

variable=\\\\
echo "$variable"        # \\
                        # Второй и четвертый escape заэкранированы.
                        # Это нормально.
```

Заэкранированный пробел предотвращает разделение слов в списке аргументов команды.

```
file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
# Список файлов, как аргумента(ов) команды.

# Добавляем два файла в список, и весь список.
ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list

echo
"------------------------------------------------------------------------"

# Что произойдет, если мы заэкранируем несколько пробелов?
ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
# Ошибка: первые три файла объединятся в единый аргумент 'ls -l'
# потому что два заэкранированных пространства предотвращают разделение

#+ аргумента (слова).
```

Экранирование также обеспечивает написание многострочной команды. Как правило, каждая отдельная строка представляет собой различные команды, но экранирование конца строки *экранирует символ перевода новой строки*, и последовательность команд продолжается на следующей строке.

```
(cd /source/directory && tar cf - . ) | \
(cd /dest/directory && tar xpvf -)
# Повторение команды копирования дерева директории Alan Cox,
# но разделенной на две строки для повышения удобочитаемости.

# Как альтернатива:
tar cf - -C /source/directory . |
tar xpvf - -C /dest/directory
# Смотрите примечание ниже.
# (Спасибо Stéphane Chazelas.)
```

Если строка сценария заканчивается **|**, знаком конвейера, то ****, экранирование, не является строго необходимым. Но это является хорошим стилем программирования - всегда экранировать окончание строки кода, который

продолжается на следующей строке.

```
echo "foo
bar"
#foo
#bar

echo

echo 'foo
bar'    # Нет никакого различия.
#foo
#bar

echo

echo foo\
bar     # Символ новой строки заэкранирован.
#foobar

echo

echo "foo\
bar"     #  Здесь \ по прежнему интерпретируется как экранирование в слабых
         #+ кавычках.
#foobar

echo

echo 'foo\
bar'     # Символ \ воспринимается буквально из-за строгих кавычек.
#foo\
#bar

# Примеры предоставлены Stéphane Chazelas.
```

Глава 6. Выход и статус выхода

... это «темный угол» Bourne shell, а все люди пользуются им

--Chet Ramey

Команда **exit**, как и в программе Си, завершает сценарий. Она возвращает значение предоставляемое родительским процессом сценария.

Каждая команда возвращает *статус выхода* (иногда упоминаемый, как *возвращаемый статус* или *код выхода*). Успешно завершившаяся команда возвращает `0`, в то время как при неудаче возвращается *ненулевое* значение, которое обычно интерпретируется, как *код ошибки*. Правильно завершаемые команды, программы и утилиты UNIX, после успешного завершения, возвращают код выхода `0`, хотя есть некоторые исключения.

Точно так же функционирует сценарий, и сам сценарий возвращает статус выхода. Статус выхода определяет последняя команда, выполняемая в функции или в сценарии. Внутри сценария, команда `exit nnn` может быть использована для доставки статуса выхода `nnn` в оболочку (`nnn` должно быть целым числом в диапазоне от `0 - 255`).

Когда сценарий завершается с **exit** без параметра, то статусом выхода сценария будет статус выхода последней команды, выполненной в сценарии (перед **exit**).

```
#!/bin/bash

КОМАНДА_1

...

ПОСЛЕДНЯЯ_КОМАНДА

# Будет выход со статусом последней команды.

exit
```

Эквивалентом чистого **exit** является **exit $?** или даже просто пропуск **exit**.

```
#!/bin/bash

КОМАНДА_1
```

```
...

ПОСЛЕДНЯЯ_КОМАНДА

# Будет выход со статусом последней команды.

exit $?

#!/bin/bash

КОМАНДА_1

...

ПОСЛЕДНЯЯ_КОМАНДА

# Будет выход со статусом последней команды.
```

$? считывает статус выхода последней команды. После выполнения функции, $? возвращает статус выхода последней команды, выполненной функцией. Это способ Bash предоставления функцией «возвращаемого значения». [1]

После выполнения ***конвейера***, $? возвращает статус выхода последней команды.

После завершения сценария, $?, в командной строке, возвращает статус выхода сценария, которым, по соглашению, является `0` при успешном завершении, или целым числом, в диапазоне `1-255`, в случае ошибки, то есть статус выхода, последней, выполненной в сценарии, команды,.

Пример 6-1. Выход/статус выхода

```
#!/bin/bash

echo hello
echo $?    #  Статус выхода 0, возвращаемый в случае успешного выполнения
           #+ команды.

lskdf      # Не распознанная команда.
echo $?    # Возвращаемый, не нулевой, статус выхода означает не успешное
           #+ выполнение команды.

echo

exit 113   # Возвращает в оболочку 113.
           # Для проверки, после завершения сценария, введите «echo $?».

#  По соглашению, выход 0 указывает на успешное завершение, в то время как
#+ выход ненулевого значения означает ошибку или аномальное состояние.
#  См. Приложение "Особые значения кодов выхода".
```

$? особенно полезна для проверки результата выполнения команды в сценарии (см. Пример

16-35 и Пример 16-20).

! - является логическим НЕ, отменяет результат проверки или команды, а это влияет на их статус выхода.

Пример 6-2. Отрицание условия с помощью !

```
true    # Встроенное "true" .
echo "статус выхода \"true\" = $?"     # 0

! true
echo "статус выхода \"! true\" = $?"   # 1
# Обратите внимание, что между "!" и командой должен быть пробел.
#    !true   приведет к сообщению об ошибке «команда не найдена»
#
# Префиксный оператор команды '!' вызывает механизм истории Bash.

true
!true
# В этот раз нет ошибки, но нет и отрицания чего-либо.
# Только повторение предыдущей команды (true).


# ============================================================ #
# Предшествующий ! _конвейеру_ инвертирует возвращаемый статус
выхода.
ls | bogus_command     # bash: bogus_command: команда не найдена
echo $?                # 127

! ls | bogus_command   # bash: bogus_command: команда не найдена
echo $?                # 0
# Обратите внимание, что ! не изменяет выполнение конвейера.
# Изменяет только статус выхода.
# ============================================================ #

# Спасибо Stéphane Chazelas и Kristopher Newsome.
```

Отдельные коды статуса выхода имеют *зарезервированные значения* и не должны указываться пользователем в сценарии.

Примечания

[1] В тех случаях, когда нет кода *возврата* прерывания функции.

Глава 7. Проверки

Содержание

Каждый, достаточно полный, язык программирования имеет возможность проверять условие, а затем выполнять действие в соответствии с результатом проверки. Bash имеет команду **`test`**, заключение *операторов* в различные, в т.ч. и круглые, скобки и конструкцию ***if/then***.

7.1. Конструкции проверки

- Конструкция ***if/then*** проверяет, является ли `0` (нуль) *статусом выхода* списка команд (`0`, по соглашению UNIX, означает "успех"), и если да, то выполняет одну или несколько команд.
- Так же существует специальная команда, вызываемая **`[`** (***левой квадратной*** скобкой). Это синоним **`test`**, ***встроенная*** по соображениям эффективности. Эта команда считает свои аргументы выражениями сравнения или файловой проверкой и возвращает статус выхода, соответствующий результату сравнения (**`0`** для ***true***, **`1`** для ***false***).
- С версии 2.02, Bash представляет *команду расширения проверки* **`[[...]]`**, которая выполняет сравнения, в манере более знакомой программистам из других языков. Обратите внимание, что **`[[`** является зарезервированным словом (***keyword***), а не командой.

  Bash понимает **`[[ $a -lt $b ]]`** как единый элемент возвращающий статус выхода.
- **`((...))`** и конструкция **`let…`** возвращают *статус выхода*, *оцениваемый расширением не нулевого значения*, *соответствующий арифметическому выражению*.

Таким образом, конструкции ***арифметических расширений*** могут использоваться для выполнения ***арифметических сравнений***.

```
(( 0 && 1 ))                  # Логическое И
echo $?     # 1     ***
# И так ...
let "num = (( 0 && 1 ))"
echo $num   # 0
# Но ...
let "num = (( 0 && 1 ))"
echo $?     # 1     ***


(( 200 || 11 ))               # Логическое ИЛИ
echo $?     # 0     ***
# ...
let "num = (( 200 || 11 ))"
echo $num   # 1
let "num = (( 200 || 11 ))"
echo $?     # 0     ***


(( 200 | 11 ))                # Побитовое ИЛИ
echo $?                       # 0     ***
# ...
let "num = (( 200 | 11 ))"
echo $num                     # 203
let "num = (( 200 | 11 ))"
echo $?                       # 0     ***

# Конструкция «let»  возвращает тот же статус выхода, что и двойные
#+ скобки арифметического расширения.
```

Опять же, обратите внимание, что *статус выхода* арифметического выражения — *не является значением ошибки*.

```
var=-2 && (( var+=2 ))
echo $?                   # 1

var=-2 && (( var+=2 )) && echo $var
                          # $var не будет выведено!
```

- ***if*** может проверять любые команды, а не только условия заключенные в скобки.

```
if cmp a b &> /dev/null  # Подавление вывода.
then echo "Файлы a и b одинаковые."
else echo "Файлы a и b отличаются."
fi

# Часто используемая конструкция "if-grep":
# ----------------------------------
if grep -q Bash file
  then echo "Файл содержит, по меньшей мере одно вхождение в Bash."
fi
```

```
word=Linux
letter_sequence=inu
if echo "$word" | grep -q "$letter_sequence"
# Опция grep "-q" подавляет вывод.
then
  echo "$letter_sequence находится в $word"
else
  echo "$letter_sequence не находится в $word"
fi


if КОММАНДА_ЧЬИМ_СТАТУСОМ_ВЫХОДА_ЯВЛЯЕТСЯ_0_ЕСЛИ_НЕ_ПРОИЗОШЛА_ОШИБКА
  then echo "Команда правильна."
  else echo "Команда не правильна."
fi
```

Эти два примера предоставлены *Stéphane Chazelas.*

Пример 7-1. Что такое истина?

```
#!/bin/bash

#  Совет:
#  Если вы не уверены, что можно оценить определенное условие, проверяйте
#+ его проверкой if.

echo

echo "Проверка \"0\""
if [ 0 ]      # ноль
then
  echo "0 это true."
else          # Или, это ...
  echo "0 это false."
fi            # 0 это true.

echo

echo "Проверка \"1\""
if [ 1 ]      # один
then
  echo "1 это true."
else
  echo "1 это false."
fi            # 1 это true.

echo

echo "Проверка \"-1\""
if [ -1 ]     # минус один
then
  echo "-1 это true."
else
  echo "-1 это false."
fi            # -1 это true.

echo
```

```
echo "Проверка \"NULL\""
if [ ]        # NULL (пустое условие)
then
  echo "NULL это true."
else
  echo "NULL это false."
fi            # NULL это false.

echo

echo "Проверка \"xyz\""
if [ xyz ]    # Строка
then
  echo "Случайная строка это true."
else
  echo "Случайная строка это false."
fi            # Случайная строка это true.

echo

echo "Проверка \"\$xyz\""
if [ $xyz ]   # Проверка, что $xyz это null, а...
              # не просто инициализированная переменная.
then
  echo "Не инициализированная переменная это true."
else
  echo " Не инициализированная переменная это false."
fi            # Не инициализированная переменная это false.

echo

echo "Проверка \"-n \$xyz\""
if [ -n "$xyz" ]            # Более педантично.
then
  echo "Не инициализированная переменная это true."
else
  echo " Не инициализированная переменная это false."
fi            # Не инициализированная переменная это false.

echo


xyz=          # Инициализирована, но присвоено значение null.

echo "Проверка \"-n \$xyz\""
if [ -n "$xyz" ]
then
  echo "Null переменная это true."
else
  echo " Null переменная это false."
fi            # Null переменная это false.


echo


# Когда "false" это true?

echo "Проверка \"false\""
if [ "false" ]              #  Принимает "false" как строку ...
then
```

```
  echo "\"false\" это true." #+ и эта проверка true.
else
  echo "\"false\" это false."
fi            # "false" это true.

echo

echo "Проверка \"\$false\""  # Снова, не инициализированная переменная.
if [ "$false" ]
then
  echo "\"\$false\" это true."
else
  echo "\"\$false\" это false."
fi            # "$false" это false.
              # Теперь получен нужный результат.

#  Что получится при проверке не инициализированной переменной "$true"?

echo

exit 0
```

Упражнение. Объяснение поведения в Примере 7-1, выше**.**

```
if [ условие-правдиво ]
then
   команда1
   команда2
   ...
else  # Или else ...
      # Добавляется выполнение блока кода по умолчанию, если проверка
      #+ условие-правдиво является ложным.
   команда3
   команда4
   ...
fi
```

Когда ***if*** и ***then*** находятся на одной строке с условием проверки, то точка с запятой прекращает действие оператора ***if***. Оба ***if*** и ***then*** это ключевые слова. Ключевое слово (или команда) уже начало действовать, а до начала действия нового оператора на той же строке, действие старого оператора необходимо прекратить.

```
if [ -x "$filename" ]; then
```

Else if* и *elif

elif

`elif` это сокращение для ***else if***. Эффект заключается во встраивании конструкции внутрь внешней конструкции ***if/then***.

```
if [ условие1 ]
then
   команда1
   команда2
   команда3
elif [ условие2 ]
# То же, что и else if
then
   команда4
   команда5
else
   команда_по_умолчанию
fi
```

Конструкция **`if test условие-правдиво`** это точный эквивалент **`if [ условие-правдиво ]`**. Как это происходит: левая скобка **`[`**, представляющая собой токен (ключ), [1], вызывает команду **`test`**. Закрывающая правая скобка,**`]`**, в операторе ***if/test***, не является строго необходимой, однако, требуется в более новых версиях Bash.

Команда **`test`** является ***builtin (встроенной)*** Bash, которой проверяются типы файлов и производится сравнение строк. Таким образом, в сценарии Bash, **`test`** не вызывает внешний бинарный файл `/usr/bin/test`, являющийся частью пакета ***sh-utils***. Так же, **`[`** не вызывает `/usr/bin/[`, который ссылается на `/usr/bin/test`.

```
bash$ type test
test это builtin оболочки
bash$ type '['
[ это builtin оболочки
bash$ type '[['
[[ это ключевое слово оболочки
bash$ type ']]'
]] это ключевое слово оболочки
bash$ type ']'
bash: type: ]: такого нет
```

Если, по какой причине, вы хотите использовать `/usr/bin/test` в сценарии Bash, то задавайте полный путь к файлу.

Пример 7-2. Эквивалентность `test`, `/usr/bin/test`, `[ ]` и `/usr/bin/[`

```
#!/bin/bash

echo

if test -z "$1"
then
  echo "Нет аргументов командной строки."
```

```
else
  echo "Первый аргумент командной строки это $1."
fi

echo

if /usr/bin/test -z "$1"      # Эквивалентно встроенному "test".
#  ^^^^^^^^^^^^               # Но указывается полный путь.
then
  echo "Нет аргументов командной строки."
else
  echo "Первый аргумент командной строки это $1."
fi

echo

if [ -z "$1" ]                # Функционально идентично блоку кода выше.
#   if [ -z "$1"    тоже будет работать, но... Bash отреагирует на
#+  недостающую закрывающую скобку выводом сообщения об ошибке.
then
  echo "Нет аргументов командной строки."
else
  echo "Первый аргумент командной строки это $1."
fi

echo

if /usr/bin/[ -z "$1" ]       # Снова, функционально идентично коду выше.
# if /usr/bin/[ -z "$1"       # Работает, но выдает сообщение об ошибке.
#                               Примечание:
#                               Это исправлено в Bash версии 3.x.
then
  echo "Нет аргументов командной строки."
else
  echo "Первый аргумент командной строки это $1."
fi

echo

exit 0
```

Конструкция **[[ ]]** в Bash более универсальна, чем **[ ]**. Это команда расширения проверки, взятая из *ksh88*.

* * *

Между **[[** и **]]** не происходит расширения имени файла или разделения слова, но возможна подстановка параметров и команд.

```
file=/etc/passwd

if [[ -e $file ]]
then
  echo "Файл пароля существует."
fi
```

Использование конструкции проверки **[[...]]**, в отличии от **[...]**, предотвращает множество логических ошибок в сценариях. Например, операторы **&&**, **||**, **<** и **>** работают в **[[ ]]**, но выдают ошибку внутри конструкции **[ ]** .

Арифметическая оценка восьмеричной/шестнадцатеричной константы в конструкции **[[...]]** происходит автоматически.

```
# [[ Оценка восьмеричных и шестнадцатеричных чисел ]]
# Спасибо Moritz Gronbach, за это указание.


decimal=15
octal=017   # = 15 (десятичное)
hex=0x0f    # = 15 (десятичное)

if [ "$decimal" -eq "$octal" ]
then
  echo "$decimal равно $octal"
else
  echo "$decimal не равно $octal"       # 15 не равно 017
fi      # Не оценивается в [ одинарных скобках ]!


if [[ "$decimal" -eq "$octal" ]]
then
  echo "$decimal равно $octal"                # 15 равно 017
else
  echo "$decimal не равно $octal"
fi      # Оценивается в [[ двойных скобках ]]!

if [[ "$decimal" -eq "$hex" ]]
then
  echo "$decimal равно $hex"                  # 15 равно 0x0f
else
  echo "$decimal не равно $hex"
fi      # [[ $hexadecimal ]] так же оценивается!
```

Для *команды*, следующей за **if**, квадратные скобки проверки (**[ ]** или **[[ ]]**) не являются строго необходимыми.

```
dir=/home/bozo

if cd "$dir" 2>/dev/null; then   #  "2>/dev/null" Скрывает сообщение
                                 #+ об ошибке.
  echo "Теперь находитесь в $dir."
else
  echo "Не перешли в $dir."
fi
```

Конструкция "**if** *КОМАНДА*" возвращает статус выхода КОМАНДА.

Аналогично, условие в квадратных скобках проверки может находиться

самостоятельно, без **if**, когда используется в комбинации с конструкцией **`list` (список)**.

```
var1=20
var2=22
[ "$var1" -ne "$var2" ] && echo "$var1 не равно $var2"

home=/home/bozo
[ -d "$home" ] || echo "Директория $home отсутствует."
```

Конструкция **`(( ))`** расширяет и вычисляет арифметические выражения. Если выражение равно нулю, возвращается статус выхода `1`, или «**false**». Не нулевое выражение возвращает статус выхода `0`, или «**true**». Это контрастирует с использованием ранее обсужденной конструкции **`test`** и **`[ ]`**.

Пример 7-3. Арифметическая проверка с помощью (( ))

```
#!/bin/bash
# arith-tests.sh
# Арифметические проверки.

# Конструкция (( ... )) оценивает и проверяет числовые выражения.
# Статус выхода противоположен конструкции [ ... ]!

(( 0 ))
echo "Статус выхода \"(( 0 ))\" это $?."         # 1

(( 1 ))
echo "Статус выхода \"(( 1 ))\" это $?."         # 0

(( 5 > 4 ))                                      # истинно
echo "Статус выхода \"(( 5 > 4 ))\" это $?."     # 0

(( 5 > 9 ))                                      # ложно
echo "Статус выхода \"(( 5 > 9 ))\" это $?."     # 1

(( 5 == 5 ))                                     # истинно
echo "Статус выхода \"(( 5 == 5 ))\" это $?."    # 0
# (( 5 = 5 ))  выдаст сообщение об ошибке.

(( 5 - 5 ))                                      # 0
echo "Статус выхода \"(( 5 - 5 ))\" это $?."     # 1

(( 5 / 4 ))                                      # Деление, отлично
echo "Статус выхода \"(( 5 / 4 ))\" это $?."     # 0

(( 1 / 2 ))                                      # Результат деления < 1.
echo "Статус выхода \"(( 1 / 2 ))\" это $?."     # Округляется до 0.
                                                 # 1

(( 1 / 0 )) 2>/dev/null                          # Неправильное деление на 0.
#           ^^^^^^^^^^^
echo "Статус выхода \"(( 1 / 0 ))\" это $?."     # 1

# Какой эффект дает "2>/dev/null"?
# Что произойдет, если мы его удалим?
# Попробуйте удалить его, затем перезапустите сценарий.
```

```
# ===================================== #

# Конструкция (( ... )) так же полезна в проверке if-then.

var1=5
var2=4

if (( var1 > var2 ))
then #^      ^      Обратите внимание: Не $var1, $var2. Почему?
  echo "$var1 больше, чем $var2"
fi     # 5 больше, чем 4

exit 0
```

Примечания

[1] *Токен* — это символ или короткая строка с прилагаемым к нему специальным значением (***мета смыслом***). В Bash, определенные токены, такие как **[** и **.** (команда точка), могут расширять ключевые слова и команды.

7.2. Операторы проверки файлов

Возвращается *true* если...

-e файл ***существует***

-a файл ***существует***

Эффект идентичен **-e**. «Устаревшее», [1] и использование не рекомендуется.

-f файл является ***регулярным*** файлом (не директорией или файлом устройства)

-s файл имеет ***не нулевой размер***

-d файл является ***директорией***

-b файл является ***блочным устройством***

-c файл является ***символьным устройством***

```
device0="/dev/sda2"    # / (корневая директория)
if [ -b "$device0" ]
then
  echo "$device0 это блочное устройство."
```

```
fi

# /dev/sda2 это блочное устройство.



device1="/dev/ttyS1"   # карта модема PCMCIA.
if [ -c "$device1" ]
then
  echo "$device1 это символьное устройство."
fi

# /dev/ttyS1 это символьное устройство.
```

-p файл является ***PIPE***

```
function show_input_type()
{
   [ -p /dev/fd/0 ] && echo PIPE || echo STDIN
}

show_input_type "Input"                          # STDIN
echo "Input" | show_input_type                   # PIPE

# Пример предоставлен Carl Anderson.
```

-h файл является ***символической ссылкой***

-L файл является ***символической ссылкой***

-S файл является ***сокетом***

-t файл (***дескриптор***) связан с ***терминальным устройством***

Опция используемая для проверки в данном сценарии, является ли терминалом `stdin` **`[-t 0]`** или `stdout` **`[-t 1]`**.

-r файл имеет права на чтение (*для пользователя, запустившего проверку*)

-w файл имеет права на запись (*для пользователя, запустившего проверку*)

-x файл имеет права на выполнение (*для пользователя, запустившего проверку*)

-g файл или директорию с установленным флагом **`sgid`**

Если директория имеет флаг `sgid`, то файл, созданный в этой директории, принадлежит группе, которой принадлежит директория, и не обязательно группе пользователя, создавшего файл. Это может быть полезно в общей директории рабочей группы.

-u файл с установленным флагом (**suid**)

Бинарник, с установленным флагом `suid`, является собственностью ***root*** и работает с привилегиями ***root***, даже если его запускает обычный пользователь. [2] Это полезно для исполняемых файлов (таких как ***pppd*** и ***cdrecord***), которые получают доступ к аппаратному обеспечение системы. Без флага `suid`, эти бинарные файлы не могут быть вызваны не *root* пользователем.

```
-rwsr-xr-t    1 root       178236 Oct  2  2000 /usr/sbin/pppd
```

Файл с установленным флагом *suid* в своих правах имеет **s**.

-k установлен *sticky bit*

Флаг сохранения текстового режима, известный как `sticky` бит, представляет собой особый тип прав файла. Если файл имеет этот установленный флаг, то файл, для быстрого доступа к нему, будет храниться в кэш-памяти. [3] Если установлен для директории, то он ограничивает право записи. Установка sticky бита добавляет **t** в листинге прав файла или директории. Он не дает владельцам файлов изменять или удалять отдельные файлы в данной директории.

```
drwxrwxrwt    7 root         1024 May 19 21:26 tmp/
```

Если пользователь не является владельцем директории для которой установлен sticky бит, но у него есть права на запись в этой директории, он может удалить в ней только те файлы, которыми он владеет. Это не позволяет пользователям случайно перезаписывать или удалять файлы друг друга в общедоступной директории, например в `/tmp`. (Владелец директории или *root* может, конечно, удалять или переименовать файлы.)

-O вы собственник файла

-G `id` группы файла такой же, как ваш

-N файл изменен с момента последнего чтения

f1 -nt f2 файл *f1* новее, чем *f2*

f1 -ot f2 файл *f1* старее, чем *f2*

f1 -ef f2 файлы *f1* и *f2* это жесткие ссылки другого файла

! "нет" -- меняет смысл указанных выше проверок (возвращает `true`, если условие отсутствует).

Пример 7-4. Проверка на мертвые ссылки

```
#!/bin/bash
# broken-link.sh
# Написан Lee bigelow <ligelowbee@yahoo.com>
# Используется в ABS Guide с разрешения.

#  Чистый сценарий оболочки для поиска битых символических ссылок и вывода их
#+ в кавычках, чтобы они могли быть переданы xargs и уничтожены :)
#+ например, sh broken-link.sh /somedir /someotherdir|xargs rm
#
#  Этот способ будет получше:
#
#  find "somedir" -type l -print0|\
#  xargs -r0 file|\
#  grep "broken symbolic"|
#  sed -e 's/^\|: *broken symbolic.*$/"/g'
#
#+ но... не в чистом Bash.
#  Внимание: будьте осторожны с файловой системой  /proc и любыми
#+ ссылками на нее!
########################################################################

#  Если в сценарий не переданы аргументы устанавливающие
#+ директорию_для_поиска, то поиск осуществляется в текущей директории.
#+ В противном случае директорией_для_поиска является соответствующая
#+ переданным аргументам.
########################

[ $# -eq 0 ] && directorys=`pwd` || directorys=$@


#  Настройка функции linkchk для проверки директории на содержание в ней
#+ файлов,  являющимися не существующими ссылками, затем вывод их в кавычках.
#  Если один из элементов в директории является поддиректорией, то отправляем
#+ эту поддиректорию в функцию проверки ссылок.
##########

linkchk () {
    for element in $1/*; do
      [ -h "$element" -a ! -e "$element" ] && echo \"$element\"
      [ -d "$element" ] && linkchk $element
    # Ну да, '-h' проверка символических ссылок, '-d' директории.
    done
}

#  Каждый аргумент, переданный сценарию, передается в функцию linkchk(),
#+ если действительно является директорией. Если нет, то
#+ выводится сообщение об ошибке.
###################
for directory in $directorys; do
    if [ -d $directory ]
        then linkchk $directory
        else
            echo "$directory не является директорией"
            echo "Usage: $0 dir1 dir2 ..."
    fi
done

exit $?
```

Пример 31-1, Пример 11-8, Пример 11-3, Пример 31-3 и пример A-1 также показывают

примеры операторов проверки файла.

Примечания

[1] Издание 1913 года *Словаря Webster*:

```
Deprecate
...

To pray against, as an evil;
to seek to avert by prayer;
to desire the removal of;
to seek deliverance from;
to express deep regret for;
to disapprove of strongly.
```

[2] Имейте в виду, что бинарные файлы с *`suid`* могут открыть дыры в безопасности. Флаг *`suid`* не оказывает влияния на сценарий оболочки.

[3] На системах Linux `sticky bit` для файлов больше не используется, но только для директорий

7.3. Другие операторы сравнения

Бинарный оператор сравнения сравнивает две *переменные* или *величины*. Обратите внимание, что для сравнения ***целых чисел*** и ***строк*** используют ***разные наборы операторов***.

Сравнения ЦЕЛЫХ ЧИСЕЛ

`-eq` равно

```
if [ "$a" -eq "$b" ]
```

`-ne` не равно

```
if [ "$a" -ne "$b" ]
```

`-gt` больше, чем...

```
if [ "$a" -gt "$b" ]
```

`-ge` больше, чем... или равно

```
if [ "$a" -ge "$b" ]
```

`-lt` меньше, чем...

```
if [ "$a" -lt "$b" ]
```

-le меньше, чем... или равно

```
if [ "$a" -le "$b" ]
```

< меньше, чем... (в двойных круглых скобках)

```
(("$a" < "$b"))
```

<= меньше, чем... или равно (в двойных круглых скобках)

```
(("$a" <= "$b"))
```

> больше, чем... (в двойных круглых скобках)

```
(("$a" > "$b"))
```

>= больше, чем.. или равно (в двойных круглых скобках)

```
(("$a" >= "$b"))
```

сравнения СТРОК

= равно

```
if [ "$a" = "$b" ]
```

Обратите внимание на ***пробелы*** вокруг **=**.

if ["$a"="$b"] это не то же самое, что выше.

== равно

```
if [ "$a" == "$b" ]
```

Это синоним **=**.

Оператор сравнения **==** ведет себя различно в двойных и в одинарных скобках проверки.

```
[[ $a == z* ]]   #  Истинно, если $a начинается с "z" (шаблон
                 #+ соответствия).
[[ $a == "z*" ]] #  Истинно, если $a равно z* (буквальное
                 #+ соответствие).

[ $a == z* ]     #  Подстановка файла и разделение слова пропусками.
[ "$a" == "z*" ] #  Истинно если $a равно z* (буквальное
                 #+ соответствие).
```

```
# Спасибо Stéphane Chazelas
```

!= не равно

```
if [ "$a" != "$b" ]
```

Этот оператор используется шаблоном соответствия в конструкции **[[...]]**.

< меньше, чем..., в алфавитном порядке ASCII

```
if [[ "$a" < "$b" ]]

if [ "$a" \< "$b" ]
```

Обратите внимание, что в конструкции **[ ]** нужно "**<**" заэкранировать.

> больше, чем... в алфавитном порядке ASCII

```
if [[ "$a" > "$b" ]]

if [ "$a" \> "$b" ]
```

Обратите внимание, что в конструкции **[ ]** нужно "**>**" заэкранировать .

См. Пример 27-11 о применении этого оператора сравнения.

-z строка является ***null***, т.е. имеет нулевую длину

```
 String=''   # Строка переменной нулевой длины ("null").

if [ -z "$String" ]
then
  echo "\$String является null."
else
  echo "\$String НЕ является null."
fi     # $String является null.
```

-n строка **не** ***null***.

Проверка **-n** требует, что бы строка в скобках проверки была ***заключена в кавычки***. Использование строки без кавычек с ***!*** ***-z***, или даже просто, строки без кавычек, только в скобках проверки (см. Пример 7-6) обычно работает, однако, это плохая практика. *Всегда* заключайте проверяемые строки в кавычки. [1]

Пример 7-5. Числовые и строковые сравнения

```
#!/bin/bash

a=4
b=5

#  Здесь «a» и «b» могут рассматриваться либо как целые числа, либо как строки.
#  Есть некоторые размытия между числовыми и строковыми сравнениями,
#+ поскольку переменные Bash не являются строго типизированными.

#  Bash допускает целочисленные операции и сравнение переменных, значение
#+ которых полностью состоит из символов целых чисел.
#  Поосторожней, однако.

echo

if [ "$a" -ne "$b" ]
then
  echo "$a не равно $b"
  echo "(числовое сравнение)"
fi

echo

if [ "$a" != "$b" ]
then
  echo "$a не равно $b."
  echo "(сравнение строк)"
  #     "4"  != "5"
  # ASCII 52 != ASCII 53
fi

# В данном конкретном случае, оба "-ne" и "!=" работают.

echo

exit 0
```

Пример 7-6. Проверка, имеет ли строка значение *null*

```
#!/bin/bash
#  Проверка строк null и строк, не заключенных в кавычки.

# С помощью   if [ ... ]

# Если строка не инициализирована, значит она не имеет заданного значения.
# Это состояние называется «null» (не то же самое, что ноль!).

if [ -n $string1 ]    # string1 не объявлена или инициализирована.
then
  echo "Строка \"string1\" не является null."
else
  echo "Строка \"string1\" является null."
fi                    # Не правильный результат.
#  Показывает $string1, как не являющуюся null, не смотря на то, что
```

```
#+ она не инициализирована.

echo

# Попробуем снова.

if [ -n "$string1" ]  # В этот раз, $string1 заключена в кавычки.
then
  echo "Строка \"string1\" не является null."
else
  echo "Строка \"string1\" является null."
fi                    # В скобках проверки строка заключена в кавычки!

echo

if [ $string1 ]       # В этот раз, $string1 остается голой.
then
  echo " Строка \"string1\" не является null."
else
  echo " Строка \"string1\" является null."
fi                    # Прекрасно работает.
# Оператор проверки [...] только определяет, имеет ли строка значение null.
# Однако это хорошая практика, заключать в кавычки (if [ "$string1" ]).
#
# Как поясняет Stephane,
#    if [ $string1 ]    имеет один аргумент, "]"
#    if [ "$string1" ]  имеет два аргумента, пустую "$string1" и "]"


echo


string1=initialized

if [ $string1 ]       # Снова, $string1 без кавычек.
then
  echo "Строка \"string1\" не является null."
else
  echo "Строка \"string1\" является null."
fi                    # Опять, правильный результат.
# Все же, лучше заключать в кавычки ("$string1"), потому что ...


string1="a = b"

if [ $string1 ]       # Опять, $string1 без кавычек.
then
  echo " Строка \"string1\" не является null."
else
  echo " Строка \"string1\" является null."
fi                    #  Без кавычек, "$string1" дает не правильный
                      #+ результат!

exit 0   # Спасибо так же Florian Wisser, за предупреждения.
```

Пример 7-7. *zmore*

```
#!/bin/bash
```

```
# zmore

# просмотр сжатых файлов фильтром 'more'.

E_NOARGS=85
E_NOTFOUND=86
E_NOTGZIP=87

if [ $# -eq 0 ] # такой же эффект, как и у:  if [ -z "$1" ]
# $1 может существовать, но быть пустым:  zmore "" arg2 arg3
then
  echo "Usage: `basename $0` filename" >&2
  # Вывод сообщения об ошибке в stderr.
  exit $E_NOARGS
  # Возвращает 85, как статус выхода сценария (код ошибки).
fi

filename=$1

if [ ! -f "$filename" ]   # В кавычках, $filename,  может содержать пробелы.
then
  echo "Файл $filename не найден!" >&2   # Сообщение об ошибке в stderr.
  exit $E_NOTFOUND
fi

if [ ${filename##*.} != "gz" ]
# Используются скобки для подстановки переменной.
then
  echo "Файл $1 не является сжатым!"
  exit $E_NOTGZIP
fi

zcat $1 | more

# Используется фильтр 'more'.
# При желании можно заменить на «less»,.

exit $?   # Сценарий возвращает статус выхода конвейера.
#  На самом деле "exit $?"  является ненужным, в любом случае сценарий
#+ возвратит статус выхода последней выполняемой команды.
```

Составные сравнения

-a логическое И

exp1 **-a** *exp2* возвращает истину, если *оба* `exp1` *и* `exp2` истинны.

-o логическое ИЛИ

exp1 **-o** *exp2* возвращает истину, если одно `exp1` *или* `exp2` истинно

Они похожи на операторы сравнения Bash **&&** и **||**, используемые в ***двойных скобках***.

```
[[ условие1 && условие2 ]]
```

Операторы **-o** и **-a** применяются с командой **test** или в одинарных скобках проверки.

```
if [ "$expr1" -a "$expr2" ]
then
  echo "Оба expr1 и expr2 истинны."
else
  echo "Одно, expr1 или expr2, ложно."
fi
```

Но, как указывает *rihad*:

```
[ 1 -eq 1 ] && [ -n "`echo true 1>&2`" ]   # истинно
[ 1 -eq 2 ] && [ -n "`echo true 1>&2`" ]   # (нет вывода)
# ^^^^^^^ Ложное условие. Пока все как ожидается.

# Однако ...
[ 1 -eq 2 -a -n "`echo true 1>&2`" ]       # истинно
# ^^^^^^^ Ложное условие. Так почему вывод "истинно"?

# Это потому, что оба условия находятся в общих скобках?
[[ 1 -eq 2 && -n "`echo true 1>&2`" ]]     # (нет вывода)
# Нет не так.

#  Видимо && и || "замыкаются накоротко", в то время, как -a и -o

#+ нет.
```

Обратитесь к Примеру 8-3, Примеру 27-17, и Примеру A-29, для того, что бы увидеть составные операторы сравнения в действии.

Примечания

[1] Как указывает S.C., в составной проверке не может быть достаточным даже заключение строки переменной в кавычки. **[-n "$string" -o "$a" = "$b"]** может привести к ошибке в некоторых версиях Bash, если `$string` пустая. Безопасный способ состоит в добавлении дополнительного символа, в, возможно, пустую переменную, **["x$string" != x -o "x$a" = "x$b"]** ("x" отменяют).

7.4. Вложенные условия проверок *if/then*

Проверка условий с помощью конструкции **if/then** может быть вложенной. Чистый результат эквивалентен использованию составного оператора сравнения.

```
a=3

if [ "$a" -gt 0 ]
then
  if [ "$a" -lt 5 ]
  then
```

```
    echo "Значение \"a\" находится где-то между 0 и 5."
  fi
fi

# Тот же результат, как:

if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]
then
  echo "Значение \"a\"находится где-то между 0 и 5."
fi
```

Пример 37-4 и Пример 17-11 демонстрируют вложенные проверки условий **`if/then`**.

7.5. Проверим ваши знания о проверках

Общесистемный файл `xinitrc` может использоваться для запуска X-сервера. Этот файл содержит ряд проверок ***`if/then`***. Далее выдержка из «древней» версии ***xinitrc*** (Red Hat 7.1, или где-то так).

```
if [ -f $HOME/.Xclients ]; then
  exec $HOME/.Xclients
elif [ -f /etc/X11/xinit/Xclients ]; then
  exec /etc/X11/xinit/Xclients
else
     # безопасные настройки, хотя они не нужны здесь,
     # (в Xclients есть откаты), не повредят.
     xclock -geometry 100x100-5+5 &
     xterm -geometry 80x50-50+150 &
     if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.html ]; then
            netscape /usr/share/doc/HTML/index.html &
     fi
fi
```

Объясните конструкции проверок в приведенном фрагменте, затем изучите обновленную версию файла `/etc/X11/xinit/xinitrc`, и проанализируйте конструкции проверки ***`if/then`***. Вам может понадобиться заглянуть вперед на обсуждение ***grep***, ***sed*** и ***регулярных выражений***.

Глава 8. Операции и смежные темы

Содержание

8.1. Операторы

Присваивание

Присваивают значения переменной Инициализация или изменение значения переменной

= Универсальный оператор присваивания, который работает с числовыми и строковыми назначениями.

```
var=27
category=minerals  # Без пробелов спереди и сзади "=".
```

Не путайте оператор присваивания "**=**" с оператором проверки **=**.

```
#   =  как оператор проверки

if [ "$строка1" = "$строка2" ]
then
   команда
fi

#  if [ "X$string1" = "X$string2" ] безопаснее, предотвращает
#+ сообщение об ошибке, если одна из переменных пустая.
#  (Присоединяются символы отмены «X».)
```

числовые операторы

+ плюс

- минус

***** умножение

/ деление

****** возведение в степень

```
# Bash, версия 2.02, представление оператора возведения в степень "**".
```

```
let "z=5**3"    # 5 * 5 * 5
echo "z = $z"   # z = 125
```

% модуль, или *mod* (возвращает *остаток* от операции деления целых чисел)

```
bash$ expr 5 % 3
2
```

5/3 = 1, с остатком 2

Среди прочего, этот оператор находит использование для создания чисел в пределах определенного диапазона (см. Пример 9-11 и Пример 9-15) и форматирования вывода программы (см. Пример 27-16 и Пример A-6). Он даже может использоваться для создания простых чисел, (см. Пример A-15). Модуль удивительно часто появляется в числовых выражениях.

Пример 8-1. Наибольший общий делитель

```
#!/bin/bash
# gcd.sh: Наибольший общий делитель
#         Используется алгоритм Евклида

#  "Наибольший общий делитель" (нод) двух целых чисел
#+ это наибольшее целое число, на которое делятся оба числа, не оставляя
#+ никакого остатка.

#  Алгоритм Евклида использует последовательное деление.
#    В каждом проходе,
#+      делимое <--- делитель
#+      делитель <--- остаток
#+   до остатка = 0.
#    нод = делимому, в конечном проходе.
# Прекрасное обсуждение алгоритма Евклида смотри:
#+ Jim Loy's site, http://www.jimloy.com/number/euclids.htm.

# ------------------------------------------------------
# Проверка аргументов
ARGS=2
E_BADARGS=85

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` первое-число второе-число"
  exit $E_BADARGS
fi
# ------------------------------------------------------

gcd ()                    # Функция нод (прим. переводчика)
{

  dividend=$1             #  Делимое. Назначаем произвольно.
  divisor=$2              #! Делитель. Неважно, какое число из двух
                          #+ больше. Почему не важно?

  remainder=1             #  Остаток. Если внутри скобок проверки
                          #+ используется не инициализированная
```

```
                          #+ переменная, то это приведет к сообщению
                          #+ об ошибке.

  until [ "$remainder" -eq 0 ]
  do    #  ^^^^^^^^^^  Нужно сначала инициализировать!
    let "remainder = $dividend % $divisor"
    dividend=$divisor     # Теперь повторяем с 2 меньшими числами.
    divisor=$remainder
  done                    # Алгоритм Евклида

}                         # Последнее $dividend это нод (gcd).


gcd $1 $2

echo; echo "НОД чисел $1 и $2 = $dividend"; echo


# Упражнения:
# ---------
# 1) Проверьте аргументы командной строки, чтобы убедиться, что они
#+   являются целыми числами и завершите сценарий с соответствующим
#+   сообщением об ошибке, если не числа не целые.
# 2) Перепишите функцию gcd() используя локальные переменные.

exit 0
```

+= *плюс-равно* (увеличивает переменную на константу) [1]

`let "var += 5"` в результате *var* увеличится на 5.

-= *минус-равно* (уменьшает переменную на константу)

***=** *раз-равно* (умножает переменную на константу)

`let "var *= 4"` в результате *var* умножится на 4.

/= *слеш-равно* (делит переменную на константу)

%= *mod-равно* (остаток от деления переменной на константу)

Числовые операторы часто находятся в выражениях ***expr*** *или* ***let****.*

Пример 8-2. Использование арифметических операций

```
#!/bin/bash
# Считаем до 11 10-ю различными способами.

n=1; echo -n "$n "

let "n = $n + 1"   # let "n = n + 1"  то же работает.
echo -n "$n "
```

```
: $((n = $n + 1))
#  ":" необходимо, потому что в противном случае Bash пытается
#+ интерпретировать "$((n = $n + 1))" как команду.
echo -n "$n "

(( n = n + 1 ))
#  Более простая альтернатива способу выше.
#  Спасибо за это указание David Lombard.
echo -n "$n "

n=$(($n + 1))
echo -n "$n "

: $[ n = $n + 1 ]
#  ":"  необходимо, потому что в противном случае Bash пытается
#+ интерпретировать "$[ n = $n + 1 ]" как команду.
#  Работает, даже если "n" была объявлена как строка.
echo -n "$n "

n=$[ $n + 1 ]
#  Работает, даже если "n" была объявлена как строка.
#* Избегайте конструкции этого типа, т.к. она устарела и не переносима.
#  Спасибо, Stephane Chazelas.
echo -n "$n "

# Теперь операторы увеличения с Си-стиле.
# Спасибо Frank Wang, за это указание.

let "n++"           # let "++n"  то же работает.
echo -n "$n "

(( n++ ))           # (( ++n ))  то же работает.
echo -n "$n "

: $(( n++ ))        # : $(( ++n )) то же работает.
echo -n "$n "

: $[ n++ ]          # : $[ ++n ]  то же работает.
echo -n "$n "

echo

exit 0
```

Целые переменные, в старых версиях Bash, обозначались длинными (***long***) (32-битными) числами, в диапазоне от -2147483648 до +2147483647. Операция, которая принимала переменную вне этого диапазона, выдавала ошибку.

```
echo $BASH_VERSION   # 1.14

a=2147483646
echo "a = $a"        # a = 2147483646
let "a+=1"           # Увеличивается "a".
echo "a = $a"        # a = 2147483647
let "a+=1"           # Снова увеличивается "a", после предела.
echo "a = $a"        # a = -2147483648
                     #      ОШИБКА: вне диапазона,
                     #+     и устанавливается самый левый бит, бит
```

```
                        #+     знака, делая результат
                        #+     отрицательным.
```

Версии Bash >= 2.05b поддерживают 64-битные целые числа.

Bash не понимает вычислений с плавающей точкой. Это относится и к числам, в виде строки, содержащим десятичную точку.

```
a=1.5

let "b = $a + 1.3"  # Ошибка.
# t2.sh: let: b = 1.5 + 1.3: синтаксическая ошибка выражения
#                            (токен ошибки ".5 + 1.3")

echo "b = $b"       # b=1
```

С помощью **bc**, сценарий может производить расчеты с плавающей точкой или функциями математической библиотеки.

Битовые операторы. Битовые операторы редко применяются в сценариях оболочки. Их основное использование, это управление и проверка значений, считанных из портов или сокетов. "Bit flipping" является более актуальным для компилируемых языков, таких как Си и C++, предоставляя прямой доступ к аппаратным средствам системы. Тем не менее, см. гениальное использование *vladz* битовых операторов в его сценарии base64.sh (Пример A-54).

битовые операторы.

<< побитовый сдвиг влево (умножение на 2 для каждой смены позиции)

<<= *сдвиг-влево-равно*

`let "var <<= 2"` результат `var` смещается влево на 2 бита (умножается на 4)

>> побитовый сдвиг вправо (деление на 2 для каждой смены позиции)

>>= *сдвиг-вправо-равно* (инверсия **<<=**)

& битовое И

&= битовое И-*равно*

| битовое ИЛИ

|= битовое ИЛИ-*равно*

~ битовое НЕ

^ битовое исключающее ИЛИ (XOR)

^= битовое ИЛИ-*равно*

логические (булевы) операторы

! НЕ

```
if [ ! -f $FILENAME ]
then
  ...
```

&& И

```
if [ $условие1 ] && [ $условие2 ]
#  То же:  if [ $условие1 -a $условие2 ]
#  Возвращает истинно, если оба, условие1 и условие2, истинны...

if [[ $условие1 && $условие2 ]]    # То же работает.
#  Обратите внимание, что оператор && не допускается внутрь скобок
#+ конструкции [ ... ].
```

Также может использоваться, в зависимости от контекста, в ***and list*** для объединения команд.

|| ИЛИ

```
if [ $условие1 ] || [ $условие2 ]
# То же:  if [ $условие1 -o $условие2 ]
# Возвращает истинно, если одно из условие1 или условие2 истинно...

if [[ $условие1 || $условие2 ]]    # То же работает.
#  Обратите внимание, что оператор || не допускается внутрь скобок
#+ конструкции [ ... ], но применяется внутри [[ … ]] (прим.          #+
переводчика).
```

Bash проверяет *статус выхода* каждого оператора, связанного с логическим оператором.

Пример 8-3. Соединение проверки условий с помощью && и ||

```
#!/bin/bash
```

```
a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
  echo "Проверка #1 успешна."
else
  echo "Проверка #1 провалена."
fi

# ОШИБКА:  if [ "$a" -eq 24 && "$b" -eq 47 ]
#+         при попытке выполнить ' [ "$a" -eq 24 '
#+         не удается найти соответствие ']'.
#
# ВНИМАНИЕ:  if [[ $a -eq 24 && $b -eq 24 ]]  работает.
#  Двойные скобки проверки if являются более
#+ гибкими, чем одинарные скобки.
#    ("&&" имеет отличное значение в строке 17 от строки 6.)
#    Спасибо Stephane Chazelas, за указание на это.


if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
  echo "Проверка #2 успешна."
else
  echo "Проверка #2 провалена."
fi


#  Опции -a и -o предоставляют альтернативное
#+ соединение проверки  условий.
#  За это указание спасибо Patrick Callahan.


if [ "$a" -eq 24 -a "$b" -eq 47 ]
then
  echo "Проверка #3 успешна."
else
  echo "Проверка #3 провалена."
fi


if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
  echo "Проверка #4 успешна."
else
  echo "Проверка #4 провалена."
fi


a=rhino
b=crocodile
if [ "$a" = rhino ] && [ "$b" = crocodile ]
then
  echo "Проверка #5 успешна."
else
  echo "Проверка #5 провалена."
fi

exit 0
```

Операторы **&&** и **||** также находят применение в арифметическом контексте.

```
bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
1 0 1 0
```

прочие операторы

, Оператор запятая

Оператор запятая связывает вместе две или более арифметические операции. Все операции обрабатываются (возможно с *побочными эффектами*). [2]

```
let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
echo "t1 = $t1"           ^^^^^^  # t1 = 11
# t1 присваивается значение результата последней операции. Почему?

let "t2 = ((a = 9, 15 / 3))"      # Присваивание "a" и вычисление "t2".
echo "t2 = $t2    a = $a"         # t2 = 5    a = 9
```

Оператор запятая находит применение, главным образом, в циклах. См. Пример 11-13.

Примечания

[1] В другом контексте, **+=** может служить в качестве оператора *объединения строк*. Это может быть полезно при *изменении переменных среды*.

[2] *Побочные эффекты* это, конечно же, непреднамеренные -- и, обычно, нежелательные -- последствия.

8.2. Числовые константы

Сценарий оболочки понимает число как десятичное (по основанию 10), если это число не имеет специального префикса или нотации. `0`, предшествующий числу, указывает на ***восьмеричное*** число (по основанию 8). `0x`, предшествующий числу, указывает на ***шестнадцатеричное*** число (основание 16). Номер со встроенным **#** оценивается как *BASE#NUMBER* (диапазон и нотационные ограничения).

Пример 8-4. Представление числовых констант

```
#!/bin/bash
# numbers.sh: Представление чисел по различным основаниям.

# Десятичное: по умолчанию
let "dec = 32"
echo "десятичное число = $dec"             # 32
# Здесь нет ничего не обычного.

# Восьмеричное: перед числом стоит '0' (ноль)
let "oct = 032"
echo "восьмеричное число = $oct"           # 26
# Выводит результат в десятичном виде.
# --------- ------ -- -------

# Шестнадцатеричное: перед числом стоит '0x' или '0X'
let "hex = 0x32"
echo "шестнадцатеричное число = $hex"      # 50

echo $((0x9abc))                           # 39612
#     ^^      ^^   двойные скобки арифметического расширения/оценки
# Выводит результат в десятичном виде.

# Другие основания: BASE#NUMBER
# BASE (ОСНОВАНИЕ) между 2 и 64.
# NUMBER (ЧИСЛО) должен использоваться символ из диапазона BASE, см. ниже.


let "bin = 2#111100111001101"
echo "двоичное число = $bin"               # 31181

let "b32 = 32#77"
echo "число по основанию 32 = $b32"        # 231

let "b64 = 64#@_"
echo "число по основанию 64 = $b64"        # 4031
#  Эта нотация работает только для ограниченного количества символов ASCII
#+ (2-64).
#  10 цифр + 26 символов в нижнем регистре + 26 символов в верхнем регистре
#+ + @ + _

echo

echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
                                           # 1295 170 44822 3375
#  Важное примечание:
#  --------------
#  Использование чисел вне диапазона, указанного значения основания, выдает
#+ сообщение об ошибке.

let "bad_oct = 081"
# (Частичный) вывод сообщения об ошибке:
#  bad_oct = 081: значение слишком велико для указанного основания
#+ (error token is "081")
#  Восьмеричные числа используют цифры в диапазоне 0 - 7.

exit $?   # Выходное значение = 1 (ошибка)

# Спасибо Rich Bartell и Stephane Chazelas за уточнение.
```

8.3. Конструкция двойных круглых скобок

Подобно команде **let**, конструкция **((...))** допускает арифметические расширения и оценку. В простой форме, **a=$((5 + 3))**, **a** присваивается **5 + 3**, или **8**. Хотя, конструкция двойных скобок, это так же механизм Си-стиля, позволяющий управлять переменными в Bash, например **((var++))**.

Пример 8-5. Управление переменными в стиле Си

```
#!/bin/bash
# c-vars.sh
# Управление переменными в стиле Си с помощью конструкции (( ... )).

echo

(( a = 23 ))  #  Присваивание значения, в стиле Си,
              #+ пробелы с обеих сторон "=".
echo "a (инициализированное значение) = $a"   # 23

(( a++ ))     #  Пост-инкремент (увеличение после) 'a', стиль Си.
echo "a (после a++) = $a"       # 24

(( a-- ))     #  Пост-декремент (уменьшение после) 'a', стиль Си.
echo "a (после a--) = $a"       # 23


(( ++a ))     #  Пре-инкремент (увеличение перед) 'a', стиль Си.
echo "a (перед ++a) = $a"       # 24

(( --a ))     #  Пре-декремент (уменьшение перед) 'a', стиль Си.
echo "a (перед --a) = $a"       # 23

echo

########################################################
#  Обратите внимание, что, как и в Си, пре- и пост-декрементные операторы
#+ имеют различные эффекты.

n=1; let --n && echo "True" || echo "False"  # Ложно
n=1; let n-- && echo "True" || echo "False"  # Истинно

#  Спасибо Jeroen Domburg.
########################################################

echo

(( t = a<45?7:11 ))   # Тройной оператор стиля Си.
#       ^  ^ ^
echo "Если a < 45, тогда t = 7, если нет, то t = 11."  # a = 23
echo "t = $t "                                 # t = 7

echo

# -----------------
# Предупреждение Пасхальное яйцо!
# -----------------
```

```
#  Chet Ramey, кажется, разобрался в куче недокументированных конструкций
#+ Си-стиля в Bash (на самом деле, взятых в большом количестве из ksh).
#  В документации Bash, Ramey называет ((...)) арифметической оболочкой,
#+ а это далеко выходит за рамки обсуждаемого.
#  Извините, Chet, за раскрытие Вашего секрета.

# См. Использование конструкции (( ... )) в циклах "for" и "while".

# Работает с версией 2.04 Bash или более поздними.

exit
```

См. Пример 11-13 и Пример 8-4.

8.4. Приоритет операторов

В сценарии, операции выполняются в порядке *приоритета*: *вначале* выполняются операции с *высоким* приоритетом, а *потом* операции с более *низким* приоритетом. [1]

Таблица 8-1. Приоритет операторов

Оператор	Значение	Пояснение
		ВЫСОКИЙ ПРИОРИТЕТ
var++ var--	Пост-инкремент, пост-декремент	Операторы стиля Си
++var --var	Пре-инкремент, пре-декремент	
! ~	отрицание	логическое/побитовое, меняет на противоположный смысл следующего оператора
******	возведение в степень	арифметическая операция
*** / %**	умножение, деление, модуль	арифметическая операция
+ -	сложение, вычитание	арифметическая операция
<< >>	сдвиг влево, вправо	побитовый
-z -n	*одиночное* сравнение	строка это/это-не null
-e -f -t -x, etc.	*одиночное* сравнение	проверка файла
< -lt > -gt <= -le >= -ge	*составное* сравнение	строка и целое число
-nt -ot -ef	*составное* сравнение	проверка файла
== -eq != -ne	равенство/неравенство	Операторы проверки, строка и целое число

Оператор	Значение	Пояснение
&	И	битовое
^	XOR	*исключающее* ИЛИ, битовое
|	ИЛИ	битовое
&& -a	И	логическое, составное сравнение
|| -o	ИЛИ	логическое, составное сравнение
?:	тройной оператор	Стиль Си
=	присваивание	(не путать с проверкой равенства)
***= /= %= += -= <<= >>= &=**	комбинационное присваивание	умножить-равно, разделить-равно, модуль-равно, и т.д..
,	запятая	links a sequence of operations
		НИЗКИЙ ПРИОРИТЕТ

На практике, все что вам действительно нужно запомнить, заключается в следующем:

- Мантра "**M**y **D**ear **A**unt **S**ally" (**m**ultiply (умножение), **d**ivide (деление), **a**dd (сложение), **s**ubtract (вычитание)) для семейства арифметических операций.
- *Составные* логические операторы **&&** , **||**, **-a** и **-o** имеют ***низкий*** приоритет.
- Приоритет порядка вычисления операторов *равенства* обычно *слева направо*.

Теперь давайте попытаемся использовать наши знания приоритета операторов для анализа пары строк из файла `/etc/init.d/functions`, находящегося в дистрибутиве *Fedora Core* Linux.

```
while [ -n "$remaining" -a "$retry" -gt 0 ]; do

#  На первый взгляд это выглядит довольно сложно.


# Разобьем условия:
while [ -n "$remaining" -a "$retry" -gt 0 ]; do
#       --условие 1---- ^^ -условие 2----

#  Если переменная "$remaining" не нулевой длины
#+      И (-a)
#+ переменная "$retry" не нулевой длины
#+ то
#+ [ выражение-условий-внутри-скобок ] возвращает успех (0)
#+ и цикл while выполняется повторно.
#  ============================================================
#  Оценка "условия 1" и "условия 2" происходит ***до*** их сравнения И.
#+ Почему? Потому что И (-a) имеет более низкий приоритет
#+ чем операторы -n и -gt,
#+ и поэтому  оценивает *после* них.

#####################################################################
```

```
if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then


# Опять, разделяем условия:
if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then
#    --условие 1---------- ^^ --условие 2-------

#  Если файл "/etc/sysconfig/i18n" существует
#+      И (-a)
#+ переменная $NOLOCALE не нулевой длины
#+ то
#+ [ выражение-проверки-условий-внутри-скобок ] возвращает успех (0)
#+ и выполняются следующие команды.
#
#  Как и раньше, И (-a) оценивает *после*,
#+ потому что имеет  приоритет ниже операторов проверки
#+ в скобках.
#  ============================================================
#  Обратите внимание:
#  ${NOLOCALE:-} это расширяемый параметр, что кажется излишним.
#  Но, если $NOLOCALE не будет объявлено, ей будет присвоено значение *null*.
#  Это дает различие в некоторых контекстах.
```

Во избежание путаницы или ошибки в сложной последовательности операторов проверки, помещайте последовательности в двойные квадратные скобки.

```
if [ "$v1" -gt "$v2"  -o  "$v1" -lt "$v2"  -a  -e "$filename" ]
# Не понятно, что здесь происходит ...

if [[ "$v1" -gt "$v2" ]] || [[ "$v1" -lt "$v2" ]] && [[ -e
"$filename" ]]
# Гораздо лучше -- проверки условий группируются в логические
#+разделы.
```

Примечания

[1] *Приоритет (Precedence)*, в этом контексте, имеет примерно тот же смысл, что и *приоритет (priority)*.

Часть 3. За пределами основ

Содержание

Глава 9. Другой взгляд на переменные

Содержание

Используемые должным образом, переменные могут добавить сценарию мощности и гибкости. А это требует изучения их тонкостей и нюансов.

9.1. Внутренние переменные

*Встроенные (**Builtin**) переменные:*

переменные, влияющие на поведение сценария

$BASH Путь к бинарному файлу Bash

```
bash$ echo $BASH
/bin/bash
```

$BASH_ENV Переменная окружения (***environmental***), указывающая Bash для чтения загружаемый файл, на него будет ссылаться сценарий.

$BASH_SUBSHELL Переменная, указывающее уровень *subshell (подоболочки)*. Это новое дополнение Bash в версии 3. См. Пример 21-1.

$BASHPID ***ID*** *процесса* текущего экземпляра Bash. Это не то же самое, что переменная **$$,** но, зачастую, выдает тот же результат.

```
bash4$ echo $$
11015


bash4$ echo $BASHPID
11015


bash4$ ps ax | grep bash4
11015 pts/2    R      0:00 bash4
```

Но ...

```
#!/bin/bash4

echo "\$\$ вне subshell = $$"                                    # 9602
echo "\$BASH_SUBSHELL  вне subshell = $BASH_SUBSHELL"            # 0
echo "\$BASHPID вне subshell = $BASHPID"                         # 9602

echo

( echo "\$\$ в subshell = $$"                                    # 9602
  echo "\$BASH_SUBSHELL в subshell = $BASH_SUBSHELL"             # 1
  echo "\$BASHPID в subshell = $BASHPID" )                       # 9603
  # Обратите внимание, что $$ возвращает PID родительского процесса.
```

$BASH_VERSINFO[n] 6-элементный ***массив***, содержащий сведения об установленной версии Bash. Это похоже на $BASH_VERSION, ниже, но более подробно.

```
# Информация о версии Bash:

for n in 0 1 2 3 4 5
do
  echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
done

# BASH_VERSINFO[0] = 3                      # Старшей версии нет.
# BASH_VERSINFO[1] = 00                     # Младшей версии нет.
# BASH_VERSINFO[2] = 14                     # Уровень исправлений.
# BASH_VERSINFO[3] = 1                      # Встроенная версия.
# BASH_VERSINFO[4] = release                # Статус выпуска.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Архитектура
                                            # (что и $MACHTYPE).
```

$BASH_VERSION Версия Bash, установленная в системе

```
bash$ echo $BASH_VERSION
3.2.25(1)-выпуск
```

```
tcsh% echo $BASH_VERSION
BASH_VERSION: Не определена переменная.
```

Проверка **$BASH_VERSION** является хорошим способом определения запущенного командного интерпретатора. $SHELL не всегда дает правильный ответ.

$CDPATH Разделенный двоеточиями список путей поиска, доступных для команды **cd**, аналогично действию переменной $PATH для бинарных файлов. Переменная **$CDPATH** может находиться в локальном файле ~/.bashrc.

```
bash$ cd bash-doc
bash: cd: bash-doc: Нет такого файла или директории


bash$ CDPATH=/usr/share/doc
bash$ cd bash-doc
/usr/share/doc/bash-doc


bash$ echo $PWD
/usr/share/doc/bash-doc
```

$DIRSTACK Верхнее значение стека директории [1] (управляемого `pushd` и `popd`).

Эта встроенная переменная соответствует команде `dirs`, однако `dirs` показывает все содержимое стека директории.

$EDITOR Редактор, обычно вызываемый сценарием по умолчанию, ***vi*** или ***emacs***.

$EUID «эффективный» ID номер пользователя.

Идентификационный номер, независимый, идентифицирует текущего пользователя, иногда с помощью **su**.

$EUID не обязательно то же самое, что и `$UID`.

$FUNCNAME Имя текущей функции

```
xyz23 ()
{
  echo "Сейчас выполняется $FUNCNAME."  # сейчас выполняется xyz23.
}

xyz23

echo "FUNCNAME = $FUNCNAME"         # FUNCNAME =
                                    # Значение Null вне функции.
```

См. Пример A-50.

$GLOBIGNORE Список шаблонов подстановки файлов, которые будут исключены из совпадений в ***подстановке***.

$GROUPS Группы, к которым принадлежит текущий пользователь

Это список (массив) идентификаторов групп текущего пользователя, как записано в */etc/passwd* и */etc/group*.

```
root# echo $GROUPS
0


root# echo ${GROUPS[1]}
1
```

```
root# echo ${GROUPS[5]}
6
```

$HOME Домашняя директория пользователя, обычно */home/username* (см. Пример 10-7).

$HOSTNAME Команда *hostname* присваивает имя хосту системы *при загрузке* сценария init. Однако функция *gethostname()* устанавливает **$HOSTNAME** внутренней переменной Bash. См. Пример 10-7.

$HOSTTYPE Тип хоста

Подобно $MACHTYPE идентифицирует аппаратное обеспечение.

```
bash$ echo $HOSTTYPE
i686
```

$IFS внутренний разделитель полей (**i**nternal **f**ield **s**eparator)

Эта переменная позволяет Bash распознавать поля или границы слов, когда он интерпретирует символы строками.

По умолчанию, $IFS это ***пробелы*** (*пробелы*, *табуляции* и *переводы строки*), но они могут быть заменены, например, разделением данных в файле запятыми. Обратите внимание, что **$*** использует первый обрабатываемый символ в **$IFS**. См. Пример 5-1.

```
bash$ echo "$IFS"

($IFS по умолчанию отображает пустую строку.)


bash$ echo "$IFS" | cat -vte
 ^I$
 $
(Выводит пробел, здесь одно пустое место, ^I [горизонтальная табуляция]
  и новая строка, и отображается "$" - конец строки.)


bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
(Читает команды из строки и назначает любые аргументы позиционным
параметрам.)
```

Установка **$IFS** устраняет пробелы в пути имени.

```
IFS="$(printf '\n\t')"   # Из David Wheeler.
```

$IFS обрабатывает другие символы как пробелы.

Пример 9-1. $IFS и пробелы

```
#!/bin/bash
# ifs.sh


var1="a+b+c"
var2="d-e-f"
var3="g,h,i"

IFS=+
# Знак '+' будет интерпретироваться как разделитель.
echo $var1     # a b c
#                ^ ^
echo $var2     # d-e-f
echo $var3     # g,h,i

echo

IFS="-"
# Возвращаем знак «+» к интерпретации по умолчанию.
# Теперь как разделитель будет интерпретироваться знак '-'.
echo $var1     # a+b+c
echo $var2     # d e f
#                ^ ^
echo $var3     # g,h,i

echo

IFS=","
# Запятая ',' будет интерпретироваться как разделитель.
# Знак «-» возвращается к интерпретации по умолчанию.
echo $var1     # a+b+c
echo $var2     # d-e-f
echo $var3     # g h i
#                ^ ^

echo

IFS=" "
# Символ пустого места будет интерпретироваться как разделитель.
# Запятая «,» возвращается к интерпретации по умолчанию.
echo $var1     # a+b+c
echo $var2     # d-e-f
echo $var3     # g,h,i

# ============================================================ #

# Однако ...
# $IFS обрабатывает пробелы иначе, чем другие знаки.

вывод_каждого_аргумента_в_отдельной_строке()
{
```

```
  for arg
  do
    echo "[$arg]"
  done #  ^    ^   Находится внутри скобок, для лучшего восприятия.
}

echo; echo "IFS=\" \""
echo "-------"

IFS=" "
var=" a  b c   "
#    ^ ^^   ^^^
вывод_каждого_аргумента_в_отдельной_строке $var
# вывод_каждого_аргумента_в_отдельной_строке `echo " a  b c   "`
# [a]
# [b]
# [c]


echo; echo "IFS=:"
echo "-----"

IFS=:
var=":a::b:c:::"               # Шаблон, как выше,
#    ^ ^^   ^^^                #+ но с заменой" " на ":" ...
вывод_каждого_аргумента_в_отдельной_строке $var
# []
# [a]
# []
# [b]
# [c]
# []
# []

# Обратите внимание, «пустые» скобки.
# То же самое происходит с разделителем полей «FS» в awk.


echo

exit
```

(Большое спасибо, Stéphane Chazelas, для разъяснения и примеры выше.)

См. поучительные примеры использования **$IFS**: Пример 16-41, Пример 11-8 и Пример 19-14.

$IGNOREEOF Игнорирование **EOF**: Количество последних файлов (Ctrl+D) игнорирующихся оболочкой до выхода.

$LC_COLLATE Часто задается в файлах *.bashrc* или */etc/profile*, эта переменная определяет порядок сопоставления расширения имени файла и соответствие шаблону. При неправильном обращении, **LC_COLLATE** может привести к непредвиденным результатам в подстановке шаблонов имен файлов.

Начиная с версии 2.05, Bash, в подстановке шаблонов, больше не делается различий между строчными и прописными буквами в диапазоне символов указанных в скобках. Например, **ls [A-M]*** будет соответствовать обоим `File1.txt` и `file1.txt`. Чтобы вернуться к общепринятому поведению скобок соответствия, установите **LC_COLLATE** в `C` командой **export LC_COLLATE=C** в *`/etc/profile`* и/или *`~/.bashrc`*.

$LC_CTYPE Эта внутренняя переменная управляет интерпретацией символов в подстановке шаблона и соответствии шаблону.

$LINENO Эта переменная - номер строки сценария, в которой появляется данная переменная. Она имеет значение только в пределах сценария, в котором она появляется, и нужна, главным образом, для отладки.

```
# *** НАЧАЛО ОТЛАДОЧНОГО БЛОКА ***
last_cmd_arg=$_  # Сохраняем.

echo "В строке с номером $LINENO, переменная \"v1\" = $v1"
echo "Последний обработанный аргумент команды = $last_cmd_arg"
# *** КОНЕЦ ОТЛАДОЧНОГО БЛОКА ***
```

$MACHTYPE Тип машины

Определяет системное оборудование

```
bash$ echo $MACHTYPE
i686
```

$OLDPWD Старая рабочая директория («"**OLD**-**P**rint-**W**orking-**D**irectory», предыдущая директория, в которой вы находились).

$OSTYPE Тип операционной системы

```
bash$ echo $OSTYPE
linux
```

$PATH Путь к бинарным файлам, обычно `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, и т.д..

Когда дана команда, оболочка автоматически выполняет поиск в хэш таблице

директорий списка путей исполнения. Путь хранится в переменной среды, **$PATH**, списке директорий, разделенных двоеточиями. Как правило система хранит определенные **$PATH** в */etc/profile* или *~/.bashrc* (см. Приложение H).

```
bash$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

PATH=${PATH}:/opt/bin** добавляет директорию /opt/bin в текущий путь. В сценарии может быть целесообразно временно добавлять директорию в путь именно таким образом. Когда сценарий завершается, он восстанавливает оригинальные **$PATH (дочернего процесса, так как сценарий, может не изменять окружающую среду родительского процесса, оболочки).

Текущая "рабочая директория", **./**, обычно опускается из **$PATH**, как мера безопасности.

$PIPESTATUS Переменная массива, передающая статус(ы) выхода последнего выполненного конвейера в основном режиме.

```
bash$ echo $PIPESTATUS
0

bash$ ls -al | bogus_command
bash: bogus_command: команда не найдена
bash$ echo ${PIPESTATUS[1]}
127

bash$ ls -al | bogus_command
bash: bogus_command: команда не найдена
bash$ echo $?
127
```

Элементы массива **$PIPESTATUS** передают статус выхода каждой правильно выполненной команды в конвейере. **$PIPESTATUS[0]**передает статус выхода *первой* команды в конвейере, **$PIPESTATUS[1]** — статус выхода *второй* команды, и т.д.

Переменная **$PIPESTATUS** может содержать ошибочное значение 0 логина оболочки (в версиях Bash до 3.0).

```
tcsh% bash

bash$ who | grep nobody | sort
bash$ echo ${PIPESTATUS[*]}
0
```

```
```

Строки выше, содержащиеся в сценарии произведут ожидаемый вывод `0 1 0`.

Спасибо, Wayne Pollock за указание на это и предоставление приведенного выше примера.

Переменная **$PIPESTATUS** в некоторых контекстах дает неожиданные результаты.

```
bash$ echo $BASH_VERSION
3.00.14(1)-release

bash$ $ ls | bogus_command | wc
bash: bogus_command: команда не найдена
 0       0       0

bash$ echo ${PIPESTATUS[@]}
141 127 0
```

Chet Ramey приписывает вывод выше поведению **ls**. Если **ls** пишет в *конвеер*, выход которого не читается, то SIGPIPE убивает этот вывод, и ее статус выхода будет 141. В противном случае ее статус выхода равен 0, как ожидается. Это также относится к **tr**.

$PIPESTATUS это «летучая» переменная. Она должна быть захвачена сразу же после конвейера, до вмешательства любой другой команды.

```
bash$ $ ls | bogus_command | wc
bash: bogus_command: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
0 127 0

bash$ echo ${PIPESTATUS[@]}
0
```

Опция **pipefail** может оказаться полезной в тех случаях, когда **$PIPESTATUS** не дает нужную информацию.

$PPID **$PPID** процесса это **ID** (pid) родительского процесса [2]

Сравните с командой **pidof**.

$PROMPT_COMMAND Переменная содержащая команду, которая выполняется до строки

приглашения, на экран выводится **$PS1**.

$PS1 Это приглашение (***prompt***), видимое в командной строке.

$PS2 Второй элемент приглашения, выглядит, как приглашение к вводу. На экране "**>**".

$PS3 Третий элемент приглашения, выводимый в цикле **select** (см. Пример 11-30).

$PS4 Четвертый запрос, появляющийся в начале каждой строки вывода, при вызове сценария с опцией `-x` [*подробная трассировка*]. Он отображается как "**+**".

Как помощь в отладке, может быть полезна для включения диагностической информации в **$PS4**.

```
P4='$(чтение мусора time < /proc/$$/schedstat; echo "@@@ $time @@@ " )'
# Предложено Erik Brandsberg.
set -x
# Следуют другие команды ...
```

$PWD Рабочая директория (директория, в которой сейчас находитесь)

Это аналог встроенной команды **pwd**.

```
#!/bin/bash

E_WRONG_DIRECTORY=85

clear # Очистка экрана.

TargetDirectory=/home/bozo/projects/GreatAmericanNovel

cd $TargetDirectory
echo "Удаление неиспользуемых файлов в $TargetDirectory."

if [ "$PWD" != "$TargetDirectory" ]
then    # Берегитесь случайного удаления в не верно указанной директории.
  echo "Не верная директория!"
  echo "$PWD, а не $TargetDirectory!"
  echo "Выручило!"
  exit $E_WRONG_DIRECTORY
fi

rm -rf *
rm .[A-Za-z0-9]*    # Удаление файлов, начинающихся с точки (скрытых).
# rm -f .[^.]* ..?*   удаляет файлы, начинающиеся с нескольких точек.
# (shopt -s dotglob; rm -f *)  так же работает.
# Спасибо, S.C. за разъяснение.

#  Файловое имя (`basename`) может содержать все символы в диапазоне
#+ от 0 – 255, за исключением "/".
#  Удаление файлов, начинающихся со странных символов, таких как -,
#+ остается в качестве упражнения.
#+ (Подсказка: rm ./-странное_имя или rm -- -странное_имя)
result=$?   # Результат операций удаления. Если успешно = 0.
```

```
echo
ls -al              # Еще файлы остались?
echo "Сделано."
echo "Старые файлы в $TargetDirectory удалены."
echo

# Здесь, при необходимости, какие-то другие операции.

exit $result
```

$REPLY Значение по умолчанию, когда вместо переменной предоставляется **read**. Также применимо для меню **select**, но предоставляет только номер элемента переменной, а не само значение переменной.

```
#!/bin/bash
# reply.sh

# REPLY это значение по умолчанию для команды 'read'.

echo
echo -n "Какие ваши любимые овощи? "
read

echo "Вашими любимыми овощами являются $REPLY."
#  REPLY сохраняет значение последней "read", если, и только если,
#+ не предоставлена ни одна переменная.

echo
echo -n "Какие ваши любимые фрукты? "
read fruit
echo "Ваш любимый фрукт это $fruit."
echo "но..."
echo "Значение \$REPLY это все еще $REPLY."
#  $REPLY по-прежнему установлена в прежнем значении, поскольку
#+ переменная $fruit была передана как новое значение "read".

echo

exit 0
```

$SECONDS Время работы сценария в секундах.

```
#!/bin/bash

TIME_LIMIT=10
INTERVAL=1

echo
echo "Нажмите Control-C для выхода через $TIME_LIMIT секунд."
echo

while [ "$SECONDS" -le "$TIME_LIMIT" ]
```

```
do   #   $SECONDS это внутренняя переменная оболочки.
  if [ "$SECONDS" -eq 1 ]
  then
    units=second
  else
    units=seconds
  fi

  echo "Этот сценарий работает $SECONDS $units."
  #  На медленных или перегруженных машинах, сценарий может сбрасывать
  #+ отсчет по истечению указанного выше времени.
  sleep $INTERVAL
done

echo -e "\a"  # Звуковой сигнал!

exit 0
```

$SHELLOPTS Список ***включеных*** опций оболочки, переменная ***readonly*** *(только для чтения)*.

```
bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs
```

$SHLVL Уровень глубины вложения оболочки Bash. [3] Если в командной строке, **$SHLVL** это 1, то в сценарии он вырастает до 2.

Эта переменная *не подвластна* подоболочке. Используйте $BASH_SUBSHELL, когда необходимо указать вложенность подоболочки.

$TMOUT Если переменной окружения **$TMOUT** присваивается не нулевое значение `time`, то приглашение командной строки будет ожидать `$time` секунд. Дальше перейдет к выходу.

С версии 2.05b Bash, возможно использовать **$TMOUT** в сценариях в комбинации с **read**.

```
# Работает в сценариях Bash, версии 2.05b и поздних.

TMOUT=3    # Время приглашения истечет через 3 секунды.

echo "Какая Ваша любимая песня?"
echo "Быстрее, у Вас на ответ только $TMOUT секунд!"
read song

if [ -z "$song" ]
```

```
then
  song="(нет ответа)"
  # Ответ по умолчанию.
fi

echo "Ваша любимая песня это $song."
```

Есть и другие, более сложные, способы реализации ввода в сценарии. Одним из вариантов является создание временного цикла, чтобы сигнализировать сценарию истечение времени. Также нужна обработка сигналов временным циклом с процедурой генерации прерываний **`trap`** (см. Пример 32-5) (вот так!).

Пример 9-2. Ввод по времени

```
#!/bin/bash
# timed-input.sh

# TMOUT=3    Работает так же, как и в новейших версиях Bash.

TIMER_INTERRUPT=14
TIMELIMIT=3  # В данном случае три секунды.
             # Можно установить другое значение.

PrintAnswer()
{
  if [ "$answer" = TIMEOUT ]
  then
    echo $answer
  else       # В данном случае не хочется смешивать.
    echo "Ваше любимое вегетарианское это $answer"
    kill $!  #  Убивает больше не нужную функцию TimerOn,
             #+ запущенную в фоновом режиме.
             #  $! это PID последнего запущенного задания в фоновом режиме.
  fi

}


TimerOn()
{
  sleep $TIMELIMIT && kill -s 14 $$ &
  # Ждет 3 секунды, затем посылает сигнал сценарию.
}


Int14Vector()
{
  answer="TIMEOUT"
  PrintAnswer
  exit $TIMER_INTERRUPT
}

trap Int14Vector $TIMER_INTERRUPT
# Прерывает таймер (14), извращено для наших целей.

echo "Какой Ваш любимый овощ? "
```

```
TimerOn
read answer
PrintAnswer


#  Правда, это запутанная реализация ограничения времени перед вводом.
#  Тем не менее, опция  '-t' в «read» упрощает эту задачу.
#  См. сценарий "t-out.sh".
#  Однако, как насчет сроков ввода не только одного пользователя,
#+ но всего сценария?

#  Если нужно что-то действительно элегантное ...
#+ рассмотрите написание приложения на Си или C++,
#+ с помощью соответствующих библиотечных функций, таких как 'alarm'
#+ и 'setitimer.'

exit 0
```

Альтернативой может быть использование ***stty***.

Пример 9-3. Еще раз, ввод по времени

```
#!/bin/bash
# timeout.sh

#  Написан Stephane Chazelas,
#+ и изменен автором документа.

INTERVAL=5                # Время ожидания

timedout_read() {
  timeout=$1
  varname=$2
  old_tty_settings=`stty -g`
  stty -icanon min 0 time ${timeout}0
  eval read $varname      # или просто read $varname
  stty "$old_tty_settings"
  # См. Справочные страницы "stty."
}

echo; echo -n "Как Ваше имя? Быстрее! "
timedout_read $INTERVAL your_name

#  Это может не работать на каком-то типе терминала.
#  Макс. время ожидания зависит от терминала.
#+ (зачастую 25.5 секунд).

echo

if [ ! -z "$your_name" ]  # Если имя введено до окончания таймаута ...
then
  echo "Вас зовут $your_name."
else
  echo "Время вышло."
fi

echo
```

```
# Поведение этого сценария несколько отличается от "timed-input.sh."
# При любом нажатии клавиши, счетчик сбрасывается.

exit 0
```

Возможно, проще использовать опцию `-t` команды **read**.

Пример 9-4. Ограничение *read* по времени

```
#!/bin/bash
# t-out.sh [время ожидания]
# Вдохновлено предложением  "syngin seven"  (спасибо).


TIMELIMIT=4         # 4 секунды

read -t $TIMELIMIT variable <&1
#                           ^^^
#  В данном случае, "<&1" нужно в Bash 1.x и 2.x,
#  и необязательно в Bash 3+.

echo

if [ -z "$variable" ]  # Это null?
then
  echo "Время вышло, переменная все еще не присвоена."
else
  echo "variable = $variable"
fi

exit 0
```

$UID ID пользователя

Идентификационный номер текущего пользователя, как записан в `/etc/passwd`

Это реальный **id** текущего пользователя, даже если пользователь временно присвоил другой идентификатор с помощью **su**. `$UID` является переменной ***readonly*** (только для чтения), не может быть изменен в командной строке или сценарии и является аналогом встроенного **id**.

Пример 9-5. Я root?

```
#!/bin/bash
# am-i-root.sh:   Я root или нет?

ROOT_UID=0   # Root имеет $UID 0.

if [ "$UID" -eq "$ROOT_UID" ]  # Вы стали реально "root"?
then
```

```
  echo "Вы root."
else
  echo "Вы просто обыкновенный пользователь (но мама все равно вас любит)."
fi

exit 0


# ============================================================== #
# Код ниже не будет выполняться, потому что сценарий уже завершен.

# Это другой способ:

ROOTUSER_NAME=root

username=`id -nu`              # Или...   username=`whoami`
if [ "$username" = "$ROOTUSER_NAME" ]
then
  echo "Рути, тути, тути. Вы root."
else
  echo "Вы просто обычный парень."
fi
```

См. также Пример 2-3.

Переменные $ENV, $LOGNAME, $MAIL, $TERM, $USER и $USERNAME *не являются* ***встроенными*** Bash. Однако они часто устанавливаются как переменные среды в Bash или ***логинами*** загрузки файлов. $SHELL, имя пользовательского логина оболочки, может быть установлена из /etc/passwd или сценария "***init***", и не является встроенной Bash.

```
tcsh% echo $LOGNAME
bozo
tcsh% echo $SHELL
/bin/tcsh
tcsh% echo $TERM
rxvt

bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
rxvt
```

Позиционные параметры

$0, $1, $2, и т. д. ***Позиционные параметры*** передаваемые из командной строки сценарию, передаваемые функции или присваиваемые переменной (см. Пример 4-5 и Пример 15-16).

$# ***Количество аргументов*** командной строки [4] или позиционных параметров (см. Пример 36-2).

$* ***Все*** позиционные параметры, рассматриваемые как ***одно слово***

"$*" должна быть заключена в кавычки.

$@ То же, что и $*, но каждый параметр является строкой заключенной в кавычки, то есть, параметры передаваемые нетронутыми, без интерпретации или расширения. Кроме того, это означает, что *каждый параметр в списке аргументов рассматривается как* ***отдельное слово*** (буквально, по-буквенно).

Конечно, **"$@"** должна быть заключена в кавычки.

Пример 9-6. ***arglist*****: Листинг аргументов с $* и $@**

```
#!/bin/bash
# arglist.sh
# Сценарий вызывается несколькими аргументами, типа "one two three" ...

E_BADARGS=85

if [ ! -n "$1" ]
then
  echo "Usage: `basename $0` аргумент1 аргумент2 т.д."
  exit $E_BADARGS
fi

echo

index=1          # Инициализация начала отсчета.

echo "Список аргументов  \"\$*\":"
for arg in "$*"  # Не будет работать нормально, если "$*" не в кавычках.
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $* рассматривает все аргументы как одно слово.
echo "Список всех аргументов рассматривается как одно слово."

echo

index=1          # Сброс счетчика.
                 # Что случится, если забудем сбросить?

echo "Список аргументов \"\$@\":"
for arg in "$@"
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $@ видит каждый аргумент как отдельное слово.
echo "Список всех аргументов рассматривается как отдельные слова."

echo
```

```
index=1          # Сброс счетчика.

echo "Список аргументов с \$* (не в кавычках):"
for arg in $*
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             #  Не заключенная в кавычки $* рассматривает аргументы
                 #+ как отдельные слова.
echo "Список аргументов рассматривается как отдельные слова."

exit 0
```

После **shift**, **$@** содержит оставшиеся параметры командной строки, но без $1, который утрачивается.

```
#!/bin/bash
# Вызываем ./scriptname 1 2 3 4 5

echo "$@"    # 1 2 3 4 5
shift
echo "$@"    # 2 3 4 5
shift
echo "$@"    # 3 4 5

# Каждый "shift" удаляет параметр $1.
# "$@" содержит остальные параметры.
```

Специальный параметр **$@** находит применение в качестве инструмента при фильтрации входных данных сценариев. Конструкция **cat "$@"** , в качестве параметров сценария, принимает ввод в сценарий либо из **stdin**, либо из файлов данных. См. Пример 16-24 и Пример 16-25.

Параметры **$*** и **$@** иногда имеют противоречивое и загадочное поведение, в зависимости от настроек **$IFS**.

Пример 9-7. Не последовательное поведение $* и $@

```
#!/bin/bash

#  Неустойчивое поведение "$*" и "$@" внутренних переменных Bash, в
#+ зависимости от того, заключены они в кавычки или нет.
#  Демонстрирует несовместимую обработку  разделения слов и символов
#+ перевода строки.


set -- "First one" "second" "third:one" "" "Fifth: :one"
# Устанавливаем аргументы сценария, $1, $2, $3, и т.д.

echo

echo 'IFS не изменяется, используем "$*"'
c=0
for i in "$*"               #  В кавычках
```

```
do echo "$((c+=1)): [$i]"   #  Эта строка остается в каждом случае
                           #+ неизменной.
                           #  Выводятся аргументы.
done
echo ---

echo 'IFS не изменяется, используем $*'
c=0
for i in $*                # Без кавычек
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS не изменяется, используем "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS не изменяется, используем $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

IFS=:
echo 'IFS=":", используем "$*"'
c=0
for i in "$*"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", используем $*'
c=0
for i in $*
do echo "$((c+=1)): [$i]"
done
echo ---

var=$*
echo 'IFS=":", используем "$var" (var=$*)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", используем $var (var=$*)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo 'IFS=":", используем $var (var="$*")'
c=0
for i in $var
```

```
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", используем "$var" (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", используем "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", используем $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

var=$@
echo 'IFS=":", используем $var (var=$@)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", используем "$var" (var=$@)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

::var="$@"
echo 'IFS=":", используем "$var" (var="$@")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", используем $var (var="$@")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done

echo

# Попробуйте этот сценарий с ksh или zsh -y.

exit 0

#  Этот пример сценария написан Stephane Chazelas,
```

```
#+ и слегка изменен автором документа.
```

Параметры **$*** и **$@** различаются только когда заключены в двойные кавычки.

Пример 9-8. $* и $@ при пустом $IFS

```
#!/bin/bash

#  Если $IFS объявлена, но пуста,
#+ то "$*" и "$@" не выводят позиционные параметры, как ожидается.

mecho ()       # Вывод позиционных параметров.
{
echo "$1,$2,$3";
}


IFS=""         # Объявлена, но пуста.
set a b c      # Позиционные параметры.

mecho "$*"     # abc,,
#                 ^^
mecho $*       # a,b,c

mecho $@       # a,b,c
mecho "$@"     # a,b,c

#  Поведение $* и $@ при пустой $IFS  зависит от версий Bash или sh.
#  Поэтому не целесообразно полагаться на их «функциональность» в сценарии.


# Спасибо, Stephane Chazelas.

exit
```

Прочие специальные параметры

$- Флаги, передаваемые сценарию (с помощью **set**). См. Пример 15-16.

Эта конструкция *ksh*, изначально, адаптированная в Bash, и, к сожалению, она, похоже, не надежно работает в сценариях Bash. Одним из возможных использований ее является самопроверка сценария на интерактивность.

$! **PID** последнего задания (ID процесса) запущенного в ***фоновом*** режиме

```
LOG=$0.log

КОМАНДА1="sleep 100"

echo "Запись в журнал PID команд работающих в фоновом режиме сценария: $0" >>
"$LOG"
# Таким образом, они могут быть проверены, и, при необходимости,
#+ уничтожены.
echo >> "$LOG"

# Команды записи в журнал.

echo -n "PID \"$КОМАНДА1\":  " >> "$LOG"
${КОМАНДА1} &
echo $! >> "$LOG"
# PID значения "sleep 100":  1506

# Спасибо, Jacques Lederer, за разъяснение.
```

С помощью **$!** контролируется выполнение задания:

```
possibly_hanging_job & { sleep ${TIMEOUT}; eval 'kill -9 $!' &> /dev/null; }
# Принудительное завершение плохо ведущей себя программы.
# Полезно, например, в сценариях инициализации (init).

# Спасибо, Sylvain Fourmanoit, за творческое использование переменной "$!".
```

Или по другому:

```
# Пример Matthew Sage.
# Используется с разрешения.

TIMEOUT=30   # Значение времени ожидания в секундах
count=0

possibly_hanging_job & {
        while ((count < TIMEOUT )); do
                eval '[ ! -d "/proc/$!" ] && ((count = TIMEOUT))'
                # в /proc ищется информация о запущенных процессах.
                # "-d" проверка существования (существует ли директория).
                # Поэтому ожидается вывод проблем выполняемого задания.
                ((count++))
                sleep 1
        done
        eval '[ -d "/proc/$!" ] && kill -15 $!'
        # Если запущенное задание зависает, то оно убивается.
}

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

#  Тем не менее, это может не сработать как положено, если после «hanging_job'
#+ запустится другой процесс ...
#  В таком случае, неправильно выполняемое задание может быть убито.
#  Ariel Meragelman предлагает следующие исправления.
```

```
TIMEOUT=30
count=0
# Время ожидания в секундах
possibly_hanging_job & {

while ((count < TIMEOUT )); do
  eval '[ ! -d "/proc/$lastjob" ] && ((count = TIMEOUT))'
  lastjob=$!
  ((count++))
  sleep 1
done
eval '[ -d "/proc/$lastjob" ] && kill -15 $lastjob'

}

exit
```

$_ Специальная переменная, присваивающая последний аргумент предыдущей выполненной команды.

Пример 9-9. Подчеркнутая переменная

```
#!/bin/bash

echo $_              #  /bin/bash
                     #  Только вызов /bin/bash запускает сценарий.
                     #  Обратите внимание, что будет варьироваться в
                     #+ зависимости от того, как вызывается сценарий

du >/dev/null        #  Так как нет никакого вывода команды.
echo $_              #  du

ls -al >/dev/null    #  Так как нет никакого вывода команды.
echo $_              #  -al  (последний аргумент)

:
echo $_              #  :
```

$? Статус выхода команды, функции или самого сценария (см. Пример 24-7)

$$ **ID** процесса (PID) самого сценария. [5] Переменная **$$** часто находит применение в сценариях при создании «уникальных» имен временных файлов (см. Пример 32-6, Пример 16-31 и Пример 15-27). Она гораздо проще вызова **mktemp**.

Примечания

[1] Регистр стека представляет собой набор последовательной памяти, сохраняющей (***pushed***) и извлекающей (***popped***) значения в обратном порядке. Последнее, сохраненное

значение будет являться первым для извлечения. Иногда это называют ***LIFO*** (*последнее-выходит-первым*) или стеком ***pushdown***.

[2] PID выполняемого текущего сценария - конечно **$$**.

[3] Несколько аналогий рекурсии, в контексте вложенности, относится к шаблону, встроенному в больший шаблон. Одно из определений вложенности, согласно изданию 1913г. словаря Вебстера, это прекрасно иллюстрирует: *«коллекция коробок, ящичков, или тому подобного, оконечного размера, каждый из которых вложен в другой, побольше.»*

[4] Слова «аргумент» и «параметр» часто взаимозаменяемы. В контексте настоящего документа, они имеют один и тот же точный смысл: *переменная или функция, передаваемая в сценарий.*

[5] В сценарии, внутри *subshell*, **$$** ***возвращает PID сценария***, а не *subshell*.

9.2. Ввод переменных: `declare` или `typeset`

`declare` или ***`typeset`*** это встроенные команды, являющиеся точными синонимами, изменяющие свойства переменных. В некоторых языках программирования - это очень слабая форма *ввода* [1]. Команда ***`declare`*** специфицирована для Bash версии 2 или более поздних. Команда ***`typeset`*** работает еще и в сценариях ***ksh***.

Опции `declare`/`typeset`

`-r` ***readonly(только для чтения)***

(*`declare -r var1`* работает так же, как и ***`readonly`*** *`var1`*)

Это примерный эквивалент квалификатора типа ***const*** в Си. При попытке изменения значения переменной ***readonly,*** происходит сбой с выводом сообщения об ошибке.

```
declare -r var1=1
echo "var1 = $var1"   # var1 = 1

(( var1++ ))          # x.sh: строка 4: var1: переменная readonly
```

-i ***integer(целое число)***

```
declare -i number
# Сценарий будет обрабатывать последующие вхождения «number» как целого числа.

number=3
echo "Number = $number"     # Number = 3

number=three
echo "Number = $number"     # Number = 0
# Пытается вычислить строку «three» как целое число.
```

Возможны некоторые арифметические операции при объявления целочисленных переменных, когда нет необходимости в использовании ***expr*** или ***let***.

```
n=6/3
echo "n = $n"       # n = 6/3

declare -i n
n=6/3
echo "n = $n"       # n = 2
```

-a ***array(массив)***

```
declare -a indices
```

Переменная *indices* будет рассматриваться как ***массив***.

-f ***function(s)(функция(и))***

```
declare -f
```

Строка в сценарии **declare -f** без аргументов, вызывает список ***всех*** функций, ранее объявленных в этом сценарии.

```
declare -f function_name
```

declare -f function_name в сценарии перечисляет только указанные функции.

-x export (экспорт)

```
declare -x var3
```

Объявляет переменную, как доступную для экспорта за пределы среды самого сценария.

-x var=$value

```
declare -x var3=373
```

Команда ***declare*** позволяет присвоить значение переменной тем же оператором, которым устанавливались ее свойства.

Пример 9-10. Использование declare для вывода переменных

```
#!/bin/bash

func1 ()
{
  echo Это функция.
}

declare -f        # Перечисляет функции выше.

echo

declare -i var1   # var1 это целое число.
var1=2367
echo "var1 объявлено как $var1"
var1=var1+1       # Целочисленное объявление устраняет необходимость в «let».
echo "var1 увеличивает $var1 на 1."
# Попробуем изменить переменную, объявленную как целое число.
echo "Пробуем изменить var1 на значение с плавающей запятой, 2367.1."
var1=2367.1       # В результате сообщение об ошибке, переменная не изменена.
echo "var1 все еще $var1"

echo

declare -r var2=13.36            # 'declare' разрешает устанавливать свойства
                                 #+ переменной и одновременно
                                 #+ присваивать ей значение.
echo "var2 объявлена как $var2"  # Пробуем изменить переменную readonly.
var2=13.37                       # Сообщение об ошибке и выход из сценария.

echo "var2 все еще $var2"       # Эта строка не будет выполняться.

exit 0                           # Сценарий здесь не выходит.
```

Использование встроенного ***declare*** ограничивает область действия переменной.

```
foo ()
{
FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar   # Вывод на экран bar.
```

Однако . . .

```
foo (){
declare FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar  # Ничего не выводится на экран.


# Спасибо за это указание Michael Iatrou.
```

9.2.1. Другое использование для *declare*

Команда ***declare*** может быть полезна в определении переменных, ***окружения*** или т.п. Она может быть особенно полезна при работе с ***массивами***.

```
bash$ declare | grep HOME
HOME=/home/bozo


bash$ zzy=68
bash$ declare | grep zzy
zzy=68


bash$ Colors=([0]="purple" [1]="reddish-orange" [2]="light green")
bash$ echo ${Colors[@]}
purple reddish-orange light green
bash$ declare | grep Colors
Colors=([0]="purple" [1]="reddish-orange" [2]="light green")
```

Примечания

[1] В этом контексте *ввод* переменной означает ее классифицирование и ограничение ее свойств. Например, если переменная *объявлена* или *введена* как целое число, то ей больше не доступны операции ***со строками***.

```
declare -i intvar

intvar=23
echo "$intvar"   # 23
intvar=stringval
echo "$intvar"   # 0
```

9.3. $RANDOM: генерация случайных целых чисел

Грешен тот, кто пытается создавать случайные числа путем уменьшения.

--John von Neumann

`$RANDOM` это внутренняя ***функция*** Bash (не постоянная), которая возвращает *псевдослучайные* [1] целые числа в диапазоне от 0 до 32767. Она *НЕ* должна использоваться для создания ключей шифрования.

Пример 9-11. Генерация случайных чисел

```
#!/bin/bash

# $RANDOM возвращает различные случайные целые числа при каждом вызове.
# Номинальный диапазон: 0 - 32767 (16-битное целое число).

MAXCOUNT=10
count=1

echo
echo " Случайные числа $MAXCOUNT:"
echo "-----------------"
while [ "$count" -le $MAXCOUNT ]      #  Создание 10 ($MAXCOUNT) случайных
                                      #+ целых чисел.
do
  number=$RANDOM
  echo $number
  let "count += 1"  # Увеличиваем счетчик.
done
echo "-----------------"

#  Если вам нужны случайные целые числа в пределах определенного диапазона,
#+ используйте оператор «modulo».
# Он возвращает остаток от операции деления.

RANGE=500

echo

number=$RANDOM
let "number %= $RANGE"
#           ^^
echo "Случайное число меньше $RANGE  ---  $number"

echo
```

```
#  Если вам нужно случайное целое число больше, чем определенная нижняя
#+ граница, то настройте проверку отбрасывающую все числа ниже этого значения.

FLOOR=200

number=0   # Инициализация
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
done
echo "Случайное число больше, чем $FLOOR ---  $number"
echo

   # Рассмотрим простую альтернативу циклу выше, а именно:
   #       let "number = $RANDOM + $FLOOR"
   # Устраняющую цикл while и запускающуюся быстрее.
   # Но здесь могут возникнуть проблемы. Какие?


#  Объединим оба метода, выше, для получения случайного числа между двумя
#+ пределами.
Number=0   # Инициализация
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
  let "number %= $RANGE"  #  $number нижний предел $RANGE.
done
echo "Случайное число между $FLOOR и $RANGE ---  $number"
echo


# Генерирование двойного выбора, то есть значения «истинно» или «ложно».
BINARY=2
T=1
number=$RANDOM

let "number %= $BINARY"
#  Обратите внимание, что  let "number >>= 14"  дает более случайное
#+ распределение(сдвигает вправо все, кроме последней двоичной цифры).
if [ "$number" -eq $T ]
then
  echo "TRUE"
else
  echo "FALSE"
fi

echo


# Генерирование броска костей.
SPOTS=6   # Модуль 6 задает диапазон 0 - 5.
          # Увеличение на 1 дает желаемый диапазон 1 - 6.
          # Спасибо за упрощение, Paulo Marcel Coelho Aragao.
die1=0
die2=0
# Было бы лучше просто установить SPOTS=7, а не добавлять 1?
# Почему да и почему нет?

#  Бросок каждой кости производится отдельно, а поэтому дает
#+ правильные коэффициенты.
```

```
    let "die1 = $RANDOM % $SPOTS +1" # Бросаем первую кость.
    let "die2 = $RANDOM % $SPOTS +1" # Бросаем вторую кость.
    #  Какие арифметические операции выше, имеют больший приоритет --
    #+ модуль (%) или сложение (+)?


let "throw = $die1 + $die2"
echo "Бросок костей = $throw"
echo


exit 0
```

Пример 9-12. Выбор случайной карты из колоды

```
#!/bin/bash
# pick-card.sh

# Это пример выбора случайных элементов массива.


# Выбрать карту, любую карту.

Suites="Пики
Буби
Черви
Кресты"

Denominations="2
3
4
5
6
7
8
9
10
Валет
Дама
Король
Туз"

# Обратите внимание, что переменные занимают разные строки.


suite=($Suites)                # Инициализация переменной массива.
denomination=($Denominations)

num_suites=${#suite[*]}        # Подсчет количества элементов.
num_denominations=${#denomination[*]}

echo -n " Из ${denomination[$((RANDOM%num_denominations))]}"
echo ${suite[$((RANDOM%num_suites))]}


# $bozo sh pick-cards.sh
# Валет пик
```

```
# Спасибо, "jipe," за указание использования этого $RANDOM.
exit 0
```

Пример 9-13. Моделирование броуновского движения

```
#!/bin/bash
# brownian.sh
# Автор: Mendel Cooper
# Дата: 10/26/07
# Лицензия: GPL3

#  ---------------------------------------------------------------
#  Этот сценарий моделирует броуновское движение: случайное перемещение
#+ взвешенных частиц в жидкости, зависимость их движения от случайных течений
#+ и столкновений.  Известное как «Движение пьяного», разговорно-народное.

#  Он может также рассматриваться, как урезанное моделирование доски Гальтона,
#+ наклонной доски с узором из колышков, по которой один за одним скатываются
#+ шарики. В нижней части находятся лузы или ловушки, в которые эти шарики, в
#+ конце концов, попадают.
#  Понимайте это как своего рода минимальный вариант игры Pachinko.
#  Запустив сценарий вы увидите,что  большинство шариков группируются вокруг
#+ центральной лузы. Это согласуется с ожидаемым биномиальным распределением.
#  В моделировании доски Гальтона, сценарий учитывает такие параметры, как
#+ угол наклона доски, трение качения шариков, углы отскока, и эластичность
#+ колышков.
#  Насколько это влияет на точность моделирования?
#  ---------------------------------------------------------------

PASSES=500            #  Количество взаимодействующих частиц/шариков.
ROWS=10               #  Число «столкновений».
RANGE=3               #  0 - 2 выводимый диапазон $RANDOM.
POS=0                 #  Позиция лево/право.
RANDOM=$$             #  Получение создаваемых случайных чисел из PID
                      #+ сценария.

declare -a Slots      # Массив, содержащий совокупность результатов проходов.
NUMSLOTS=21           # Количество луз в нижней части доски.


Initialize_Slots () { # Обнуление всех элементов массива.
for i in $( seq $NUMSLOTS )
do
  Slots[$i]=0
done

echo                  #  В начале выполнения пустая строка.
  }


Show_Slots () {
echo; echo
echo -n " "
for i in $( seq $NUMSLOTS )   # Достаточность выводимых элементов массива.
do
  printf "%3d" ${Slots[$i]}   # Выделяем три места на каждый результат.
done
```

```
echo # Ряд луз:
echo " |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|"
echo "                                 ||"
echo #  Обратите внимание, что если счет в любой лузе превысит 99,
     #+ то это испортит вывод.
     #  Запускаем только (!) 500 проходов, как правило, этого хватает.
  }


Move () {               #  Перемещение одного элемента вправо/влево, или
                        #+ оставление на месте.
  Move=$RANDOM          #  Насколько случаен $RANDOM? Давайте посмотрим ...
  let "Move %= RANGE"   #  Нормализуем диапазон от 0 - 2.
  case "$Move" in
    0 ) ;;                      # Ничего, т.е., остался на месте.
    1 ) ((POS--));;             # Влево.
    2 ) ((POS++));;             # Вправо.
    * ) echo -n "Ошибка ";;   # Аномалия! (Не должна происходить.)
  esac
  }


Play () {                   # Один проход (в цикле).
i=0
while [ "$i" -lt "$ROWS" ]   # Одно событие в ряду.
do
  Move
  ((i++));
done

SHIFT=11                    # Почему 11, а не 10?
let "POS += $SHIFT"         # Смещение "нулевой позиции" в центр.
(( Slots[$POS]++ ))         # ОТЛАДКА: echo $POS

# echo -n "$POS "

  }


Run () {                    # Вне цикла.
p=0
while [ "$p" -lt "$PASSES" ]
do
  Play
  (( p++ ))
  POS=0                     # Сброс на 0. Почему?
done
  }


# --------------
# main ()
Initialize_Slots
Run
Show_Slots
# --------------

exit $?

#  Упражнения:
```

```
#  ---------
#  1) Покажите результаты в виде вертикальной диаграммы, или, как альтернативы,
#+    скейтеграммы.
#  2) Измените сценарий для использования /dev/urandom вместо $RANDOM.
#     Сделается ли результат более случайным?
#  3) Обеспечьте какую-нибудь 'анимацию' или графический вывод для каждого
#     шарика в игре.
```

Jipe подсказал способ для генерации случайных чисел в пределах диапазона.

```
#  Создание случайного числа между 6 и 30.
   rnumber=$((RANDOM%25+6))

#  создание случайного числа в одном и том же диапазоне 6 - 30,
#+ но число должно делится без остатка на 3.
   rnumber=$(((RANDOM%30/3+1)*3))

#  Обратите внимание, что это не всегда работает.
#  Завершается неудачей, если $RANDOM%30 возвращает 0.

#  Frank Wang предложил альтернативу:
   rnumber=$(( RANDOM%27/3*3+6 ))
```

Bill Gradwohl привел улучшенную формулу, которая работает для положительных чисел.

```
rnumber=$((RANDOM%(max-min+divisibleBy))/divisibleBy*divisibleBy+min))
```

Здесь Bill представляет универсальную функцию, которая возвращает случайное число из двух заданных.

Пример 9-14. Случайное из заданных значений

```
#!/bin/bash
# random-between.sh
# Случайные числа между двумя указанными значениями.
# Сценарий Bill Gradwohl, с небольшими изменениями автора документа.
# Исправление строк 187 и 189 Anthony Le Clezio.
# Используется с разрешения.


randomBetween() {
   #  Генерация положительных или отрицательных случайных чисел
   #+ между $min и $max
   #+ и делимых на $divisibleBy.
   #  Дает «достаточно случайное» распределение возвращаемых значений.
   #
   #  Bill Gradwohl - Oct 1, 2003

   syntax() {
   # Функции включенные в функцию.
      echo
      echo    "Синтаксис: randomBetween [min] [max] [multiple]"
      echo
      echo -n "Ожидает до 3 передаваемых параметров, "
      echo    "но все они совершенно не обязательны."
```

```
   echo    "min это минимальное значение"
   echo    "max это максимальное значение"
   echo -n "multiple указывает какой должен быть ответ "
   echo     "множества этого значения."
   echo    "т.е. ответ должен без остатка делится на это число."
   echo
   echo    "Если какое-то значение отсутствует, по умолчанию выводится как "
   echo -n "области: 0 32767 1 "
   echo -n "Успешное завершение возвращает 0, "
   echo    "не успешное завершение возвращает"
   echo    "функцию синтаксиса и 1."
   echo -n "Ответом является возвращаемый шаблон подстановки переменной "
   echo    "randomBetweenAnswer"
   echo -n "Отрицательные значения для любого переданного параметра "
   echo    "обрабатываются корректно."
}

local min=${1:-0}
local max=${2:-32767}
local divisibleBy=${3:-1}
# Значения по умолчанию, в случае, если параметры не переданы функции.

local x
local spread

# Удостоверяемся, что значение divisibleBy положительное.
[ ${divisibleBy} -lt 0 ] && divisibleBy=$((0-divisibleBy))

# Санитарная проверка.
if [ $# -gt 3 -o ${divisibleBy} -eq 0 -o  ${min} -eq ${max} ]; then
   syntax
   return 1
fi

# Посмотрим не поменялись ли местами min и max.
if [ ${min} -gt ${max} ]; then
   # Меняем их.
   x=${min}
   min=${max}
   max=${x}
fi

#  Если min не делится без остатка на $divisibleBy, то исправляем min,
#+ чтобы находилось в пределах диапазона.
if [ $((min/divisibleBy*divisibleBy)) -ne ${min} ]; then
   if [ ${min} -lt 0 ]; then
      min=$((min/divisibleBy*divisibleBy))
   else
      min=$((((min/divisibleBy)+1)*divisibleBy))
   fi
fi

#  Если max не делится без остатка на $divisibleBy, то исправляем max,
#+ чтобы находилось в пределах диапазона.
if [ $((max/divisibleBy*divisibleBy)) -ne ${max} ]; then
   if [ ${max} -lt 0 ]; then
      max=$((((max/divisibleBy)-1)*divisibleBy))
   else
      max=$((max/divisibleBy*divisibleBy))
   fi
fi
```

```
   #  ------------------------------------------------------------------
   #  Теперь сделаем настоящую работу.

   #  Обратите внимание, что чтобы получить правильное распределение для
   #+ конечных точек, диапазону случайных значений будет позволено находиться
   #+ между abs(max-min)+divisibleBy, а не только abs(max-min)+1.
   #  Незначительное увеличение  надлежаще распределит конечные точки.
   #  Использование замененной формулы abs(max-min)+1 будет по-прежнему давать
   #+ правильные ответы, но случайность этих ответов будет значительно ниже,
   #+ чем при использовании правильной формулы.
   #  ------------------------------------------------------------------

   spread=$((max-min))
   #  Omair Eshkenazi указывает, что эта проверка необходима, поскольку max и
   #+ min, уже были заменены друг другом.
   [ ${spread} -lt 0 ] && spread=$((0-spread))
   let spread+=divisibleBy
   randomBetweenAnswer=$(((RANDOM%spread)/divisibleBy*divisibleBy+min))

   return 0

   #  Как поясняет Paulo Marcel Coelho Aragao
   #+ когда $max и $min не делятся на$divisibleBy,
   #+ формула будет не правильной.
   #
   #  Он предлагает другую формулу:
   #    rnumber = $((RANDOM%(max-min+1)+min)/divisibleBy*divisibleBy))

}

# Давайте проверим функцию.
min=-14
max=20
divisibleBy=3


#  Создаем массив ожидаемых ответов и проверок, чтобы убедиться, что мы
#+ получаем, по крайней мере, одно число из каждого ответа, в достаточно
#+ длинном цикле.
declare -a answer
minimum=${min}
maximum=${max}
   if [ $((minimum/divisibleBy*divisibleBy)) -ne ${minimum} ]; then
      if [ ${minimum} -lt 0 ]; then
         minimum=$((minimum/divisibleBy*divisibleBy))
      else
         minimum=$((((minimum/divisibleBy)+1)*divisibleBy))
      fi
   fi


   #  Если max не делится без остатка на $divisibleBy, то исправляем max,
   #+ чтобы находилось в пределах диапазона.

   if [ $((maximum/divisibleBy*divisibleBy)) -ne ${maximum} ]; then
      if [ ${maximum} -lt 0 ]; then
         maximum=$((((maximum/divisibleBy)-1)*divisibleBy))
      else
         maximum=$((maximum/divisibleBy*divisibleBy))
      fi
```

```
   fi


#  Нам нужно создать только положительный массив индексов, поэтому
#+ нам нужно перемещение, которое будет гарантировать
#+ положительные результаты.

disp=$((0-minimum))
for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
   answer[i+disp]=0
done


# Прогоним цикл большое количество раз, чтобы увидеть, что получится.
loopIt=1000   #  Автор сценария предлагает 100000,
              #+ но на это потребуется довольно много времени.

for ((i=0; i<${loopIt}; ++i)); do

   #  Обратите внимание, что здесь min и max указаны в обратном порядке,
   #+ чтобы сделать правильное, в этом случае, функционирование.

   randomBetween ${max} ${min} ${divisibleBy}

   # Сообщение об ошибке, если ответ будет не ожидаемый.
   [ ${randomBetweenAnswer} -lt ${min} -o ${randomBetweenAnswer} -gt ${max} ] \
   && echo MIN or MAX error - ${randomBetweenAnswer}!
   [ $((randomBetweenAnswer%${divisibleBy})) -ne 0 ] \
   && echo ошибка ДЕЛЕНИЯ - ${randomBetweenAnswer}!

   # Сохраняем ответ в статистику.
   answer[randomBetweenAnswer+disp]=$((answer[randomBetweenAnswer+disp]+1))
done


# Давайте проверим результаты

for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
   [ ${answer[i+disp]} -eq 0 ] \
   && echo "Мы никогда не получим ответ от $i." \
   || echo "${i} произошло ${answer[i+disp]} раз."
done


exit 0
```

Насколько случаен $RANDOM? Лучший способ проверить это - написать сценарий, который отслеживает распределения «случайных» чисел порожденных $RANDOM. Давайте бросим кость $RANDOM раз...

Пример 9-15. Бросание одной кости с RANDOM

```
#!/bin/bash
# Насколько случаен RANDOM?

RANDOM=$$       #  Перезапуск генератора случайных чисел, для использования ID
                #+ процесса сценария.
```

```
PIPS=6          # Кость имеет 6 плоскостей.
MAXTHROWS=600   # Увеличьте, если у вас много свободного времени.
throw=0         # Количество бросков кости.

ones=0          #  Необходимо установить счетчик в ноль,
twos=0          #+ так как не присвоенная переменная имеет значение null,
                #+ а НЕ ноль.
threes=0
fours=0
fives=0
sixes=0

print_result ()
{
echo
echo "ones =   $ones"
echo "twos =   $twos"
echo "threes = $threes"
echo "fours =  $fours"
echo "fives =  $fives"
echo "sixes =  $sixes"
echo
}

update_count()
{
case "$1" in
  0) ((ones++));;   # Так как  бросок кости это не «ноль», то соответствует 1.
  1) ((twos++));;   # А это 2.
  2) ((threes++));; # Так далее.
  3) ((fours++));;
  4) ((fives++));;
  5) ((sixes++));;
esac
}

echo


while [ "$throw" -lt "$MAXTHROWS" ]
do
  let "die1 = RANDOM % $PIPS"
  update_count $die1
  let "throw += 1"
done

print_result

exit $?

#  Выпадения должны распределиться равномерно, предполагая случайный RANDOM.
#  С $MAXTHROWS равным 600, все должно группироваться вокруг 100,
#+ плюс-минус 20 или около того.
#  Имейте в виду, что RANDOM это ***псевдослучайный***  генератор.
#  Случайность - это глубокая и сложная тема.
#  Достаточно большие "случайные" последовательности могут демонстрировать
#+ хаотическое и другое "неслучайное" поведение.

# Упражнение (легкое):
# ---------------
```

```
# Перепишите этот сценарий, для подбрасывания монеты 1000 раз.
# Выберите "ОРЕЛ" и "РЕШКА."
```

Как мы видели в предыдущем примере, при вызове, лучше каждый раз перезапускать генератор RANDOM. Используя один и тот же источник для RANDOM он повторяет тот же ряд цифр. [2] (Это совпадает с поведением функции ***random()*** в Си)

Пример 9-16. Смена источника RANDOM

```
#!/bin/bash
# seeding-random.sh: Seeding the RANDOM variable.
# v 1.1, reldate 09 Feb 2013

MAXCOUNT=25       # Количество генерируемых чисел.
SEED=

random_numbers ()
{
local count=0
local number

while [ "$count" -lt "$MAXCOUNT" ]
do
  number=$RANDOM
  echo -n "$number "
  let "count++"
done
}

echo; echo

SEED=1
RANDOM=$SEED      # Устанавливаем начальным RANDOM случайно созданное число.
echo "Случайное начальное число = $SEED"
random_numbers


RANDOM=$SEED      # Другие начальные числа RANDOM ...
echo; echo "Снова, с другим случайным начальным числом ..."
echo "Случайное начальное число = $SEED"
random_numbers    # ... воспроизводит ту же серию чисел.
                  #
                  # Когда  полезно дублировать 'случайные' серии?

echo; echo

SEED=2
RANDOM=$SEED      # Пробуем снова, но с другим начальным числом ...
echo "Случайный начальное число = $SEED"
random_numbers    # ... дает отличающиеся серии чисел.

echo; echo

# RANDOM=$$  это начальное число RANDOM из id процесса сценария.
# Так же возможно начальное число RANDOM из команд 'time' или 'date'.

# Пофантазируем...
```

```
SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }'| sed s/^0*//)
#  Псевдо-случайный вывод извлекается из /dev/urandom
#+ (файла устройства системы псевдо-случайных чисел), затем
#+ "od" преобразует в строку чисел для вывода (восьмеричное представление),
#+ затем "awk" извлекает только одно число для начального числа, и,
#+ наконец, "sed" удаляет все начальные нули.
RANDOM=$SEED
echo "Случайное начальное число = $SEED"
random_numbers

echo; echo

exit 0
```

Файл псевдо-устройства */dev/urandom* предоставляет способ создания намного большего количества "псевдослучайных" чисел, чем переменная $RANDOM. **dd if=/dev/urandom of=targetfile bs 1 count XX** создает файл хорошо рассеянных псевдослучайных чисел. Однако, для присвоения этих чисел переменной в сценарии требуется обходной путь, например фильтрация через **od** (пример выше, Пример 16-14 и Пример A-36), или даже конвейер в **md5sum** (см. Пример 36-16).

Есть также другие способы генерации псевдослучайных чисел в сценарии. **Awk** предоставляется удобным средством для этого.

Пример 9-17. Псевдослучайные числа, с помощью awk

```
#!/bin/bash
#  random2.sh: Возвращение псевдослучайных чисел в диапазоне 0 - 1,
#+ до 6 десятичных. Например: 0.822725
#  С помощью функции awk rand().

AWKSCRIPT=' { srand(); print rand() } '
#           Команды(ы)/параметры передаваемые в awk
#  Обратите внимание, что srand() меняет начальное число генератора
#+ случайных чисел awk.


echo -n "Случайное число между 0 и 1 = "

echo | awk "$AWKSCRIPT"
# Что произойдет, если вы оставите вывод «echo»?  ?

exit 0


# Упражнения:
# ---------

# 1) С помощью конструкции цикла, выведите 10 различных случайных чисел.
#    (Подсказка: нужно менять начальное число функции srand() на другой
#+   начальное число  в каждом проходе цикла. Что произойдет, если вы
#+   не сделаете этого?).

# 2) С помощью множителя целого числа, как фактора масштабирования,
#+   создайте случайные числа в диапазоне от 10 до 100.
# 3) Как и в упражнении #2, выше, создайте в этот раз случайные целые
```

```
#+   числа.
```

Команда **date** также поддается генерации *псевдослучайных целых последовательностей.*

Примечания

[1] Истинная 'случайность', если она вообще существует, может быть найдена только с помощью некоторых, полностью понятых, природных явлений, таких как радиоактивный распад. Компьютеры только имитируют случайность, а создание компьютером последовательности 'случайных' чисел поэтому и называют - *псевдослучайными*.

[2] Начальное число псевдослучайной числовой последовательности компьютерной графики можно считать идентификационной меткой. Например, подумайте о псевдослучайной серии с начальным числом 23, как *Series #23*.

Свойством серии псевдослучайных чисел является величина цикла, прежде чем он начнет повторяться. Хороший генератор псевдослучайных чисел будет производить серии с очень длинным циклом.

Глава 10. Управление переменными

Содержание

10.1. Управление строками

Bash поддерживает удивительное количество операций обработки строк. К сожалению, эти средства не имеют единой направленности. Одни, это подмножества ***подстановки параметров***, а другие попадают под функциональность команды UNIX ***expr***. Это приводит к несогласованности синтаксиса команд и перекрытию функций, не говоря уже о путанице.

Длина (размер) строки

${#string}

*expr **length** $string*

Это эквиваленты ***strlen()*** в *Cи*.

expr "$string" : '.'*

```
stringZ=abcABC123ABCabc

echo ${#stringZ}                 # 15 символов
echo `expr length $stringZ`      # 15
echo `expr "$stringZ" : '.*'`    # 15
```

Пример 10-1. Вставка пустой строки между абзацами в текстовом файле

```
#!/bin/bash
# paragraph-space.sh
# Ver. 2.1, Reldate 29Jul12 [исправлено]

# Вставляет пустую строку с одним интервалом между абзацами текстового файла.
# Usage: $0 <FILENAME

MINLEN=60        # Изменить это значение? Это решает вызов.
#  Предполагаем, строка короче, чем $MINLEN символов
#+ будет окончанием азаца. См. упражнения ниже.

while read line  # Все строки, сколько имеет входной файл ...
do
  echo "$line"   # Вывод самой строки.
```

```
  len=${#line}
  if [[ "$len" -lt "$MINLEN" && "$line" =~ [*{\.}]$ ]]
# if [[ "$len" -lt "$MINLEN" && "$line" =~ \[*\.\] ]]
# Обновление Bash испортило предыдущую версию этого сценария. Ой!
# Спасибо, Halim Srama, указавшему на это и предложившему исправление.
    then echo    #  Добавляем пустую строку сразу после короткой строки,
  fi             #+ уничтожая точки.
done

exit

# Упражнения:
# ---------
#  1) Сценарий обычно вставляет пустую строку в конце
#+    целевого файла. Исправьте это.
#  2) Строка 17 только рассматривает точки как предложение уничтожить.
#     Измените ее, чтобы включать другие общие символы окончания предложения,
#+    такие как ?, !, и ".
```

Размер (длина) Substring (содержимого в строке) соответствующей началу строки

expr ***match*** *"$string" '$substring'*

$substring это ***регулярное выражение***.

expr "$string" : '$substring'

$substring это ***регулярное выражение***.

```
stringZ=abcABC123ABCabc
#       |------|
#       12345678

echo `expr match "$stringZ" 'abc[A-Z]*.2'`   # 8
echo `expr "$stringZ" : 'abc[A-Z]*.2'`       # 8
```

`index` (Индекс)

expr index $string $substring

Числовая позиция в *$string* первого совпадающего символа *$substring* .

```
stringZ=abcABC123ABCabc
#       123456 ...
echo `expr index "$stringZ" C12`             # 6
                                             # Позиция C (первый совпадающий
                                             #+ символ)/

echo `expr index "$stringZ" 1c`              # 3
# 'c' (в позиции #3) первой совпадает до '1'.
```

Примерно эквивалентно ***strchr()*** в *Cи*.

Извлечение Substring

${string:position}

Извлекает содержимое строки *$string* начиная с *$position*.

Если параметрами $string являются "*" или "@", то извлекаются ***позиционные параметры***, [1] начиная с $position.

${string:position:length}

Извлекает *$length* символов *$string* начиная с позиции *$position*.

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       Отсчет с 0.

echo ${stringZ:0}                            # abcABC123ABCabc
echo ${stringZ:1}                            # bcABC123ABCabc
echo ${stringZ:7}                            # 23ABCabc

echo ${stringZ:7:3}                          # 23A
                                             # Три символа substring.


# Возможно ли начинать отсчет от правой части строки?

echo ${stringZ:-4}                           # abcABC123ABCabc
# По умолчанию это полная строка, как ${parameter:-default}.
# Однако ...

echo ${stringZ:(-4)}                         # Cabc
echo ${stringZ: -4}                          # Cabc
#              ^   пробел
#  Теперь это работает.
#  Параметр позиции помещен в круглых скобках или перед ним
#+ добавлен пробел.

# Спасибо, Dan Jacobson, за разъяснение.
```

Аргументы *position* и *length* могут быть "параметризированны," то есть представлены как переменные, а не как числовые константы.

Пример 10-2. Создание 8-символьной "случайной" строки

```
#!/bin/bash
# rand-string.sh
# Создание 8-символьной "случайной" строки.

if [ -n "$1" ]  #  Если имеется аргумент командной строки,
then            #+ то он устанавливается как начало строки.
  str0="$1"
```

```
else            #  Иначе, как начало строки, используем PID сценария.
  str0="$$"
fi

POS=2  # Начало в строке с позиции 2.
LEN=8  # Извлекаются 8 символов.

str1=$( echo "$str0" | md5sum | md5sum )
#  Двойное             ^^^^^^   ^^^^^^
#+ конвейер и еще конвейер в md5sum.

randstring="${str1:$POS:$LEN}"
#                   ^^^^ ^^^^ Можно параметризовать

echo "$randstring"

exit $?

# bozo$ ./rand-string.sh my-password
# 1bdd88c4

#  Нет, это не рекомендую,
#+ как способ создания возможности взлома паролей.
```

Если параметром `$string` является "*" или "@", то извлекается максимально `$length` позиционных параметров, начиная с `$position`.

```
echo ${*:2}          # Выводит второй и все следующие поз. параметры.
echo ${@:2}          # Так же, как и выше.

echo ${*:2:3}        # Выводит три поз. параметра, начиная со второго.
```

expr substr $string $position $length

Извлекает *$length* символов из *$string* начиная с *$position*.

```
stringZ=abcABC123ABCabc
#       123456789......
#       Отсчет с 1.

echo `expr substr $stringZ 1 2`              # ab
echo `expr substr $stringZ 4 3`              # ABC
```

expr match "$string" '\($substring\)'

Извлекает *$substring* от начала *$string*, где *$substring* это ***регулярное выражение***.

expr "$string" : '\($substring\)'

Извлекает *$substring* от начала *$string*, где *$substring* это ***регулярное выражение***.

```
stringZ=abcABC123ABCabc
#       =======

echo `expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
echo `expr "$stringZ" : '\(.......\)'`                   # abcABC1
# Все эти формы дают одинаковый результат.
```

expr match "$string" '.*\($substring\)'

Извлекает *$substring* от конца *$string*, где *$substring* это ***регулярное выражение***.

expr "$string" : '.*\($substring\)'

Извлекает *$substring* от конца *$string*, где *$substring* это ***регулярное выражение***.

```
stringZ=abcABC123ABCabc
#               ======

echo `expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)'`    # ABCabc
echo `expr "$stringZ" : '.*\(......\)'`                       # ABCabc
```

Удаление *Substring*

${string#substring}

Удаляет короткие соответствия *$substring* от начала *$string*.

${string##substring}

Удаляет длинные соответствия *$substring* от начала *$string*.

```
stringZ=abcABC123ABCabc
#       |----|          короткое
#       |----------|    длинное

echo ${stringZ#a*C}      # 123ABCabc
# Вырезает короткие соответствия между 'a' и 'C'.

echo ${stringZ##a*C}     # abc
# Вырезает длинные соответствия между 'a' и 'C'.


# Вы можете параметризовать substrings.
```

```
X='a*C'

echo ${stringZ#$X}      # 123ABCabc
echo ${stringZ##$X}     # abc
                        # Как и выше.
```

${string%substring}

Удаляет короткие соответствия *$substring* от конца *$string*.

Например:

```
# Переименование всех файлов в $PWD с суффиксом "TXT" в суффикс "txt".
# Например, "file1.TXT" станет "file1.txt" ...

SUFF=TXT
suff=txt

for i in $(ls *.$SUFF)
do
  mv -f $i ${i%.$SUFF}.$suff
  #  Оставляет без изменений все, *кроме* короткого шаблона соответствия,
  #+ начиная от правой стороны переменной $i...
done ### При желании все можно объединить в "однострочник".

# Благодарность Rory Winston.
```

${string%%substring}

Удаляет длинные соответствия *$substring* от конца *$string*.

```
stringZ=abcABC123ABCabc
#                   ||     короткое
#        |------------|     длинное

echo ${stringZ%b*c}      # abcABC123ABCa
# Вырезает короткие соответствия между 'b' и 'c', от окончания $stringZ.

echo ${stringZ%%b*c}     # a
# Вырезает длинные соответствия между 'b' и 'c', от окончания $stringZ..
```

Этот оператор используется для ***создания имен файлов***.

Пример 10-3. Преобразование форматов графических файлов, с изменением имени файла

```
#!/bin/bash
#  cvt.sh:
#  Конвертирование всех файлов изображений MacPaint директории
#+ в формат "pbm".
```

```
#  Используем бинарные файлы "macptopbm" из пакета "netpbm",
#+ сопровождаемого Brian Henderson (bryanh@giraffe-data.com).
#  Netpbm это обычная часть большинства дистрибутивов Linux.

OPERATION=macptopbm
SUFFIX=pbm          # Новый суффикс имен файлов.

if [ -n "$1" ]
then
  directory=$1      # Имя директории задается как аргумент сценария...
else
  directory=$PWD    # Иначе используется текущая рабочая директория.
fi

#  Предполагается, что все файлы в целевой директории это файлы изображений
#+ MacPaint, с суффиксом имени файла '.mac'.

for file in $directory/*    # Подстановка шаблона имени файла.
do
  filename=${file%.*c}      #  Удаление суффикса ".mac" имени файла
                            #+ ('.*c' соответствие всему
                            #+ между '.' и 'c', включительно).
  $OPERATION $file > "$filename.$SUFFIX"
                            # Перенаправление преобразования нового имени.
  rm -f $file               # Удаление оригинальных файлов после конвертации.
  echo "$filename.$SUFFIX"  # Вывод журнала всего происходившего в stdout.
done

exit 0

# Упражнение:
# --------
#  Как можно заметить, этот сценарий преобразует *все* файлы в
#+ текущей рабочей директории.
#  Измените работу на *только* файлы с суффиксом ".mac".


# *** А здесь другой способ того же самого. *** #

#!/bin/bash
# Пакетное преобразование в различные графические форматы.
# Предполагается, что imagemagick установлен (стандартно в Linux).

INFMT=png   # Может быть tif, jpg, gif, и т.д.
OUTFMT=pdf  # Может быть tif, jpg, gif, pdf,  и т.д.

for pic in *"$INFMT"
do
  p2=$(ls "$pic" | sed -e s/\.$INFMT//)
  # echo $p2
    convert "$pic" $p2.$OUTFMT
    done

exit $?
```

Пример 10-4. Конвертирование потоковых файлов аудио в *ogg*

```
#!/bin/bash
# ra2ogg.sh: Конвертирование потоковых файлов аудио (*.ra) в ogg.

#  Используется "mplayer" программа медиа плейер:
#      http://www.mplayerhq.hu/homepage
#  Используется библиотека "ogg" и "oggenc":
#      http://www.xiph.org/
#
#  Этому сценарию может понадобиться установка соответствующих кодеков,
#+ таких как sipr.so ...
#  Возможно и пакета compat-libstdc++.


OFILEPREF=${1%%ra}      # Удаление суффикса "ra".
OFILESUFF=wav           # Суффикс файлов wav.
OUTFILE="$OFILEPREF""$OFILESUFF"
E_NOARGS=85

if [ -z "$1" ]          # Должен быть указан файл для преобразования.
then
  echo "Usage: `basename $0` [имя файла]"
  exit $E_NOARGS
fi


###############################################################################
mplayer "$1" -ao pcm:file=$OUTFILE
oggenc "$OUTFILE"  #  Автоматически добавляется правильное расширение файла
                   #+ oggenc.
###############################################################################

rm "$OUTFILE"      # Удаляем промежуточные файлы *.wav.
                   # Если вы хотите сохранить их, то закомментируйте строку
#+ выше.

exit $?

#  Примечание:
#  ----
#  На веб-сайте, просто нажав на *.ram, потоковые аудио файлы, обычно
#+ загружается только URL фактического *.ra аудио файла.
#  Можно использовать "wget" или что-то похожее для скачивания
#+ самого файла *.ra.


#  Упражнения:
#  ---------
#  Этот сценарий преобразует только *.ra файлы.
#  Добавьте гибкости, позволяя использовать *.ram и другие имена файлов.
#
#  Если вы действительно амбициозны, разверните сценарий для автоматической
#+ загрузки и преобразования потоковых аудио файлов.
#  Задайте URL-адрес пакетной загрузки потоковых аудио файлов (с помощью "wget')
#+ и конвертируйте их на лету.
```

Простая эмуляция **`getopt`** с использованием конструкций извлечения ***substring***.

Пример 10-5. Эмуляция *getopt*

```
#!/bin/bash
# getopt-simple.sh
# Автор: Chris Morgan
# Используется в ABS Guide с разрешения.


getopt_simple()
{
    echo "getopt_simple()"
    echo "Параметры '$*'"
    until [ -z "$1" ]
    do
      echo "Обрабатываемые параметры: '$1'"
      if [ ${1:0:1} = '/' ]
      then
          tmp=${1:1}               # Вырезаем ведущие '/' ...
          parameter=${tmp%%=*}     # Извлекаем имя.
          value=${tmp##*=}         # Извлекаем значение.
          echo "Параметр: '$parameter', значение: '$value'"
          eval $parameter=$value
      fi
      shift
    done
}

# Передаем все опции getopt_simple().
getopt_simple $*

echo "test это '$test'"
echo "test2 это '$test2'"

exit 0  # См. Так же измененную версию этого сценария UseGetOpt.sh.

---

sh getopt_example.sh /test=value1 /test2=value2

Параметры '/test=value1 /test2=value2'
Обработка параметра: '/test=value1'
Параметр: 'test', значение: 'value1'
Обработка параметра: '/test2=value2'
Параметр: 'test2', значение: 'value2'
test это 'value1'
test2 это 'value2'
```

Замена *Substring*

`${string/substring/replacement}`

Замена ***первого*** соответствия *`$substring`* на *`$replacement`*. [2]

`${string//substring/replacement}`

Замена ***всех*** соответствий *`$substring`* на *`$replacement`*.

```
stringZ=abcABC123ABCabc

echo ${stringZ/abc/xyz}       # xyzABC123ABCabc
                              # замена первого соответствия 'abc' на 'xyz'.

echo ${stringZ//abc/xyz}      # xyzABC123ABCxyz
                              # Замена всех соответствий 'abc' на  'xyz'.

echo  ---------------
echo "$stringZ"               # abcABC123ABCabc
echo  ---------------
                              # Сама строка не изменяется!

# Могут ли быть соответствие и замена строк параметризованы?
match=abc
repl=000
echo ${stringZ/$match/$repl}  # 000ABC123ABCabc
#              ^      ^         ^^^
echo ${stringZ//$match/$repl} # 000ABC123ABC000
# Да!          ^      ^        ^^^         ^^^

echo

# Что происходит, если строка $replacement отсутствует?
echo ${stringZ/abc}           # ABC123ABCabc
echo ${stringZ//abc}          # ABC123ABC
# Происходит простое удаление.
```

${string/#substring/replacement}

Если соответствие *$substring* в ***начале*** окончания*$string*, подстанавливается *$replacement* на место *$substring*.

${string/%substring/replacement}

Если соответствие *$substring* в ***конце*** окончания *$string*, подстанавливается *$replacement* на место *$substring*.

```
stringZ=abcABC123ABCabc

echo ${stringZ/#abc/XYZ}       # XYZABC123ABCabc
                               # Замена соответствия в начале 'abc' на 'XYZ'.

echo ${stringZ/%abc/XYZ}       # abcABC123ABCXYZ
                               # Замена соответствия в конце 'abc' на 'XYZ'.
```

10.1.1. Обработка строк с помощью *awk*

Сценарий Bash может вызывать средство обработки строки ***awk***, как альтернативу использованию встроенных операций.

Пример 10-6. Альтернативные способы извлечения и поиска *substring*

```
#!/bin/bash
# substring-extraction.sh

String=23skidoo1
#      012345678    Bash
#      123456789    awk
# Обратите внимание на отличие систем индексации строк:
# В Bash номером первого символа строки является 0.
# В Awk  номером первого символа строки является 1.

echo ${String:2:4} # позиция 3 (0-1-2), величиной 4 символа
                   # перенос

# В awk эквивалентом ${string:pos:length} является substr(string,pos,length).
echo | awk '
{ print substr("'"${String}"'",3,4)      # перенос
}
'
#  Передаем конвейеру пустое  "echo", чтобы дать awk фиктивный ввод,
#+ и, таким образом, делаем ненужным задание имени файла.

echo "----"

# И аналогично:

echo | awk '
{ print index("'"${String}"'", "skid")      # 3
}                                           # (перенос начала на позицию 3)
'   # awk эквивалентно "expr index" ...

exit 0
```

10.1.2. Дополнительные ссылки

Для получения более подробной информации об управлении `string` в сценарии, см. Раздел 10.2 и в *соответствующий раздел* списка команды ***`expr`***.

Примеры сценариев:

1. Пример 16-9
2. Пример 10-9
3. Пример 10-10
4. Пример 10-11
5. Пример 10-13
6. Пример A-36
7. Пример A-41

Примечания

[1] Это относится к аргументам командной строки или параметрам, передаваемым *функцub*.

[2] Обратите внимание, что *$substring* и *$replacement* могут относиться либо к текстовым строкам, либо к переменным, в зависимости от контекста. См. первый пример использования.

10.2. Подстановка параметров

Управление и/или расширение переменных

${parameter}

То же, что и *$parameter*, т.е., значение переменной *parameter*. В некоторых контекстах работает только менее двусмысленная форма *${parameter}*.

Может использоваться для объединения строк переменных

```
your_id=${USER}-on-${HOSTNAME}
echo "$your_id"
#
echo "Old \$PATH = $PATH"
PATH=${PATH}:/opt/bin  # Добавляет /opt/bin в $PATH расширяя сценарий.
echo "New \$PATH = $PATH"
```

${parameter-default}**, **${parameter:-default}

Если параметр не установлен, то используется умолчание

```
var1=1
var2=2
# var3 не установлена.

echo ${var1-$var2}   # 1
echo ${var3-$var2}   # 2
#           ^          Обратите внимание на префикс $.



echo ${username-`whoami`}
```

```
# Выводит на экран `whoami`, если переменная $username еще не установлена.
```

${parameter-default}* и *${parameter:-default} почти эквивалентны. Дополнение: разница существует, только тогда, когда *parameter* был объявлен, но является пустым (**null**).

```
#!/bin/bash
# param-sub.sh

#  Объявление переменной влияет
#+ на запуск умолчания, даже если
#+ переменная имеет значение null.

username0=
echo "username0 объявлено, но ей присвоено значение null."
echo "username0 = ${username0-`whoami`}"
# Не выводится.

echo

echo username1 не было объявлено.
echo "username1 = ${username1-`whoami`}"
# Будет выведено имя пользователя.

username2=
echo "username2 объявлена, но присвоено значение null."
echo "username2 = ${username2:-`whoami`}"
#                           ^
# Будет выведено, потому что в условии проверки : стоит перед -.
# Сравните с первым примером, выше.


#

# Еще раз:

variable=
# Переменная объявлена, но присвоено значение null.

echo "${variable-0}"    # (нет вывода)
echo "${variable:-1}"   # 1
#              ^

unset variable

echo "${variable-2}"    # 2
echo "${variable:-3}"   # 3

exit 0
```

Конструкция *параметр по умолчанию* находит использование в замещении «отсутствующих» аргументов командной строки в сценариях.

```
DEFAULT_FILENAME=generic.data
filename=${1:-$DEFAULT_FILENAME}
```

```
#  Если не указано иное, работает следующий блок команд
#+ файла "generic.data".
#  Начало-блока-команд
#  ...
#  ...
#  ...
#  Окончание-блока-команд



#  Из примера "hanoi2.bash":
DISKS=${1:-E_NOPARAM}   # Необходимо указать количество дисков.
#  Присваиваем $DISKS параметр командной строки $1,
#+ или $E_NOPARAM, если параметр не присвоен.
```

См. так же Пример 3-4, Пример 31-2 и Пример A-6.

Сравните этот способ с использованием ***and list*** *для расширения аргумента командной строки по умолчанию.*

`${parameter=default}`, `${parameter:=default}`

> Если параметр не присвоен, то ***присваивает*** его *умолчание.*

Обе формы почти эквивалентны. `:` делает различие только тогда, когда `$parameter` был объявлен и его значение ***null***, [1], как и выше.

```
echo ${var=abc}   # abc
echo ${var=xyz}   # abc
# $var уже было присвоено abc, поэтому она не изменяется.
```

`${parameter+alt_value}`, `${parameter:+alt_value}`

> Если параметр присвоен, используется значение **`alt_value`**, в противном случае - пустая строка (***null string***).

Обе формы почти эквивалентны. `:` делает различие только тогда, когда `$parameter` был объявлен и его значение ***null***, как и выше.

```
echo "###### \${parameter+alt_value} ########"
echo

a=${param1+xyz}
echo "a = $a"      # a =

param2=
a=${param2+xyz}
echo "a = $a"      # a = xyz

param3=123
```

```
a=${param3+xyz}
echo "a = $a"      # a = xyz

echo
echo "###### \${parameter:+alt_value} ########"
echo

a=${param4:+xyz}
echo "a = $a"      # a =

param5=
a=${param5:+xyz}
echo "a = $a"      # a =
# Результат отличается от a=${param5+xyz}

param6=123
a=${param6:+xyz}
echo "a = $a"      # a = xyz
```

${parameter?err_msg}, ${parameter:?err_msg}

Если параметр присвоен, то используется он, в противном случае выводится *err_msg* и *сценарий завершается* со статусом выхода 1.

Обе формы почти эквивалентны. `:` делает различие только тогда, когда *`$parameter`* был объявлен и его значение ***null***, как и выше.

Пример 10-7. Использование подстановки параметров и сообщений об ошибках

```
#!/bin/bash

#  Проверка некоторых системных переменных окружения.
#  Это хорошее профилактическое обслуживание.
#  Например, если  $USER, имя субъекта в консоли, не задано, то машина
#+ не распознает вас.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
  echo "Имя машины $HOSTNAME."
  echo "Вы $USER."
  echo "Ваша домашняя директория $HOME."
  echo "Ваша почта INBOX находится в $MAIL."
  echo
  echo "Если Вы читаете это сообщение,"
  echo "то установлены рискованные  переменные окружения."
  echo
  echo

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

#  Конструкция ${variablename?} то же может проверять
#+ присвоение переменных в сценарии.

ThisVariable=Value-of-ThisVariable
#  Обратите внимание, что именам строковых переменных
#+ могут быть присвоены символы.
```

```
: ${ThisVariable?}
echo "Значение ThisVariable это $ThisVariable".

echo; echo


: ${ZZXy23AB?"ZZXy23AB не присвоено."}
#  Так как ZZXy23AB не присвоен, то сценарий
#+ завершается с сообщением об ошибке

# Можно указать текст сообщения об ошибке.
# : ${variablename?"СООБЩЕНИЕ ОБ ОШИБКЕ"}


# Те же результаты:   dummy_variable=${ZZXy23AB?}
#                     dummy_variable=${ZZXy23AB?"ZXy23AB не присвоено."}
#
#                     echo ${ZZXy23AB?} >/dev/null

#  Сравните эти способы проверки задания переменной с «set -u»...



echo "Если Вы не видите это сообщение, значит сценарий завершился."

HERE=0
exit $HERE   # Выход сценария НЕ здесь.

# На самом деле, этот сценарий должен возвращать статус выхода (echo $?) 1.
```

Пример 10-8. Подстановка параметров и сообщение "usage"

```
#!/bin/bash
# usage-message.sh

: ${1?"Usage: $0 ARGUMENT"}
#  Здесь сценарий завершается, если параметр командной строки отсутствует,
#+ с последующим выводом сообщения об ошибке.
#  usage-message.sh: 1: Usage: usage-message.sh ARGUMENT

echo "Эти две строки выводятся только если задан параметр командной строки."
echo "Параметр командной строки = \"$1\""

exit 0  #  Выход будет здесь только в том случае, если присутствует параметр
        #+ командной строки.

# Проверьте статус выхода, с, так  и без, параметра командной строки.
# Если параметр командной строки есть, то "$?" возвратит 0.
# Если нет, то "$?" возвратит 1.
```

Подстановка параметра и/или расширение. Следующее выражение является дополнением строковых операций **`match`** *`in`* **`expr`** (см. Пример 16-9). Конкретно используется в разборе пути имен файлов.

Размер (длина) переменной/удаления *Substring*

${#var}

Размер строки (количество символов в $var). Для массива *${#array}* это размер первого элемента в массиве.

Исключения:

- ***${#*}*** и ***${#@}*** задают *числовые позиционные параметры*.
- Для массива, ***${#array[*]}*** и ***${#array[@]}*** задают *количество элементов* в массиве

Пример 10-9. Размер переменной

```
#!/bin/bash
# length.sh

E_NO_ARGS=65

if [ $# -eq 0 ]  #  Необходимы аргументы командной строки для демонстрации
                 #+ работы сценария.
then
  echo "Вызовите этот сценарий с одним, или более, аргументом командной
строки."
  exit $E_NO_ARGS
fi

var01=abcdEFGH28ij
echo "var01 = ${var01}"
echo "Размер var01 = ${#var01}"
# Теперь, давайте попробуем вставить пробел.
var02="abcd EFGH28ij"
echo "var02 = ${var02}"
echo "Размер var02 = ${#var02}"

echo "Количество аргументов командной строки переданных сценарию = ${#@}"
echo "Количество аргументов командной строки переданных сценарию = ${#*}"

exit 0
```

${var#Pattern}, ${var##Pattern}

${var#Pattern} Удаляет из $var *короткие* совпадающие части $Pattern, от *начала* $var.

${var##Pattern} Удаляет из $var *длинные* совпадающие части $Pattern, от *начала* $var.

Иллюстрация использования из Примера A-7:

```
# Пример функции из примера "days-between.sh".
# Вырезаем начальные нули из переданных аргументов.

strip_leading_zero () #  Вырезаем возможные начальные нули
{                     #+ из переданных аргументов.
  return=${1#0}       #  "1" означает, что "$1" – переданный аргумент.
}                     #  "0" это то, что удаляем из "$1" – вырезаемые нули.
```

Manfred Schwarb предлагает более сложный вариант выше:

```
strip_leading_zero2 () #  Удаляем возможные ведущие нули, иначе Bash будет
                       #+ интерпретировать такие числа как восьмеричные
                       #+ значения.
{
  shopt -s extglob     #  Включаем расширенную подстановку шаблонов.
  local val=${1##+(0)} #  Используем локальную переменную, для длинных
                       #+ соответствий серий 0-ей.
  shopt -u extglob     #  Выключаем расширенную подстановку шаблонов.
  _strip_leading_zero2=${val:-0}
                       # Если будет введен 0, то вместо 0, возвратится " ".
}
```

Другая иллюстрация использования:

```
echo `basename $PWD`         # Basename текущей рабочей директории.
echo "${PWD##*/}"           # Basename текущей рабочей директории.
echo
echo `basename $0`           # Имя сценария.
echo $0                      # Имя сценария.
echo "${0##*/}"              # Имя сценария.
echo
filename=test.data
echo "${filename##*.}"       # Данные
                             # Расширение filename.
```

${var%Pattern}, ${var%%Pattern}

${var%Pattern} удаляет из $var небольшие совпадающие части $Pattern от окончания $var.

${var%%Pattern} Удаляет из $var большие совпадающие части $Pattern от окончания $var.

В версию 2 Bash добавлены дополнительные опции.

Пример 10-10. Соответствие шаблону при подстановке параметров

```
#!/bin/bash
# patt-matching.sh

#  Соответствие шаблону использующего операторы # ## % %% в подстановке
#+ параметров.

var1=abcd12345abc6789
pattern1=a*c  # * (wild card) соответствие всему между a - c.

echo
echo "var1 = $var1"           # abcd12345abc6789
echo "var1 = ${var1}"         # abcd12345abc6789
                              # (альтернативная форма)
echo "Количество символов в ${var1} = ${#var1}"
echo

echo "pattern1 = $pattern1"   # a*c  (все между 'a' и 'c')
echo "--------------"
echo '${var1#$pattern1}  =' "${var1#$pattern1}"    #         d12345abc6789
# Возможные короткие соответствия, удаляет первые 3 символа  abcd12345abc6789
#                                      ^^^^^^             |-|
echo '${var1##$pattern1} =' "${var1##$pattern1}"   #                 6789
# Возможные длинные соответствия, удаляет первые 12 символов  abcd12345abc6789
#                                      ^^^^^^              |----------|

echo; echo; echo

pattern2=b*9            # Все между 'b' и '9'
echo "var1 = $var1"     # Все еще  abcd12345abc6789
echo
echo "pattern2 = $pattern2"
echo "--------------"
echo '${var1%pattern2}  =' "${var1%$pattern2}"     #     abcd12345a
# Возможные короткие соответствия, удаляет задние 6 символов  abcd12345abc6789
#                                      ^^^^^^                       |----|
echo '${var1%%pattern2} =' "${var1%%$pattern2}"    #     a
# Возможные длинные соответствия, удаляет задние 12 символов  abcd12345abc6789
#                                      ^^^^^^               |-------------|

# Запомните, # и ## работают с левого конца (от начала) строки,
#            % и %% работают с правого конца.

echo

exit 0
```

Пример 10-11. Переименование расширений файлов

```
#!/bin/bash
# rfe.sh: Переименование расширений фалов.
#
#         rfe старое_расширение новое_расширение
#
# Пример:
# Переименовать все файлы *.gif в рабочей директории в *.jpg,
#          rfe gif jpg
```

```
E_BADARGS=65

case $# in
  0|1)             # Вертикальная черта означает, в этом контексте, "ИЛИ".
  echo "Usage: `basename $0` старый_суффикс_файла новый_суффикс_файла"
  exit $E_BADARGS  # Если аргументов 0 или 1, то выручает.
  ;;
esac


for filename in *.$1
# Просматривает список файлов, заканчивая первым аргументом.
do
  mv $filename ${filename%$1}$2
  #  Удаляет часть имени соответствующую первому аргументу,
  #+ затем добавляет второй аргумент.
done

exit 0
```

Расширение переменных/замена *Substring*

Эти конструкции адаптированны из *ksh*.

`${var:pos}` Расширение переменной *`var`*, начиная смещение от *`pos`*.

`${var:pos:len}` Расширение переменной *`var`* до максимального *`len`* символов, смещая от *`pos`*. См. Пример A-13, как пример творческого использования данного оператора.

`${var/Pattern/Replacement}` Первое соответствие *`Pattern`*, в *`var`*, заменяется на *`Replacement`*.

Если *`Replacement`* отсутствует, то первое соответствие *`Pattern`* заменяется *ничем*, т.е. удаляется.

`${var//Pattern/Replacement}` **Глобальная замена.** Все соответствия *`Pattern`* в *`var`* заменяются на *`Replacement`*.

Как и выше, если *`Replacement`* отсутствует, то все совпадения *`Pattern`* заменяются *ничем*, т.е. удаляются.

Пример 10-12. Анализ произвольных строк с помощью соответствия шаблону

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-*}
```

```
echo "var1 ( удаляется все, включая первое соответствие) = $t"
#  t=${var1#*-}  работает так же,
#+ # короткие соответствия строки,
#+ а * соответствие всему предшествующему, включая пустую строку.
# (Спасибо Stephane Chazelas,за разъяснение.)

t=${var1##*-*}
echo "Если var1 содержит \"-\",то возвращается пустая строка...   var1 = $t"


t=${var1%*-*}
echo "var1 ( удаляется все от конца) = $t"

echo

# ----------------------------------------
path_name=/home/bozo/ideas/thoughts.for.today
# ----------------------------------------
echo "path_name = $path_name"
t=${path_name##/*/}
echo "path_name, удаляет префикс (путь к файлу) = $t"
# В данном конкретном случае t=`basename $path_name` имеет такой же эффект.
#  t=${path_name%/}; t=${t##*/}   Это более общее решение,
#+ Но иногда не срабатывает.
#  Если $path_name оканчивается переводом строки, то `basename $path_name` не
#+ сработает, как в приведенное выше выражении.
# (Спасибо S.C.)

t=${path_name%/*.*}
# Такой же эффект как у  t=`dirname $path_name`
echo "path_name, удаляет суффикс (имя файла) = $t"
#  Но в некоторых случаях, таких как "../", "/foo////", # "foo/", "/"
#+ произойдет сбой.
#  Удаление суффикса (имени файла), особенно когда у basename нет суффикса, а
#+ работа производится с dirname, также является сложным вопросом.
# (Спасибо S.C.)

echo

t=${path_name:11}
echo "$path_name, удаляются первые 11 символов = $t"
t=${path_name:11:5}
echo "$path_name, удаляются первые 11 символов,выводятся 5 = $t"

echo

t=${path_name/bozo/clown}
echo "$path_name \"bozo\" заменяется на \"clown\" = $t"
t=${path_name/today/}
echo "$path_name удаляется \"today\" = $t"
t=${path_name//o/O}
echo "$path_name все o становятся заглавными = $t"
t=${path_name//o/}
echo "$path_name все o удаляются = $t"

exit 0
```

${var/#Pattern/Replacement} Если *префикс* `var` соответствует `Pattern`, то `Pattern` заменяется на `Replacement`.

${var/%Pattern/Replacement} Если *суффикс* `var` соответствует `Pattern`, то `Pattern` заменяется на `Replacement`.

Пример 10-13. Соответствие шаблонов префиксу или суффиксу строки

```
#!/bin/bash
# var-match.sh:
# Демонстрация соответствия шаблонов префиксу/суффиксу строки

v0=abc1234zip1234abc    # Оригинальная переменная.
echo "v0 = $v0"         # abc1234zip1234abc
echo

# Соответствие префиксу (началу) строки.
v1=${v0/#abc/ABCDEF}    # abc1234zip1234abc
                        # |-|
echo "v1 = $v1"         # ABCDEF1234zip1234abc
                        # |----|

# Соответствие суффиксу (концу) строки.
v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
                        #              |-|
echo "v2 = $v2"         # abc1234zip1234ABCDEF
                        #               |----|

echo

#  --------------------------------------------------
#  Должны совпадать начало/конец строки, в противном случае не заменяется.
#  --------------------------------------------------
v3=${v0/#123/000}       # Соответствует, но не началу.
echo "v3 = $v3"         # abc1234zip1234abc
                        # НЕ ЗАМЕНЯЕТСЯ.
v4=${v0/%123/000}       # Соответствует, но не окончанию.
echo "v4 = $v4"         # abc1234zip1234abc
                        # НЕ ЗАМЕНЯЕТСЯ.

exit 0
```

${!varprefix*}**, **${!varprefix@} Соответствие имен всех ранее объявленных переменных, начиная с `varprefix`.

```
# Это вариация на косвенную ссылку, но с * или @.
# Эта фича добавлена в Bash, версии 2.04.

xyz23=все_равно
xyz24=

a=${!xyz*}         #  Расширяет *имена* объявленных переменных
# ^ ^   ^           + начинающихся с "xyz".
```

```
echo "a = $a"       #  a = xyz23 xyz24
a=${!xyz@}          #  То же, что и выше.
echo "a = $a"       #  a = xyz23 xyz24

echo "---"

abc23=что_то_другое
b=${!abc*}
echo "b = $b"       #  b = abc23
c=${!b}             #  Теперь более знакомый тип косвенной ссылки.
echo $c             #  что_то_другое
```

Примечания

[1] Если в не интерактивном сценарии *`$parameter`* имеет значение ***null***, он прервется со статусом выхода **`127`** (код ошибки Bash для «command not found»).

Глава 11. Циклы и ветвления

Кому нужно это повторение, женщина?

--Shakespeare, `Othello`

Содержание

11.2. Вложенные циклы
11.3. Управление циклом
11.4. Проверки и ветвления

Операции с блоками кода являются ключом структурированных и упорядоченных сценариев оболочки. Инструментами для того являются конструкции циклов и ветвлений.

11.1. Циклы

Цикл это блок кода, повторяющий [1] список команд до тех пор, пока управляющее условие цикла истинно.

Циклы `for`

`for arg in [list]`

Это одна из основных конструкций цикла. Она значительно отличается от ее аналога в Си.

```
for аргумент in [список]
do
 команда(ы)...
done
```

В процессе каждого прохода цикла, *`аргумент`* последовательно принимает значение каждой переменной в *`списке`*.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# Проход 1 цикла, arg = $var1
# Проход 2 цикла, arg = $var2
# Проход 3 цикла, arg = $var3
# ...
# Проход N цикла, arg = $varN

# Аргументы в [списке] заключаются в кавычки для предотвращения разделения
#+слов.
```

Аргумент *`list`* может содержать символы шаблона постановки

Если ***`do`*** находится на одной строке с ***`for`***, то после ***`list`*** необходима **точка с запятой**.

```
for arg in [list] ; do
```

Пример 11-1. Простые циклы *for*

```
#!/bin/bash
# Список планет.

for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
```

```
  echo $planet  # Каждая планета будет на отдельной строке.
done

echo; echo

for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
    # Все планеты будут на одной строке.
    # Весь «list», заключенный в кавычки, создает одну переменную.
    # Почему? В переменные включены пробелы.
do
  echo $planet
done

echo; echo "Опа! Pluto это не большая планета!"

exit 0
```

Каждый элемент **[list]** может содержать несколько параметров. Это полезно при обработке групп параметров. В таких случаях с помощью команды ***set*** (см. Пример 15-16) принудительно анализируется каждый элемент **[list]** и каждый компонент объявляется позиционным параметром.

Пример 11-2. Цикл *for* с двумя параметрами в каждом элементе [list]

```
#!/bin/bash
# Снова планеты.

# Связь имени каждой планеты с расстоянием от Солнца.

for planet in "Mercury 36" "Venus 67" "Earth 93"  "Mars 142" "Jupiter 483"
do
  set -- $planet  #  Анализ переменной "planet"
                  #+ и объявление позиционных параметров.
  #  "--" предотвращает неприятные сюрпризы, если $planet равно null или
  #+ начинается с тире.

  #  Возможно, потребуется сохранить оригинальные позиционные параметры,
  #+ так как они будут перезаписываться.
  #  Одним из способов является использование массива,
  #         original_params=("$@")

  echo "$1              $2,000,000 миль от Солнца"
  #----две  табуляции---добавляют нули в параметр $2
done

# (Спасибо S.C., за дополнительные разъяснения.)

exit 0
```

В цикле ***for*** в ***[list]*** может использоваться переменная .

Пример 11-3. *Fileinfo:* обработка списка файлов, находящегося в переменной

```
#!/bin/bash
# fileinfo.sh

FILES="/usr/sbin/accept
/usr/sbin/pwck
/usr/sbin/chroot
/usr/bin/fakefile
/sbin/badblocks
/sbin/ypbind"     # Список интересующих файлов.
                  # Помещаем пустой файл, /usr/bin/fakefile.

echo

for file in $FILES
do

  if [ ! -e "$file" ]       # Проверяем существование файла.
  then
    echo "$file не существует."; echo
    continue                # переход к следующему.
   fi

  ls -l $file | awk '{ print $8 "       размер файла: " $5 }'  # Вывод 2 полей.
  whatis `basename $file`   # Файловая информация.
  # Обратите внимание, что для этого необходимо установить базу данных Whatis.
  # Что бы это сделать, запустите /usr/bin/makewhatis из-под root.
  echo
done

exit 0
```

`[list]`, в цикле ***for***, может быть параметризирован.

Пример 11-4. Обработка параметризированного `list` файла

```
#!/bin/bash

filename="*txt"

for file in $filename
do
 echo "Содержимое $file"
 echo "---"
 cat "$file"
 echo
done
```

Если **`[list]`** в цикле ***for*** содержит символы шаблонов подстановки (***** и **?**) использующихся при расширении имени файла, то происходит подстановка.

Пример 11-5. Обработка файлов циклом *for*

```
#!/bin/bash
```

```
# list-glob.sh: Создание [list] в цикле, с помощью "globbing" ...
# Globbing = расширяемое имя файла.

echo

for file in *
#           ^  Bash выполняет расширение имени файла выражения,
#+             которое признает шаблон подстановки.
do
  ls -l "$file"  # Список всех файлов в $PWD (в текущей директории).
  #  Напомним, что символу шаблона подстановки "*" соответствует любое имя,
  #+ однако, в 'шаблоне подстановки' (globbing), ему не соответствуют файлы
  #+ начинающиеся с точки.
  #  Если шаблон не соответствует файлу, он расширяется в себя (т.е. в *).
  #  Перед этим установите опцию nullglob
  #+   (shopt -s nullglob).
  #  Спасибо S.C.
done

echo; echo

for file in [jx]*
do
  rm -f $file    # В $PWD удаляются только файлы начинающиеся с "j" или "x".
  echo "Удаленные файлы \"$file\"".
done

echo

exit 0
```

Отсутствие части **`in [list]`** в цикле ***for*** заставляет цикл оперировать ***$@*** – позиционными параметрами. Очень хорошей иллюстрацией этого является Пример A-15. См. также Пример 15-17.

Пример 11-6. Отсутствие `in [list]` в цикле *for*

```
#!/bin/bash

#  Вызовите этот сценарий с/без аргументов
#+ и посмотрите, что получится.

for a
do
 echo -n "$a "
done

#  'in list' отсутствует, поэтому цикл оперирует '$@'
#+ (аргументом списка командной строки, включая пробелы).

echo

exit 0
```

Для создания **[list]** в цикле ***for*** позволяется использовать подстановку команд. См. также Пример 16-54, Пример 11-11 и Пример 16-48.

Пример 11-7. Создание [list] в цикле *for* командой подстановки

```
#!/bin/bash
#  for-loopcmd.sh: Создание [list] цикла for
#+ командой подстановки.

NUMBERS="9 7 3 8 37.53"

for number in `echo $NUMBERS`  # for number in 9 7 3 8 37.53
do
  echo -n "$number "
done

echo
exit 0
```

Вот более сложный пример использования команды подстановки для создания **[list]**.

Пример 11-8. Замена двоичных файлов *grep*

```
#!/bin/bash
# bin-grep.sh: Поиск совпадающих строк в двоичном файле.

# "grep" заменяет двоичные файлы.
# Эффект подобен "grep -a"

E_BADARGS=65
E_NOFILE=66

if [ $# -ne 2 ]
then
  echo "Usage: `basename $0` искомая_строка файла"
  exit $E_BADARGS
fi

if [ ! -f "$2" ]
then
  echo "Файл \"$2\" не существует."
  exit $E_NOFILE
fi


IFS=$'\012'       # Предложено Anton Filippov.
                  # было:  IFS="\n"
for word in $( strings "$2" | grep "$1" )
# Команда "strings" выводит список строк двоичных файлов.
# Вывод конвейером передается "grep", которая сравнивает необходимые строки.
do
  echo $word
done

# Как пояснил S.C., строки 23 - 30 могут быть заменены облегченной
```

```
#    строкой "$2" | grep "$1" | tr -s "$IFS" '[\n*]'

#  Попробуйте что-то вроде  "./bin-grep.sh mem /bin/ls"
#+ для воплощения этого сценария.

exit 0
```

Больше из того же.

Пример 11-9. Список всех системных пользователей

```
#!/bin/bash
# userlist.sh

PASSWORD_FILE=/etc/passwd
n=1           # Номер пользователя

for name in $(awk 'BEGIN{FS=":"}{print $1}' < "$PASSWORD_FILE" )
# Разделитель полей = :  ^^^^^^
# Вывод первого поля             ^^^^^^^^
# Считываются данные из файла паролей /etc/passwd  ^^^^^^^^^^^^^^^^
do
  echo "USER #$n = $name"
  let "n += 1"
done


# USER #1 = root
# USER #2 = bin
# USER #3 = daemon
# ...
# USER #33 = bozo

exit $?

#  Обсуждение:
#  ----------
#  Как обычный пользователь может прочитать /etc/passwd или запустить сценарий?
#+ (Подсказка:  Проверьте права доступа к /etc/passwd.)
#  Это дыра в безопасности? Почему да, или почему нет?
```

Еще один пример результата подстановки команд **`[list]`**.

Пример 11-10. Проверка авторства всех бинарных файлов в директории

```
#!/bin/bash
# findstring.sh:
# Поиск заданной строки в двоичных файлах заданного каталога.

directory=/usr/bin/
fstring="Free Software Foundation"  # Смотрим, какие файлы из FSF.

for file in $( find $directory -type f -name '*' | sort )
```

```
do
  strings -f $file | grep "$fstring" | sed -e "s%$directory%%"
  #  В выражении "sed"
  #+ нужно разделять нормальным "/" разделителем,
  #+ потому что "/" может быть одним из отфильтровываемых символов.
  #  Невыполнение этого выдает сообщение об ошибке. (Попробуй.)
done

exit $?

#  Упражнение (легкое):
#  --------------
#  Преобразуйте этот сценарий для принятия параметра из командной строки
#+  $directory и $fstring.
```

Последний пример подстановки команд **[list]**, но теперь "команда" это функция.

```
generate_list ()
{
  echo "one two three"
}

for word in $(generate_list)  # Пусть «word» захватывает вывод функции.
do
  echo "$word"
done

# one
# two
# three
```

Вывод цикла ***for*** может быть передан посредством конвейера команде или командам.

Пример 11-11. Список символических ссылок в директории

```
#!/bin/bash
# symlinks.sh:  Список символических ссылок в директории


directory=${1-`pwd`}
#  По умолчанию текущая рабочая директория,
#+ если не указывается другая.
#  Эквивалентно блоку кода ниже.
# ----------------------------------------------------------
# ARGS=1                # Ожидается один аргумент из командной строки.
#
# if [ $# -ne "$ARGS" ]  # Если аргумент не 1...
# then
#   directory=`pwd`      # текущая рабочая директория
# else
#   directory=$1
```

```
# fi
# ------------------------------------------------------------

echo "Символические ссылки в директории \"$directory\""

for file in "$( find $directory -type l )"   # -type l = символическая ссылка
do
  echo "$file"
done | sort                                  #  В ином случае список файлов
                                              #+ не сортируется.
#  Строго говоря цикл не является действительно необходимым,
#+ поскольку выходные данные команды «find» расширяются в одно слово.
#  Однако это легко понять и, таким образом, проиллюстрировать.

#  Как пояснил Dominik 'Aeneas' Schnitzer,
#+ ошибочное не заключение в кавычки $( find $directory -type l )
#+ завалит именами файлов имеющих пробелы.

exit 0


# ------------------------------------------------------------
# Jean Helou предлагает следующую альтернативу:

echo "символические ссылки в директории \"$directory\""
# Архивируем текущий IFS.
OLDIFS=$IFS
IFS=:

for file in $(find $directory -type l -printf "%p$IFS")
do     #                              ^^^^^^^^^^^^^^^^
       echo "$file"
       done|sort

# A, James "Mike" Conley изменил код Helou:

OLDIFS=$IFS
IFS='' # IFS в значении null означает «без переноса слов»
for file in $( find $directory -type l )
do
  echo $file
  done | sort

#  Это работает в случае «патологических» имен директорий, имеющих
#+ встроенные двоеточия.
#  «Также исправляет случай патологического имени директории, также имеющей в
#+ имени двоеточие (или пробелы, как в предыдущем примере).»
```

`stdout` цикла может быть перенаправлен в файл, как продемонстрировано в следующем примере.

Пример 11-12. Символические ссылки директории, сохраняемые в файл

```
#!/bin/bash
# symlinks.sh: Список символических ссылок директории.

OUTFILE=symlinks.list                         # сохраняемый файл
```

```
directory=${1-`pwd`}
#  По умолчанию - текущая рабочая директория,
#+ если не указана другая.


echo "Символические ссылки директории \"$directory\"" > "$OUTFILE"
echo "---------------------------" >> "$OUTFILE"

for file in "$( find $directory -type l )"    # -type l = символические ссылки
do
  echo "$file"
done | sort >> "$OUTFILE"                     # stdout цикла перенаправляется
#           ^^^^^^^^^^^^^                       в сохраняемый файл.

# echo "Выходной файл = $OUTFILE"

exit $?
```

Существует альтернативный синтаксис для цикла for, который будет выглядеть очень знакомым для программистов на Си. Для этого потребуются *двойные скобки*.

Пример 11-13. Цикл *for* в стиле Си

```
#!/bin/bash
# Несколько способов подсчета до 10.

echo

# Обычный синтаксис.
for a in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$a "
done

echo; echo

# +=========================================+

# С помощью "seq" ...
for a in `seq 10`
do
  echo -n "$a "
done

echo; echo

# +=========================================+

# С помощью скобок расширения ...
# Bash, version 3+.
for a in {1..10}
do
  echo -n "$a "
done
```

```
echo; echo

# +==========================================+

# Теперь сделаем то же самое, используя синтаксис Си.

LIMIT=10

for ((a=1; a <= LIMIT ; a++))  # Двойные скобки и голый "LIMIT"
do
  echo -n "$a "
done                           # Конструкция заимствована из ksh93.

echo; echo

# +========================================================================+

#  Давайте используем «оператор запятая» Си, для приращения двух
#+ переменных  одновременно.

for ((a=1, b=1; a <= LIMIT ; a++, b++))
do  # Запятая объединяет операции.
  echo -n "$a-$b "
done

echo; echo

exit 0
```

См. также Пример 27-16, Пример 27-17 и Пример A-6.

Теперь, цикл *for* используемый в "реальной" жизни.

Пример 11-14. Использование *efax* в пакетном режиме

```
#!/bin/bash
# Факс (должен быть установлен пакет 'efax').

EXPECTED_ARGS=2
E_BADARGS=85
MODEM_PORT="/dev/ttyS2"   # На Вашей машине может отличаться.
#                 ^^^^^      По умолчанию, порт карты модема PCMCIA.

if [ $# -ne $EXPECTED_ARGS ]
# Проверка правильного количества аргументов командной строки.
then
   echo "Usage: `basename $0` телефон# текстовый файл"
   exit $E_BADARGS
fi


if [ ! -f "$2" ]
then
  echo "Файл $2 не является текстовым."
  #     Не обыкновенный файл, или не существует.
  exit $E_BADARGS
```

```
fi


fax make $2              #  Создаем из текстовых файлов файлы в формате fax.

for file in $(ls $2.0*)  #  Объединяем преобразованные файлы.
                         #  Используется символ подстановки ("шаблон
                         #+ подстановки" имени файла)
                         #+ в списке переменных.
do
  fil="$fil $file"
done

efax -d "$MODEM_PORT"  -t "T$1" $fil   # Наконец работа сделана.
# Попробуйте добавить -o1, если строка выше не заработает.


#  Как пояснил S.C., цикл for может быть убран с помощью
#     efax -d /dev/ttyS2 -o1 -t "T$1" $2.0*
#+ но в этом случае не будет так поучительно [оскал].

exit $?   # Кроме того, efax посылает диагностические сообщения в stdout.
```

Ключевые слова **do** и **done** ограничивают командной блок цикла ***for***. Однако в определенных контекстах, они могут быть опущены, путем обрамления командного блока ***фигурными скобками***.

```
for((n=1; n<=10; n++))
# Отсутствует do!
{
  echo -n "* $n *"
}
# done отсутствует!


# Выводится:
# * 1 ** 2 ** 3 ** 4 ** 5 ** 6 ** 7 ** 8 ** 9 ** 10 *
# А, echo $? возвратило 0, поэтому Bash не увидел ошибки.

echo

#  Но обратите внимание, что в классическом цикле for:
#+ for n in [list] ...
#+ необходима заключающая точка с запятой.

for n in 1 2 3
{  echo -n "$n "; }
#              ^

# Спасибо за это указание YongYe.
```

while

Эта конструкция проверяет условие в верхней части цикла, и продолжает цикл, до тех

пор, пока условие истинно (возвращает статус выхода 0). В отличие от цикла ***for***, цикл ***while*** находит применение в тех случаях, когда количество итераций (повторений цикла) *заранее не известно*.

```
while [ условие ]
do
 команда(ы)...
done
```

Конструкция прямых скобок в цикле ***while***, это не более чем наш старый друг, скобки проверки **test**, используемые в проверке ***if/then***. На самом деле, цикл ***while*** может использовать конструкцию более универсальных **двойных скобок** (*while [[условие]]*).

Как и в случае с циклом ***for***, при размещении ***do*** на той же строке цикла, что и *условия*, необходима *точка с запятой*.

```
while [ условие ];do
```

Обратите внимание, что *скобки проверки* не являются *обязательными* в цикле ***while***. См., например, конструкцию getopts.

Пример 11-15. Простой цикл *while*

```
#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
#      ^                    ^
# Пробелы, потому что это "скобки проверки" ...
do
  echo -n "$var0 "        # -n подавляет перевод строки.
  #             ^          Пробелы, для разделения выводимых чисел.

  var0=`expr $var0 + 1`   # var0=$(($var0+1)) то же работает.
                          # var0=$((var0 + 1)) то же работает.
                          # let "var0 += 1"    то же работает.
done                      # Другие способы то же будут работать.

echo

exit 0
```

Пример 11-16. Другой цикл *while*

```
#!/bin/bash

echo
                              # Эквивалентно:
while [ "$var1" != "end" ]    # while test "$var1" != "end"
```

```
do
  echo "Введите переменную #1 (end - выход) "
  read var1                     # Не читается 'read $var1' (почему?).
  echo "Переменная #1 = $var1"   # Потому что нужны кавычки "#" ...
  # Если ввели 'end', оно выведется на экран.
  # Т.к. не будет проверяться условие завершения верхней частью цикла.
  echo
done

exit 0
```

Цикл ***while*** может иметь несколько условий. Завершение цикла определяет только *конечное* условие. Для этого потребуется немного отличающийся синтаксис цикла.

Пример 11-17. Цикл *while* с несколькими условиями

```
#!/bin/bash

var1=unset
previous=$var1

while echo "начальная переменная = $previous"
      echo
      previous=$var1
      [ "$var1" != end ] # Проверка, что бы $var1 была начальной.
# В *while * имеются четыре условия, но только последнее управляет циклом.
# Вычисляется *последний* статус выхода.
do
  echo "Введите переменную #1 (end - выход) "
  read var1
  echo "variable #1 = $var1"
done

# Попытайтесь выяснить, как это работает.
# Это чуть-чуть сложнее.

exit 0
```

Как в цикле ***for***, так и в цикле ***while*** может использоваться синтаксис Си, использующий конструкцию двойных круглых скобок (см. также Пример 8-5).

Пример 11-18. Синтаксис Си в цикле *while*

```
#!/bin/bash
# wh-loopc.sh: Счет до 10 циклом "while".

LIMIT=10                 # 10 проходов (итераций).
a=1

while [ "$a" -le $LIMIT ]
do
  echo -n "$a "
  let "a+=1"
```

```
done                    # Пока никаких сюрпризов.

echo; echo

# +==========================================================+

# Теперь повторим с помощью синтаксиса Си.

((a = 1))      # a=1
# Двойные скобки допускают пробелы при задании переменной, как в Си.

while (( a <= LIMIT ))   #  Двойные скобки,
do                       #+ а "$" перед переменной не нужно.
  echo -n "$a "
  ((a += 1))             # Можно "a+=1"
  # Конечно да.
  # Двойные скобки синтаксиса Си допускают приращение переменной.
done

echo

# Программисты на Си и Java могут чувствовать себя в Bash как дома.

exit 0
```

В скобках проверки, цикл ***while*** может вызывать функцию.

```
t=0

условие ()
{
  ((t++))

  if [ $t -lt 5 ]
  then
    return 0  # истинно
  else
    return 1  # ложно
  fi
}

while условие
#     ^^^^^^^^^
#     Функция вызывает — четыре повторения цикла.
do
  echo "Продолжается: t = $t"
done

# Продолжается: t = 1
# Продолжается: t = 2
# Продолжается: t = 3
# Продолжается: t = 4
```

Подобно конструкции ***if-test***, цикл ***while*** может опускать скобки проверки.

```
while условие
do
   команда(ы) ...
done
```

Путем принудительного соединения команды ***read*** с циклом ***while***, мы получим удобную конструкцию **while read**, полезную для чтения и анализа файлов.

```
cat $filename |     # Ввод из файла.
while read строка   # До тех пор, пока есть строки для чтения...
do
  ...
done

# ========== Пример фрагмента сценария "sd.sh" ========== #

  while read value   # Читается один пункт данных за раз.
  do
    rt=$(echo "scale=$SC; $rt + $value" | bc)
    (( ct++ ))
  done

  am=$(echo "scale=$SC; $rt / $ct" | bc)

  echo $am; return $ct   # Данная функция "возвращает" ДВА значения!
  #  Внимание: Этот трюк не сработает если $ct > 255!
  #  Что бы обрабатывать большее число пунктов данных,
  #+ просто закомментируйте «return $ct» выше.
} <"$datafile"   # Направление из файла данных.
```

Цикл ***while*** может в конце перенаправлять `stdin` в файл **<** .

Цикл ***while*** может передавать `stdin` конвейеру.

until

Эта конструкция проверяет состояние верхней части цикла и продолжает цикл до тех пор, пока это условие имеет значение ***ложно*** (в отличии от цикла ***while***).

```
until [ истинное_условие ]
do
 команда(ы)...
done
```

Обратите внимание, что цикл проверки ***until***, проверяет условие в верхней части цикла перед каждым проходом, в отличии от аналогичных конструкций в некоторых языках программирования.

Как и в случае с циклом ***for***, размещение ***do*** на той же строке, что и условия проверки, требует точки с запятой.

```
until [ истинное_условие ]; do
```

Пример 11-19. Цикл *until*

```
#!/bin/bash

END_CONDITION=end

until [ "$var1" = "$END_CONDITION" ]
# Здесь, в верхней части цикла, находится условие проверки.
do
  echo "Введите переменную #1 "
  echo "($END_CONDITION для выхода)"
  read var1
  echo "variable #1 = $var1"
  echo
done

#                     ---                         #

#  Как и с циклами "for" и "while",
#+ цикл "until" допускает конструкции проверки Си.

LIMIT=10
var=0

until (( var > LIMIT ))
do  # ^^ ^     ^     ^^   Без прямых скобок, без префикса переменной $.
  echo -n "$var "
  (( var++ ))
done    # 0 1 2 3 4 5 6 7 8 9 10


exit 0
```

Как выбрать между циклами ***for***, ***while*** и ***until***? В *Cu*, как правило, цикл ***for*** используется когда заранее известно число повторений. В Bash, ситуация неоднозначна. В Bash цикл ***for*** более свободно структурирован и более гибок, чем его эквивалент в других языках. Поэтому, не стесняйтесь использовать любой тип цикла, позволяющего выполнить работу более простейшим образом.

Примечания

[1] *Итерация*: Повторное выполнение команды или группы команд, обычно - но не всегда, пока заданное условие не выполнилось, или до выполнения заданного условия.

11.2. Вложенные циклы

Вложенный цикл это цикл в цикле, внутренний цикл в теле внешнего цикла. Как это работает: первый проход внешнего цикла вызывает внутренний цикл, выполняемый до завершения. Затем второй проход внешнего цикла снова вызывает внутренний цикл. Это повторяется до тех пор, пока внешний цикл не завершится. Конечно, прерывание любого - внутреннего или внешнего цикла, прерывает процесс.

Пример 11-20. Вложенный цикл

```
#!/bin/bash
# nested-loop.sh: Вложенные циклы "for".

outer=1             # Установка счетчика внешнего цикла.

# Запуск внешнего цикла.
for a in 1 2 3 4 5
do
  echo "Передача $outer внешнего цикла."
  echo "---------------------"
  inner=1           # Установка счетчика внутреннего цикла.

  # ===============================================
  # Запуск внутреннего цикла.
  for b in 1 2 3 4 5
  do
    echo "Передача $inner внутреннего цикла."
    let "inner+=1"  # Увеличение счетчика внутреннего цикла.
  done
  # Окончание внутреннего цикла.
  # ===============================================

  let "outer+=1"    # Увеличение счетчика внешнего цикла.
  echo              # Пробелы между выходными блоками в проходах
#+ внешнего цикла.
done
# Окончание внешнего цикла.

exit 0
```

См. Пример 27-11 для иллюстрации вложенных циклов ***while***, а Пример 27-13 показывает цикл ***while***, вложенный в цикл ***until***.

11.3. Управление циклом

Tournez cent tours, tournez mille tours,

Tournez souvent et tournez toujours . . .

--Verlaine, "Chevaux de bois"

Команды влияющие на поведение цикла

`break`, `continue`

Команды управления циклом **`break`** и **`continue`** [1] точно соответствуют их аналогам в других языках программирования. Команда ***`break`*** завершает цикл (прерывает его), а ***`continue`*** передает управление следующей *итерации* цикла, пропуская все остальные команды в этом конкретном цикле.

Пример 11-21. Эффекты *break* и *continue* в цикле

```
#!/bin/bash

LIMIT=19  # Верхний предел

echo
echo "Вывод чисел от 1 до 20 (но не 3 и 11)."

a=0

while [ $a -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # Исключает 3 и 11.
 then
   continue      # Пропуск остальных итераций этого конкретного цикла.
 fi

 echo -n "$a "   # Это не будет выполняться для 3 и 11.
done

# Упражнение:
# Почему цикл выводит числа до 20?

echo; echo

echo Вывод чисел от 1 до 20, но что-то происходит после вывода 2.

##########################################################################

# Тот же цикл, но заменим 'continue' на 'break'.

a=0

while [ "$a" -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -gt 2 ]
 then
   break  # Прерываем всю остальную часть цикла.
 fi

 echo -n "$a "
done

echo; echo; echo
```

```
exit 0
```

При необходимости команда ***break*** может принимать параметр. Обычно ***break*** завершает только внутренний цикл, в котором она находится, а ***break N*** прерывает *N* уровней цикла.

Пример 11-22. Прерывание нескольких уровней цикла

```
#!/bin/bash
# break-levels.sh: Прерывание циклов.

# "break N" прерывает N уровней циклов.

for outerloop in 1 2 3 4 5
do
  echo -n "Group $outerloop:   "

  # ----------------------------------------------------------
  for innerloop in 1 2 3 4 5
  do
    echo -n "$innerloop "

    if [ "$innerloop" -eq 3 ]
    then
      break  # Попробуем break 2, чтобы увидеть, что случиться.
             # («Прервутся» внешний и внутренний циклы.  )
    fi
  done
  # ----------------------------------------------------------

  echo
done

echo

exit 0
```

Команда ***continue***, подобно ***break***, может принимать параметр. Простое ***continue*** удаляет короткие текущие итерации своего цикла и начинает следующие. ***continue N*** завершает все оставшиеся итерации на своем уровне цикла и передает управление следующей итерации в цикле, находящимся на *N* уровней выше.

Пример 11-23. Продолжение более высокого уровня цикла

```
#!/bin/bash
# Команда "continue N" передает работу N-ому уровню цикла.

for outer in I II III IV V           # внешний цикл
do
  echo; echo -n "Group $outer: "

  # --------------------------------------------------------------------
  for inner in 1 2 3 4 5 6 7 8 9 10  # внутренний цикл
```

```
  do

    if [[ "$inner" -eq 7 && "$outer" = "III" ]]
    then
      continue 2  # Передача управления циклу 2-го уровня, "внешнему циклу".
                  # Замените строку выше простым 'continue',
                  # чтобы увидеть нормальное поведение цикла.
    fi

    echo -n "$inner "  # 7 8 9 10 не вывело "Group III."
  done
  # ------------------------------------------------------------------

done

echo; echo

# Упражнение:
# Придумайте практическое использование для 'continue N' в сценарии.

exit 0
```

Пример 11-24. Использование *continue N* в реальных задачах

```
# Albert Reiner предоставил пример использования "continue N":
# ------------------------------------------------------------

#  Предположим, необходимо запустить большое количество задач, с любыми
#+ данными, которые должны быть обработаны в файлах, заданных по шаблону имени,
#+ в директории. Есть несколько машин, которые имеют доступ к этой директории,
#+ и я хочу распределить задачи между этими разными "ящиками".
#  Поэтому, обычно, для передачи следующей задачи, ожидается завершение задачи
#+ в каждом ящике:

while true
do
  for n in .iso.*
  do
    [ "$n" = ".iso.opts" ] && continue
    beta=${n#.iso.}
    [ -r .Iso.$beta ] && continue
    [ -r .lock.$beta ] && sleep 10 && continue
    lockfile -r0 .lock.$beta || continue
    echo -n "$beta: " `date`
    run-isotherm $beta
    date
    ls -alF .Iso.$beta
    [ -r .Iso.$beta ] && rm -f .lock.$beta
    continue 2
  done
  break
done

exit 0

#  Детали, в частности sleep N, характерны для моего приложения,
#+ но общей картиной является:
```

```
while true
do
  for задание in {шаблон}
  do
    {задача уже выполнена или выполняется} && continue
    {отмечает задачу как выполняемую, выполняет задачу, отмечает задачу как
     выполненную}
    continue 2
  done
  break        # Или что-то вроде `sleep 600', чтобы избежать прекращения.
done

#  Таким образом сценарий остановится только тогда, когда задачи
#+ выполнятся (включая задачи, которые были добавлены во время
#+ выполнения). С помощью соответствующих lockfiles они могут выполняться
#+ на нескольких машинах одновременно, без дублирования вычислений,
#+ [запущенных на пару часов в моем случае, поэтому я хочу избежать этого].
#+ Кроме того, так как поиск каждый раз начинается с самого начала, сразу
#+ можно кодировать приоритеты в именах файлов. Конечно, это можно было бы
#+ сделать и без «continue 2», но тогда пришлось бы  проверять выполнение
#+ некоторых задач (и необходимо было бы немедленно искать следующую
#+ задачу), или нет (в этом случае мы прервались бы или уснули на долгое
#+ время до проверки для новой задачи).
```

Конструкцию **`continue N`** трудно понять и сложно использовать в любом значимом контексте. Ее, вероятно, лучше избегать.

Примечания

[1] Это встроенные функции оболочки, в то время как другие команды цикла, такие как **`while`** и **`case`** - являются ключевыми словами (***keywords***).

11.4. Проверки и ветвления

Конструкции **`case`** и **`select`** технически не являются циклами, поскольку они не производят повторений выполнения блока кода. Однако они, как циклы, направляют поток программы согласно условий в верхней или нижней части блока.

Управление потоком программы в блоке кода

`case (in)/esac`

Конструкция **`case`** является аналогом оболочки сценариев конструкции *`switch`* в Си/C++. Она позволяет, ветвление единственным блоком кода, в зависимости от условий проверки. Она служит, своего рода, сокращением для нескольких операторов ***if/then/else*** и соответствующим инструментом для создания меню.

```
case "$variable" in

"$условие1" )
команда...
;;

"$условие2" )
команда...
;;

esac
```

- Заключение переменных в кавычки не является обязательным, поскольку разделения слов не происходит.
- Каждая строка проверки завершается *правой скобкой* **)**. [1]
- Каждый блок условий заканчивается *двойной точкой с запятой* **;;**.
- Если условие проверки *истинно*, то выполняются соответствующие команды и блок **case** завершается.
- Весь блок **case** заканчивается **esac** (***case*** наоборот).

Пример 11-25. Применение *case*

```
#!/bin/bash
# Проверка диапазонов символов.

echo; echo "Нажмите клавишу, а затем нажмите возврат."
read Keypress

case "$Keypress" in
  [[:lower:]]   ) echo "Строчная буква";;
  [[:upper:]]   ) echo "Прописная буква";;
  [0-9]         ) echo "Цифра";;
  *             ) echo "Пунктуация, пробелы, или иное";;
esac      #  Допускаются диапазоны символов в [квадратных скобках], или
          #+ диапазоны POSIX в [[ двойных квадратных скобках ]].

#  В начальной версии этого примера,
#+ проверка строчных и прописных символов выглядела так
#+ [a-z] и [A-Z].
#  Но это не всегда работает в некоторых локалях и/или дистрибутивах Linux.
#  POSIX более портабелен (платформенно переносим).
#  Спасибо Frank Wang за указание на это.

#  Упражнение:
#  --------
#  При остановке сценария, он принимает одно нажатие клавиши, а затем
```

```
#+ прекращается.
#  Измените сценарий так, чтобы он принимал повторный ввод,
#+ сообщал о каждом нажатии клавиши и прекращался только при нажатии «X».
#  Подсказка: заключите все в цикл "while".

exit 0
```

Пример 11-26. Создание меню с помощью *case*

```
#!/bin/bash

# Черновая адресная база данных

clear # Очистка экрана.

echo "          Список контактов"
echo "          ------- ----"
echo "Выберите одно из следующих лиц:"
echo
echo "[E]vans, Roland"
echo "[J]ones, Mildred"
echo "[S]mith, Julie"
echo "[Z]ane, Morris"
echo

read person

case "$person" in
# Обратите внимание – переменная ***заключена*** *в кавычки*.

  "E" | "e" )
  # Доступен ввод в верхнем или нижнем регистре.
  echo
  echo "Roland Evans"
  echo "4321 Flash Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;
# Обратите внимание, что каждую опцию завершает двойная точка с запятой.

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Ex-girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Добавьте записи для Smith & Zane later.

          * )
```

```
   # Опция по умолчанию.
   # Здесь возможен пустой ввод (нажатие возврата).
   echo
   echo "Пока в базе нет данных."
  ;;

esac

echo

#  Упражнение:
#  --------
#  Измените сценарий так, что бы он принимал очередной ввод,
#+ вместо прекращения, после вывода единственного адреса.

exit 0
```

Использование ***case*** может включать в себя проверку параметров командной строки.

```
#! /bin/bash

case "$1" in
  "") echo "Usage: ${0##*/} <filename>"; exit $E_PARAM;;
                      # Параметры командной строки отсутствуют,
                      # или первый параметр пустой.
# Обратите внимание, что ${0##*/} это  подстановка параметра ${var##шаблон}.
                      # Конечным результатом является $0.

  -*) FILENAME=./$1;;   #  Если filename, переданное как аргумент ($1),
                        #+ начинается с тире,
                        #+ то заменяет его на ./$1
                      #+ что бы дальнейшие команды не интерпретировали
                      #+ его в качестве опции.

  * ) FILENAME=$1;;   # В противном случае, $1.
esac
```

Вот более простой пример обработки параметров командной строки:

```
#! /bin/bash

while [ $# -gt 0 ]; do    # Пока не закончатся параметры ...
  case "$1" in
    -d|--debug)
              # параметр "-d" или "--debug"?
              DEBUG=1
              ;;
    -c|--conf)
              CONFFILE="$2"
              shift
              if [ ! -f $CONFFILE ]; then
                echo "Ошибка: Этот файл не существует!"
                exit $E_CONFFILE     # Файл не найден, ошибка.
              fi
              ;;
  esac
  shift       # Проверка следующих установленных параметров.
done
```

```
#  Из сценария Stefano Falsetto's "Log2Rot",
#+ часть его пакета "rottlog".
#  Используется с разрешения.
```

Пример 11-27. Использование *подстановки команд* для создания переменной *case*

```
#!/bin/bash
# case-cmd.sh: Использование подстановки команд для создания переменной case

case $( arch ) in   # $( arch ) Возвращает архитектуру машины.
                    # Эквивалентно 'uname -m' ...
  i386 ) echo "80386-based machine";;
  i486 ) echo "80486-based machine";;
  i586 ) echo "Pentium-based machine";;
  i686 ) echo "Pentium2+-based machine";;
  *    ) echo "Other type of machine";;
esac

exit 0
```

Конструкция ***case*** может фильтровать шаблоны подстановки строки

Пример 11-28. Простой поиск совпадающих строк

```
#!/bin/bash
# match-string.sh: Простой поиск совпадающих строк
#                  с помощью конструкции 'case'.

match_string ()
{ # Извлечение совпадающих строк.
  MATCH=0
  E_NOMATCH=90
  PARAMS=2     # Функции необходимы 2 аргумента.
  E_BAD_PARAMS=91

  [ $# -eq $PARAMS ] || return $E_BAD_PARAMS

  case "$1" in
  "$2") return $MATCH;;
  *   ) return $E_NOMATCH;;
  esac

}


a=one
b=two
c=three
d=two


match_string $a     # неверное количество параметров
echo $?             # 91
```

```
match_string $a $b  # нет совпадения
echo $?             # 90

match_string $b $d  # совпадение
echo $?             # 0


exit 0
```

Пример 11-29. Проверка вводимых букв

```
#!/bin/bash
# isalpha.sh: Фильтрование строк с помощью структуры "case".

SUCCESS=0
FAILURE=1   #  Было FAILURE=-1, но Bash больше не поддерживает
            #+ отрицательные возвращаемые значения.

isalpha ()  # Проверка в веденной строке буквенный ли *первый символ*.
{
if [ -z "$1" ]               # Нет переданных аргументов?
then
  return $FAILURE
fi

case "$1" in
  [a-zA-Z]*) return $SUCCESS;;  # Начинается с буквы?
  *        ) return $FAILURE;;
esac
}             # Сравните это с функцией"isalpha ()" в Си.


isalpha2 ()   # Проверка *вся ли строка* состоит из букв.
{
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
               *) return $SUCCESS;;
  esac
}

isdigit ()    # Проверка *вся ли строка* состоит из цифр.
{             # Иными словами, проверка целочисленности переменной.
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
    *[!0-9]*|"") return $FAILURE;;
               *) return $SUCCESS;;
  esac
}



check_var ()  # Интерфейс isalpha ().
{
if isalpha "$@"
then
```

```
  echo "\"$*\" начинается с буквенного символа."
  if isalpha2 "$@"
  then        # Нет смысла в проверке, если первый символ не является буквой.
    echo "\"$*\" содержит только буквенные символы."
  else
    echo "\"$*\" содержит, по крайней мере, один не буквенный символ."
  fi
else
  echo "\"$*\" начинается с не буквенного символа."
              # Если нет переданных аргументов, то так же "не буква".
fi

echo

}

digit_check ()  # Интерфейс isdigit ().
{
if isdigit "$@"
then
  echo "\"$*\" содержит только цифры [0 - 9]."
else
  echo "\"$*\" содержит по крайней мере один не цифровой символ."
fi

echo

}

a=23skidoo
b=H3llo
c=-What?
d=What?
e=$(echo $b)   # Подстановка команды.
f=AbcDef
g=27234
h=27a34
i=27.34

check_var $a
check_var $b
check_var $c
check_var $d
check_var $e
check_var $f
check_var     # Аргумент не передан, что произойдет?
#
digit_check $g
digit_check $h
digit_check $i


exit 0        # Сценарий улучшен S.C.

# Упражнение:
# --------
#  Напишите функцию 'isfloat ()' для проверки цифр с плавающей точкой.
#  Подсказка: Функция повторяет 'isdigit ()',
#+ но добавьте обязательную проверку наличия десятичной точки.
```

select

Конструкция ***select***, заимствованая из Korn Shell, и является еще одним инструментом для создания меню.

```
select переменная [in list]
do
 команда...
 break
done
```

Пользователю предлагается ввести один из вариантов из списка переменной. Обратите внимание, что по умолчанию, вместо $PS3, ***select*** использует приглашение **(#?)**, хотя это можно изменить.

Пример 11-30. Создание меню с помощью *select*

```
#!/bin/bash

PS3='Выберите любимый овощ: ' # Строка приглашения.
                              # В другом случае, по умолчанию #?.

echo

select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
  echo
  echo "Любимый овощ $vegetable."
  echo "Гадость!"
  echo
  break  # Что случиться, если отсюда убрать 'break'?
done

exit

# Упражнение:
# --------
#  Исправьте этот сценарий, что бы принимался пользовательский ввод, не
#+ указанный оператором «select».
#  Например, если пользователь введет "peas,"
#+ сценарий должен выдать "Извините. Этого в меню нет."
```

Если **in *list*** отсутствует, то ***select*** использует список переданных сценарию аргументов из командной строки **($@)** или функцию, содержащую конструкцию ***select***.

Сравните это с поведением

```
for переменная [in list]
```

конструкции с отсутствующим **in *list***.

Пример 11-31. Создание меню с помощью *select* в функции

```
#!/bin/bash

PS3='Выберите любимый овощ: '

echo

choice_of()
{
select vegetable
#  [in list] отсутствует, поэтому 'select' использует аргументы переданные
#+ функции.
do
  echo
  echo "Любимый овощ $vegetable."
  echo "Гадость!"
  echo
  break
done
}

choice_of beans rice carrots radishes rutabaga spinach
#         $1    $2   $3      $4       $5       $6
#         передаваемые функции choice_of()

exit 0
```

См. также Пример 37-3.

Примечания

[1] Шаблон совпадения строк может также начинаться с **(** слевой скобки, придавая более упорядоченный вид.

```
case $( arch ) in   # $( arch ) returns machine architecture.
  ( i386 ) echo "80386-based machine";;
# ^      ^
  ( i486 ) echo "80486-based machine";;
  ( i586 ) echo "Pentium-based machine";;
  ( i686 ) echo "Pentium2+-based machine";;
  (    * ) echo "Other type of machine";;
esac
```

Глава 12. Подстановка команд

Подстановка команд присваивает вывод команды [1] или даже нескольких команд; она буквально вставляет вывод команды в другой контекст. [2]

Классическая форма подстановки команд использует *обратные кавычки* (`...`). Команды внутри обратных кавычек (обратных галочек) образуют текст командной строки.

```
script_name=`basename $0`
echo "Название этого сценария $script_name."
```

Вывод команды может быть использован как аргумент для другой команды, для присваивания переменной и даже для создания списка аргументов в цикле *for*.

```
rm `cat filename`   # "filename" содержит список файлов для удаления.
#
#  S. C. указывает, что "слишком большой список аргументов"  может привести к
#+ ошибке.
#  Лучше так             xargs rm -- < filename
# ( -- охватывает те случаи, когда "filename" начинается с "-" )

textfile_listing=`ls *.txt`
#  Переменная содержит имена всех файлов *.txt в текущей рабочей директории.
echo $textfile_listing

textfile_listing2=$(ls *.txt)   # Другая форма подстановки команд.
echo $textfile_listing2
#  Тот же результат.

#  Возможной проблемой с вводом списка файлов в одну строку является
#+ перевод строк.
#
# Более безопасным способом присвоения параметру списка файлов является массив.
#      shopt -s nullglob    #  Если не совпадает,
                            #+ filename ничем не расширяется.
#      textfile_listing=( *.txt )
#
# Спасибо S.C.
```

Команда подстановки вызывает ***subshell (подоболочку)***.

Команда подстановки может приводить к ***разделению слова***.

```
COMMAND `echo a b`     # 2 аргумента: a и b -разделенное слово (буквальное)

COMMAND "`echo a b`"   # 1 аргумент: "a b" — одно (целое) слово

COMMAND `echo`         # нет аргументов

COMMAND "`echo`"       # один пустой аргумент

# Спасибо S.C.
```

Даже когда слова не разделяются, подстановка команд может удалять конечные символы.

```
# cd "`pwd`"  # Это должно работать всегда.
# Однако...
```

```
mkdir 'директория с символом перевода строки в конце
'

cd 'директория с символом перевода строки в конце
'

cd "`pwd`"  # Сообщение об ошибке:
#  bash: cd: /tmp/file с символом перевода строки в конце: Нет такого файла
#+ или директории

cd "$PWD"   # Прекрасно работает.


old_tty_setting=$(stty -g)   # Сохранение старых настроек терминала.
echo "Нажмите кнопку "
stty -icanon -echo           # Отключение режима терминала "canonical".
                             # Так же отключается *локальное* echo.
key=$(dd bs=1 count=1 2> /dev/null)   # С помощью 'dd' определяется нажатие.
stty "$old_tty_setting"      # Восстановление старых настроек.
echo "Вы нажали ${#key} клавиш."  # ${#variable} = число символов в $variable
#  Нажав любую клавишу, за исключением RETURN, получите вывод "Вы нажали 1
#+ клавишу."
#  Нажав RETURN получите "Вы нажали 0 клавиш."
#  Символ новой строки съедается подстановкой команд.

#  Фрагмент кода Stéphane Chazelas.
```

С помощью ***echo***, при выводе *без кавычек* присвоенной, с помощью подстановки команд, переменной, удаляется и конечный символ перевода строки из вывода переназначенной команды. Это может привести к неприятным сюрпризам.

```
dir_listing=`ls -l`
echo $dir_listing     # без кавычек

# Ожидаем прекрасно упорядоченный список.

# Однако получаем вот это:
# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1
bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13
wi.sh

# Перевод строки исчез.


echo "$dir_listing"   # заключаем в кавычки
# -rw-rw-r--    1 bozo       30 May 13 17:15 1.txt
# -rw-rw-r--    1 bozo       51 May 15 20:57 t2.sh
# -rwxr-xr-x    1 bozo      217 Mar  5 21:13 wi.sh
```

Подстановка команд позволяет даже присваивать переменной содержимое файла, используя

либо ***перенаправление*** или команду ***cat***.

```
variable1=`<file1`       #  Присвоение "variable1" содержимого "file1".
variable2=`cat file2`    #  Присвоение "variable2" содержимого "file2".
                         #  Это, однако, порождает новый процесс,
                         #+ поэтому эта строка кода выполняется
                         #+ медленнее, чем в версии выше.

#  Обратите внимание, что переменные внутри себя могут содержать  пробелы,
#+ или даже (ужас) управляющие символы.

#  Нет необходимости явного присваивания переменной.
echo "` <$0`"            # Выводит сценарий в stdout.


#  Выдержки из системного файла /etc/rc.d/rc.sysinit
#+ (установленного Red Hat Linux)


if [ -f /fsckoptions ]; then
        fsckoptions=`cat /fsckoptions`
...
fi
#
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
             hdmedia=`cat /proc/ide/${disk[$device]}/media`
...
fi
#
#
if [ ! -n "`uname -r | grep -- "-"`" ]; then
       ktag="`cat /proc/version`" # В двойных кавычках!
...
fi
#
#
if [ $usb = "1" ]; then
    sleep 5
    mouseoutput=`cat /proc/bus/usb/devices` 2>/dev/null|grep -E
"^I.*Cls=03.*Prot=02"`
    kbdoutput=`cat /proc/bus/usb/devices` 2>/dev/null|grep -E
"^I.*Cls=03.*Prot=01"`
...
fi
```

Не присваивайте переменной содержимое *большого* текстового файла, только если у вас есть для этого очень веские основания. Не присваивайте переменной содержимое *бинарного* файла, даже в шутку.

Пример 12-1. Идиотские трюки сценария

```
#!/bin/bash
```

```
# stupid-script-tricks.sh: Не делайте это дома, народ.
# Из "Stupid Script Tricks," Том I.

exit 99  ### Закомментируйте эту строку, если хватит смелости.

dangerous_variable=`cat /boot/vmlinuz`   # Сжатое ядро Linux.

echo "длина строки \$dangerous_variable = ${#dangerous_variable}"
# длина строки $dangerous_variable = 794151
# (Новейшие ядра больше.)
# Так не подсчитывается: 'wc -c /boot/vmlinuz'.

# echo "$dangerous_variable"
# Не пытайтесь повторить это! Сценарий зависнет.


#  Автор документа осознает вредность приложений с присвоенным
#+ переменной содержимого бинарного файла.

exit 0
```

Обратите внимание, что не происходит *переполнения буфера*. Это один из случаев, когда интерпретируемый язык, например Bash, обеспечивает более высокий уровень защиты от ошибок программиста, чем компилируемый язык.

Подстановка команд позволяет присваивать переменной вывод ***цикла***. Ключом к этому является захват вывода команды **echo** внутри цикла.

Пример 12-2. Создание переменной из цикла

```
#!/bin/bash
# csubloop.sh: Присвоение переменной вывода цикла.

variable1=`for i in 1 2 3 4 5
do
  echo -n "$i"                 #  Команда 'echo' ***решающая***
done`                          #+ для подстановки команд.

echo "variable1 = $variable1"  # variable1 = 12345


i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"                 # Опять ***необходимо*** 'echo'.
  let "i += 1"                 # Увеличение.
done`

echo "variable2 = $variable2"  # variable2 = 0123456789

#  Демонстрирует возможность включения объявления переменной внутри цикла.

exit 0
```

Подстановка команд расширяет набор инструментов доступных в Bash. Это всего лишь вопрос написания программы или сценария, которая выводит в `stdout` (подобно инструментам UNIX) и записи этого вывода в переменную.

```
#include <stdio.h>

/*  "Hello, world." C program  */

int main()
{
  printf( "Hello, world.\n" );
  return (0);
}
```

```
bash$ gcc -o hello hello.c
```

```
#!/bin/bash
# hello.sh

greeting=`./hello`
echo $greeting
```

```
bash$ sh hello.sh
Hello, world.
```

Форма **$(…)**, при подстановке команд, позволяет обходиться без обратных кавычек.

```
output=$(sed -n /"$1"/p $file)   # Из примера "grp.sh".

# Присвоение переменной содержимое текстового файла.
File_contents1=$(cat $file1)
File_contents2=$(<$file2)        # Bash допускает и так.
```

Форма подстановки команд **$(...)** рассматривает *двойной обратный слэш* по другому, в отличии от **`...`**.

```
bash$ echo `echo \\`


bash$ echo $(echo \\)
\
```

Форма подстановки команд **$(. . .)** допускает вложения. [3]

```
word_count=$( wc -w $(echo * | awk '{print $8}') )
```

И что-то посложнее...

Пример 12-3. Поиск анаграмм

```
#!/bin/bash
# agram2.sh
# Пример вложений подстановки команд.

#  Используется утилита "anagram"
#+ являющаяся частью пакета словаря автора "yawl".
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  http://bash.deta.in/yawl-0.3.2.tar.gz

E_NOARGS=86
E_BADARG=87
MINLEN=7

if [ -z "$1" ]
then
  echo "Usage $0 LETTERSET"
  exit $E_NOARGS         # Сценарию нужен аргумент командной строки.
elif [ ${#1} -lt $MINLEN ]
then
  echo "Аргумент должен иметь хотя бы $MINLEN букв."
  exit $E_BADARG
fi



FILTER='.......'         # Должно быть хотя бы 7 букв.
#       1234567
Anagrams=( $(echo $(anagram $1 | grep $FILTER) ) )
#          $(     $(  вложенные подкоманды.  ) )
#        (            присваивание массива       )

echo
echo "Найдены буквы ${#Anagrams[*]}  анаграмм "
echo
echo ${Anagrams[0]}      # Первая анаграмма.
echo ${Anagrams[1]}      # Вторая анаграмма.
                         # и т.д.

# echo "${Anagrams[*]}"  # Список всех анаграмм одной строкой ...

# Обратитесь к главе Массивы что бы понять, что здесь происходит.

# См. Так же сценарий agram.sh для упражнения в поиске анаграмм.

exit $?
```

Примеры подстановки команд в сценариях:

1. Пример 11-8
2. Пример 11-27
3. Пример 9-16
4. Пример 16-3
5. Пример 16-22
6. Пример 16-17
7. Пример 16-54
8. Пример 11-14
9. Пример 11-11
10. Пример 16-32
11. Пример 20-8
12. Пример A-16
13. Пример 29-3
14. Пример 16-47
15. Пример 16-48
16. Пример 16-49

Примечания

[1] В *подстановке команд*, ***команда*** может быть внешней системной командой, внутренней встроенной сценария, или даже функцией сценария.

[2] В технически более правильном смысле, *подстановка команды* извлекает `stdout` команды, а затем присваивает его переменной с помощью оператора **=**.

[3] На самом деле, вложение в *обратные кавычки* также возможно, но только с экранированием внутренних обратных кавычек, как и указывает John Default.

```
word_count=` wc -w \`echo * | awk '{print $8}'\` `
```

Глава 13. Арифметические расширения

Арифметические расширения представляют собой мощный инструмент для выполнения в сценариях арифметических операций (с целыми числами). Перевод строки в числовое выражение осуществляется относительно просто, используя *обратные кавычки*, *двойные круглые скобки* или ***let***.

Варианты

Арифметическое расширение с обратными кавычками (часто используется в сочетании с **expr**).

```
z=`expr $z + 3`          # Команда 'expr' выполняет расширение.
```

Арифметическое расширение в ***двойных скобках*** и с помощью ***let***.

Использование обратных кавычек в арифметическом расширении было заменено на двойные скобки **((...))** и **$((...))**, а также на очень удобную конструкцию **let**.

```
z=$(($z+3))
z=$((z+3))                                  #  То же правильно.
                                            #  В двойных скобках,
                                            #+ параметр разыменован, но
                                            #+ это по желанию.

# $(( ВЫРАЖЕНИЕ )) это арифметическое выражение.
# Не следует путать с подстановкой команд.

# Можно также производить операции в двойных скобках без присваивания.

  n=0
  echo "n = $n"                             # n = 0

  (( n += 1 ))                              # Увеличение.
# (( $n += 1 )) Это не правильно!
  echo "n = $n"                             # n = 1
```

```
let z=z+3
let "z += 3"  #  Кавычки разрешают использовать пробелы при присваивании
              #+ переменной.
              #  Оператор 'let' выполняет арифметическое вычисление,
              #+ а не расширение.
```

Примеры арифметического расширения в сценариях:

1. Пример 16-9
2. Пример 11-15
3. Пример 27-1
4. Пример 27-11
5. Пример A-16

Глава 14. Время перерыва

Этот странный маленький антракт дает читателю возможность расслабиться и, возможно, немного посмеяться.

Пользователь Linux, привет! Вы читаете то, что принесет вам удачу и счастье. Просто перешлите по электронной почте копию этого документа 10 вашим друзьям. Перед тем, как сделать копии, отправьте 100-строчный сценарий Bash, в нижней части этого письма, первому в списке. Затем удалите имя и добавьте свое в нижней части списка.

Wilfred P. из Бруклина удалось отправить его десять копий, и проснувшись на следующее утро, обнаружил, что род его деятельности изменился на «программист COBOL». L. Howard и Ньюпорт-Ньюс, послал десять копий и в течение месяца обнаружил оборудование достаточное для построения 100-узлового кластера Beowulf для игры в Tuxracer. Amelia V. из Чикаго рассмеялся на это письмо и прервала цепочку. Вскоре после этого вспыхнул ее монитор, и теперь она тратит свои дни на написание документации для MS Windows.

Отправьте ваши десять копий сегодня! Не разрывайте цепь!
Сделайте копии в течение 48 часов.

Вежливый 'NIX "куки счастья", с некоторыми изменениями и многими извинениями

Часть 4. Команды

Владение командами Linux является незаменимой прелюдией написания эффективных сценариев.

Этот раздел охватывает следующие команды:

A

ac, adduser, agetty, agrep, ar, arch, at, autoload, awk

B

badblocks, banner, basename, batch, bc, bg, bind, bison, builtin, bzgrep, bzip2

C

cal, caller, cat, cd, chmod, chfn, chgrp, chkconfig, chmod, chown, chroot, cksum, clear, clock, cmp, col, colrm, column, comm, command, compgen, complete, compress, coproc, cp, cpio, cron, crypt, csplit, cu, cut

D

date, dc, dd, debugfs, declare, depmod, df, dialog, diff, diff3, diffstat, dig, dirname, dirs, disown, dmesg, doexec, dos2unix, du, dump, dumpe2fs

E

e2fsck, echo, egrep, enable, enscript, env, eqn, eval, exec, exit (Связанная тема: **exit status), expand, export, expr**

F

factor, false, fsck, fdformat, fdisk, fg, fgrep, file, find, finger, flex, flock, fmt, fold, free, fsck, ftp, fuser

G

getfacl, getopt, getopts, gettext, getty, gnome-mount, grep, groff, groupmod, groups (Связанная тема: переменная **$GROUPS), gs, gzip**

H

halt, hash, hdparm, head, help, hexdump, host, hostid, hostname (Связанная тема: переменная **$HOSTNAME**), **hwclock**

I

iconv, id (Связанная тема: переменная **$UID**), **ifconfig, info, infocmp, init, insmod, install, ip, ipcalc, iptables, iwconfig**

J

jobs, join, jot

K

kill, killall

L

last, lastcomm, lastlog, ldd, lessmore, let, lex, lid, ln, locate, lockfile, logger, logname, logout, logrotate, look, losetup, lp, ls, lsdev, lsmod, lsof, lspci, lsusb, ltrace, lynx, lzcat, lzma

M

m4, mail, mailstats, mailto, make, MAKEDEV, man, mapfile, mcookie, md5sum, merge, mesg, mimencode, mkbootdisk, mkdir, mkdosfs, mke2fs, mkfifo, mkisofs, mknod, mkswap, mktemp, mmencode, modinfo, modprobe, more, mount, msgfmt, mv

N

nc, netconfig, netstat, newgrp, nice, nl, nm, nmap, nohup, nslookup

O

objdump, od, openssl

P

passwd, paste, patch (Связанная тема: **diff), pathchk, pax, pgrep, pidof, ping, pkill, popd, pr, printenv, printf, procinfo, ps,**

pstree, ptx, pushd, pwd (Связанная тема: переменная **$PWD)**

Q

quota

R

rcp, rdev, rdist, read, readelf, readlink, readonly, reboot, recode, renice, reset, resize, restore, rev, rlogin, rm, rmdirreboot, rmmod, route, rpm, rpm2cpio, rsh, rsync, runlevel, run-parts, rx, rz

S

sar, scp, script, sdiff, sed, seq, service, set, setfacl, setquota, setserial, setterm, sha1sum, shar, shopt, shred, shutdown, size, skill, sleep, slocate, snice, sort, source, sox, split, sq, ssh, stat, strace, strings, strip, stty, su, sudo, sum, suspend, swapoff, swapon, sx, sync, sz

T

tac, tail, tar, tbl, tcpdump, tee, telinit, telnet, TeX, texexec, time, times, tmpwatch, top, touch, tput, tr, traceroute, true, tset, tsort, tty, tune2fs, type, typeset

U

ulimit, umask, umount, uname, unarc, unarj, uncompress, unexpand, uniq, units, unlzma, unrar, unset, unsq, unzip, uptime, usbmodules, useradd, userdel, usermod, users, usleep, uucp, unzip, uudecode, uuencode, uux

V

vacation, vdir, vmstat, vrfy

W

w, wait, wall, watch, wc, wget, whatis, whereis, which, who, whoami, whois, write

X

xmessage, xargs, xrandr, xz

Y

yacc, yes

Z

zcat, zdiff, zdump, zegrep, zenity, zfgrep, zgrep, zip

Содержание

Глава 15. Внутренние и встроенные команды

«***Builtin***» (Встроенный ресурс) — это **команда**, содержащаяся в наборе инструментов Bash, буквально *встроенная*. Это сделано либо по соображениям производительности -- встроенные команды выполняются быстрее, чем внешние, которые обычно требуют порождения [1] *дочернего* процесса --либо потому, что специфике ***builtin*** нужен *прямой доступ* к внутренним компонентам оболочки.

Запущенный командой или на самой оболочкой новый подпроцесс, для выполнения определенной задачи (или размножения), называется ***форком***. Этот новый процесс является ***дочерним*** или ***потомком***, а процесс, от которого он был ответвлен является ***родителем***. Родительский процесс продолжает выполняться, во время работы дочернего процесса.

Обратите внимание, что *родительский* процесс получает ID *дочернего* процесса, и, таким образом, передает ему аргументы, но никак не в обратном порядке. Это может создавать проблемы, которые сложно отследить.

Пример 15-1. Сценарий, который порождает несколько своих экземпляров

```
#!/bin/bash
# spawn.sh


PIDS=$(pidof sh $0)  # ID процессов различных экземпляров этого сценария.
P_array=( $PIDS )    # Помещение их в массив (почему?).
echo $PIDS           # Вывод ID родительского и дочерних процессов.
let "instances = ${#P_array[*]} - 1"  # Подсчет элементов, уменьшая на 1.
                                      # Почему вычитается 1?
echo "Запущено $instances экземпляр(ов) этого сценария."
```

```
echo "[Нажмите Ctl-C для выхода.]"; echo


sleep 1              # Ожидание.
sh $0                # Сыграй снова, Sam.

exit 0               # Не обязательно; сценарий никогда сюда не доберется.
                     # Почему?

#  После выхода при помощи Ctl-C,
#+ все порожденные экземпляры  сценария умрут?
#  Если да, то почему?

# Примечания:
# ----
# Будьте осторожны, не запускайте этот сценарий слишком надолго.
# Он пожирает слишком много системных ресурсов.
```

Как правило *встроенные* команды Bash, когда они выполняются в сценарии, ***не порождают*** подпроцессы. Внешние системные команды или фильтры в сценарии обычно ***порождают*** подпроцессы.

Встроенная (***builtin***) команда может быть синонимом системной команды с тем же именем, но Bash переопределяет ее внутри. Например команда **echo** в Bash не является тем же самым, что `/bin/echo`, хотя поведение их почти идентично.

```
#!/bin/bash

echo "Эта строка использует встроенную команду \"echo\"."
/bin/echo "Эта строка использует системную команду /bin/echo."
```

Ключевые слова (***keyword***) это *зарезервированные* слова, токены или операторы. Ключевые слова имеют особое значение в оболочке и в действительности являются строительными блоками синтаксиса оболочки. Например, ***for***, ***while***, ***do***, и ***!*** это ключевые слова. Подобно *встроенным* командам, ключевые слова жестко запрограммированы в Bash, но в отличие от ***builtin***, ключевое слово само по себе является не командой, а *частью конструкции команды*. [2]

I/O (Команды Вода/Вывода)

echo

выводит (в `stdout`) выражение или переменную (См. Пример 4-1).

```
echo Hello
echo $a
```

Для вывода заэкранированных символов ***echo*** необходима опция **-e**. См. Пример 5-2.

Обычно, каждая команда ***echo*** выводит в терминале новую строку, но опция **-n** это подавляет.

echo может использоваться для передачи последовательности команд через конвейер.

```
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
then
  echo "$VAR содержит последовательность substring \"txt\""
fi
```

echo, в комбинации с подстановкой команд, может присваивать переменную.

```
a=`echo "HELLO" | tr A-Z a-z`
```

См. также Пример 16-22, Пример 16-3, Пример 16-47 и Пример 16-48.

Помните, что ***echo `команда`*** удаляет все символы перевода строки создаваемые выводом *команда* .

Переменная **$IFS** (внутренний разделитель полей) обычно содержит **\n** (перевод строки) в качестве одного из набора символов пробелов. Поэтому Bash разбивает вывод *команда* символами пробелов на аргументы передавая в ***echo***. Затем ***echo*** выводит эти аргументы разделенные пробелами.

```
bash$ ls -l /usr/share/apps/kjezz/sounds
-rw-r--r--    1 root     root         1407 Nov  7  2000 reflect.au
 -rw-r--r--    1 root     root          362 Nov  7  2000 seconds.au


bash$ echo `ls -l /usr/share/apps/kjezz/sounds`
total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 reflect.au -rw-r--r-- 1 root
root ...
```

Итак, как мы можем вставить перевод строки в пределах выводимой строки символов?

```
# Вставляем перевод строки?
echo "Почему эту строку \n не разбивает на две строки?"
# Не разбивает.

# Попробуем что-нибудь еще.

echo

echo $"Строка текста содержащая
перевод строки."
# Выводит как две различные строки (встроенный перевод строки).
```

```
# А, действительно, необходим префикс переменной '$'?

echo

echo "Эта строка разбита
на две строки."
# Нет, '$' не требуется.

echo
echo "---------------"
echo

echo -n $"Другая строка текста содержащая
перевод строки."
# Выводятся как две различные строки (встроенный перевод строки).
# Даже опции -n не удается здесь подавить новую строку.

echo
echo
echo "---------------"
echo
echo

# А вот следующее не работает как ожидалось.
# Почему? Подсказка: Присваивание переменной.
string1=$"Еще одна строка текста содержащая
перевод строки (может быть)."

echo $string1
# Еще одна строка текста содержащая перевод строки (может быть).
#                                  ^
# Перевод строки становится пробелом.

# Спасибо Steve Parker за разъяснение.
```

Это встроенная команда оболочки, отличающаяся от `/bin/echo`, хотя является аналогичной.

```
bash$ type -a echo
echo is a shell builtin
 echo is /bin/echo
```

printf

Команда форматированного вывода ***printf***, это расширенное ***echo***. Она является ограниченным вариантом библиотечной функции *printf()* языка Си и ее синтаксис несколько отличается.

printf *форматированная строка... параметр...*

В Bash это *встроенная* версия команды `/bin/printf` или `/usr/bin/printf`. См.

Справочные страницы **printf** (системной команды) для более глубоко изучения.

Старые версии Bash могут не поддерживать **printf**.

Пример 15-2. *printf* в действии

```
#!/bin/bash
# Демонстрация работы printf

declare -r PI=3.14159265358979 # Переменная только для чтения, т.е., константа.
declare -r DecimalConstant=31373

Message1="Greetings,"
Message2="Earthling."

echo

printf "Пи с 2 десятичными знаками = %1.2f" $PI
echo
printf "Пи с 9 десятичными знаками = %1.9f" $PI  # Даже правильно округляет.

printf "\n"                                      # Перевод строки,
                                                 # Эквивалентен 'echo' . . .

printf "Constant = \t%d\n" $DecimalConstant  # Вставка табуляции (\t).

printf "%s %s \n" $Message1 $Message2

echo

# ============================================#
# Моделирование функции Си, sprintf().
# Загрузка переменной с форматированной строкой.

echo

Pi12=$(printf "%1.12f" $PI)
echo "Пи с 12 десятичными знаками = $Pi12"      # Ошибка округления!

Msg=`printf "%s %s \n" $Message1 $Message2`
echo $Msg; echo $Msg

#  Что произошло, функция «sprintf» теперь может
#+ быть доступна для Bash как загружаемый модуль,
#+ но это не переносится.

exit 0
```

Форматированные сообщения об ошибках — это полезное приложение **printf**

```
E_BADDIR=85

var=nonexistent_directory
```

```
error()
{
  printf "$@" >&2
  # Форматирование передаваемых позиционных параметров, и передача их stderr.
  echo
  exit $E_BADDIR
}

cd $var || error $"Не перешли в %s." "$var"

# Спасибо S.C.
```

См. также Пример 36-17.

read

"Считывает" значение переменной со *stdin*, то есть интерактивно извлекает ввод с клавиатуры. Опция **-a** позволяет считывать полученный массив переменных (см. Пример 27-6).

Пример 15-3. Присваивание переменной с помощью *read*

```
#!/bin/bash
# "Считывание" переменных.

echo -n "Введите значение переменной 'var1': "
# Опция -n в echo подавляет перевод строки.

read var1
# Обратите внимание на отсутствие '$' перед var1.

echo "var1 = $var1"

echo

# Одним объявлением 'read' может присвоить несколько переменных.
echo -n "Введите значения переменных 'var2' и 'var3' "
echo =n "(разделенных пробелами или табуляцией): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
#  Если ввели только одно значение,
#+ другая переменная(ые) останутся не присвоенными (null).

exit 0
```

При отсутствии передачи переменной в ***read*** осуществляется ввод в переменную **$REPLY**.

Пример 15-4. Что случится, если у *read* не будет переменной

```
#!/bin/bash
# read-novar.sh

echo
```

```
# ------------------------- #
echo -n "Введите значение: "
read var
echo "\"var\" = "$var""
# Все как ожидается.
# ------------------------- #

echo

# ------------------------------------------------------------------ #
echo -n "Введите другое значение: "
read           #  Переменная для 'read' отсутствует, поэтому...
               #+ Ввод 'read' присваивается по умолчанию переменной $REPLY.
var="$REPLY"
echo "\"var\" = "$var""
# Это эквивалентно первому блоку кода.
# ------------------------------------------------------------------ #

echo
echo "========================="
echo


#  Этот пример похож на сценарий 'reply.sh'.
#  Тем не менее, это одно показывает, что $REPLY доступна
#+ переменной обычным способом, даже после «read».


# ================================================================== #

#  Иногда, возможно, нужно отменить чтение первого значения.
#  В таких случаях просто игнорируется переменная $REPLY.

{ # Блок кода.
read            # Строка 1, игнорируется.
read line2      # Строка 2, сохраняется в переменную.
  } <$0
echo "Строка 2 этого сценария:"
echo "$line2"   #   # read-novar.sh
echo            #   #!/bin/bash  строка проигнорирована.

# См. Так же сценарий soundcard-on.sh.

exit 0
```

Обычно, ввод `\` подавляет перевод строки во время ввода в **`read`**. Опция **`-r`** вызывает символ `\`, чтобы интерпретировать в буквальном смысле.

Пример 15-5. Многострочный ввод в *read*

```
#!/bin/bash

echo

echo "Введите строку завершив ее \\, а затем нажмите <ENTER>."
echo "Теперь введите вторую строку (в этот раз без \\), и снова нажмите
```

```
<ENTER>."

read var1     #     "\" подавляет перевод строки при чтении $var1.
              #     первая строка \
              #     вторая строка

echo "var1 = $var1"
#     var1 = первая строка   вторая строка

#  В каждой строке, заканчивающейся "\", вы получаете приглашение
#+ к вводу следующей строки, для продолжения ввода символов в var1.

echo; echo

echo "Введите другую строку заканчивающуюся \\ , затем нажмите <ENTER>."
read -r var2  # Опция -r заставляет читать буквально "\".
              #     первая строка \

echo "var2 = $var2"
#     var2 = первая строка \

# Запись данных прекращается с первым нажатием  <ENTER>.

echo

exit 0
```

Команда **read** имеет некоторые интересные опции, которые позволяют выводить приглашение, и даже считывать данные не дожидаясь нажатия **ENTER**.

```
# Чтение нажатых клавиш без нажатия ENTER.

read -s -n1 -p "Нажмите клавишу " keypress
echo; echo "Была нажата клавиша "\"$keypress\""."

# Опция -s опция означает не выводить ввод на экран.
# Опция -nN означает принимать только N вводимых символов.
# Опция -p  означает, вывод на экран приглашения перед чтением ввода.

#  Пользоваться этими параметрами сложно, поскольку они должны идти в
#+ определенном порядке.
```

Кроме того, опция `-n` в **read** позволяет обнаруживать нажатие **клавиш со стрелками** и некоторых других необычных клавиш.

Пример 15-6. Обнаружение нажатия клавиш со стрелками

```
#!/bin/bash
# arrow-detect.sh: Обнаружение нажатий служебных клавиш.
# Спасибо Sandro Magi, за то, что показал как это сделать.

# ---------------------------------------------
```

```
# Коды символов создаваемых нажатием клавиш.
arrowup='\[A'
arrowdown='\[B'
arrowrt='\[C'
arrowleft='\[D'
insert='\[2'
delete='\[3'
# ----------------------------------------

SUCCESS=0
OTHER=65

echo -n "Нажмите клавишу...  "
# Может потребоваться нажать ENTER, если нажата не перечисленная выше клавиша.
read -n3 key                      # Считывается 3 символа.

echo -n "$key" | grep "$arrowup"  # Проверка обнаружения кода символа.
if [ "$?" -eq $SUCCESS ]
then
  echo "Нажата клавиша Up-arrow."
  exit $SUCCESS
fi

echo -n "$key" | grep "$arrowdown"
if [ "$?" -eq $SUCCESS ]
then
  echo "Нажата клавиша Down-arrow."
  exit $SUCCESS
fi

echo -n "$key" | grep "$arrowrt"
if [ "$?" -eq $SUCCESS ]
then
  echo "Нажата клавиша Right-arrow."
  exit $SUCCESS
fi

echo -n "$key" | grep "$arrowleft"
if [ "$?" -eq $SUCCESS ]
then
  echo "Нажата клавиша Left-arrow."
  exit $SUCCESS
fi

echo -n "$key" | grep "$insert"
if [ "$?" -eq $SUCCESS ]
then
  echo "Нажата клавиша \"Insert\"."
  exit $SUCCESS
fi

echo -n "$key" | grep "$delete"
if [ "$?" -eq $SUCCESS ]
then
  echo "Нажата клавиша \"Delete\"."
  exit $SUCCESS
fi


echo " Нажата иная клавиша."
```

```
exit $OTHER

# ======================================== #

#  Mark Alexander привел упрощенную версию сценария выше (спасибо!).
#  Он устраняет необходимость в grep.

#!/bin/bash

  uparrow=$'\x1b[A'
  downarrow=$'\x1b[B'
  leftarrow=$'\x1b[D'
  rightarrow=$'\x1b[C'

  read -s -n3 -p "Нажмите клавишу со стрелкой: "

  case "$x" in
  $uparrow)
     echo "Вы нажали up-arrow"
     ;;
  $downarrow)
     echo " Вы нажали down-arrow"
     ;;
  $leftarrow)
     echo " Вы нажали left-arrow"
     ;;
  $rightarrow)
     echo " Вы нажали right-arrow"
     ;;
  esac

exit $?

# ======================================== #

# Более простая альтернатива от Antonio Macchi.

#!/bin/bash

while true
do
  read -sn1 a
  test "$a" == `echo -en "\e"` || continue
  read -sn1 a
  test "$a" == "[" || continue
  read -sn1 a
  case "$a" in
    A)  echo "up";;
    B)  echo "down";;
    C)  echo "right";;
    D)  echo "left";;
  esac
done

# ======================================== #

#  Упражнение:
#  --------
#  1) Добавьте определение клавиш "Home," "End," "PgUp," и "PgDn".
```

Опция `-n` **`read`** не определяет нажатие клавиши **ENTER** (перевод строки).

Опция `-t` **`read`** разрешает ввод в течении определенного времени (см. Пример 9-4 и Пример A- 41).

Опция `-u` принимает ***дескриптор*** целевого файла.

Команда **`read`** также может 'считывать' значение переменной из файла перенаправленного в `stdin`. Если файл содержит более чем одну строку, то переменной присваивается только первая строка. Если **`read`** имеет более одного параметра, то каждой строке этих переменных присваивается ***межстрочный*** пробел. ***Внимание***!

Пример 15-7. Применение *read* с *перенаправлением файла*

```
#!/bin/bash

read var1 <data-file
echo "var1 = $var1"
# var1 присваивает всю первую строку вводимую из файла "data-file"

read var2 var3 <data-file
echo "var2 = $var2   var3 = $var3"
# Обратите внимание, на не интуитивное поведение «read».
# 1) Переходим назад к началу ввода из файла.
# 2) Каждой переменной присваиваются соответствующие строки,
#    разделяемые пробелами, а не все строки текста.
# 3) Последняя переменная получает остаток строки.
# 4) Если  будет установлено больше переменных, чем первых строк файла
#    завершающихся пробелами, то последние переменные останутся пустыми.

echo "-----------------------------------------------"

# Как решить данную проблему с помощью цикла:
while read line
do
  echo "$line"
done <data-file
# Спасибо Heiner Steven за разъяснения.

echo "-----------------------------------------------"

# Если вы не хотите иметь пробелы по умолчанию, то с помощью $IFS (переменной
# Internal Field Separator) строки вводимые в "read" разбиваются.

echo "Список всех пользователей:"
OIFS=$IFS; IFS=:       # /etc/passwd для разделения полей использует ":".
while read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done </etc/passwd   # Перенаправление I/O.
IFS=$OIFS              # Восстановление оригинальной $IFS.
# Это фрагмент кода Heiner Steven.
```

```
#  Переменная $IFS, присваиваемая внутри цикла, устраняет необходимость
#+ сохранения оригинальной $IFS во временной переменной.
#  Спасибо Dim Segebart, за разъяснение.
echo "-----------------------------------------------"
echo "Список всех пользователей:"

while IFS=: read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done </etc/passwd   # Перенаправление I/O.

echo
echo "\$IFS все еще $IFS"

exit 0
```

Вывод конвейера в ***read***, с помощью ***echo***, не присваивает переменные.

Тем не менее, вывод конвейера из **cat**, *кажется*, работает.

```
cat file1 file2 |
while read line
do
   echo $line
done
```

Однако, как продемонстрировал Bjön Eriksson:

Пример 15-8. Проблемы при чтении из конвейера

```
#!/bin/sh
# readpipe.sh
# Этот пример предоставлен Bjon Eriksson.

### shopt -s lastpipe

last="(null)"
cat $0 |
while read line
do
    echo "{$line}"
    last=$line
done

echo
echo "+++++++++++++++++++++"
printf "\nГотово, последняя $last\n" #  Вывод этой строки изменится
                                    #+ если раскомментировать строку 5.
                                    #  (Bash, версии 4.2 требуется -ge.)

exit 0  # Завершение кода.
        # Следует(частичный) вывод сценария.
```

```
        # 'echo' предоставляет дополнительные скобки.

###############################################

./readpipe.sh

{#!/bin/sh}
{last="(null)"}
{cat $0 |}
{while read line}
{do}
{echo "{$line}"}
{last=$line}
{done}
{printf "nГотово, последнее: $lastn"}


Готово, последнее: (null)

Переменная (последняя) присваивается в пределах цикла/subshell,
но ее значение, вне цикла, не сохраняется.
```

Сценарий *gendiff*, который можно найти в `/usr/bin` многих дистрибутивов Linux, передает конвейером вывод ***find*** в конструкцию ***while read***.

```
find $1 \( -name "*$2" -o -name ".*$2" \) -print |
while read f; do
. . .
```

В поле ввода ***read*** возможно *вставлять* текст (но не несколько строк!). См. Пример A-38.

Файловая система

cd

Знакомая команда изменения директории **cd,** находит использование в сценариях, где выполнение команды требует пребывания в указанной директории.

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

[цитата из приведенного ранее примера Alan Cox]

Опция `-P` (**p**hysical) **cd** вызывает игнорирование символических ссылок .

cd - изменяет на **$OLDPWD**, предыдущую рабочую директорию .

Команда **cd** не работает, как ожидается, при наличии двух слешей.

```
bash$ cd //
bash$ pwd
//
```

Выводом, конечно же, будет `/`. Это проблема как командной строки, так и сценария.

pwd

Вывод рабочей директории. Показывает пользовательскую (или сценария) текущую директорию (см. Пример 15-9). Эффект полностью идентичен чтению значения встроенной переменной **$PWD**.

pushd, **popd**, **dirs**

Данный набор команд — это механизм закладок рабочих директорий, средство перемещения вперед и назад упорядоченным образом по директориям. Стек ***pushdown*** используется для отслеживания имен директорий. При помощи опций возможны различные манипуляции со стеком директорий.

pushd dir-name помещает путь *dir-name* в стек директорий (в верхнюю часть стека) и одновременно изменяет текущую рабочую директорию на *dir-name*

popd Удаляет (извлекает) путь директории из *верхней части* стека директорий и одновременно изменяет текущую рабочую директорию на директорию находящуюся в данный момент *в верхней части стека*.

dirs перечисляет содержимое стека директорий (сравните с переменной $DIRSTACK). Успешное **pushd** или **popd** автоматически вызывает **dirs**.

Сценарии, требующие каких-то изменений в текущей рабочей директории без жесткого задания имени изменяющейся директории, могут использовать эти команды. Обратите внимание, что не явная переменная массива **$DIRSTACK**, доступна из сценария и включает в себя содержимое стека директорий.

Пример 15-9. Изменение текущей рабочей директории

```
#!/bin/bash

dir1=/usr/local
dir2=/var/spool

pushd $dir1
# Производит автоматически 'dirs' (стек списка директорий в stdout).
echo "Теперь в директории `pwd`." # Для 'pwd' используются обратные кавычки.

# Теперь что-то сделаем в директории 'dir1'.
pushd $dir2
echo "Теперь в директории `pwd`."

# Теперь что-то сделаем в директории 'dir2'.
```

```
echo "Верхняя запись в массиве DIRSTACK -это $DIRSTACK."
popd
echo "Теперь обратно в директории `pwd`."

# Теперь что-то сделаем в директории 'dir1'.
popd
echo "Теперь снова в исходной рабочей директории `pwd`."

exit 0

# Что случится, если не сделать 'popd' — и выйти из сценария?
# В какой директории Вы окажитесь? Почему?
```

Переменные

`let`

Команда **`let`** выполняет *арифметические* операции над переменными.[3] В большинстве случаев она функционирует как менее сложная версия **`expr`**.

Пример 15-10. Давайте посчитаем с *let*.

```
#!/bin/bash

echo

let a=11            # То же, что и 'a=11'
let a=a+5           # Эквивалентно  let "a = a + 5"
                    # (Двойные кавычки и пробелы делают более читабельным.)
echo "11 + 5 = $a"  # 16

let "a <<= 3"       # Эквивалентно  let "a = a << 3"
echo "\"\$a\" (=16) смещение влево на 3 позиции = $a"
                    # 128

let "a /= 4"        # Эквивалентно  let "a = a / 4"
echo "128 / 4 = $a" # 32

let "a -= 5"        # Эквивалентно  let "a = a - 5"
echo "32 - 5 = $a"  # 27

let "a *=  10"      # Эквивалентно  let "a = a * 10"
echo "27 * 10 = $a" # 270

let "a %= 8"        # Эквивалентно  let "a = a % 8"
echo "270 по модулю 8 = $a  (270 / 8 = 33, остаток $a)"
                    # 6


# Допускает ли "let" операторы в стиле Си?
# Да, так же, как делается в конструкции ((...)) двойных скобок.

let a++             # Увеличение (после) в стиле Си.
echo "6++ = $a"     # 6++ = 7
let a--             # Уменьшение в стиле Си.
echo "7-- = $a"     # 7-- = 6
```

```
# Конечно, ++a, и т.п., так же позволительно ...
echo


# Тройной оператор.

# Обратите внимание, что $a это 6, см. выше.
let "t = a<7?7:11"   # Истинно
echo $t  # 7

let a++
let "t = a<7?7:11"   # Ложно
echo $t  #     11

exit
```

Команда ***let*** , в определенных контекстах, может возвращать удивительный статус выхода.

```
# Пояснения Evgeniy Ivanov:

var=0
echo $?     # 0
            # Как и ожидалось.

let var++
echo $?     #  1
            #  Команда выполнена успешно,
            #+ тогда почему не $?=0 ???
            #  Аномалия!

let var++
echo $?     # 0
            # Как и ожидалось.


# Точно так же ...

let var=0
echo $?     # 1
            # Команда выполнена успешно,
            #+ тогда почему не $?=0 ???

#  Однако, как указывает Jeff Gorak,
#+ это часть спецификации «let»...
#  "Если последний АРГУМЕНТ имеет значение 0, let возвращает 1;
#+ в противном случае let возвращает 0." ['help let']
```

eval

eval *arg1* [*arg2*] ... [*argN*]

Комбинирует аргументы в выражении или списке выражений и затем *оценивает* их. Любые переменные внутри выражения расширяются. Конечным результатом является ***преобразование строки в команду***.

Команда **eval** может использоваться для создания кода в командной строке или в сценарии.

```
bash$ command_string="ps ax"
bash$ process="ps ax"
bash$ eval "$command_string" | grep "$process"
26973 pts/3    R+     0:00 grep --color ps ax
 26974 pts/3    R+     0:00 ps ax
```

Каждый вызов ***eval*** приводит к переоценке его аргументов.

```
a='$b'
b='$c'
c=d

echo $a             # $b
                    # Первый уровень.
eval echo $a        # $c
                    # Второй уровень.
eval eval echo $a   # d
                    # Третий уровень.

# Спасибо E. Choroba.
```

Пример 15-11. Демонстрация эффекта *eval*

```
#!/bin/bash
# Применение "eval" ...

y=`eval ls -l`  #  Подобно y=`ls -l` ,но переводы строки удаляются,
echo $y         #+ потому что «выводимая» переменная не заключена в кавычки.
echo
echo "$y"       #  Переводы строки сохраняются, когда переменная
                #+ заключается в кавычки.

echo; echo

y=`eval df`     #  Подобно y=`df`,
echo $y         #+ но переводы строки удаляются.

#  Если удаляются переводы строк, при анализе вывода их легко можно
#+ снова воспроизвести (восстановить) с помощью таких утилит, как "awk".

echo
echo "==========================================================="
echo

eval "`seq 3 | sed -e 's/.*/echo var&=ABCDEFGHIJ/'`"
# var1=ABCDEFGHIJ
# var2=ABCDEFGHIJ
# var3=ABCDEFGHIJ

echo
echo "==========================================================="
```

```
echo


# Теперь посмотрим, что можно сделать полезного с помощью "eval" ...
# (Благодарность E. Choroba!)

version=3.4     #  Можно ли разделить версию на старшую
                #+ и младшую части одной командой?
echo "version = $version"
eval major=${version/./;minor=}     #  Заменяем '.' в версии на ';minor='
                                    #  Подставляем поля '3; minor=4'
                                    #+ таким образом eval делает
                                    #+ minor=4, major=3
echo Major: $major, minor: $minor   #  Major: 3, minor: 4
```

Пример 15-12. Выбор между переменными с помощью *eval*

```
#!/bin/bash
# arr-choice.sh

#  Передача аргументов в функцию для выбора
#+ одной конкретной переменной из группы.

arr0=( 10 11 12 13 14 15 )
arr1=( 20 21 22 23 24 25 )
arr2=( 30 31 32 33 34 35 )
#       0  1  2  3  4  5      Число элементов (начинаются с нуля)


choose_array ()
{
  eval array_member=\${arr${array_number}[element_number]}
  #                  ^       ^^^^^^^^^^^^
  #  Используем eval для создания имени переменной,
  #+ в данном конкретном случае, имени массива.

  echo "Элемент $element_number массива $array_number это $array_member"
} #  Функция перезаписывает при приеме параметров.

array_number=0    # Первый массив.
element_number=3
choose_array      # 13

array_number=2    # Третий массив.
element_number=4
choose_array      # 34

array_number=3    # Null массив (arr3 не существует).
element_number=4
choose_array      # (null)

# Спасибо Antonio Macchi за разъяснения.
```

Пример 15-13. Вывод параметров командной строки

```
#!/bin/bash
# echo-params.sh

# Сценарий вызывается несколькими параметрами из командной строки.
# Например:
#     sh echo-params.sh first second third fourth fifth

params=$#              # Количество параметров командной строки.
param=1                # Начальный первый параметр командной строки.

while [ "$param" -le "$params" ]
do
  echo -n "Параметр командной строки "
  echo -n \$$param     #  Задается только *имя* переменной.
#         ^^^          #  $1, $2, $3, и т.д.
                       #  Почему?
                       #  \$ экранирует первую "$"
                       #+ поэтому она понимается буквально,
                       #+ а $param разименовывается в "$param"...
                       #+ ... как и ожидалось.
  echo -n " = "
  eval echo \$$param   #  Задаем *значение* переменной.
# ^^^^       ^^^       #  "eval" принудительно *оценивает* \$$
                       #+ как косвенную ссылку на переменную.

(( param ++ ))         # Далее.
done

exit $?

# =================================================

$ sh echo-params.sh first second third fourth fifth
Параметр командной строки $1 = first
Параметр командной строки $2 = second
Параметр командной строки $3 = third
Параметр командной строки $4 = fourth
Параметр командной строки $5 = fifth
```

Пример 15-14. Принудительный выход

```
#!/bin/bash
# Принудительный выход путем прекращения ppp.
# Для соединения dialup, конечно.

# Сценарий должен запускаться из-под root.

SERPORT=ttyS3
#  В зависимости от аппаратного обеспечения и даже версии ядра,
#+ порт модема на вашей машине может отличаться --
#+ /dev/ttyS1 или /dev/ttyS2.


killppp="eval kill -9 `ps ax | awk '/ppp/ { print $1 }'`"
#                     -------- ID процесса ppp --------

$killppp                    # Эта переменная стала командой.
```

```
# Следующие действия должны быть сделаны из-под root.

chmod 666 /dev/$SERPORT      # Восстановление прав r+w, или как?
#  Так как работа SIGKILL на ppp изменяет права доступа к последовательном
#+  порту, мы восстановили эти права в предыдущее состояние.

rm /var/lock/LCK..$SERPORT   #  Удаляем файл блокировки последовательного
                             #+ порта. Зачем?

exit $?

# Упражнения:
# ---------
# 1) Необходимо проверить является ли пользователь, вызывающий сценарий -
#+   root.
# 2) Прежде чем убить процесс, проверьте, запущен ли этот процесс.
# 3) Напишите альтернативную версию этого сценария на основе "fuser":
#+      if [ fuser -s /dev/modem ]; then ...
```

Пример 15-15. Версия *rot13*

```
#!/bin/bash
# Версия "rot13" использующая 'eval'.
# Сравните с "rot13.sh".

setvar_rot_13()              # Перестановка элементов "rot13"
{
  local varname=$1 varvalue=$2
  eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
}


setvar_rot_13 var "foobar"   # Запускаем "foobar" через rot13.
echo $var                    # sbbone

setvar_rot_13 var "$var"     # Запускаем "sbbone" через rot13.
                             # Обратно в оригинальную переменную.
echo $var                    # foobar

# Пример Stephane Chazelas.
# Изменения автора документа.

exit 0
```

Вот еще один пример использования ***eval*** для *оценки* сложного выражения, это одна из самых ранних версий сценария игры тетрис YongYe.

```
eval ${1}+=\"${x} ${y} \"
```

В Примере A-53, с помощью ***eval***, элементы массива преобразуются в список команд.

Команда ***eval*** является старой версией *косвенной ссылки*.

```
eval var=\$$var
```

Команда **eval** может быть использована в *фигурных скобках расширения* для параметризации.

Команда **eval** рискованна, и, как правило, ее избегают, когда существует альтернатива. **eval $COMMANDS** выполняет содержимое **COMMANDS**, которое может содержать весьма неприятные сюрпризы, как **rm -rf ***. Запуск незнакомого кода с помощью **eval**, написанного неизвестными лицами - **опасно**.

set

Команда **set** изменяет значение внутренних переменных/опций сценария. Она может использоваться для переключения *опционных флагов*, которые помогают определить поведение сценария. Еще одно применение заключается в *сбросе позиционных параметров*, которые сценарий воспринимает как результат работы команды (**set `command`**). Сценарий может анализировать поля вывода команды.

Пример 15-16. Использование *set* позиционными параметрами

```
#!/bin/bash
# ex34.sh
# Сценарий "set-test"

# Сценарий вызывается с тремя параметрами командной строки,
# например, "sh ex34.sh one two three".

echo
echo "Позиционные параметры до  set \`uname -a\` :"
echo "Аргумент командной строки #1 = $1"
echo "Аргумент командной строки #2 = $2"
echo "Аргумент командной строки #3 = $3"


set `uname -a` # Устанавливает позиционные параметры в вывод
               # команды `uname -a`

echo
echo +++++
echo $_        # +++++
# Установленные в сценарии флаги.
echo $-        # hB
#                Аномальное поведение?
echo
```

```
echo "Позиционные параметры после  set \`uname -a\` :"
# $1, $2, $3, и т.д. переопределены в результате `uname -a`
echo "Поле #1 'uname -a' = $1"
echo "Поле #2 'uname -a' = $2"
echo "Поле #3 'uname -a' = $3"
echo \#\#\#
echo $_        # ###
echo

exit 0
```

Еще развлечение с позиционными параметрами.

Пример 15-17. Реверс позиционных параметров

```
#!/bin/bash
# revposparams.sh: Реверс позиционных параметров.
# Сценарий Dan Jacobson, со стилистическими изменениями автора документа.


set a\ b c d\ e;
#     ^      ^     Экранированные пробелы
#       ^ ^        Не экранированные пробелы
OIFS=$IFS; IFS=:;
#              ^   Сохранение старой IFS и установка новой.

echo

until [ $# -eq 0 ]
do          #      Шаг через позиционные параметры.
  echo "### k0 = "$k""     # До
  k=$1:$k;  #      Добавляет каждый позиционный параметр переменной цикла.
#     ^
  echo "### k = "$k""      # После
  echo
  shift;
done

set $k  #  Установка новых позиционных параметров.
echo -
echo $# #  Количество позиционных параметров.
echo -
echo

for i   #  Опускаем "in list" присваивания переменной -- i --
        #+ позиционных параметров.
do
  echo $i  # Выводим на экран новые позиционные параметры.
done

IFS=$OIFS  # Восстанавливаем IFS.

#  Вопрос:
#  Необходимо ли устанавливать новый IFS, разделитель внутренних полей,
#+ для правильной работы этого сценария?
#  Что случится, если этого не сделать? Попробуйте.
#  И, почему с помощью нового IFS -- двоеточия — в строке 17,
```

```
#+ производится добавление в переменную цикла?
#  Какова цель этого?

exit 0

$ ./revposparams.sh

### k0 =
### k = a b

### k0 = a b
### k = c a b

### k0 = c a b
### k = d e c a b

-
3
-

d e
c
a b
```

Вызов **set**, без каких-либо параметров или аргументов, просто перечисляет все переменные *окружения* и другие переменные, которые были инициализированы.

```
bash$ set
AUTHORCOPY=/home/bozo/posts
 BASH=/bin/bash
 BASH_VERSION=$'2.05.8(1)-release'
 ...
 XAUTHORITY=/home/bozo/.Xauthority
 _=/etc/bashrc
 variable22=abc
 variable23=xzy
```

С помощью опции **set --** содержимому переменной явно присваиваются позиционные параметры. Если переменная отсутствует, то **--** сбрасывает позиционные параметры.

Пример 15-18. Переназначение позиционных параметров

```
#!/bin/bash

variable="one two three four five"

set -- $variable
# Присвоение позиционных параметров содержимому "$variable".

first_param=$1
second_param=$2
shift; shift        # Сдвиг двух последних позиционных параметров.
```

```
# shift 2             То же работает
remaining_params="$*"

echo
echo "Первый параметр = $first_param"             # one
echo "Второй параметр = $second_param"            # two
echo "Остальные параметры = $remaining_params"    # three four five

echo; echo

# Снова.
set -- $variable
first_param=$1
second_param=$2
echo "Первый параметр = $first_param"             # one
echo "Второй параметр = $second_param"            # two

# =============================================================

set --
# Сброс позиционных параметров, если переменная не указана.

first_param=$1
second_param=$2
echo "Первый параметр = $first_param"             # (значение null)
echo "Второй параметр = $second_param"            # (значение null)

exit 0
```

См. также Пример 11-2 и Пример 16-56.

`unset`

Команда **`unset`** удаляет переменную оболочки, устанавливая в значение ***null***. Обратите внимание, что эта команда *не влияет на позиционные параметры*.

```
bash$ unset PATH

bash$ echo $PATH


bash$
```

Пример 15-19. "Сброс" переменной

```
#!/bin/bash
# unset.sh: Сброс переменной.

variable=hello                      #  Объявлена.
echo "variable = $variable"

unset variable                      #  Сброс.
                                    #  В данном конкретном случае,
                                    #+ эффект:   variable=
echo "(unset) variable = $variable"  #  $variable это null.
```

```
if [ -z "$variable" ]               #  Попробуйте проверить длину строки.
then
  echo "\$variable имеет нулевой размер."
fi

exit 0
```

В большинстве контекстов не объявление переменной и ее сброс - *эквивалентны*. Однако конструкция подстановки параметров *`${parameter:-default}`* это различает.

export

Команда **`export`** [4] создает переменные, доступные всем дочерним процессам запущенного сценария или оболочки. Одно из главных применений команды **`export`** - в *загрузке* файлов, инициализации и создании доступных *переменных* среды для последующих пользовательских процессов.

К сожалению, не является способом экспорта переменных обратно, в родительский процесс, в процесс, который их вызывал или вызвал сценарий или оболочку.

Пример 15-20. Применение *export* для передачи переменной во встроенный сценарий *awk*

```
#!/bin/bash

#  Еще одна версия сценария «column totaler» (col-totaler.sh), которая
#+ добавляет указанную колонку (чисел) в заданном файле.
#  Он использует окружающую среду для передачи переменной сценария в «awk»...
#+ и помещает сценарий awk в переменную.


ARGS=2
E_WRONGARGS=85

if [ $# -ne "$ARGS" ] # Проверка необходимых аргументов командной строки.
then
   echo "Usage: `basename $0` файл номер_колонки"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#=====Так же, как оригинальный сценарий, до этого момента=====#

export column_number
# Экспорт номера колонки в среду, поэтому она доступна для поиска.
```

```
# ------------------------------------------------
awkscript='{ total += $ENVIRON["column_number"] }
END { print total }'
# Да, переменная может содержать сценарий awk.
# ------------------------------------------------

# Теперь запустите сценарий awk.
awk "$awkscript" "$filename"

# Спасибо Stephane Chazelas.

exit 0
```

Она позволяет объявлять и экспортировать переменные одной операцией, типа **`export var1=xxx`**.

Однако, как отмечает Greg Keraunen, в некоторых ситуациях может иметь другой эффект, чем объявление переменной, а затем ее экспорт.

```
bash$ export var=(a b); echo ${var[0]}
(a b)

bash$ var=(a b); export var; echo ${var[0]}
a
```

При экспорте переменная может потребовать специальной обработки. См. Пример M-2.

`declare`, `typeset`

Команды **`declare`** и **`typeset`** объявляют и/или ограничивают переменные.

`readonly`

То же, что **`declare -r`**, устанавливает переменную только для чтения, или, по существу, константой. При попытке изменить переменную выводится сообщение об ошибке. Это аналог оболочки квалификатора типа `const` языка Си.

`getopts`

Этот мощный инструмент анализа аргументов командной строки передаваемых сценарию. Является аналогом внешней команды Bash **`getopt`** и библиотеки функции **`getopt`**, знакомых программистам Си. Позволяет передавать и объединения несколько опций [5] и связывать аргументы сценария (например **`scriptname -abc -e/usr/local`**).

Конструкция **`getopts`** использует две неявные переменные. `$OPTIND` - указатель аргумента (`OPTion INDex`) и аргумент `$OPTARG` (`OPTion ARGument`) привязки к опции

(необязательный). Двоеточие, после названия опции при объявлении тегов этой опции, является объединяющим аргументом.

Конструкция **getopts** обычно используется в цикле ***while***, при одновременной обработке параметроов и аргументы, затем увеличивает неявную переменную `$OPTIND` для дальнейшей передачи.

1. Аргументам, переданным из командной строки в сценарий, должен предшествовать дефис (**-**). Это префикс - который позволяет **getopts** распознавать аргументы командной строки в качестве *опции*. В самом деле, **getopts** не будет обрабатывать аргументы без префикса **-** и прекратит обработку опций на первом попавшемся аргументе без него.

2. Шаблон **getopts** незначительно отличается от стандартного цикла ***while***, в том, что он не имеет скобок условий.

3. Конструкция **getopts** является весьма функциональный заменой традиционной внешней команды **getopt** .

```
while getopts ":abcde:fg" Опции
# Первоначальное объявление.
# a, b, c, d, e, f, и g являются ожидаемыми опциями (флагами).
#  : после опции 'e' показывает, что с ней будет передан
#+ дополнительный   аргумент.
do
  case $Option in
    a ) # Что нибудь сделать с переменной 'a'.
    b ) # Что нибудь сделать с переменной 'b'.
    ...
    e)  # Что нибудь сделать с 'e', а так же с $OPTARG,
        # который является связанный аргументом, передаваемым с опцией 'e'.
    ...
    g ) # Что нибудь сделать с переменной 'g'.
  esac
done
shift $(($OPTIND - 1))
# Перемещение указателя аргумента далее.

# Все это не так сложно, как это выглядит <усмешка>.
```

Пример 15-21. Чтение опций/аргументов передаваемых в сценарий с помощью *getopts*.

```
#!/bin/bash
# ex33.sh: Упражнения с getopts и OPTIND
#          Сценарий изменен 10/09/03 Bill Gradwohl.


# Здесь мы увидим, как 'getopts' производит аргументы командной строки
#+ сценария.
# Аргументы анализируются как 'опции' (флаги) и связанные аргументы.
```

```
# Попробуйте вызвать этот сценарий с:
#   'scriptname -mn'
#   'scriptname -oq qOption' (qOption может быть произвольная строка.)
#   'scriptname -qXXX -r'
#
#   'scriptname -qr'
#+    -Неожиданный результат, принимает «r» в качестве аргумента для опции "q"
#   'scriptname -q -r'
#+    - Неожиданный результат, как и выше
#   'scriptname -mnop -mnop'  -Неожиданный результат
#   (OPTIND это ненадежное начало, из которого выходит опция.)
#
#  Если параметр ожидает аргумент ("флаг:"), то он будет захватывать все
#+ последующее в командной строке.

NO_ARGS=0
E_OPTERROR=85

if [ $# -eq "$NO_ARGS" ]    # Сценарий вызыван с отсутствующими аргументами
                            #+ командной строки?
then
  echo "Usage: `basename $0` опции (-mnopqrs)"
  exit $E_OPTERROR          # Выход и объяснение использования.
                            # Использование: scriptname -опции
                            # Примечание: тире (-) необходимо
fi


while getopts ":mnopq:rs" Option
do
  case $Option in
    m     ) echo "Сценарий #1: опция -m-   [OPTIND=${OPTIND}]";;
    n | o ) echo "Сценарий #2: опция -$Option-   [OPTIND=${OPTIND}]";;
    p     ) echo "Сценарий #3: опция -p-   [OPTIND=${OPTIND}]";;
    q     ) echo "Сценарий #4: опция -q-\
                  с аргументами \"$OPTARG\"   [OPTIND=${OPTIND}]";;
    #  Обратите внимание, что опция «q» должна иметь связанный аргумент, в
    #+ противном случае она, по умолчанию, понижается.
    r | s ) echo "Сценарий #5: option -$Option-";;
    *     ) echo "Не выбранный вариант.";;   # Умолчание.
  esac
done

shift $(($OPTIND - 1))
#  Уменьшает указатель аргумента,  поэтому указывает на следующий аргумент.
#  $1 теперь ссылается на первую не опцию поставляемую в командную строку, если
#+ таковая существует.

exit $?

#   Как утверждает Билл Gradwohl, "Механизм getopts позволяет указать:
#+  scriptname -mnop -mnop , но нет никакого надежного способа различить то,
#+  что пришло с помощью OPTIND".
#   Но обходные пути существуют.
```

Поведение сценария

source, **.** (команда «точка»)

Эта команда, при вызове из командной строки, выполняет сценарий. В сценарии **`source file-name`** загружается файл `file-name`. Файл *источник* (команда ***точка***) *импортирует*, добавляет, код в сценарий (то же эффект, как у директивы **`#include`** в программах Си). В результате, строки кода «источника», как будто бы физически, присутствуют в теле сценария. Это полезно в ситуациях, когда несколько сценариев используют общие файлы данных или библиотечные функции.

Пример 15-22. "Включение" файла данных

```
#!/bin/bash
#  Обратите внимание, что этот пример должен вызываться bash, т.е.,
#+ bash ex38.sh, а не sh ex38.sh !

. data-file    # Загрузка data-file.
# Тот же эффект, как и у "source data-file", но более переносимый.

#  Файл "data-file" должен находиться в текущей рабочей директории,
#+ поскольку он вызывается его основным именем.

# Теперь давайте сошлемся на некоторые данные из этого файла.

echo "variable1 (из data-file) = $variable1"
echo "variable3 (из data-file) = $variable3"

let "sum = $variable2 + $variable4"
echo "Сумма variable2 + variable4 (из data-file) = $sum"
echo "message1 (из data-file) это \"$message1\""
#         Экранирование кавычек   ^          ^
echo "message2 (из data-file) это \"$message2\""

print_message Это сообщение выводит на экран функцию data-file.


exit $?
```

Файл `data-file` в Примере 15-22, выше, должен находиться в той же директории .

```
# Эти данные файла загружаются сценарием.
# Файл может содержать переменные, функции, и т.п.
# Он загружается командами 'source' или '.' из оболочки сценария.

# Объявляются несколько переменных.

variable1=23
variable2=474
variable3=5
variable4=97

message1="Привет из *** строка $LINENO *** данных файла!"
message2="Пока достаточно. До свидания."

print_message ()
{   # Вывод на экран любого, передаваемого ему, сообщения.
```

```
  if [ -z "$1" ]
  then
    return 1 # Ошибка, при отсутствии аргумента.
  fi

  echo

  until [ -z "$1" ]
  do             # Шаг через аргументы, передаваемые функции.
    echo -n "$1" # Вывод на экран всех аргументов одновременно, подавляя
                 #+переводы строк.
    echo -n " "  # Вставляются пробелы между словами.
    shift        # Далее.
  done

  echo

  return 0
}
```

Если файл-*источник* сам является исполняемым сценарием, он будет запущен, затем возвращается управление сценарию, вызвавшему его. *Источник* исполняемого сценария может использовать для этого **`return`**.

Аргументы могут (не обязательно) передаваться файлу-источнику в качестве *позиционных параметров*.

```
source $filename $arg1 arg2
```

Это возможно даже для сценария, который сам является источником, хотя это, наверное, не имеет практического применения.

Пример 15-23. Пример (бесполезного) сценария, который сам является источником

```
#!/bin/bash
# self-source.sh: сценарий, «рекурсивно» являющийся источником для сам себя.
# Из "Stupid Script Tricks," Том II.

MAXPASSCNT=100    # Максимальное количество выполняемых проходов.

echo -n  "$pass_count  "
#  Первый выполняемый проход, только выводит на экран два пробела,
#+ т.к. $pass_count все еще не объявлена.

let "pass_count += 1"
#  Предполагается, что не инициализированная переменная $pass_count
#+ может быть увеличена с первого раза.
#  Это работает в Bash и pdksh, но
#+ обладает не переносимым (и возможно опасным) поведением.
#  Лучше до увеличения $pass_count присвоить ей 0.

while [ "$pass_count" -le $MAXPASSCNT ]
do
  . $0   # Сценарий, являющийся "источником", вызывает сам себя.
         # ./$0 (настоящая рекурсия) здесь не работает. Почему?
done
```

```
#  То, что здесь происходит, на самом деле не является  рекурсией,
#+ так как сценарий эффективно «расширяет» сам себя, т.е. создает
#+ новый блок кода с каждым проходом цикла «while» в строке 20, с каждым
#+ «источником».
#
#  Конечно, сценарий определяет каждый новый 'источник' строки "#!"
#+ как комментарий, а не как начало нового сценария.

echo

exit 0   # Чистым эффектом является счет от 1 до 100.
         # Очень впечатляет.

# Упражнение:
# --------
# Напишите сценарий, который использует этот трюк для чего-то полезного.
```

exit

Безоговорочно завершает сценарий. [6] Команда **exit** может при необходимости принять целочисленный аргумент, который возвращается в оболочку, как статус выхода сценария. Это хорошая практика, для завершения всех простых сценариев **exit 0**, указывающим на успешное выполнение.

Если сценарий завершается **exit** отсутствующего аргумента, то статусом выхода сценария является статус выхода последней команды, выполненной в сценарии, не считая **exit**. Это эквивалентно **exit $?**.

Команда **exit** может также использоваться для завершения subshell.

exec

Встроенная оболочка заменяющая текущий процесс на указанную команду. Обычно, когда оболочка встречает команду, то она *порождает* дочерний процесс для ее выполнения. С помощью встроенной **exec**, оболочка не порождает дочерний процесс, а заменяет оболочку командой *exec*. При использовании в сценарии, она она осуществляет принудительный выход из сценария, при завершении работы команды **exec**. [7]

Пример 15-24. Эффект *exec*

```
#!/bin/bash

exec echo "Выход \"$0\" на строке $LINENO."   # Здесь выход из сценария.
# $LINENO это внутренняя переменная Bash, устанавливающая номер строки.

# ---------------------------------
# Остальные строки не выполняются.
```

```
echo "Это echo ничего не выводит."

exit 99                         #  Сценарий здесь не выходит.
                                #  Проверьте значение exit, после завершения
                                #+ сценария, командой 'echo $?'.
                                #  Оно *не* будет 99.
```

Пример 15-25. Сценарий который порождает сам себя

```
#!/bin/bash
# self-exec.sh

# Примечание: Установите права этого сценария 555 или 755,
#       затем вызовите ./self-exec.sh или sh ./self-exec.sh.

echo

echo "Эта строка появляется в сценарии ОДНАЖДЫ, и не исчезает."
echo "PID этого сценария по-прежнему $$."
#     Показывает, что subshell не порождается.

echo "=================== Нажмите Ctl-C для выхода ==================="

sleep 1

exec $0   #  Появляется другой экземпляр этого же сценария,
          #+ который заменяет предыдущий.

echo "Эта строка не будет выведена!"  # Почему не будет?

exit 99                             # Здесь нет выхода!
                                    # Код выхода не 99!
```

Exec также служит для переназначения файловых дескрипторов. Например, **exec <zzz-file** заменяет `stdin` файлом *zzz-file*.

Опция *-exec* для поиска это НЕ то же самое, что встроенная команда **exec**.

shopt

Эта команда позволяет изменять *опции оболочки* на лету (См. Пример 25-1 и Пример 25-2). Она часто появляется в начальных файлах Bash, но может использоваться и в сценариях. Необходима версия Bash 2 или более поздняя.

```
shopt -s cdspell
# Позволяет незначительные ошибки в написании имен директорий с 'cd'
# Опция -s установка, -u сброс.
```

```
cd /hpme  # Упс! Опечатка '/home'.
pwd       # /home
          # Оболочка исправляет опечатку.
```

`caller`

Ввод команды **`caller`** внутрь ***функции*** выводит в `stdout` информацию о том, кто *вызвал* эту функции.

```
#!/bin/bash

function1 ()
{
  # Внутри function1 ().
  caller 0   # Расскажет нам о ней.
}

function1    # Строка 9 сценария.

# 9 main test.sh
# ^                 Номер строки из которой вызвана функция.
#   ^^^^            Вызвана из "main" (основной) части сценария.
#        ^^^^^^^    Имя вызывающего сценария.

caller 0     # Не имеет никакого эффекта, потому что она не внутри функции.
```

Команда **`caller`** также может возвращать информацию о *вызывающем* из сценария ***источника*** в другом сценарии. Аналогичная функция, это "вызов подпрограммы".

Эта команда может оказаться полезной при отладке.

Команды

`true`

Команда, которая возвращает *успешный* (*нулевой*) ***статус выхода***, но ничего не делает.

```
bash$ true
bash$ echo $?
0
```

```
# Бесконечный цикл
while true   # псевдоним для ":"
do
   операция-1
   операция-2
   ...
   операция-n
   # Необходим способ прерывания цикла или сценарий зависает.
done
```

`false`

Команда, возвращает статус *неудачного* выхода, но ничего не делает.

```
bash$ false
bash$ echo $?
1
```

```
# Проверка "ложности"
if false
then
  echo "ложная оценка \"истинно\""
else
  echo "ложная оценка \"ложно\""
fi
# ложная оценка "ложно"


# Ложный цикл (null цикл)
while false
do
   # Следующий код не выполняется.
   операция-1
   операция-2
   ...
   операция-n
   # Ничего не произошло!
done
```

`type` [команда]

Подобна внешней команде **`which`**, ***`type cmd`*** определяет "***`cmd`***". В отличии от **`which`**, **`type`** это встроенная команда Bash. Полезная опция `-a` для **`type`** определяет *`keywords`* и *`builtins`*, а также местонахождение системных команд с одинаковыми именами.

```
bash$ type '['
[ это встроенная в оболочку команда
bash$ type -a '['
[  это встроенная в оболочку команда
 [ это /usr/bin/[


bash$ type type
type  это встроенная в оболочку команда
```

Команда **`type`** может быть полезна для проверки, существования определенной команды.

`hash` [команды]

Записывает названия путей указанных команд - в *хэш таблицы* оболочки [8], - поэтому оболочке или сценарию уже не надо искать **`$PATH`** при последующих вызовах этих команд. Когда **`hash`** вызывается без аргументов, она просто перечисляет хэшированные команды. Опция `-r` сбрасывает хэш-таблицу.

`bind`

Встроенная команда **`bind`** отображает или изменяет привязку клавиш *readline* [9].

`help`

Краткое описание использования оболочки встроенной команды. Это аналог **`whatis`**, только для внутренних команд. Вывод справочной информации получил необходимые обновления в выпуске версии 4 Bash.

```
bash$ help exit
exit: exit [n]
    Выход оболочки со статусом N.  Если N опущено, статусом выхода будет
статус последней выполненной команды.
```

Примечания

[1] Как указывает Nathan Coulter, "в то время, как порождение процесса является простой операцией, выполнение новой программы, в недавно порожденном дочернем процессе, увеличивает затраты".

[2] Исключением является команда **`time`**, являющаяся, в официальной документации Bash, ключевым словом (keyword).

[3] Обратите внимание, что ***`let`*** не может быть использована для установки строковых переменных.

[4] Информация *`Export`* делает доступность в более общем контексте. Смотрите также **`scope`**.

[4] *Опция* является аргументом, который выступает в качестве флага включения или выключения поведения сценариев. Аргумент, связанный с определенной опцией (флагом), указывает поведение, в зависимости от того включена или выключена опция (флаг).

[5] Технически, **`exit`** только завершает процесс (или оболочку) в котором он запущен, а ***не*** *родительский* процесс.

[6] Если **`exec`** используется для переназначения *файловых дескрипторов*.

[7] ***Хэширование*** — это способ поиска созданных ключей для данных, хранящихся в таблице. *Сами элементы данных*, «используются» для создания ключей, с помощью одного из ряда простых математических *алгоритмов* (способов).

Преимуществом *хэширования* является скорость. Недостатком - возможность *коллизии*, когда одному ключу соответствует более одного элемента данных.

Примеры хэширования смотрите Пример A-20 и Пример A-21.

[9] Библиотека ***readline*** используется Bash для чтения ввода в интерактивной оболочке.

15.1. Команды управления задачами

Некоторые из команд управления задачами принимают *идентификатор задания*, как аргумент. См. таблицу в конце главы.

`jobs`

Перечисляет задания, запущенные в фоновом режиме, выдавая *номер задания*. Не так полезна, как **`ps`**.

В ней слишком легко спутать *задания* и *процессы*. Некоторые *встроенные* команды, такие как **`kill`**, **`disown`** и **`wait`**, принимают в качестве аргумента либо номер задания, либо номер процесса. Команды **`fg`**, **`bg`** и **`jobs`** принимают только номер задания.

```
bash$ sleep 100 &
[1] 1384

bash $ jobs
[1]+  Запущена                 sleep 100 &
```

"`1`" это номер задания (задания поддерживаемого текущей оболочкой). "`1384`" это **PID** или *номер ID процесса* (процесса поддерживаемого системой). Что бы убить задание/процесс, работает **`kill %1`**или **`kill 1384`**.

Спасибо S.C.

`disown`

Удаление заданий из таблицы активных заданий оболочки.

`fg`, `bg`

Команда **`fg`** переключает задание запущенное в фоновом режиме в основной режим. Команда **`bg`** перезапускает приостановленное задание и выполняет его в фоновом режиме. Если номер задания не указан, команды **`fg`** или **`bg`** работают с текущими запущенными заданиями.

`wait`

Приостанавливает выполнение сценария до прекращения всех заданий, запущенных в фоновом режиме, или до завершения номера задания или ID процесса, указанного как параметр. Возвращает статус выхода ждущей команды.

Можно использовать команду **`wait`** для предотвращения выхода сценария до окончания фонового задания (создающего процесс сироту).

Пример 15-26. Ожидание завершения перед продолжением

```
#!/bin/bash

ROOT_UID=0   # Только пользователь с $UID 0 имеет права root.
E_NOTROOT=65
E_NOPARAMS=66

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Запускать этот сценарий должен root."
  # "Запустите шутку на сон грядущий."
  exit $E_NOTROOT
fi

if [ -z "$1" ]
then
  echo "Usage: `basename $0` искомая строка"
  exit $E_NOPARAMS
fi


echo "Обновление базы данных 'locate'..."
echo "Это может занять некоторое время."
updatedb /usr &     # Должно запускаться из-под root.

wait
# Остальная часть сценария не запускается до завершения «updatedb».
# Нужно, чтобы база данных обновилась до нахождения имени файла.

locate $1

#  Без команды «wait», в худшем случае, сценарий завершится,
#+ в то время как «updatedb» все еще будет работать,
#+ оставаясь процессом сиротой.

exit 0
```

При необходимости **`wait`** может принимать идентификатор задания в качестве

аргумента, например, ***wait%1*** или ***wait $PPID***. [1] См. таблицу id заданий.

Команда запущенная в сценарии в фоновом режиме с амперсандом (**&**), может вызывать зависание сценария до нажатия **ENTER**. Это, кажется, происходит с командами, которые пишут в `stdout`. Это главная неприятность.

```
#!/bin/bash
# test.sh

ls -l &
echo "Done."
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
 _
```

> Как это объясняет Walter Brameld IV:
>
> Насколько я понимаю, такие сценарии на самом деле не зависают. Это только так кажется, потому что фоновая команда записывает текст в консоли после приглашения. У пользователя складывается впечатление, что запрос никогда не выведется на экран. Вот последовательность событий:
>
> 1. Сценарий запускает фоновую команду.
> 2. Сценарий завершается.
> 3. Оболочка отображает приглашение.
> 4. Фоновая команда продолжает выполняться и пишет текст в консоль.
> 5. Фоновая команда завершается
> 6. Пользователь не видит приглашения в нижней части вывода и думает, что сценарий завис.

Размещение **wait** после фоновой команды, как представляется, это исправляет.

```
#!/bin/bash
# test.sh

ls -l &
echo "Done."
wait
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
```

Перенаправление вывода команды в файл или даже в */dev/null* также исправляет эту проблему.

suspend

Имеет эффект аналогичный **Control-Z**, но она приостанавливает оболочку (родительский процесс оболочки должен возобновиться в указанное время).

logout

Выход из оболочки, дополнительно указывается статус выхода.

times

Предоставляет статистические данные о прошедшем системном времени выполняющихся команд, в следующем виде:

```
0m0.020s 0m0.020s
```

Эта возможность имеет сравнительно ограниченную ценность, так как она не является общей для профиля и базовых сценариев оболочки.

kill

Принудительное завершение процесса, отправка процессу соответствующего сигнала о *прекращении* (см. Пример 17-6).

Пример 15-27. Сценарий, который убивает сам себя

```
#!/bin/bash
# self-destruct.sh

kill $$  # Этот сценарий убивает свой собственный процесс.
         # Напомню, что "$$" это PID процесса.

echo "Эта строка не будет выведена."
# Вместо этого оболочка отправляет в stdout сообщение «Прекращено».

exit 0   # Нормальный выход? Нет!

#  После того, как этот сценарий преждевременно завершается,
#+ какой возвращается статус выхода?
#
# sh self-destruct.sh
# echo $?
# 143
#
# 143 = 128 + 15
#             сигнал TERM
```

Kill -l список всех сигналов (как в файле /usr/include/asm/signal.h). **kill -9** это ***обязательный***

kill, завершающий процесс, который упорно отказывается умирать с простым **kill**. Иногда работает **kill -15**. Процесс-*зомби*, то есть, дочерний процесс, который прекращен, но, как родительский процесс, не убит (пока), и не может быть убит пользователем -- не льзя убить то, что уже мертво -- а **init**, обычно, рано или поздно вычистит его.

killall

Команда **killall** убивает запущенный процесс не по ***ID*** процесса, а по имени. Если существует несколько экземпляров какой-то запущенной команды, то ***killall*** этой командой прекращает все экземпляры.

Это относится к команде **killall** в `/usr/bin`, а не **killall** сценария в `/etc/rc.d/init.d`.

command

Директива **command** *отключает* псевдонимы и функции для команды сразу же после нее.

```
bash$ command ls
```

Это одна из трех директив оболочки, которые обрабатывают действия команд сценария. Остальные **builtin** и **enable**.

builtin

Вызывает внутреннюю **BUILTIN_COMMAND** запускающую *BUILTIN_COMMAND* как встроенную оболочку, временно сразу отключает функции и внешние системные команды с таким же названием.

enable

Включает или отключает встроенную команду оболочки. Например, *enable -n kill* отключает встроенную команду оболочки **kill,** поэтому, когда Bash впоследствии сталкивается с ***kill***, он вызывает внешнюю команду `/bin/kill`.

Опция `-a` *включает* список всех встроенных команд оболочки, с/без указания их включения. Опция `-f filename` позволяет *включить* загрузку встроенной команды в

качестве модуля общей библиотеки (DLL) из правильно скомпилированного объектного файла. [2].

autoload

Это порт автозагрузчика ***ksh*** в Bash. Функция с объявленной **autoload** будет загружаться из внешнего файла при его первом вызове. [3] Это экономит ресурсы системы.

Обратите внимание, что *autoload* не является частью установленного ядра Bash. Она должна быть загружена с помощью *enable -f* (см. выше).

Таблица 15-1. Идентификаторы заданий

Обозначение	Значение
%N	Номер задания [N]
%S	Вызов (командной строкой) задания начинающегося строкой *S*
%?S	Вызов (командной строкой) задания содержащего в себе строку *S*
%%	"текущее" задание (последняя задание остановленное в основном режиме или запущенное в фоновом режиме)
%+	"текущее" задание (последняя задание остановленное в основном режиме или запущенное в фоновом режиме)
%-	Последнее задание
$!	Последний фоновый процесс

Примечания

[1] Это относится только к *дочерним процессам*.

[2] Источник Си для нескольких загружаемых встроенных команд, как правило, находится в директории `/usr/share/doc/bash-?.??/functions` .

Обратите внимание, что опция `-f`, в **enable** *не переносима* на все системы.

[3] Тот же самый эффект, как у **autoload** может быть достигнут с **typeset -fu.**

Глава 16. Внешние фильтры, программы и команды

Содержание

16.7. Команды управления терминалом
16.8. Математические команды
16.9. Различные команды

Команды стандарта UNIX делают сценарии более универсальными. Сила сценариев в системных командах и директивах оболочки с простыми конструкциями программирования.

16.1. Основные команды

Первая команда, которую узнает новичок

`ls`

Основная файловая команда «*`list`*». Нельзя недооценивать силу этой скромной команды. Например с помощью опции рекурсии `-R`, **`ls`** выводит древовидный список структуры директории. Другими полезными опциями являются сортировка списка по размеру файла `-S`, сортировка по времени изменения файлов `-t`, сортировка по номерам (числовым) версий встроенных в именах файлов `-v`, [1] `-b` выводит экранирующие символы, а `-i` выводит иноды файлов (см. Пример 16-4).

```
bash$ ls -l
-rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter10.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter11.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter12.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter1.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter2.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter3.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Chapter_headings.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Preface.txt


bash$ ls -lv
 total 0
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Chapter_headings.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Preface.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter1.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter2.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter3.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter10.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter11.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter12.txt
```

При попытке вывода несуществующего файла команда **`ls`** возвращает *не нулевой* статус выхода

```
bash$ ls abc
ls: abc: Нет такого файла или директории


bash$ echo $?
2
```

Пример 16-1. Создание оглавления записи CDR диска с помощью *ls*

```
#!/bin/bash
# ex40.sh (burn-cd.sh)
# Сценарий автоматической записи CDR.


SPEED=10         #  Возможно использовать более высокую
                 #+ скорость,поддерживаемую вашим оборудованием.
IMAGEFILE=cdimage.iso
CONTENTSFILE=contents
# DEVICE=/dev/cdrom     Для старых версий cdrecord
DEVICE="1,0,0"
DEFAULTDIR=/opt  # Директория содержащая данные для записи.
                 # Убедитесь, что она существует.
                 # Упражнение: Добавьте проверку этого.

# Используется пакет Joerg Schilling "cdrecord":
# http://www.fokus.fhg.de/usr/schilling/cdrecord.html

#  Если этот сценарий вызывается обычным пользователем, возможно,
#+ потребуется suid cdrecord chmod u+s /usr/bin/cdrecord, как у root.
#  Конечно, это создаст дыру в безопасности, хотя относительно небольшую.

if [ -z "$1" ]
then
  IMAGE_DIRECTORY=$DEFAULTDIR
  # Директория по умолчанию, если не задана из командной строки.
else
    IMAGE_DIRECTORY=$1
fi

# Создание файла "Содержание".
ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
# Опция "l" задает "полный" список файла.
# Опция "R" делает список рекурсивным.
# Опция "F" указывает типы файлов (директории оканчиваются символом /).
echo "Создается содержание."

# Создаем файл образа для прожига на CDR.
mkisofs -r -o $IMAGEFILE $IMAGE_DIRECTORY
echo "Создается образ файловой системы ISO9660 ($IMAGEFILE)."

# Прожиг CDR.
echo "Записывается диск."
echo "Пожалуйста подождите, это займет какое-то время."
wodim -v -isosize dev=$DEVICE $IMAGEFILE
#  В новых дистрибутивах Linux, утилита "wodim" обладает
#+ функциональностью "cdrecord."
exitcode=$?
echo "Exit code = $exitcode"

exit $exitcode
```

`cat`, `tac`

`cat`, акроним *concatenate* (объединение), перечисляет файлы в `stdout`. В

комбинации с перенаправлением (**>** или **>>**), она обычно используется для объединения файлов.

```
# Использование 'cat'
cat filename                         # Перечисление файла.

cat file.1 file.2 file.3 > file.123   # Объединяет три файла в один.
```

Опция `-n` команды **`cat`** вставляет нумерацию перед всеми строками целевого файла(ов). Опция `-b` нумерует только ***не*** пустые строки. Опция `-v` выводит непечатаемые символы, с помощью записи **^**. Опция `-s` сжимает несколько последовательных пустых строк в одну пустую строку.

См. Пример 16-28 и Пример 16-24.

В ***конвейере*** она может быть более эффективно *перенаправление* `stdin` в файл, вместо ***`cat file`***.

```
cat filename | tr a-z A-Z

tr a-z A-Z < filename
#  Тот же эффект, но запускает на один процесс
#+ меньше, а также освобождает от конвейера.
```

`tac`, это **`cat`** наоборот, перечисляет файл с конца.

`rev`

переворачивает каждую строку файла и выводит на `stdout`. Эффект не такой же, как у **`tac`**, она сохраняет порядок следования строк, но переворачивает каждую наоборот (отображая зеркально).

```
bash$ cat file1.txt
This is line 1.
 This is line 2.


bash$ tac file1.txt
This is line 2.
 This is line 1.


bash$ rev file1.txt
.1 enil si sihT       # Задом на перед
 .2 enil si sihT
```

`cp`

Команда копирования файлов. **`cp file1 file2`** копирует `file1` в `file2`, если `file2` не существует — *создает* его (См. Пример 16-6).

Особенно полезны: флаг архивирования (для копирования дерева директорий) `-a`, флаг обновления `-u` (предотвращающий перезапись одинаково названных новых файлов), и флаги рекурсии `-r`/`-R`.

```
cp -u source_dir/* dest_dir
#  "Синхронизирует" dest_dir с source_dir
#+  путем копирования всех новых и ранее не существующих
файлов.
```

mv

Команда перемещения файла. Эквивалентна комбинации **cp** и **rm**. Может использоваться для перемещения нескольких файлов директории, или даже переименования директории. Некоторые примеры использования **mv** в сценариях см. Пример 10-11 и Пример A-2.

При использовании в не интерактивных сценариях, **mv** принимает опцию обхода пользовательского ввода `-f` (*force*).

Когда директория перемещается в существующую директорию, то она становится поддиректорией в директории назначения.

```
bash$ mv source_directory target_directory

bash$ ls -lF target_directory
total 1
 drwxrwxr-x    2 bozo  bozo      1024 May 28 19:20
source_directory/
```

rm

Удаление (стирание) файла или файлов. Опция `-f` принудительно удаляет даже файлы ***readonly*** и полезна для обхода пользовательского ввода в сценарии.

Команда `rm`, сама по себе, не в состоянии удалять имена начинающиеся с тире. Почему? Потому что `rm` принимает тире префикса файла в качестве опции.

```
bash$ rm -badname
rm: неправильная опция -- b
 Попробуйте `rm --help' для получения справки.
```

Одним из решений является начать с " **--** " (флага *окончания опций*).

```
bash$ rm -- -badname
```

```
```

Другой способ заключается в предварении удаляемого имени файла ***точкой со слэш.***

```
bash$ rm ./-badname
```

При использовании флага рекурсии `-r`, эта команда удаляет все файлы в дереве директорий начиная с текущей директории. Неосторожное **`rm -rf *`** может уничтожить большой кусок структуры директорий.

`rmdir`

Удаление директории. Для успешного выполнения этой команды директория должна быть пустой (не содержать файлы - включая и *файлы с точкой* [2]).

`mkdir`

Создание новой директории. Например, **`mkdir -p project/programs/December`** создает именованную директорию. Опция `-p` автоматически создает все необходимые родительские директории.

`chmod`

Изменяет атрибуты существующего файла или директории (См. Пример 15-14).

```
chmod +x filename
# Делает "filename" исполняемым для всех пользователей.

chmod u+s filenamecattacrev
# Устанавливает бит "suid" в атрибутах прав "filename".
# Обычный пользователь может выполнять "filename" с правами владельца.
# (Это не относится к сценариям оболочки.)


chmod 644 filename
#  Устанавливает "filename" доступным для чтения/записи владельцу, чтение
#+ для всех остальных (восьмизначное).

chmod 444 filename
#  Устанавливает "filename" для всех только для чтения.
#  Изменение файла (например, текстовым редактором)
#+ пользователем, не являющемуся владельцем файла (кроме  root),
#+ не допускается, и даже владелец файла должен принудительно сохранять
#+ файл при его изменении.
#  Те же ограничения применяются для удаления файла.


chmod 1777 directory-name
#  Разрешается всем читать, записывать, и выполнять в этой директории,
#+ но также устанавливается «sticky bit».
#  Это означает, что только владелец директории,
```

```
#+ владелец файла, и, конечно же, root может удалять
#+ любой обычный файл в этой директории.

chmod 111 directory-name
#  Дает всем права на выполнение в этой директории.
#  Это означает, что можно выполнять и ЧИТАТЬ файлы в этой директории
#+ (право на выполнение обязательно включает в себя право на чтение,
#+ потому что нельзя выполнить файл, не будучи в состоянии прочитать его).
#  Но нельзя перечислять файлы или искать их командой «find».
#  Эти ограничения не относятся к root.

chmod 000 directory-name
#  Нет разрешений на все в этой директории.
#  Нельзя в ней читать, записывать или выполнять файлы.
#  Нельзя даже вывести  список файлов содержащихся в ней или «cd» в нее.
#  Но можно переименовывать (mv) директории
#+ или удалять их (rmdir) если они пусты.
#  Вы можете даже символически ссылаться на файлы в директории, но не
#+ можете читать, записывать или выполнять символические ссылки.
#  Эти ограничения не относятся к root.
```

chattr

Изменение атрибутов файла. Это аналогично ***chmod*** выше, но с другими опциями и другим синтаксисом вызова, и работает только в файловых системах *ext2/ext3*.

Одной из особенно интересных опций `chattr` является **i**. **chattr +i filename** помечает файл как ***неизменяемый***. Этот файл не может быть изменен, удален и нельзя делать ссылки на него, ***даже для root***. Этот атрибут может установить или удалить только *root*. Аналогичным образом опция помечает файл как "*только для добавления*".

```
root# chattr +i file1.txt


root# rm file1.txt

rm: удалить защищенный от записи файл `file1.txt'? y
 rm: не возможно удалить `file1.txt': Нет прав на операцию
```

Если файл имеет установленный атрибут **s** (безопасный), то при его удалении его блок перезаписывается двоичными нулями. [3]

Если файл имеет атрибут **u** (восстанавливаемый), то когда он будет удален, его содержимое еще можно извлечь (восстановить).

Если файл имеет установленный атрибут **c** (сжатие), то он будет автоматически сжиматься при записи на диск и распаковываться при чтении.

Атрибуты файла установленные **chattr** не показываются в списке файлов (**ls -l**).

`ln`

Создает ссылки на *уже существующие* файлы. "Ссылка", являющаяся ссылкой на файл, является альтернативным названием этого файла. Команда **`ln`** позволяет ссылаться на связанный файл более чем одним именем и является отличной альтернативой псевдонимам (см. Пример 4-6).

`ln` создает только ссылку, указатель на файл, размером всего в несколько байт.

Команда **`ln`** наиболее часто используется с флагом **`-s`**, символическая или "мягкая" ссылка. Преимущества использования флага `-s` в том, что он позволяет ссылаться на другие файловые системы или директории в них.

Синтаксис команды немного сложен. Например: **`ln -s oldfile newfile`** уже существующий *`oldfile`* ссылается на вновь создаваемую ссылку *`newfile`*.

Если файл `newfile` уже существовал ранее, результатом будет сообщение об ошибке.

Какой тип ссылки использовать?

Как объясняет John Macdonald:

Оба [вида ссылок] обеспечивают двойную ссылку -- если вы измените содержимое файла, использующего любое имя, ваши изменения повлияют на исходное имя и, либо на жесткое, либо на мягкое новое имя. Различия между ними видны при работе на более высоком уровне. Преимуществом ***жесткой*** ссылки является полная независимость от старого имени -- если вы удалите или переименуете старое имя, это не повлияет на жесткую ссылку, которая продолжит указывать на данные, в то время как оставшаяся мягкая ссылка, будет указывать на старое, не существующее, имя. Преимущество ***мягкой*** ссылки заключается в том, что она может ссылаться на другую файловую систему (как будто это просто ссылка на имя файла, а не фактические данные). И, в отличие от жесткой ссылки, символическая ссылка может ссылаться на директорию.

Ссылки дают возможность вызывать сценарий (или любой другой тип исполняемого файла) с несколькими именами, и этот сценарий будет работать так, как он был вызван.

Пример 16-2. Hello или Good-bye

```
#!/bin/bash
# hello.sh: Выводит "hello" или "goodbye"
#+          в зависимости от вызова.

# Сделаем ссылку на этот сценарий в текущей рабочей директории ($PWD):
#    ln -s hello.sh goodbye
# Теперь попробуем вызвать этот сценарий обоими способами:
# ./hello.sh
```

```
# ./goodbye


HELLO_CALL=65
GOODBYE_CALL=66

if [ $0 = "./goodbye" ]
then
  echo "Good-bye!"
  # Другие команды типа goodbye, в зависимости от обстоятельств.
  exit $GOODBYE_CALL
fi

echo "Hello!"
# Другие команды типа hello, в зависимости от обстоятельств.
exit $HELLO_CALL
```

man, **info**

Команды доступа к справочным и информационным страницам системных команд и установленных утилит. Насколько это возможно, *информационные* страницы обычно содержат более подробные описания, чем *справочные* страницы.

Были различные попытки «автоматизировать» написание *справочных* страниц. Сценарий, который делает предварительный первый шаг в этом направлении - Пример A-39.

Примечания

[1] Опция `-v` также по порядку сортирует имена файлов по префиксам в *верхнем* и *нижнем* регистрах.

[2] *Файл с точкой* это файл, имя которого начинается с *точки*, вот так `~/.Xdefaults`. Такие файлы не отображаются в обычных листингах **ls**, (хотя **ls -a** показывает их), и они не могут быть удалены путем случайного **rm -rf ***. Файлы с точкой обычно используются как установочные и конфигурационные файлы в домашнем каталоге пользователя.

[3] Эта особенность еще не полностью реализована в версиях файловой системы ext2/ext3. Проверьте документацию **find** вашего дистрибутива Linux.

16.2. Сложные команды

Команды для более продвинутых пользователей

find

- **exec *COMMAND* \;**

Выполняет *COMMAND* для каждого файла, который соответствует **find**. Последовательность команд завершается **;** ("**;**" ***экранируется*** при передачи ее оболочкой в find буквально, без интерпретации ее, как специального символа).

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
 /home/bozo/misc/irmeyc.txt
 /home/bozo/test-scripts/1.txt
```

Если *COMMAND* содержит **{}**, то **find** подставляет "**{}**" вместо полного пути имени выбранного файла.

```
find ~/ -name 'core*' -exec rm {} \;
# Удаляет из всех файлов содержимое core из домашней директории

#+ пользователя.


find /home/bozo/projects -mtime -1
#                               ^   Обратите внимание на знак минус!
#  Список всех файлов дерева директории /home/bozo/projects
#+ которые были изменены вчера (текущий_день минус 1).
#
find /home/bozo/projects -mtime 1
#  То же, но изменены *точно* один день назад.
#
#  mtime = время последнего изменения целевого файла
#  ctime = время последнего изменения статуса («chmod» или иным
#          образом)
#  atime = время последнего обращения

DIR=/home/bozo/junk_files
find "$DIR" -type f -atime +5 -exec rm {} \;
#                           ^          ^^
#  Фигурные скобки являются заполнителем для выводимого пути имени "find."
#
#  Удаляются все файлы в "/home/bozo/junk_files"
#+ к которым не обращались *хотя бы* 5 дней (знак плюс ... +5).
#
#  где "-type filetype"
#  f = обычный файл
#  d = директория
#  l = символическая ссылка, и т.д.
#
#  (Справочные и информационные страницы 'find' содержат список опций.)


find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-
9]*' {} \;

# Ищем все IP адреса (xxx.xxx.xxx.xxx) в файлах директории /etc.
```

```
# Там указаны лишние. Они быть отфильтрованы?

# Возможно:

find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
| grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
#
#  [:digit:] это один из классов символов в стандарте POSIX 1003.2

# Спасибо Stéphane Chazelas.
```

Опцию `-exec` в **`find`** не следует путать с встроенной командой оболочки ***`exec`***.

Пример 16-3. *Badname*, исключение имен файлов в текущей директории содержащих плохие символы и пробелы.

```
#!/bin/bash
# badname.sh
# Удаление файлов в текущей директории содержащих плохие символы.

for filename in *
do
  badname=`echo "$filename" | sed -n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
# badname=`echo "$filename" | sed -n '/[+{;"\=?~()<>&*|$]/p'` то же работает.
# Удаляем файлы, содержащие такие гадости: + { ; " \ = ? ~ ( ) < > & * | $

  rm $badname 2>/dev/null
#             ^^^^^^^^^^^ Убирает сообщение об ошибке.
done

# Теперь позаботимся о файлах, содержащих все виды пробелов.
find . -name "* *" -exec rm -f {} \;
# Имя найденного пути файла которое _ищем_  заменяем на "{}".
# '\' гарантирует, что ';' интерпретируется буквально, как конец команды.

exit 0

#--------------------------------------------------------------------
# Команды ниже этой строки не будет выполнены, т.к. они после команды _exit_.

# Альтернатива сценарию выше:
find . -name '*[+{;"\\=?~()<>&*|$ ]*' -maxdepth 0 \
-exec rm -f '{}' \;
#  Опция "-maxdepth 0" гарантирует, что _find_ не будет искать
#+ в поддиректориях ниже $PWD.

# (Спасибо S.C.)
```

Пример 16-4. Удаление файла по номеру его *inode*

```
#!/bin/bash
# idelete.sh: Удаление файла по номеру его inode.

#  Это полезно, когда имя файла начинается с неправильного символа,
#+ такого как ? или -.

ARGCOUNT=1                  # Сценарию должен быть передан аргумент Filename.
E_WRONGARGS=70
E_FILE_NOT_EXIST=71
E_CHANGED_MIND=72

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_WRONGARGS
fi

if [ ! -e "$1" ]
then
  echo "Файл \""$1"\" не существует."
  exit $E_FILE_NOT_EXIST
fi

inum=`ls -i | grep "$1" | awk '{print $1}'`
# inum = номер inode (индекса узла) файла
# ---------------------------------------------------------------------
# Каждый файл имеет inode, запись, которая содержит информацию о
# его физическом адресе.
# ---------------------------------------------------------------------

echo; echo -n "Вы точно уверены, что хотите удалить\"$1\" (да/нет)? "
# Опция '-v' в 'rm' так же задает этот вопрос.
read answer
case "$answer" in
[nN]) echo "Передумали?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "Удаляется файл \"$1\".";;
esac

find . -inum $inum -exec rm {} \;
#                          ^^
#        Фигурные скобки это замещение
#+       текста выводимого "find."
echo "Файл "\"$1"\" удален!"

exit 0
```

Команда **find** также работает без опции -exec.

```
#!/bin/bash
#  Ищем suid файлы root.
#  Файл suid может указывать на дыру в безопасности,
#+ или даже на вторжение в систему.

directory="/usr/sbin"
# Можно так же попробовать /sbin, /bin, /usr/bin, /usr/local/bin, и т.п.
```

```
permissions="+4000"  # suid root (Опасно!)


for file in $( find "$directory" -perm "$permissions" )
do
  ls -ltF --author "$file"
done
```

См. Пример 16-30, Пример 3-4 и Пример 11-10 сценариев использующих **find**. Ее справочная страница обеспечивает детальное изучение этой сложной и мощной команды.

xargs

Фильтр для снабжения команды аргументами, а также инструмент для сборки самих команд. Она разбивает поток данных на небольшие куски для обработки фильтрами и командами. Считается, что это мощная замена для ***обратных кавычек***. В ситуациях, когда при *подстановке команд* происходит сбой, с ошибкой "слишком много аргументов", подстановка **xargs** часто работает. [1] Обычно **xargs** читает из потока `stdin` или из конвейера, но она также может задаваться *выводом из файла*.

По умолчанию командой для **xargs** является **echo**. Это означает, что данные, входящие посредством конвейера в **xargs**, могут иметь символы перевода строки и другие разделительные пробелы.

```
bash$ ls -l
total 0
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file1
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file2



bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file1 -rw-rw-r-- 1 bozo bozo 0
Jan...



bash$ find ~/mail -type f | xargs grep "Linux"
./misc:User-Agent: slrn/0.9.8.1 (Linux)
 ./sent-mail-jul-2005: hosted by the Linux Documentation Project.
 ./sent-mail-jul-2005: (Linux Documentation Project Site, rtf version)
 ./sent-mail-jul-2005: Subject: Criticism of Bozo's Windows/Linux article
 ./sent-mail-jul-2005: while mentioning that the Linux ext2/ext3 filesystem
 ... find
```

ls | xargs -p -l gzip сжимает каждый файл в текущей директории, по одному за раз, вызывая следующий перед каждой операцией.

Обратите внимание, что **xargs** обрабатывает аргументы, переданные ему, *по одному за раз*, последовательно.

```
bash$ find /usr/bin | xargs file
/usr/bin:          directory
```

```
/usr/bin/foomatic-ppd-options: выполняемый текст сценария perl
...
```

Интересна опция ***xargs*** `-n` *NN*, которая ограничивает число передаваемых аргументов *NN* .

ls | xargs -n 8 echo список файлов текущей директории в 8 колонках.

Другая полезная опция это `-0`, в комбинации с **find -print0** или **grep -lZ**. Она позволяет обрабатывать аргументы содержащие пробелы или кавычки.

```
find / -type f -print0 | xargs -0 grep -liwZ GUI |
xargs -0 rm -f

grep -rliwZ GUI / | xargs -0 rm -f
```

Любое из написанного выше удаляет любой файл, содержащий "GUI". *(Спасибо S.C.)*

Или:

```
cat /proc/"$pid"/"$OPTION" | xargs -0 echo
#  Форматирование вывода:  ^^^^^^^^^^^^^^^
#  Из исправленного сценария Han Holl' "get-commandline.sh"
#+ в главе "/dev и /proc".
```

Опция `-P` в ***xargs*** позволяет запускать процессы параллельно. Это ускоряет выполнение на машине с многоядерным процессором.

```
#!/bin/bash

ls *gif | xargs -t -n1 -P2 gif2png
# Конвертирует все файлы gif в текущей директории в png.

# Опции:
# =======
# -t    Вывод команды в stderr.
# -n1   Более 1 аргумента командной строки.
# -P2   Запуск двух процессов одновременно.

# Спасибо Roberto Polli, за вдохновение.
```

Пример 16-5. Logfile: Использование *xargs* для мониторинга системного журнала

```
#!/bin/bash

# Создание файла журнала в текущей директории
```

```
# из окончания файла /var/log/messages.

# Примечание: /var/log/messages должен быть общедоступным, на случай
# если этот сценарий будет вызываться обычным пользователем.
#         #root chmod 644 /var/log/messages

LINES=5

( date; uname -a ) >>logfile
# Время и имя машины
echo ------------------------------------------------------------ >>logfile
tail -n $LINES /var/log/messages | xargs | fmt -s >>logfile
echo >>logfile
echo >>logfile

exit 0

#  Примачание:
#  ----
#  Как поясняет Frank Wang,
#+ непарные кавычки (одинарные или двойные кавычки) в исходном файле
#+ могут не перевариться xargs.
#
#  Он предлагает следующее дополнение в строке 15:
#  tail -n $LINES /var/log/messages | tr -d "\"'" | xargs | fmt -s >>logfile



#  Упражнение:
#  --------
#  Измените этот сценарий для отслеживания изменений в /var/log/messages
#+ с интервалом в 20 минут.
#  Подсказка: С помощью команды "watch".
```

Как и в **find**, пара фигурных скобок выступает в качестве заполнителя заменяющей текст.

Пример 16-6. Копирование файлов текущей директории в другую

```
#!/bin/bash
# copydir.sh

#  Копирование (подробное) всех файлов текущей директории ($PWD)
#+ в директорию указанную в командной строке.

E_NOARGS=85

if [ -z "$1" ]   # Выход, если не задан аргумент.
then
  echo "Usage: `basename $0` директория для скопированного"
  exit $E_NOARGS
fi

ls . | xargs -i -t cp ./{} $1
#            ^^ ^^      ^^
#  -t опция "подробно" (выводит командную строку в stderr).
#  -i опция "замена строк".
#  {} является заполнителем вместо выводимого текста.
#  Это похоже на использование пары фигурных скобок в "find."
#
```

```
#  Список файлов в текущей директории (ls .),
#+ вывод "ls" передается как аргумент в "xargs" (опции -i -t),
#+ затем копируются (cp) эти аргументы ({}) в новую директорию ($1).
#
#  Конечный результат является точным эквивалентом
#+   cp * $1
#+ за исключением файлов со встроенными символами «пробел».

exit 0
```

Пример 16-7. Уничтожение процесса по имени

```
#!/bin/bash
# kill-byname.sh: Уничтожение процесса по имени.
# Сравните этот сценарий с kill-process.sh.

#  Например,
#+ попробуйте "./kill-byname.sh xterm" --
#+ и посмотрите все ли xterm исчезли на рабочем столе.

#  Предупреждение:
#  -------
#  Это очень опасный сценарий.
#  Небрежный запуск (особенно из-под root) может привести к потере
#+ данных и другим нежелательным эффектам.

E_BADARGS=66

if test -z "$1"  # Нет аргумента командной строки?
then
  echo "Usage: `basename $0` Процесс(ы)_для_уничтожения"
  exit $E_BADARGS
fi


PROCESS_NAME="$1"
ps ax | grep "$PROCESS_NAME" | awk '{print $1}' | xargs -i kill {} 2&>/dev/null
#                                                  ^^      ^^

# -----------------------------------------------------------------
# Примечания:
# -i опция "замена строк" xargs.
# {} заполнитель вместо заменяемого.
# 2&>/dev/null подавляет нежелательные сообщения об ошибках.
#
# Можно ли  grep "$PROCESS_NAME" заменить на pidof "$PROCESS_NAME"?
# -----------------------------------------------------------------

exit $?
# -n1   Более 1 аргумента командной строки.
#  Команда "killall"  имеет тот же эффект, что и этот сценарий,
#+ но ее использование не так поучительно.
```

Пример 16-8. Частотный анализ слов с помощью *xargs*

```
#!/bin/bash
# wf2.sh: Частотный анализ не обработанных слов в текстовом файле.

# Используется «xargs» для разделения строк текста на отдельные слова.
# Сравните этот пример со сценарием «wf.sh» позже.

# Проверка ввода файла в командной строке.
ARGS=1
E_BADARGS=85
E_NOFILE=86

if [ $# -ne "$ARGS" ]
# Сценарию передается правильное число аргументов?
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]       # Файл существует?
then
  echo "Файл \"$1\" не существует."
  exit $E_NOFILE
fi



###############################################################
cat "$1" | xargs -n1 | \
#  Список файлов, одно слово на строке.
tr A-Z a-z | \
#  Смещаем символы в нижний регистр.
sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
/g' | \
#  Фильтруем промежутки и запятые, и заменяем
#+ пространства между словами переводом строки,
sort | uniq -c | sort -nr
#  Наконец удаляем дубликаты, считаем префиксы и сортируем численно.
###############################################################

#  Он делает ту же работу, что и пример '' wf.sh", но немного

#+ более тяжело, и более медленно (почему?).

exit $?
```

expr

Универсальная оценка выражений: объединяет и оценивает аргументы в соответствии с учетом операции (аргументы должны быть разделены пробелами). Операции могут быть арифметическими, сравнения, строчными или логическими.

expr 3 + 5

возвращает 8

expr 5 % 3

возвращает 2

`expr 1 / 0`

возвращает сообщение об ошибке, `expr`: деление на ноль

Не правильная арифметическая операция.

`expr 5 \* 3`

возвращает 15

Оператор умножения должен быть ***экранирован*** при использовании в арифметическом выражении с **`expr`**.

`y=`expr $y + 1``

Увеличение значения переменной, такой же эффект, как и у **`let y=y+1`** и **`y=$((($y+1))`**. Это пример *арифметического расширения*.

`z=`expr substr $string $position $length``

Извлечение содержимого строки `$length` символов, начиная с `$position`.

Пример 16-9. Использование *expr*

```
#!/bin/bash

# Использование 'expr'
# =====================================

echo

# Арифметические операции
# ---------- ---------

echo "Арифметические операции"
echo
a=`expr 5 + 3`
echo "5 + 3 = $a"

a=`expr $a + 1`
echo
echo "a + 1 = $a"
echo "(приращение переменной)"

a=`expr 5 % 3`
# модуль
echo
echo "5 mod 3 = $a"

echo
echo
```

```
# Логические операции
# ------- ---------

#  Возвращает 1, если истинно, 0 если ложно,
#+ обратно обычному соглашению Bash.

echo "Логические операции"
echo

x=24
y=25
b=`expr $x = $y`          # Проверка равенства.
echo "b = $b"             # 0  ( $x -ne $y )
echo

a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, однако...'
echo "Если a > 10, b = 0 (ложно)"
echo "b = $b"             # 0  ( 3 ! -gt 10 )
echo

b=`expr $a \< 10`
echo "Если a < 10, b = 1 (истинно)"
echo "b = $b"             # 1  ( 3 -lt 10 )
echo
# Обратите внимание на ***экранирование*** операторов.

b=`expr $a \<= 3`
echo "Если a <= 3, b = 1 (истинно)"
echo "b = $b"             # 1  ( 3 -le 3 )
# То же, что и оператор "\>=" (больше или равно).


echo
echo


# Строковые операции
# ------ ---------

echo "Строковые операции"
echo

a=1234zipper43231
echo "Строка являющаяся \"$a\"."

# length: размер строки
b=`expr length $a`
echo "Размер \"$a\" это $b."

# index: позиция первого символа в substring
#        соответствующего символу string
b=`expr index $a 23`
echo "Числовая позиция первого символа \"2\" в \"$a\" это \"$b\"."

# substr: извлечение substring, с указанием начальной позиции & размера
b=`expr substr $a 2 6`
echo "Substring \"$a\", начинается с позиции 2,\
```

```
и величиной 6 символов это \"$b\"."


#  Поведение по умолчанию операций 'соответствия', это поиск
#+ указанных соответствий с НАЧАЛА строки.
#
#       Использование регулярных выражений ...
b=`expr match "$a" '[0-9]*'`                #  Подсчет чисел.
echo Количество цифр в начале \"$a\" это $b.
b=`expr match "$a" '\([0-9]*\)'`            #  Обратите внимание на
#                   ==      ==              #+ экранирование скобок вызова
                                            #+ соответствия substring.
echo "Цифры с начала \"$a\" это\"$b\"."

echo

exit 0
```

Оператор **:** (***null***) может заменить **match**. Например, **b=`expr $a : [0-9]*`** является точным эквивалентом **b=`expr match $a [0-9]*`** в листинге выше.

```
#!/bin/bash

echo
echo "Строковые операции использующие конструкцию \"expr \$string : \""
echo "=================================================="
echo

a=1234zipper5FLIPPER43231

echo  "Рабочая строка это:       \"`expr "$a" : '\(.*\)'`\"."
#     Экранированные скобки групповых операций.  ==  ==

#       **************************
#+         Экранированные скобки
#+         соответствия substring
#       **************************


#  Если скобки не экранировать ...
#+ то 'expr' преобразует операнд string в целое число.

echo "Размер \"$a\" это `expr "$a" : '.*'`."   # Размер string

echo "Количество цифр в начале \"$a\" это `expr "$a" : '[0-9]*'`."

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

echo

echo "Цифры в начале \"$a\" это `expr "$a" : '\([0-9]*\)'`."
```

```
#                                             ==      ==
echo "Первые 7 символов \"$a\" это `expr "$a" : '\(.......\)'`."
#     =====                                   ==      ==
#  Снова, экранированные круглые скобки принудительно
#+ проверяют совпадения  substring.
#
echo "Последние 7 символов \"$a\" это `expr "$a" : '.*\(.......\)'`."
#     =========           конец строки оператора    ^^
#  (На самом деле, означает пропустить один или несколько любых
#+ символов до указанной существующей substring.)

echo

exit 0
```

Сценарий выше иллюстрирует использование **expr** экранированных скобок -- `\( ... \)` -- группировка операторов в тандеме с регулярным выражением *анализирует* соответствие substring. Вот еще один пример, на этот раз от «реальной жизни».

```
# Удаление пробелов в начале и конце.
LRFDATE=`expr "$LRFDATE" : '[[:space:]]*\(.*\)[[:space:]]*$'`

#  Из сценария Peter Knowles "booklistgen.sh"
#+ для конвертирования файлов в формат Sony Librie/PRS-50X.
#  (http://booklistgensh.peterknowles.com)
```

Perl и ***sed***, ***awk*** намного превосходят в анализе объектов строки. ***Короткие*** «подпрограммы» ***sed*** или ***awk*** внутри сценария (см. раздел 36.2) являются более привлекательной альтернативой **expr**.

См. Раздел 10.1, что бы подробнее узнать об использовании **expr** в строковых операциях.

Примечания

[1] И даже когда **xargs** не является строго необходимым, он может ускорить выполнение команды, связанной с ***пакетной обработкой*** нескольких файлов.

16.3. Команды времени/даты

Время/дата и выбор определенного времени

date

Просто вызываемая **date** выводит дату и время в `stdout`. Эта команда становится интересна в форматировании и анализе опций.

Пример 16-10. Использование *date*

```
#!/bin/bash
# Работа команды 'date'

echo "Количество дней с начала года это `date +%j`."
# Для вызова форматирования в начале необходим '+'.
# %j это заданный день года.

echo "Количество секунд, прошедшее с 01/01/1970 это `date +%s`."
#  %s задает число секунд с начала 'эпохи UNIX',
#+ но, насколько это полезно?

prefix=temp
suffix=$(date +%s)  # Опция "+%s" в'date' является GNU-специфичной.
filename=$prefix.$suffix
echo "Временное файловое имя = $filename"
#  Хорошо для создания 'уникальных и случайных' имен файлов из времени,
#+ даже лучше, чем  использование $$.

# Что бы узнать о других опциях читайте справочные страницы 'date'.

exit 0
```

Опция `-u` задает UTC (Universal Coordinated Time).

```
bash$ date
Fri Mar 29 21:07:39 MST 2002



bash$ date -u
Sat Mar 30 04:07:42 UTC 2002
```

Эта опция облегчает вычисления времени между разными датами.

Пример 16-11. Расчет *дат*

```
#!/bin/bash
# date-calc.sh
# Автор: Nathan Coulter
# Используется в ABS Guide с разрешения (спасибо!).

MPHR=60    # Минут в часе.
HPD=24     # Часов в сутках.

diff () {
        printf '%s' $(( $(date -u -d"$TARGET" +%s) -
                        $(date -u -d"$CURRENT" +%s)))
#                       %d = день месяца.
}


CURRENT=$(date -u -d '2007-09-01 17:30:24' '+%F %T.%N %Z')
```

```
TARGET=$(date -u -d'2007-12-25 12:30:00' '+%F %T.%N %Z')
# %F = полная дата, %T = %H:%M:%S, %N = наносекунды, %Z = часовой пояс.

printf '\nIn 2007, %s ' \
       "$(date -d"$CURRENT +
        $(( $(diff) /$MPHR /$MPHR /$HPD / 2 )) дней" '+%d %B')"
#       %B = название месяца              ^ пополам
printf 'Половина между %s ' "$(date -d"$CURRENT" '+%d %B')"
printf 'и %s\n' "$(date -d"$TARGET" '+%d %B')"

printf '\nOn %s в %s, это была\n' \
        $(date -u -d"$CURRENT" +%F) $(date -u -d"$CURRENT" +%T)
DAYS=$(( $(diff) / $MPHR / $MPHR / $HPD ))
CURRENT=$(date -d"$CURRENT +$DAYS дней" '+%F %T.%N %Z')
HOURS=$(( $(diff) / $MPHR / $MPHR ))
CURRENT=$(date -d"$CURRENT +$HOURS часов" '+%F %T.%N %Z')
MINUTES=$(( $(diff) / $MPHR ))
CURRENT=$(date -d"$CURRENT +$MINUTES минут" '+%F %T.%N %Z')
printf '%s дней, %s часов, ' "$DAYS" "$HOURS"
printf '%s минут, и %s секунд ' "$MINUTES" "$(diff)"
printf 'до Рождественского ужина!\n\n'

#  Упражнение:
#  --------
#  Перепишите функцию diff() для принятия передаваемых ей параметров,
#+ вместо использования глобальных переменных.
```

Команда **`date`** имеет ряд параметров *вывода*. Например %N наносекундные порции текущего времени. Одним из интересных использований этого является генерация случайных целых чисел.

```
date +%N | sed -e 's/000$//' -e 's/^0//'
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#  Удаляются начальные и конечные нули, если они есть.
#  Длина создаваемого целого числа зависит от количества удаленных нулей.

# 115281032
# 63408725
# 394504284
```

Существуют много других опций (читай ***man date***).

```
date +%j
# Выводит на экран порядковый день года (дней, прошедших с 1 января).

date +%k%M
# Выводит на экран часы и минуты в 24-часовом формате, в одной цифровой строке.

# Параметр 'TZ' позволяет переопределять часовой пояс по умолчанию.
date                  # Mon Mar 28 21:42:16 MST 2005
TZ=EST date           # Mon Mar 28 23:42:16 EST 2005
# Спасибо Frank Kannemann и Pete Sjoberg.


SixDaysAgo=$(date --date='6 дней назад')
OneMonthAgo=$(date --date='1 месяц назад')  # 4 недели назад (не месяц!)
```

```
OneYearAgo=$(date --date='1 год назад')
```

См. также Пример 3-4 и Пример A-43.

zdump

Дамп часового пояса: выводит на экран время в указанном часовом поясе.

```
bash$ zdump EST
EST  Tue Sep 18 22:09:22 2001 EST
```

time

Вывод подробной временной статистики выполнения команды.

time ls -l / дает что-то вроде этого :

```
real    0m0.067s
 user    0m0.004s
 sys     0m0.005s
```

Смотрите также очень похожую команду **times** в предыдущем разделе.

Начиная с версии Bash 2.0, **time** стало зарезервированным словом оболочки, со слегка измененным поведением в конвейере.

touch

Утилита для обновления времени доступа/изменения файла в текущем системном времени или в другом указанном времени, полезна для создания нового файла. Команда **touch zzz** создает новый файл нулевой длины, с именем `zzz`, предполагается, что `zzz` ранее не существовал. Отметка времени пустыми файлами полезна для хранения информации о дате, например при отслеживании времени изменений в проекте.

Команда **touch** эквивалентна **: >> newfile** или **>> newfile** (для простых файлов).

Прежде чем делать **cp -u** (копирование/обновление), используйте **touch** для обновления отметки времени файлов, которые вы не хотите перезаписывать.

Например, если директория `/home/bozo/tax_audit` содержит файлы `spreadsheet-051606.data`, `spreadsheet-051706.data`, and `spreadsheet-051806.data`, то сделайте **touch**

spreadsheet*.data для защиты этих файлов от перезаписи файлами с теми же именами во время **cp -u /home/bozo/financial_info/spreadsheet*data /home/bozo/tax_audit**.

at

Команда управления заданиями **at** выполняет определенный набор команд в заданное время. Внешне она напоминает **cron**, но **at** , главным образом, полезна для выполнения одноразового набора команд.

at 2pm January 15 запрашивает набор команд для выполнения в указанное время. В любом случае все команды исполняемые сценарием, введенные пользователем, должны быть совместимы с оболочкой сценария. Ввод завершается **Ctl-D**.

С помощью опции `-f` или перенаправления ввода (**<**), **at** читает список команд из файла. Если этот файл является исполняемым сценарием, то он, конечно, должен быть не интерактивными.

```
bash$ at 2:30 am Friday < at-jobs.list
job 2 at 2000-10-27 02:30
```

batch

Команда управления заданием **batch** похожа на **at**, но она запускает список команд, понижая нагрузку на систему. Как **at**, она может читать команды из файла при помощи опции `-f`.

> Концепция *пакетной обработки* восходит к эпохе мэнфреймов. Это означает, запуск множества команд без вмешательства пользователя.

cal

Выводит аккуратно отформатированный календарь на месяц в `stdout`. Выводит текущий год или большой диапазон прошлых и будущих лет.

sleep

Это эквивалент оболочки *ожидания цикла*. Делает паузу в течение определенного количества секунд, в это время ничего не происходит. Она может быть полезна для расчета времени или в процессах, запущенных в фоновом режиме, для проверки каждого конкретного события с определенной частотой (опрос), как в Примере 32-6.

```
sleep 3     # Пауза 3 секунды.
```

Команда **sleep** по умолчанию в секундах, но можно устанавливать минуты, часы или дни.

```
sleep 3 h   # Пауза 3 часа!
```

Команда **watch** может быть лучшим выбором, чем **sleep** для запуска команд через определенные промежутки времени.

usleep

*Микро**sleep***. Это то же самое, как **sleep**, выше, но 'спит' с микросекундными интервалами. Она может быть использована для точнейших интервалов времени, или для постоянных опросов процесса в очень короткие промежутки времени.

```
usleep 30     # Пауза 30 микросекунд.
```

Эта команда является частью пакета Red Hat *initscripts/rc-scripts*.

Команда **usleep** не обеспечивает большую точность, и поэтому не подходит для определенных циклов синхронизации.

hwclock, clock

hwclock команда доступа или регулировки аппаратных команд, влияющая на текст и текстовые файлы часов машины. Некоторые опции требуют привилегий *root*. Файл начального запуска `/etc/rc.d/rc.sysinit` при загрузке использует **hwclock** для установки системного времени из часов оборудования.

Команда **clock** это синоним **hwclock**.

16.4. Команды обработки текста

Команды, влияющие на текст и текстовые файлы

sort

Утилита сортировки файлов, часто используемая в качестве фильтра в конвейере. Эта команда сортирует текстовый поток или файлы, с начала или с конца, или согласно различным ключам, или с позиций символов. С помощью опции `-m`, она объединяет отсортированные входные файлы. В *info page* перечислены большинство возможностей и параметров. См. Пример 11-10, Пример 11-11 и Пример A-8.

tsort

Топологическая сортировка, считывает две строки за раз, разделяет пробелами и сортирует в соответствии с введенным шаблоном. Первоначальной целью **tsort** являлась сортировка списка зависимостей в «древних» версиях UNIX устаревших версий компоновщика ***ld***.

Результаты **tsort** отличаются от стандартных команд сортировки выше.

uniq

Фильтр удаляющий повторяющиеся строки из сортируемого файла. Часто используется в конвейере в сочетании с **sort**.

```
cat list-1 list-2 list-3 | sort | uniq > final.list
# Объединяются файлы list,
# затем сортируются,
# удаляются повторяющиеся строки
# и результат записывается в выходной файл.
```

Опция `-c`, вначале каждой строки файла, выводит количество повторений.

```
bash$ cat testfile
Эта строка встречается только один раз.
 Эта строка встречается дважды.
 Эта строка встречается дважды.
 Эта строка встречается трижды.
 Эта строка встречается трижды.
 Эта строка встречается трижды.


bash$ uniq -c testfile
      1 Эта строка встречается только один раз.
       2 Эта строка встречается дважды.
       3 Эта строка встречается трижды.


bash$ sort testfile | uniq -c | sort -nr
      3 Эта строка встречается трижды.
       2 Эта строка встречается дважды.
       1 Эта строка встречается только один раз.
```

Команда **sort** *INPUTFILE* **| uniq -c | sort -nr** выводит список *частоты встречаемости* в файле *INPUTFILE* (опции `-nr` указывают **sort** производить сортировку с обратной нумерацией). Этот шаблон находит применение в анализе лог-файлов или словарных слов, а так же везде, где должна быть рассмотрена лексическая структура документа.

Пример 16-12. Анализ частоты встречаемости слова

```
#!/bin/bash
# wf.sh: Частотный анализ не обработанного слова в текстовом файле.
# Это более эффективная версия сценария «wf2.sh».


# Проверка входного файла в командной строке.
ARGS=1
E_BADARGS=85
E_NOFILE=86

if [ $# -ne "$ARGS" ]  # Сценарию передано правильное число аргументов?
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]       # Проверка существования файла.
then
  echo "Файл \"$1\" не существует."
  exit $E_NOFILE
fi


###############################################################
# main ()
sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
/g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
#                           ========================
#                            Частота встречаемости

#  Фильтруются разрывы и запятые, изменения пространства
#+ между словами в виде перевода каретки, сдвиг символов в
#+ нижний регистр и, наконец, рассчитывается префикс встречаемости
#+ и производится численная сортировка.

#  Arun Giridhar предлагает изменить:
#  ... | sort | uniq -c | sort +1 [-f] | sort +0 -nr
#  Добавляется ключ вторичной сортировки, поэтому каждое
#+ равное вхождение сортируется в алфавитном порядке.
#  Как он объясняет:
#  "Это эффективная сортировка, сначала не важная
#+ колонка (слово или строка, без учета регистра)
#+ а затем, наиболее важная колонка (частота)."
#
#  Как поясняет Frank Wang, эквивалент вышеуказанному
#+       ... | sort | uniq -c | sort +0 -nr
#+ является:
#+       ... | sort | uniq -c | sort -k1nr -k
###############################################################

exit 0

# Упражнения:
# ---------
# 1) Добавьте команды 'sed' для фильтрации иных знаков пунктуации,
#+   таких как точка с запятой.
# 2) Измените сценарий для фильтрации множественных
#+   пробелов и других пробелов.
```

```
bash$ cat testfile
```

```
Эта строка встречается только один раз.
 Эта строка встречается дважды
 Эта строка встречается дважды
 Эта строка встречается трижды
 Эта строка встречается трижды
 Эта строка встречается трижды


bash$ ./wf.sh testfile
      6 эта
       6 строка
       6 встречается
       1 раз
       3 трижды
       2 дважды
       1 только
       1 один
```

expand, unexpand

Фильтр **expand** преобразует табуляцию в пробелы. Часто используется в конвейере.

Фильтр **unexpand** преобразует пробелы в табуляцию. Эффект обратный **expand**.

cut

Инструмент для извлечения полей из файлов. Похож на команду **print $N** в ***awk***, но более ограничен. В сценарии может быть проще использовать **cut**, чем *awk*. Особенно важными являются опции `-d` (разделитель) и `-f` (описание поля).

Использование **cut** для получения листинга смонтированных файловых систем:

```
cut -d ' ' -f1,2 /etc/mtab
```

Использование **cut** для получения списка ОС и версии ядра:

```
uname -a | cut -d" " -f1,3,11,12
```

Использование **cut** для извлечения заголовка сообщения из папки e-mail:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
 MAKE MILLIONS WORKING AT HOME!!!
 Spam complaint
 Re: Spam complaint
```

Использование **cut** для анализа файла:

```
# Список всех пользователей в /etc/passwd.

FILENAME=/etc/passwd
```

```
for user in $(cut -d: -f1 $FILENAME)
do
  echo $user
done

# Спасибо за предоставление Oleg Philon.
```

cut -d ' ' -f2,3 filename эквивалентно **awk -F'[ ]' '{ print $2, $3 }' filename**

Возможно даже указание строки в качестве разделителя. Хитрость заключается во внедрении перевода каретки (**RETURN**) в последовательность команд.

```
bash$ cut -d'
 ' -f3,7,19 testfile
Это строка 3 testfile.
 Это строка 7 testfile.
 Это строка 19 testfile.
```

Спасибо Jaka Kranjc, за разъяснение.

См. также Пример 16-48.

paste

Инструмент для объединения различных файлов в один, многостолбцовый файл. В сочетании с **cut** полезен для создания системных лог-файлов.

```
bash$ cat items
alphabet blocks
 building blocks
 cables

bash$ cat prices
$1.00/dozen
 $2.50 ea.
 $3.75

bash$ paste items prices
alphabet blocks $1.00/dozen
 building blocks $2.50 ea.
 cables  $3.75
```

join

Рассматривайте, как особого родственника **paste**. Эта мощная утилита позволяет объединять два файла в понятный образ, который, по существу, создает простую версию реляционной базы данных.

Команда **join** работает ровно с *двумя* файлами, но вставляет вместе только строки с тегами общих полей (обычно числовая метка) и записывает результат в stdout. Файлы для объединения должны быть отсортированы в соответствии совпадающим тегам полей.

```
File: 1.data

100 Shoes
200 Laces
300 Socks


File: 2.data

100 $40.00
200 $1.00
300 $2.00
```

```
bash$ join 1.data 2.data
File: 1.data 2.data

 100 Shoes $40.00
 200 Laces $1.00
 300 Socks $2.00
```

Поле меток, в выходных данных, появляется только один раз.

head

Список от начала файла выводимый в stdout. Значение по умолчанию — первые 10 строк, но можно указать другое количество. Команда имеет ряд интересных опций.

Пример 16-13. Какие файлы являются сценариями?

```
#!/bin/bash
# script-detector.sh: Обнаружение сценариев в директории.

TESTCHARS=2    # Проверка первых 2 символов.
SHABANG='#!'   # Сценарии начинаются с "sha-bang."

for file in *  # Просмотр всех файлов в текущей директории.
do
  if [[ `head -c $TESTCHARS "$file"` = "$SHABANG" ]]
  #      head -c2       это              #!
  #  Опция '-c' в "head" выводит указанное количество
  #+ символов, а не строк (как по умолчанию).
  then
    echo "Файл \"$file\" это сценарий."
  else
    echo "Файл \"$file\" это *не* сценарий."
  fi
```

```
done

exit 0

#  Упражнения:
#  ---------
#  1) Измените этот сценарий, чтобы, для проверки наличия
#+    в ней сценариев (а не только в текущей рабочей директории),принимать
#+    в качестве необязательного аргумента директорию.
#
#  2) Как его остановить, если сценарий будет «ложно срабатывать»
#+    в сценариях на Perl, awk и других  языках сценариев.
#     Исправьте этот момент.
```

Пример 16-14. Генерация случайных 10-значных чисел

```
#!/bin/bash
# rnd.sh: Вывод 10-значных случайных чисел

# Сценарий Stephane Chazelas.

head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'


# ============================================================== #

# Анализ
# --------

# head:
# опция -c4 принимает первые 4 байта.

# od:
# Опция -N4 ограничивает вывод 4 байтами.
# Опция -tu4  выбирает десятичный формат вывода без знака.

# sed:
# Опция -n, в сочетании с флагом "p" команды "s",
# выводит только совпадающие строки.


# Автор сценария объясняет работу 'sed' следующим образом.

# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# ---------------------------------> |

# Предполагаемый вывод в  "sed" ----> |
# это 0000000 1198195154\n,
#  sed начинает читать символы: 0000000 1198195154\n.
#  Там он находит символ новой строки,
#+ но он готов к обработке первой строки (0000000 1198195154).
#  Похоже на его <диапазон> <действие>. Первое и единственное это
#  диапазон    действие
#    1         s/.* //p

#  Номер строки указан в диапазоне, поэтому выполняется действие:
#+ пытается заменить длинную строку заканчивающуюся  пробелом в строке
#  ("0000000 ") на ничего (//), и если удачно, то выводит результат
```

```
#  (здесь "p" это флаг команды "s", отличающийся действием
#+ от команды "p").
#  Теперь sed готов продолжить чтение входных данных. (Обратите внимание, что
#+ до продолжения, если опция -n не была передана, sed выведет строку
#+ еще раз).
#  Далее, sed считывает оставшиеся символы, и ищет
#+ конец файла.
#  И только теперь он готов к обработке 2-й строки (которая пронумерована '$'
#+ как завершающая, последняя).
#  Он видит не соответствие ее <диапазону>, потому и завершает задачу.
#  Несколько слов о значениях команд sed:
#  «Только в первой строке, удаление любого символа до пробела справа, потом
#+ его вывод.»

# Лучший способ это сделать:
#           sed -e 's/.* //;q'

# Здесь, два <диапазон><действия> (можно записать
#           sed -e 's/.* //' -e q):

#   диапазон                      действие
#   ничего (совпадающие строки)   s/.* //
#   ничего (совпадающие строки)   q (выход)

#  sed читает только первую вводенную строку.
#  Он выполняет действия и выводит строки (подставляемые) перед выходом
#+ (из-за действия 'q') с не переданным параметром '-n'.

# ================================================================== #

# Даже может быть проще альтернативный однострочник:
#           head -c4 /dev/urandom| od -An -tu4

exit
```

См. также Пример 16-39.

tail

Список (хвоста) от окончания файла выводимого в `stdout`. По умолчанию `10` строк, но это можно изменить при помощи опции `-n`. Обычно используется для отслеживания изменений в системных журнальных файлах, с помощью опции `-f` выводимые строки *добавляются* в конец файла.

Упражнение 16-15. Проверка системного журнала с помощью *tail*

```
#!/bin/bash

filename=sys.log

cat /dev/null > $filename; echo "Создание/очистка файла."
#  Создаем файл, если он не существует,
#+ или уменьшаем его до нулевой длины, если он существует.
#  : > filename   и   > filename так же работают.

tail /var/log/messages > $filename
# /var/log/messages должен иметь общедоступные права на чтение.
```

```
echo "$filename содержит последние строки системного журнала."

exit 0
```

Для вывода конкретных строк текстового файла, ***конвейером*** передается вывод **head** в **tail -n 1**. Например **head -n 8 database.txt | tail -n 1** вывод 8-й строки файла database.txt.

Присваивание переменной в заданном блоке текстового файла:

```
var=$(head -n $m $filename | tail -n $n)

# filename = имя файла
# m = от начала файла, количество строк до конца блока
# n = количество строк присваиваемых переменной (отрезок от
конца блока)
```

Более новые реализации **tail** опротестовывают использование старых **tail -$LINES filename**. Правильным является обычное **tail -n $LINES filename**.

См. Пример 16-5, Пример 16-39 и Пример 32-6.

grep

Средство многоцелевого назначения для поиска в файле, с **grep** пользуются ***регулярные выражения***. Первоначально это была команда/фильтр строкового редактора **ed** : **g/re/p** -- *global - regular expression - print*.

grep *шаблон* [*файл*...]

Поиск в указанном файле(ах) данных соответствующих *шаблону*, где *шаблоном* может быть обычный текст или регулярное выражение.

```
bash$ grep '[rst]ystem.$' osinfo.txt
GPL определяет распространение операционной системы Linux.
```

Если файл(ы) не указан, **grep** работает как фильтр из stdout в **конвейер**.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
 901 pts/1    S      0:00 grep clock
```

Опция **-i** ищет без учета регистра .

Опция **-w** ищет соответствия только целым словам.

Опция **-l** ищет только файлы, в которых найдены совпадения, но не совпадающие строки.

Опция **-r** (рекурсивно) ищет файлы в текущей рабочей директории и всех ее поддиректориях.

Опция **-n** выводит список совпадающих строк, вместе с номерами строк.

```
bash$ grep -n Linux osinfo.txt
2:This is a file containing information about Linux.
 6:The GPL governs the distribution of the Linux operating system.
```

Опция **-v** (или `--invert-match`) *отфильтровывает* совпадения.

```
grep pattern1 *.txt | grep -v pattern2

# Все совпадающие строки в файлах "*.txt" содержащие "pattern1",
# но ***не*** "pattern2".
```

Опция **-c** (`--count`) дает *численное количество* совпадений, а не список совпадений.

```
grep -c txt *.sgml   # (число совпадений "txt" в файлах "*.sgml")

#   grep -cz .
#            ^ точка
# означает подсчет(-c) совпадающих элементов ".", разделенных нулями (-z)
# то есть, не пустые (содержащие по крайней мере 1 символ).
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz .     # 3
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '$'   # 5
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '^'   # 5
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -c '$'    # 9
# По умолчанию, символ перевода строки (\n) является отдельным
#+ элементом соответствия.

# Обратите внимание, что опция -z является особенностью GNU "grep".


# Спасибо, S.C.
```

Опция `--color` (или `--colour`) выделяет соответствующую строку цветом (в консоли или в окне xterm). Так как `grep` выводит каждую строку, содержащую соответствие шаблону, то это позволяет увидеть в чем именно соответствие. См. Так же опцию `-o`, которая показывает только совпадающий участок строки.

Пример 16-16. Вывод строк *From* из сохраненных сообщений e-mail

```
#!/bin/bash
# from.sh
```

```
#  Эмуляция полезной утилиты 'from' в Solaris, BSD, и т.п..
#  "From" выводит строки заголовков всех сообщений e-mail вашей директории.

MAILDIR=~/mail/*               #  Переменная не в кавычках. Почему?
# Может быть проверить существование $MAILDIR:  if [ -d $MAILDIR ] ...
GREP_OPTS="-H -A 5 --color"    #  Выводит файлы, дополнительно
                               #+ выводит строки "From" в цвете.
TARGETSTR="^From"              #  Начало строки"From".

for file in $MAILDIR           #  Переменная не в кавычках.
do
  grep $GREP_OPTS "$TARGETSTR" "$file"
  #    ^^^^^^^^^^              #  Опять, переменная не в кавычках.
  echo
done

exit $?

#  Вы, возможно, пожелаете передать выходные данные этого сценария
#+ «more» или перенаправить их в файл...
```

При вызове с более чем одним заданным целевым файлом, **grep** определяет, какой файл содержит совпадения.

```
bash$ grep Linux osinfo.txt misc.txt
osinfo.txt:This is a file containing information about Linux.
 osinfo.txt:The GPL governs the distribution of the Linux operating system.
 misc.txt:The Linux operating system is steadily gaining in popularity.
```

Что бы заставить **grep** выводить имя файла при поиске единственного указанного файла, просто в качестве второго файла задайте `/dev/null`.

```
bash$ grep Linux osinfo.txt /dev/null
osinfo.txt:This is a file containing information about Linux.
 osinfo.txt:The GPL governs the distribution of the Linux operating
system.
```

Если совпадение успешно, **grep** возвращает статус выхода `0`, что делает возможным его нахождение в условии проверки в сценарии, особенно в сочетании с параметром `-q`, подавляющем вывод.

```
SUCCESS=0                      # Если поиск grep успешен
word=Linux
filename=data.file

grep -q "$word" "$filename"    #  Опция "-q"
                               #+ ничего не выводит в stdout.
if [ $? -eq $SUCCESS ]
```

```
# if grep -q "$word" "$filename"   может заменить строки 5 - 7.
then
  echo "$word найдено в $filename"
else
  echo "$word не найдено в $filename"
fi
```

Пример 32-6 демонстрирует использование **grep** для поиска слова по шаблону в системном файле журнала.

Пример 16-17. Эмуляция *grep* в сценарии

```
#!/bin/bash
# grp.sh: Простая повторная реализация grep.

E_BADARGS=85

if [ -z "$1" ]    # Проверка аргументов сценария.
then
  echo "Usage: `basename $0` шаблон"
  exit $E_BADARGS
fi

echo

for file in *     # Просмотр всех файлов в $PWD.
do
  output=$(sed -n /"$1"/p $file)  #  Подстановка команд.

  if [ ! -z "$output" ]           #  Что случится, если "$output"
                                  #+ будет без кавычек?
  then
    echo -n "$file: "
    echo "$output"
  fi           #  sed -ne "/$1/s|^|${file}: |p"  это эквивалент тому, что выше.

  echo
done

echo

exit 0

# Упражнения:
# ---------
# 1) Добавьте в вывод перевод строк, если в любом заданном файле будет найдено
#+   более чем одно совпадение.
# 2) Добавьте функциональности.
```

Может ли **grep** осуществлять поиск по двум (или более) отдельным шаблонам? Что делать, если нужно с помощью **grep** вывести все строки, содержащие «pattern1» и «pattern2», в файле или файлах, ?

Одним из способов является передача ***конвейером*** результата вывода **`grep pattern1`** в **`grep pattern2`**.

Например, задан следующий файл:

```
# Filename: tstfile

This is a sample file.
This is an ordinary text file.
This file does not contain any unusual text.
This file is not unusual.
Here is some text.
```

Теперь давайте искать в этом файле строки одновременно содержащие «**file**» и «**text**»...

```
bash$ grep file tstfile
# Filename: tstfile
 This is a sample file.
 This is an ordinary text file.
 This file does not contain any unusual text.
 This file is not unusual.

bash$ grep file tstfile | grep text
This is an ordinary text file.
 This file does not contain any unusual text.
```

Теперь, интересное использование `grep` для досуга...

Пример 16-18. Решатель кроссворда

```
#!/bin/bash
# cw-solver.sh
# На самом деле является оберткой для однострочника (строки 46).

#  Вам известно *несколько* букв слова,
#+ нужно вывести весь список правильных слов,
#+ содержащих известные буквы в указанных позициях.
#  Например:    w...i....n
#               1???5????10
# w в позиции 1, 3 неизвестных, i в 5-й, 4 неизвестных, n в конце.
# (См. Комментарий в конце сценария.)


E_NOPATT=71
DICT=/usr/share/dict/word.lst
#                    ^^^^^^^^   Это источник для поиска списка слов.
#  Словарь ASCII, одно слово на строке.
#  Если будет нужен соответствующий словарь,
#+ загрузите пакет словаря автора "yawl".
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  или
#  http://bash.deta.in/yawl-0.3.2.tar.gz


if [ -z "$1" ]   #  Если отсутствует заданный шаблон слова,
```

```
then             #+ как аргумент командной строки ...
  echo           #+ . . . то . . .
  echo "Usage:"  #+ Сообщение об использовании.
  echo
  echo ""$0" \"шаблон,\""
  echo "где \"шаблон\" в форме"
  echo "xxx..x.x..."
  echo
  echo "x'ы это известные буквы,"
  echo "а пропуски – неизвестные буквы (пробелы)."
  echo "Буквы и пробелы могут быть в любой позиции."
  echo "Например, попробуйте:   sh cw-solver.sh w...i....n"
  echo
  exit $E_NOPATT
fi

echo
# ================================================
# Здесь делается вся работа.
grep ^"$1"$ "$DICT"   # Да, только одна строка!
#    |    |
# ^ это обозначение начала слова регулярного выражения.
# $ это обозначение окончания слова регулярного выражения.

# ================================================
echo


exit $?  # Здесь сценарий завершается.
#  Если создается слишком много слов,
#+ перенаправьте вывод в файл.

$ sh cw-solver.sh w...i....n

wellington
workingman
workingmen
```

egrep — ***e****xtended (расширенная)* ***grep*** — то же самое, что и **grep -E**. Она использует несколько иной, расширенный, набор регулярных выражений, которые позволяют производить поиск более гибко. Она допускает использование логического оператора **|** (ИЛИ).

```
bash $ egrep 'matches|Matches' file.txt
Строка 1 matches.
 Строка 3 Matches.
 Строка 4 содержит matches, а так же Matches
```

fgrep — ***f****ast (быстрая)* ***grep*** — то же самое, что и **grep -F**. Это строка посимвольного поиска (без регулярных выражений), которая обычно ускоряет поиск.

В некоторых дистрибутивах Linux, **egrep** и **fgrep** являются символическими ссылками или псевдонимами для **grep**, вызываемой с

опциями -E и -F, соответственно.

Пример 16-19. Поиск определений в *Словаре Webster 1913*

```
#!/bin/bash
# dict-lookup.sh

#  Сценарий ищет определения в 1913 Webster's Dictionary.
#  Словарь Public Domain, доступный для скачивания с различных сайтов,
#+ включая Project Gutenberg (http://www.gutenberg.org/etext/247).
#
#  Перед использованием его этим сценарием, конвертируйте его из формата DOS в
#+ UNIX (только с символом перевода строки в конце строк).
#  Сохраните его, не сжимая текст ASCII.
#  Присвойте переменной DEFAULT_DICTFILE, ниже, путь/имя_файла.


E_BADARGS=85
MAXCONTEXTLINES=50                # Максимально выводимое количество строк.
DEFAULT_DICTFILE="/usr/share/dict/webster1913-dict.txt"
                                  # Путь к файлу словаря по умолчанию.
                                  # При необходимости измените.
#  Примечания:
#  ----
#  Данный выпуск 1913 Вебстер начинает каждую запись с прописной
#+ буквы (остальные символы в нижнем регистре).
#  Только *начальная строка* записи начинается таким образом,
#+ и, именно поэтому работает алгоритм поиска ниже.


if [[ -z $(echo "$1" | sed -n '/^[A-Z]/p') ]]
#  Необходимо указать хотя бы одно слово для поиска,
#+ и оно должно начинаться с заглавной буквы.
then
  echo "Usage: `basename $0` Слово-для-поиска [файл-словаря]"
  echo
  echo "Примечание: Слово для поиска должно начинаться с заглавной буквы,"
  echo "а остальная часть слова в нижнем регистре ."В некоторых дистрибутивах
Linux, egrep и fgrep являются символическими ссылками или псевдонимами для
grep, вызываемые, соответственно, опциями -E и -F.
  echo "-----------------------------------------"
  echo "Примеры: Abandon, Dictionary, Marking, и т.д."
  exit $E_BADARGS
fi


if [ -z "$2" ]                            #  В качестве аргумента этого
                                          #+ сценария можно указать другой
                                          #+ словарь.
then
  dictfile=$DEFAULT_DICTFILE
else
  dictfile="$2"
fi

# ------------------------------------------------------------
Definition=$(fgrep -A $MAXCONTEXTLINES "$1 \\" "$dictfile")
```

```
#                   Определения в форме "Слово \..."
#
#  И, да, "fgrep" достаточно быстр для поискать
#+ даже в очень большом текстовом файле.

# Теперь, вырезаем только из этого блока.

echo "$Definition" |
sed -n '1,/^[A-Z]/p' |
#  Вывод, из первой строки вывода, в первую строку следующей записи
sed '$d' | sed '$d'
#  Удаляем две последние строки вывода
#+ (пустая строка и первая строка следующей записи).
# ------------------------------------------------------------

exit $?

# Упражнения:
# ---------
# 1)  Измените сценарий так, чтобы принимать ввод букв любого
#   + типа (прописные, строчные, смешанный регистр) и преобразовывать
#   + их в приемлемый для обработки формат.
#
# 2)  Конвертируйте сценарий в приложение GUI,
#   + используя что-то типа 'gdialog' или 'zenity' ...
#   + Что бы сценарий больше не принимал аргументы из командной строки.
#
# 3)  Измените сценарий для анализа одного из других доступных словарей
#   + Public Domain, например U.S. Census Bureau Gazetteer.
```

См. также Пример A-41 быстрого поиска ***fgrep*** в большом текстовом файле.

agrep (***a****pproximate (приблизительная)* ***grep***) расширяет возможности **grep** приблизительными совпадениями. Строка поиска может отличаться на указанное количество символов от окончательных совпадений. Эта утилита не является частью ядра дистрибутивов Linux.

Для поиска в сжатых файлах используются **zgrep**, **zegrep** или **zfgrep**. Они также работают с несжатыми файлами, хотя и медленнее, чем обычные **grep**, **egrep**, **fgrep**. Они удобны для поиска в смешанных наборах файлов, где какие-то файлы сжатые, а какие-то нет.

Для поиска в файлах **bzip** используется **bzgrep**.

look

Команда **look** работает как **grep**, но осуществляет поиск с сортировкой списка слов

по «словарю». По умолчанию, **look** ищет совпадения в /usr/dict/words, но могут быть указаны и другие словари.

Пример 16-20. Проверка на правильность слов в списке

```
#!/bin/bash
# поиск: Поиск по словарю каждого слова в данных файла.

file=words.data  # Файл данных, из которого считываются проверяемые слова.

echo
echo "Проверка файла $file"
echo

while [ "$word" != end ]  # Последнее слово в файле данных.
do                        # ^^^
  read word       # Из файла данных, перенаправленного в конце цикла.
  look $word > /dev/null  # Не нужно выводить строки файла словаря.
  #  Поиск слов в файле /usr/share/dict/words
  #+ (обычно ссылается на linux.words).
  lookup=$?       # Статус выхода команды 'look'.

  if [ "$lookup" -eq 0 ]
  then
    echo "\"$word\" правильное."
  else
    echo "\"$word\" не правильное."
  fi

done <"$file"  # Перенаправление stdin в $file, т.к. отсюда происходит
"считывание".

echo

exit 0

# ------------------------------------------------------------------
# Код строк ниже не будет выполняться из-за команды 'exit' выше.


# Stephane Chazelas предлагает следующий, более краткий вариант:

while read word && [[ $word != end ]]
do if look "$word" > /dev/null
   then echo "\"$word\" правильное."
   else echo "\"$word\" не правильное."
   fi
done <"$file"

exit 0
```

sed, awk

Языки сценариев особенно подходящие для анализа текстовых файлов и вывода команд. Могут быть вставлены поодиночке или в сочетании с конвейером и сценариями оболочки.

sed

Не интерактивный «потоковый редактор», позволяет использовать множество команд **ex** в пакетном режиме. Находит применение в сценариях оболочки.

awk

Программируемое файловое извлечение и форматирование, хорош для манипуляций и/или извлечения *полей* (столбцов) из структурированных текстовых файлов. Синтаксис похож на Си.

wc

wc выводит "количество слов" в файле или потоке I/O:

```
bash $ wc /usr/share/doc/sed-4.1.2/README
13  70  447 README
[13 строк  70 слов  447 символов]
```

wc -w подсчет только слов.

wc -l подсчет только строк.

wc -c подсчет только байтов.

wc -m подсчет только символов.

wc -L только размер самой длинной строки.

С помощью **wc** подсчитаем, сколько файлов .txt находятся в текущей рабочей директории:

```
$ ls *.txt | wc -l
#  Будет работать до первого '*.txt' файла, имеющего,
#+ встроенный в его имя, символ перевод каретки.

#  Другой способ:
#      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#      (shopt -s nullglob; set -- *.txt; echo $#)

#  Спасибо S.C.
```

Передача в **total**, с помощью **wc**, размера всех файлов, чьи имена начинаются с букв в диапазоне **d - h**

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

Использование **wc** для подсчета количества слов «Linux» в основном исходном файле этой книги.

```
bash$ grep Linux abs-book.sgml | wc -l
138
```

См. также Пример 16-39 и Пример 20-8.

Некоторые команды включают некоторую функциональность **wc** с опциями.

```
... | grep foo | wc -l
# Эта часто используемая конструкция может быть отображена более лаконично.

... | grep -c foo
# Только с помощью опции "-c" (или "--count") grep.

# Спасибо S.C.
```

tr

фильтр преобразования символов.

Необходимо использовать кавычки и/или скобки. Кавычки предотвращают интерпретацию оболочкой специальных символов в последовательности команд **tr**. Квадратные скобки в кавычках предотвращают расширения оболочки.

tr "A-Z" "*" <filename или **tr A-Z * <filename** заменяют все прописные буквы в `filename` на звездочки (выводятся в `stdout`). В некоторых системах может не сработать, но **tr A-Z '[**]'** сработает.

Опция `-d` удаляет диапазон символов.

```
echo "abcdef"                  # abcdef
echo "abcdef" | tr -d b-d      # aef


tr -d 0-9 <filename
# Удаляет все цифры из файла "filename".
```

Опция `--squeeze-repeats` (или `-s`) удалит все, кроме первого знака строки последовательных символов. Эта опция полезна для удаления лишних ***пробелов***.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

Опция `-c` "дополнительно" опция *инвертирует* соответствие символов. С этой опцией **tr** действует только на те символы, которые не соответствуют указанному набору.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

Обратите внимание, что **tr** признает классы символов POSIX. [1]

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

Пример 16-21. *toupper*: Преобразование имени файла в верхний регистр.

```
#!/bin/bash
# Изменяет имя файла в верхний регистр.

E_BADARGS=85

if [ -z "$1" ]  # Обычная проверка аргумента командной строки.
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

tr a-z A-Z <"$1"

# Тот же эффект, но с использованием набора нотации символов POSIX:
#       tr '[:lower:]' '[:upper:]' <"$1"
# Спасибо S.C.

#     Или даже ...
#     cat "$1" | tr a-z A-Z
#     Или десятком других способов ...

exit 0

#  Упражнение:
#  Перепишите этот сценарий, чтобы можно было изменять
#+ файл *либо* в верхний, либо в нижний регистр.
#  Подсказка: Используйте команды "case" или "select".
```

Пример 16-22. *lowercase*: Изменение имен всех файлов в рабочей директории в нижний регистр.

```
#!/bin/bash
#
#  Меняет имя каждого файла в рабочей директории в нижний регистр.
#
#  Вдохновлено сценарием John Dubois,
#+ который Chet Ramey перенес на Bash,
#+ и значительно упрощен автором ABS Guide.
```

```
for filename in *                 # Просмотр всех файлов в директории.
do
   fname=`basename $filename`
   n=`echo $fname | tr A-Z a-z`   # Изменение имен на нижний регистр.
   if [ "$fname" != "$n" ]        #  Переименовываются только файлы, которые
                                  #+ не находятся в нижнем регистре.
   then
     mv $fname $n
   fi
done

exit $?


# Код ниже этой строки не будет выполнен из-за "exit" выше.
#-------------------------------------------------------#
# Чтобы запустить его, удалите сценарий выше строки.

#  Сценарий выше не работает с именами файлов содержащими
#+ пробелы или переводы строк.
# Stephane Chazelas предлагает следующую альтернативу:


for filename in *    # Не обязательно использовать basename,
                     # т.к. "*" не возвращает файлы содержащие "/".
do n=`echo "$filename/" | tr '[:upper:]' '[:lower:]'`
#                            Нотация символов POSIX.
#                    Слэш добавлен для того, что бы оконечные символы новой
#                    строки не удалялись путем подстановки команд.
   # Подстановка переменных:
   n=${n%/}          # Удаляем оконечный слэш из filename, добавленный выше.
   [[ $filename == $n ]] || mv "$filename" "$n"
                     # Проверка, находится ли имя файла в нижнем регистре.
done

exit $?
```

Пример 16-23. *du*: преобразование текстового файла DOS в UNIX.

```
#!/bin/bash
# Du.sh: конвертер текстовых файлов DOS в UNIX.

E_WRONGARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` имя_файла_для_ конвертации"
  exit $E_WRONGARGS
fi

NEWFILENAME=$1.unx

CR='\015'  # Возврат каретки.
           # 015 это восьмеричный код ASCII для CR.
           # Строки текстового файла DOS оканчиваются CR-LF.
           # Строки текстового файла UNIX оканчиваются только LF.
```

```
tr -d $CR < $1 > $NEWFILENAME
# Удаляются CR-ы и пишутся в новый файл.

echo "Оригинальный текстовый файл DOS это \"$1\"."
echo "Преобразованный текстовый файл UNIX это \"$NEWFILENAME\"."

exit 0

# Упражнение:
# --------
# Измените этот сценарий для конвертации из UNIX в DOS.
```

Пример 16-24. *rot13*: сверхслабое шифрование.

```
#!/bin/bash
# rot13.sh: Классический алгоритм rot13,
#           стойкое шифрование 3-летней давности
#           вскрываемое сейчас за 10 минут.

# Использовать: ./rot13.sh filename
# или     ./rot13.sh <filename
# или     ./rot13.sh и клавиатурный ввод (stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a" переходит в "n", "b" в "o" ...
#  Конструкция   cat "$@"
#+ допускает ввод из stdin или из файлов.

exit 0
```

Пример 16-25. Создание головоломки "Crypto-Quote"

```
#!/bin/bash
# crypto-quote.sh: Шифрование цитат

#  Будем шифровать известные цитаты простой побуквенной заменой.
#  Результат аналогичен головоломке «Crypto Quote»,
#+ увиденной на страницах Op Ed газеты Sunday.


key=ETAOINSHRDLUBCFGJMQPVWZYXK
# "key" это не более, чем перемешанный алфавит.
# Изменение "key" изменяет шифрование.

# Конструкция 'cat "$@"' позволяет вводить из stdin или из файлов.
# Если используется stdin, ввод прекращается Control-D.
# Иначе, указываем имя файла в качестве параметра командной строки
cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
#        | в вехн.регистр |   шифрование
# Будет работать с цитатами в нижнем, верхнем и в смешанном регистрах.
# Передача не буквенных символов останется без изменения.

# Попробуем этот сценарий с чем-нибудь вроде:
# "Nothing so needs reforming as other people's habits."
```

```
# --Марка Твена
#
# Выводится:
# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."

# Расшифровка:
# cat "$@" | tr "$key" "A-Z"


#  Этот бесхитростный шифр, при использовании только карандаша и бумаги,
#+ может быть вскрыт в среднем за 12 лет.

exit 0

#  Упражнение:
#  --------
#  Измените сценарий так, что бы он либо шифровал либо расшифровывал,
#+ в зависимости от аргументов командной строки.
```

Конечно, **tr** запутывает *код*.

```
#!/bin/bash
# jabh.sh

x="wftedskaebjgdBstbdbsmnjgz"
echo $x | tr "a-z" 'oh, turtleneck Phrase Jar!'

# Из главы Wikipedia "Just another Perl hacker".
```

Варианты *tr*

Утилита **tr** имеет два исторических варианта. В версии BSD не используются скобки (**tr a-z A-Z**), а в SysV они применяются (**tr '[a-z]' '[A-Z]'**). Версия **tr** GNU напоминает версию BSD.

fold

Фильтр, который делит вводимые строки на указанную длину. Особенно полезна с опцией `-s`, которая разделяет строки пробелами (см. Пример 16-26 и Пример A-1).

fmt

Бесхитростное форматирование файла, используемое в качестве фильтрации из конвейера «делящее» длинные строки выводимого текста.

Пример 16-26. Форматированный вывод файла.

```
#!/bin/bash
```

```
WIDTH=40                    # 40 колонок в ширину.

b=`ls /usr/local/bin`       # Выводимый файл...

echo $b | fmt -w $WIDTH

# А можно и так
#    echo $b | fold - -s -w $WIDTH

exit 0
```

См. также Пример 16-5.

Мощной альтернативой **fmt** является утилита Kamil Toman **par**, см. *http://www.cs.berkeley.edu/~amc/Par/*.

col

Это обманчивое название фильтра удаляющего обратный перевод строки из входного потока. Он также пробует заменять пробелы табуляцией. Основным использованием **col** является фильтрация вывода некоторых утилит обработки текста, таких как **groff** и **tbl**.

column

Форматирование колонки. Этот фильтр преобразует тип вывода текстового списка в «доступную для печати» таблицу, вставляя в соответствующих местах табуляцию.

Пример 16-27. Использование *column* для форматирования списка директории

```
#!/bin/bash
# colms.sh
# Незначительно измененный пример файла справочной страницы «column».


(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l | sed 1d) | column -t
#         ^^^^^^           ^^

#  "sed 1d" в конвейере удаляет первую выводимую строку,
#+ которая будет «всего        N»,
#+ где "N" общее количество файлов выводимое "ls -l".

# Опция -t в "column" доступная для печати таблица.

exit 0
```

colrm

Фильтр для удаления колонок. Удаляет колонки (символов) из файла и записывает файл, с отсутствующим рядом указанных столбцов, обратно в `stdout`. **`colrm 2 4 <filename`** удаляет со второго по четвертый символы из каждой строки текстового файла `filename`.

Если файл содержит табуляцию или непечатаемые знаки, то это может привести к непредсказуемому поведению. В таких случаях рассмотрите возможность использования **`expand`** и **`unexpand`** в конвейере перед **`colrm`**.

`nl`

Фильтр нумерации строк: **`nl filename`** передает список `filename` в `stdout`, но вставляет последовательную нумерацию в начале каждой не пустой строки. Если `filename` опущен, работает со `stdin`.

Вывод **`nl`** очень похож на **`cat -b`**, поскольку, по умолчанию, **`nl`** не выводит пустые строки.

Пример 16-28. *nl*: Самономерующийся сценарий.

```
#!/bin/bash
# line-number.sh

# Этот сценарий выводит себя дважды на stdout с пронумерованными строками.

Echo "      номер строки = $LINENO" # 'nl' понимает это как строку 4
#                                   (nl не нумерует пустые строки).
#                                   'cat -n' понимает правильно, как строку #6.

nl `basename $0`

echo; echo  # Теперь давайте попробуем с 'cat -n'

cat -n `basename $0`
# Разница в том, что 'cat -n' нумерует пустые строки.
# Обратите внимание, что 'nl -ba' делает то же самое.

exit 0
# -----------------------------------------------------------------
```

`pr`

Фильтр форматирования печати. Он разбивает отдельные файлы на страницы (или `stdout`), подходящие для точного копирования вывода при печати или для просмотра на экране. Различные опции допускают манипуляции с рядами и колонками, присоединение строк, установку границ, нумерацию строк, добавление заголовков страниц и, среди прочего, объединение файлов. Команда **`pr`** объединяет функциональность **`nl`**, **`paste`**, **`fold`**, **`column`** и **`expand`**.

`pr -o 5 --width=65 fileZZZ | more` выводит прекрасный постраничный список `fileZZZ` на экран с границами `5` и `65`.

Особенно полезна опция `-d`, принудительных двойных пробелов (эффект подобен **`sed -G`**).

`gettext`

GNU пакет **`gettext`** является набором утилит для *локализации* и перевода текста, выводимого программами, на иностранные языки. Будучи изначально предназначенным для программ Си, теперь он поддерживает целый ряд языков программирования и сценариев.

Программа **`gettext`** работает в сценариях оболочки. См. *`info page`*.

`msgfmt`

Программа для создания каталогов бинарных сообщений. Используется для *локализации*.

`iconv`

Утилита для преобразования файлов в другую кодировку (набор символов). Ее основное использование это *локализация*.

```
# Преобразование строки из UTF-8 в UTF-16 и вывод в BookList
function write_utf8_string {
    STRING=$1
    BOOKLIST=$2
    echo -n "$STRING" | iconv -f UTF8 -t UTF16 | \
    cut -b 3- | tr -d \\n >> "$BOOKLIST"
}

#  Из сценария Peter Knowles "booklistgen.sh"
#+ для преобразования файлов в формат Sony Librie/PRS-50X.
#  (http://booklistgensh.peterknowles.com)
```

`recode`

Рассматривайте ее как любительскую версию **`iconv`**, выше. Это весьма универсальная утилита для преобразования файла в различные схемы кодирования. Обратите внимание, что **`recode`** не является частью обычной установки Linux.

`TeX`, `gs`

TeX и **Postscript** — это языки разметки текста, используемые при подготовке копий для печати или форматированного вывода на дисплей.

TeX - система печати разработанная Donald Knuth. Часто бывает удобно написать

сценарий оболочки, содержащий все параметры и аргументы, передаваемые одному из этих языков разметки.

Ghostscript (**gs**) это GPL интерпретатор ***Postscript***.

texexec

Утилита для обработки *TeX* файлов *pdf*. Во многих дистрибутивах Linux находится в `/usr/bin`, является, на самом деле, оберткой оболочки, которая вызывает *Perl* для вызова *Tex*.

```
texexec --pdfarrange --result=Concatenated.pdf *pdf

#  Объединяет все файлы pdf в текущей рабочей директории
#+ в общий файл, Concatenated.pdf ...
#  (Опция --pdfarrange выполняет разбиение на страницы файла pdf. См. так же
--pdfcombine.)
#  Командная строка выше может быть параметризована и помещена в сценарий
оболочки.
```

enscript

Утилита для преобразования текстового файла в PostScript

Например **enscript filename.txt -p filename.ps** производит вывод PostScript файла `filename.ps`.

groff, tbl, eqn

Другим языком разметки текста и форматированного вывода является **groff**. Это расширенная GNU версия уважаемого пакета отображения и верстки UNIX **roff/troff**.

Утилита обработки таблиц **tbl** считается частью **groff**, ее функция заключается в преобразовании разметки таблицы в команды **groff**.

Утилита обработки формул **eqn** является также частью **groff**, и ее функция заключается в преобразовании разметки уравнения в команды **groff**.

Пример 16-29. *manview*: Просмотр форматированных справочных страниц

```
#!/bin/bash
# manview.sh: Форматирование исходных справочных страниц для просмотра.

#  Этот сценарий полезен при написании исходников man page.
#  Он позволяет видеть промежуточные результаты на лету, во время работы.

E_WRONGARGS=85
```

```
if [ -z "$1" ]
then
  echo "Usage: `basename $0` файл"
  exit $E_WRONGARGS
fi

# --------------------------
groff -Tascii -man $1 | less
# Из man page для groff.
# --------------------------

#  Если страница man page включает таблицы и/или уравнения,
#+ приведенный выше код будет «мутить».
#  В таких случаях может помочь следующая строка.
#
#   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
#
#   Спасибо S.C.

exit $?   # См. Так же сценарий "maned.sh".
```

См. также Пример A-39.

lex, **yacc**

lex - лексический анализатор, делающий программы соответствующими шаблону. Он заменил в системах Linux не проприетарный **flex**.

Утилита **yacc** создает анализатор на основе набора спецификаций. Заменила в системах Linux не проприетарный **bison**.

Примечания

[1] Это верно только для GNU версии **tr**, а не универсальной версии, часто встречающейся на коммерческих системах UNIX.

16.5. Файловые команды и команды архивирования

Архивирование

tar

Основная утилита архивирования в UNIX. [1] Изначально программа ***Tape ARchiving***, она превратилась в пакет общего назначения, который обрабатывает все виды архивирования со всеми типами указанных устройств, начиная от ленточных накопителей и кончая обычными файлами и даже `stdout` (см. Пример 3-4). GNU ***tar***

была исправлена, с целью использования различных фильтров сжатия, например: **`tar czvf archive_name.tar.gz *`**, рекурсивно архивирует и сжимает (**`gzip`**) все файлы дерева директории, за исключением *файлов с точкой*, текущей рабочей директории (`$PWD`). [2]

Некоторые полезные опции **`tar`**:

1. `-c` создать (новый архив)
2. `-x` извлечь (файлы из существующего архива)
3. `--delete` удалить (файлы из существующего архива)

   

   Эта опция не работает на устройствах с магнитной лентой.
4. `-r` добавить (файлы в существующий архив)
5. `-A` добавить (файлы *tar* в существующий архив)
6. `-t` список (содержимого существующего архива)
7. `-u` обновление архива
8. `-d` сравнить архив с указанной файловой системой
9. `--after-date` обрабатывать только файлы с отметками даты *после* указанной даты
10. `-z` сжатие архива (**`gzip`**)

    (сжатие или распаковка, в зависимости от комбинации, опциями `-c` или `-x`)
11. `-j` архивирование **`bzip2`**

При восстановлении данных из поврежденного сжатого tar-архива могут возникнуть трудности. При архивации важных файлов делайте несколько копий.

`shar`

Утилита ***Sh**ell **ar**chiving*. В архив оболочки текст или двоичные файлы объединяются без сжатия и полученный архив является по существу сценарием оболочки, с заголовком `#!/ bin/sh`, содержащим все необходимые команды разархивирования, а также сами файлы. Непечатные двоичные символы в указанных файле(ах) преобразуются, в выходном файле shar, в печатные символы ASCII. Команда **`unshar`** распаковывает архивы *shar*.

Команда **mailshar** является сценарием Bash использующим **shar** для объединения нескольких файлов в один для *e-mail*. Этот сценарий поддерживает сжатие и ***uuencoding***.

ar

Утилита для создания и управления архивами, главным образом используемая для библиотек двоичных объектных файлов.

rpm

Пакетный менеджер Red Hat, или утилита **rpm** предоставляет оболочку для исходных или бинарных архивов. Среди прочего она включает в себя команды для установки и проверки целостности пакетов.

Простое **rpm -i package_name.rpm** достаточно для установки пакета, хотя имеется достаточно много других опций

rpm -qf определяет источник происхождения пакета файлов.

```
bash$ rpm -qf /bin/ls
coreutils-5.2.1-31
```

rpm -qa предоставляет полный список всех установленных rpm пакетов в данной системе. А **rpm -qa package_name** только список пакетов соответствующих package_name.

```
bash$ rpm -qa
redhat-logos-1.1.3-1
 glibc-2.2.4-13
 cracklib-2.7-12
 dosfstools-2.7-1
 gdbm-1.8.0-10
 ksymoops-2.4.1-1
 mktemp-1.5-11
 perl-5.6.0-17
 reiserfs-utils-3.x.0j-2
 ...


bash$ rpm -qa docbook-utils
docbook-utils-0.6.9-2


bash$ rpm -qa docbook | grep docbook
docbook-dtd31-sgml-1.0-10
 docbook-style-dsssl-1.64-3
 docbook-dtd30-sgml-1.0-10
 docbook-dtd40-sgml-1.0-11
 docbook-utils-pdf-0.6.9-2
 docbook-dtd41-sgml-1.0-10
 docbook-utils-0.6.9-2
```

cpio

Специализированная команда копирования архивирования (копирование ввода-вывода), это редко встречаемая команда была заменена **tar**/**gzip**. Она до сих пор еще используется, например для перемещение дерева директории. С указанным размером блока (для копирования) может быть намного быстрее, чем **tar**.

Пример 16-30. Перемещение дерева директории с помощью *cpio*

```
#!/bin/bash

# Копирование дерева директории с помощью cpio.

# Преимущества использования 'cpio':
#   Скорость копирования. Быстрее, чем 'tar' с конвейерами.
#   Хорошо подходит для копирования специальных файлов (именованных
#+  каналов, и т.д.)  в отличии от "cp".

ARGS=2
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` указанный источник"
  exit $E_BADARGS
fi

source="$1"
destination="$2"

###########################################################################
find "$source" -depth | cpio -admvp "$destination"
#               ^^^^^          ^^^^^
#  Читайте справочные страницы 'find' и 'cpio' для
#+ расшифровки этих опций.
#  Работает только по отношению к $PWD (текущей директории)...
#+ или укажите полный путь.
###########################################################################


# Упражнение:
# --------

#  Добавьте код проверяющий статус выхода ($?) 'find | cpio'
#+ и выводящий соответствующие сообщения об ошибках, если
#+ что-то пошло не так.

exit $?
```

rpm2cpio

Эта команда извлекает архив **cpio** из *rpm*.

Пример 16-31. Распаковка архива *rpm*

```
#!/bin/bash
# de-rpm.sh: Распаковка 'rpm' архива

: ${1?"Usage: `basename $0` целевой_файл"}
# Имя 'rpm'архива должно быть указано как аргумент.

TEMPFILE=$$.cpio                # Временный файл с уникальным именем.
                                # $$ это ID процесса сценария.
a
rpm2cpio < $1 > $TEMPFILE               #  Конвертирование rpm архив в
                                        #+ архив cpio.
cpio --make-directories -F $TEMPFILE -i  #  Распаковка архива cpio.
rm -f $TEMPFILE                          #  Удаление архив cpio.

exit 0

#  Упражнение:
#  Добавьте проверку на 1) существование "целевого_файла" и
#+                      2) архива rpm.
#  Подсказка:             Проверка вывода командой 'file'.
```

pax

pax - набор инструментов обмена для переносимости архивов, облегчающая периодический бэкап файлов, и предназначенный для кросс совместимости между различными платформами UNIX. Был разработан для замены **tar** и **cpio**.

```
pax -wf daily_backup.pax ~/linux-server/files
#  Создает tar архив всех файлов в целевой директории.
#  Обратите внимание, что опции в pax должны быть в правильном порядке
#+ --  pax -fw   будет иметь другой эффект.

pax -f daily_backup.pax
#  Список файлов архива

pax -rf daily_backup.pax ~/bsd-server/files
#  Восстановление заархивированных файлов Linux машины
#+ на BSD.
```

Обратите внимание, что ***pax*** обрабатывает многие из стандартных команд архивирования и сжатия.

Сжатие

gzip

Стандартная утилита сжатия GNU/UNIX, заменившая низкоуровневую и проприетарную **compress**. Соответствующая команда распаковывания - **gunzip** эквивалентна **gzip -d**.

Опция `-c` посылает вывод **gzip** в `stdout`. Это полезно для передачи с помощью ***конвейра*** в другие команды.

Фильтр **zcat** распаковывает сжатый файл в `stdout`, для возможности ввода в конвейер или перенаправления. Это, по сути, команда **cat**, которая работает с сжатыми файлами (включая файлы, обработанные с помощью старой утилиты **compress**). Команда **zcat** эквивалентна **gzip -dc**.

В некоторых коммерческих системах UNIX, **zcat** является синонимом **uncompress -c**, и не будет работать с файлами сжатыми ***gzip***.

См. также Пример 7-7.

bzip2

Другая утилита сжатия, обычно более эффективная (но более медленная), чем **gzip**, особенно с большими файлами. Соответствующая команда распаковки это **bunzip2**.

Подобна команде **zcat**, **bzcat** распаковывает файлы *bzip2* в `stdout`.

Более новые версии **tar** были исправлены для поддержки **bzip2**.

compress, **uncompress**

Это давнишние, проприетарные утилиты сжатия коммерческих дистрибутивов UNIX. В основном их заменил более эффективный **gzip**. Linux дистрибутивы обычно содержат **compress** для совместимости, хотя **gunzip** может разархивировать файлы, сжатые **compress**.

Команда **znew** преобразует файлы *сжатые* в ***gzip***.

sq

Еще одна утилита сжатия (**sq**ueeze), фильтр, который осуществляет сортировку списков слов только ASCII. Использует обычный синтаксис для фильтра, **sq < input-file > output-file**. Быстрая, но не такая эффективная как ***gzip***. Фильтр для распаковки, соответственно **unsq**, вызываемый как **sq**.**unsqsqsq**

Вывод **sq** может быть передан конвейером в **gzip** для дальнейшего сжатия.

zip, **unzip**

Кросс-платформенная утилита архивирования и сжатия файла совместимая с DOS *pkzip.exe*. «Зипованные» архивы более распространенное средство обмена файлами в Интернет, чем «тарболлы.»

unarc, **unarj**, **unrar**

Утилиты позволяющие распаковывать архивы сжатые программами DOS *arc.exe*, *arj.exe* и *rar.exe*.

lzma, **unlzma**, **lzcat**

Высокоэффективное сжатие **L**empel-**Z**iv-**Ma**rkov. Синтаксис *lzma* подобен *gzip*. Больше информации на сайте 7-zip.

xz, **unxz**, **xzcat**

Новый, высоко эффективный инструмент сжатия, совместим с *lzma* и синтаксис вызова подобен *gzip*. Для получения дополнительной информации см. Википедию.

Файловая информация

file

Утилита для определения типов файлов. Команда **file file-name** возвратит спецификацию файла для file-name, такую как ascii text или data (данные). Она ссылается на ***магические числа***, находящиеся в /usr/share/magic, /etc/magic, или /usr/lib/magic, в зависимости от дистрибутива Linux/UNIX.

Опция -f указывает **file** запуск чтения из указанного файла для анализа списка имен файлов в **пакетном** режиме. Опция -z, при использовании сжатого целевого файла, пытается принудительно анализировать тип несжатого файла.

```
bash$ file test.tar.gz
test.tar.gz: данные сжатые gzip, извлечены,
 последнее изменение: Sun Sep 16 13:34:51 2001, os: Unix

bash file -z test.tar.gz
test.tar.gz: архив GNU (данные сжатые gzip, извлечены,
 последнее изменение: Sun Sep 16 13:34:51 2001, os: Unix)
```

```
# Поиск sh и Bash сценариев в заданной директории:

DIRECTORY=/usr/local/bin
```

```
KEYWORD=Bourne
# Bourne и Bourne-Again сценарии оболочки

file $DIRECTORY/* | fgrep $KEYWORD

# Выводится:

# /usr/local/bin/burn-cd:          Bourne-Again текстовый исполняемый
                                   сценарий оболочки
# /usr/local/bin/burnit:           Bourne-Again текстовый исполняемый
                                   сценарий оболочки
# /usr/local/bin/cassette.sh:      Bourne текстовый исполняемый
                                   сценарий оболочки
# /usr/local/bin/copy-cd:          Bourne-Again текстовый исполняемый
                                   сценарий оболочки
# ...
```

Пример 16-32. Удаление комментариев из файлов программы Си

```
#!/bin/bash
# strip-comment.sh: Удаление комментариев (/* COMMENT */) в программе Си.

E_NOARGS=0
E_ARGERROR=66
E_WRONG_FILE_TYPE=67

if [ $# -eq "$E_NOARGS" ]
then
  echo "Usage: `basename $0` файл-программы-Си" >&2 # Сообщения об ошибке в
                                                    #+ stderr.
  exit $E_ARGERROR
fi

# Проверка правильности типа файла.
type=`file $1 | awk '{ print $2, $3, $4, $5 }'`
# "file $1" выводит на экран тип файла ...
# Затем awk удаляет первое поле, имя файла ...
# Затем  результат передается переменной "type."
correct_type="ASCII C program text"

if [ "$type" != "$correct_type" ]
then
  echo
  echo "Этот сценарий работает только с файлами программ Си."
  echo
  exit $E_WRONG_FILE_TYPE
fi


# Довольно загадочный сценарий sed:
#--------
sed '
/^\/\*/d
/.*\*\//d
' $1
#--------
Легко понимается, если уделить несколько часов изучению основ sed.
```

```
#  Нужно добавить еще одну строку в сценарий sed для рассмотрения случая,
#+ когда строка кода имеет комментарий после него на той же строке.
#  Это оставим как нетривиальное упражнение.

#  Кроме того, код выше удаляет строки не комментариев с "*/" ...
#+ не желательный результат.

exit 0


# ------------------------------------------------------------------
# Код ниже этой строки не будет выполнен из-за 'exit 0' выше.

# Stephane Chazelas предлагает следующую альтернативу:

usage() {
  echo "Usage: `basename $0` файл_программы_Си" >&2
  exit 1
}

WEIRD=`echo -n -e '\377'`   # или WEIRD=$'\377'
[[ $# -eq 1 ]] || usage
case `file "$1"` in
  *"текст программы Си"*) sed -e "s%/\*%${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
     | tr '\377\n' '\n\377' \
     | sed -ne 'p;n' \
     | tr -d '\n' | tr '\377' '\n';;
  *) usage;;
esac

#  Но все еще обманывается вещами типа:
#  printf("/*");
#  или
#  /*  /* ошибочно встроенный комментарий */
#
#  Для обработки всех особых случаев (комментарии в строках, комментарии в
#+ строке, где есть\", \\" ...),
#+ единственным выходом является написание парсера Си (возможно с помощью
#+ lex и yacc?).

exit 0
```

which

Команда **which** выводит полный путь к "*команде*." Это полезно для выяснения того, установлены ли в системе определенные команды или утилиты.

$bash which rm

```
/usr/bin/r
```

Интересный эффект использования этой команды см. Пример 36-16.

whereis

Подобно **which**, выше, *команда* **whereis** выдает полный путь к "*команде*," а так же путь к ее справочной странице.

$bash whereis rm

```
rm: /bin/rm /usr/share/man/man1/rm.1.bz2

```

whatis

команда **whatis** ищет "*команду*" в базе данных *whatis*. Она полезна для выявления системных команд и важных конфигурационных файлов. Рассматривайте, как упрощенную команду **man**.

$bash whatis whatis

```
whatis               (1)  - search the whatis database for complete words

```

Пример 16-33. Знакомство с /usr/X11R6/bin

```
#!/bin/bash

# Что это за загадочные двоичные файлы в /usr/X11R6/bin?

DIRECTORY="/usr/X11R6/bin"
# Попробуйте так же "/bin", "/usr/bin", "/usr/local/bin", и т.д.

for file in $DIRECTORY/*
do
  whatis `basename $file`   # Выводит информацию о бинарных файлах.
done

exit 0

#  Примечание: Что бы это работало, вы должны создать базу данных "whatis"
#+ с /usr/sbin/makewhatis.
#  Вы можете перенаправить вывод этого сценария, например так:
#    ./what.sh >>whatis.db
#  или просмотреть постранично во время вывода stdout,
#    ./what.sh | less
```

См. также Пример 11-3.

vdir

Выводит детализированный список директории. Эффект похож на **ls -lb**.
vdirlocateslocate

Это одна из *файловых утилит* GNU.

```
bash$ vdir
total 10
```

```
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo

bash ls -l
total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo
```

locate, **slocate**

Команда **locate** выполняет поиск файлов, с помощью сохраняемой, только для этой цели, базы данных. Команда **slocate** это безопасная версия **locate** (которая может быть алиасом **slocate**).

$bash locate hickson

```
/usr/lib/xephem/catalogs/hickson.edb Эти команды извлечения или установки
управления списком доступа к файлу ( file access control list)--владелец,
группа и права доступа к файлу.
```

getfacl, **setfacl**

Команды извлечения или установки управляющие списком доступа к файлу (**f**ile **a**ccess **c**ontrol **l**ist)--владельца, группы и прав доступа к файлу.

```
bash$ getfacl *
# file: test1.txt
 # owner: bozo
 # group: bozgrp
 user::rw-
 group::rw-
 other::r--

 # file: test2.txt
 # owner: bozo
 # group: bozgrp
 user::rw-
 group::rw-
 other::r--



bash$ setfacl -m u:bozo:rw yearly_budget.csv
bash$ getfacl yearly_budget.csv
# file: yearly_budget.csv
 # owner: accountant
 # group: budgetgrp
 user::rw-
 user:bozo:rw-
 user:accountant:rw-
```

```
group::rw-
mask::rw-
other::r--
```

readlink

Выводит файл на который указывает символическая ссылка.

```
bash$ readlink /usr/bin/awk
../../bin/gawk
```

strings

С помощью команды **strings** осуществляется поиск строк в двоичном файле или файле данных, возможных для вывода. Это будет список последовательностей печатаемых символов, найденных в указанном файле. Может быть удобно для быстрой черновой проверки дампа ядра или просмотра файла неизвестного графического изображения (**strings файла изображения | more** может показать что-то вроде JFIF, файл, который будет идентифицирован, как изображение jpeg). В сценарии вывод **strings** производит синтаксический анализ с помощью **grep** или **sed**. См. Пример 11-8 и Пример 11-10.

Пример 16-34. "Улучшения" команды *strings*

```
#!/bin/bash
# wstrings.sh: "word-strings" (расширение команды "strings")
#
#  Этот сценарий фильтрует вывод «strings», сверяя его с файлом словаря.
#  Эффективно устраняет бред и шум, и выводит только слова.

# ============================================================
#                 Обычная проверка для аргументов сценария
ARGS=1
E_BADARGS=85
E_NOFILE=86

if [ $# -ne $ARGS ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]                       # Проверка существования файла.
then
    echo "Файл \"$1\" не существует."
    exit $E_NOFILE
fi
# ============================================================


MINSTRLEN=3                            #  Минимальная длина строки.
```

```
WORDFILE=/usr/share/dict/linux.words  #  Файл словаря.
#  Можно указать другой словарь в формате: одно слово на одной строке.
#  Например, пакет словаря "yawl",
#  http://bash.deta.in/yawl-0.3.2.tar.gz


wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
       tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`

#  Переводим вывод команды «strings» множеством проходов «tr».
#  "tr A-Z a-z"  преобразует в нижний регистр.
#  "tr '[:space:]'"  преобразует символы пробелов в Z.
#  "tr -cs '[:alpha:]' Z"  Преобразует не буквенные символы в Z,
#+ и сжимает несколько последовательных Z.
#  "tr -s '\173-\377' Z"  преобразует все прошлые символы 'z' в Z
#+ и сжимает несколько последовательных Z,
#+ избавляясь от всех странных символов, с  которыми не удалось
#+ справиться предыдущему переводу.
#  В конце, "tr Z ' '" преобразует все эти Z' в пробелы,
#+ которые будут рассматриваться, как разделители слов, в цикле ниже.

#  ==========================================================================
#  Обратите внимание на технику передачи/конвейера вывода «tr» самой себе,
#+ но с разными аргументами и/или опциями в каждой последовательной передаче.
#  ==========================================================================


for word in $wlist                      #  Важно:
                                        #  $wlist должно быть без кавычек.
                                        # "$wlist" не работает.
                                        #  Почему не работает?
do
  strlen=${#word}                       #  Длина строки.
  if [ "$strlen" -lt "$MINSTRLEN" ]     #  Сброс всех строк короче этой длины.
  then
    continue
  fi

  grep -Fw $word "$WORDFILE"            #  Совпадение только целых слов.
#      ^^^                              #  Опции "фиксированные строки"
                                        #+ и "целые слова".
done

exit $?
```

Сравнение

`diff`, `patch`

`diff` - гибкая утилита сравнения файлов. Она сравнивает указанные файлы последовательно, строка-за-строкой. В некоторых приложениях, например сравнении слов по словарю, полезна фильтрация файлов **`sort`** и **`uniq`**, перед передачей их в **`diff`**. **`diff file-1 file-2`** выводит отличающиеся строки файлов выводя символы перевода каретки в каждой конкретной строке.

Опция —`side-by-side` в **`diff`** выводит каждый сравниваемый файл построчно, в

отдельных столбцах, с отметкой не совпадающих строк. Опции `-c` и `-u` также делают вывод команды легким для понимания.

Имеются несколько интерфейсов для **`diff`**, например **`sdiff`**, **`wdiff`**, **`xdiff`** и **`mgdiff`**.

Команда **`diff`** возвращает статус выхода `0`, если сравниваемые файлы одинаковы и `1`, если они различаются (или `2`, при сравнении *бинарных* файлов). Это позволяет использовать **`diff`** в конструкциях проверки сценария оболочки (см. ниже)

Обычное использование **`diff`** - это вывод разницы файлов для использования с **`patch`**. Опция `-e` выводит файлы подходящие для сценариев **`ed`** или **`ex`**.

`patch`: гибкая утилита версий. Указывая какой-то файл, созданный **`diff`**, **`patch`** может обновить предыдущую версию пакета до новой версии. Гораздо удобней распространять файл относительно небольшого «**`diff`**», чем пересматривать вновь все тело пакета. «Патчи» ядра стали основным методом распространения частых релизов ядра в Linux.

```
patch -p1 <patch-file
# Принимает все изменения, перечисленные в 'patch-file'
# и применяет их к файлам, которые в нем упоминаются.
# Это обновление до более новой версии пакета.
```

Патчинг ядра:

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
# Обновление исходных текстов ядра, с помощью 'patch'.
# Из README, документации ядра Linux,
# анонимного автора (Alan Cox?).
```

Команда **`diff`** может также рекурсивно сравнивать директории (содержащих файлы).

```
bash$ diff -r ~/notes1 ~/notes2
Only in /home/bozo/notes1: file02
 Only in /home/bozo/notes1: file03
 Only in /home/bozo/notes2: file04
```

С помощью **`zdiff`** сравниваются сжатые (*gzip)* файлы.

С помощью **diffstat** создается гистограмма (график распределения точек) вывода из **diff**.

diff3, merge

diff3 - это расширенная версия **diff**, которая сравнивает три файла за раз. Эта команда возвращает статус выхода 0, после успешного выполнения, но к сожалению это не дает никакой информации о результатах сравнения.

```
bash$ diff3 file-1 file-2 file-3
====
 1:1c
   Это строка 1  "file-1".
 2:1c
   Это строка 1  "file-2".
 3:1c
   Это строка 1  "file-3"
```

Команда **merge** является интересным дополнением к *diff3*. Синтаксисом **merge** является **Mergefile file1 file2**. Результатом является вывод в Mergefile изменений, которые перешли из file1 в file2. Рассматривайте эту команду, как урезанную версию **patch**.

sdiff

Сравнение и/или редактирование двух файлов для объединения их в выходном файле. Из-за интерактивного характера, от этой команды, в сценарии будет мало пользы.

cmp

Команда **cmp** это упрощенная версия **diff**, выше. В то время как **diff** сообщает различия между двумя файлами, **cmp** просто показывает где они отличаются.

Как и **diff**, **cmp** возвращает статус выхода 0, если сравниваемые файлы являются одинаковыми и 1, если они отличаются. Это позволяет использовать ее в конструкции проверки сценария оболочки.

Пример 16-35. Сравнение двух файлов сценария с помощью *cmp*.

```
#!/bin/bash
# file-comparison.sh

ARGS=2  # В сценарии ожидается два аргумента.
E_BADARGS=85
```

```
E_UNREADABLE=86

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` file1 file2"
  exit $E_BADARGS
fi

if [[ ! -r "$1" || ! -r "$2" ]]
then
  echo "Оба сравниваемых файла должны существовать и быть доступными для
        чтения."
  exit $E_UNREADABLE
fi

cmp $1 $2 &> /dev/null
#   Перенаправление в /dev/null похоронит вывод команды "cmp".
#   cmp -s $1 $2  имеет тот же результат ("-s" флаг молчания в "cmp")
#   Спасибо за пояснение  Anders Gustavsson.
#
#  Так же работает 'diff', т.е.,
#+ diff $1 $2 &> /dev/null

if [ $? -eq 0 ]         # Проверка статуса выхода команды "cmp".
then
  echo "Файл \"$1\" идентичен файлу \"$2\"."
else
  echo "Файл \"$1\" отличается от файла \"$2\"."
fi

exit 0
```

С помощью **zcmp** файлы сжимаются (***gzip***).

comm

Универсальная утилита сравнения. Для этого файлы должны быть отсортированы.

comm *-опции first-file second-file*

comm file-1 file-2 выводит три колонки:

- колонка 1 = строки, уникальные для `file-1`
- колонка 2 = строки, уникальные для `file-2`
- колонка 3 = строки общие для обоих.

Опции позволяют подавлять вывод одной или нескольких колонок.

- `-1` подавляет колонку `1`

- `-2` подавляет колонку `2`
- `-3` подавляет колонку `3`
- `-12` подавляет обе колонки `1` и `2`, и т.п.

Эта команда полезна для сравнения «словарей» или *списков слов* - сортированных текстовых файлов с одним словом на каждой строке.

Утилиты

basename

Удаляет сведения из имени файла о пути, выводя только имя файла. Конструкция **basename $0** позволяет сценарию узнать свое имя, то есть имя, которым он был вызван. Она может быть использована в сообщениях "**usage**", если, например, сценарий вызывается без аргументов :

```
echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname

Удаляет **basename** из имени файла, выводит только сведения о пути.

basename и **dirname** могут работать с любой произвольной строкой. Аргументу, в этом случае, не нужно ссылаться на существующий файл или даже имя файла (см. Пример A-7).

Пример 16-36. ***basename*** **и** ***dirname***

```
#!/bin/bash

address=/home/bozo/daily-journal.txt

echo "Basename of /home/bozo/daily-journal.txt = `basename $address`"
echo "Dirname of /home/bozo/daily-journal.txt = `dirname $address`"
echo
echo "My own home is `basename ~/`."         # `basename ~` то же работает.
echo "The home of my home is `dirname ~/`."  # `dirname ~`  то же работает.

exit 0
```

split, csplit

Это утилиты для разделения файла на более мелкие части. Они обычно используются

для разделения больших файлов при резервном копировании на дискеты или подготовке электронной почты, или их загрузке.

Команда **csplit** разбивает файл в зависимости от *контекста*, разбиение происходит в местах соответствия шаблонов.

Пример 16-37. Сценарий копирующий себя частями

```
#!/bin/bash
# splitcopy.sh

#  Сценарий, который разбивается на части,
#+ затем собирает части в точную копию оригинала.

CHUNKSIZE=4    #  Начальный размер блока разбиваемых файлов.
OUTPREFIX=xx   #  Префиксы csplit, по умолчанию,
               #+ файлы с "xx" ...

csplit "$0" "$CHUNKSIZE"

# Несколько закомментированных строк для заполнения ...
# Строка 15
# Строка 16
# Строка 17
# Строка 18
# Строка 19
# Строка 20

cat "$OUTPREFIX"* > "$0.copy"  # Соединение частей.
rm "$OUTPREFIX"*               # Удаляем части.

exit $?
```

Кодирование и шифрование

sum, cksum, md5sum, sha1sum

Это утилиты для создания *контрольных сумм* (***checksums***). *Контрольная сумма* представляет собой число, [3] математически рассчитанное из содержимого файла, с целью проверки его целостности. Сценарий, в целях безопасности, может ссылаться на список контрольных сумм, как гарантию того, что содержимое ключевых системных файлов не было изменено или повреждено. Для безопасности приложений, используйте команду **md5sum** (**m**essage **d**igest **5** check**sum**), или еще лучше, более новую **sha1sum** (**S**ecure **H**ash **A**lgorithm). [4]

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz

bash$ echo -n "Top Secret" | cksum
3391003827 10



bash$ md5sum /boot/vmlinuz
```

```
0f43eccea8f09e0a0b2b5cf1dcf333ba  /boot/vmlinuz

bash$ echo -n "Top Secret" | md5sum
8babc97a6f62a4649716f4df8d61728f  -
```

Команда **cksum** показывает размер цели, в байтах, будь это файл или stdout.

Команды **md5sum** и **sha1sum** выводят ***mupe***, когда они получают ввод из stdout.

Пример 16-38. Проверка целостности файлов

```
#!/bin/bash
# file-integrity.sh: Проверка, были ли подделаны файлы в заданной директории.

E_DIR_NOMATCH=80
E_BAD_DBFILE=81

dbfile=File_record.md5
# Имя файла для хранения записей (файл базы данных).


set_up_database ()
{
  echo ""$directory"" > "$dbfile"
  # Запись имени директории в первую строку файла.
  md5sum "$directory"/* >> "$dbfile"
  # Добавление контрольной суммы md5 и имен файлов.
}

check_database ()
{
  local n=0
  local filename
  local checksum

  # ------------------------------------------- #
  #  Эта проверка файла, должно быть, не нужна,
  #+ но лучше перестраховаться, чем жалеть.

  if [ ! -r "$dbfile" ]
  then
    echo "Не удается прочитать контрольную сумму файла базы данных!"
    exit $E_BAD_DBFILE
  fi
  # ------------------------------------------- #

  while read record[n]
  do

    directory_checked="${record[0]}"
    if [ "$directory_checked" != "$directory" ]
    then
      echo "Директории не совпадают!"
```

```
      # Попробуйте использовать файл другой директории.
      exit $E_DIR_NOMATCH
    fi

    if [ "$n" -gt 0 ]   # Не имя директории.
    then                # Директории не совпадают
      filename[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
      #  запись md5sum перезаписывается,
      #+ сначала контрольная сумма, потом имя файла.
      checksum[n]=$( md5sum "${filename[n]}" )


      if [ "${record[n]}" = "${checksum[n]}" ]
      then
        echo "${filename[n]} без изменений."

        elif [ "`basename ${filename[n]}`" != "$dbfile" ]
              #  Пропускаем контрольную сумму файла базы данных, так как она
              #+ будет меняться с каждым вызовом сценария.
              #  ---
              #  К сожалению, это означает, что при выполнении этого
              #+ сценария в $PWD, подделка контрольной суммы
              #+ файла базы данных не будет обнаружена.
              #  Упражнение: Исправьте это.
        then
          echo "${filename[n]} : КОНТРОЛЬНАЯ СУММА ОШИБОЧНА!"
        # Файл был изменен с момента последней проверки.
        fi

      fi



    let "n+=1"
  done <"$dbfile"       # Чтение контрольной суммы файла базы данных.

}

# ============================================================ #
# main ()

if [ -z  "$1" ]
then
  directory="$PWD"      #  Если не указана,
else                    #+ используется текущая рабочая директория.
  directory="$1"
fi

clear                   # Очистка экрана.
echo " Выполнение проверки целостности файлов в $directory"
echo

# ------------------------------------------------------------------- #
  if [ ! -r "$dbfile" ] # Нужно создать файл базы данных?
  then
    echo "Создание файла базы данных, \""$directory"/"$dbfile"\"."; echo
    set_up_database
  fi
# ------------------------------------------------------------------- #

check_database          # Фактическая работа.
```

```
echo

#  Вы можете перенаправить stdout этого сценарий в файл, особенно если
#+ проверяемая директория содержит много файлов.

exit 0

#  Для более тщательной проверки целостности файла,
#+ рассмотрите пакет «Tripwire»,
#+ http://sourceforge.net/projects/tripwire/.
```

Также посмотрите Пример A-19, Пример 36-16 и Пример 10-2 творческого использования команды **md5sum**.

Были сообщения, что 128-битная **md5sum** может быть взломана, поэтому более безопасным новым дополнением является 160-битная **sha1sum**.

```
bash$ md5sum testfile
e181e2c8720c60522c4c4c981108e367  testfile


bash$ sha1sum testfile
5d7425a9c08a66c3177f1e31286fa40986ffc996  testfile
```

Консультанты по безопасности указали, что даже **sha1sum** может быть нарушена. К счастью, новые дистрибутивы Linux включают в себя команды более длиной битности **sha224sum**, **sha256sum**, **sha384sum** и **sha512sum** .

uuencode

Эта утилита кодирует двоичные файлы (изображения, звуковые файлы, сжатые файлы и т.д.) в символы ASCII, что делает их пригодными для передачи в теле сообщения электронной почты или в публикации новостей. Она особенно полезна, когда не доступно кодирование MIME (мультимедиа).

uudecode

Меняет кодировку файлов декодированных *uuencoded*, обратно, в исходные двоичные файлы.

Пример 16-39. Шифрование файлов Uudecoding

```
#!/bin/bash
# Кодирование uuencoded всех файлов в текущей рабочей директории.

lines=35        # Разрешаем 35 строк для заголовка (очень щедро).

for File in *   # Проверяем все файлы в $PWD.
do
  search1=`head -n $lines $File | grep begin | wc -w`
  search2=`tail -n $lines $File | grep end | wc -w`
```

```
  #  Uuencode файлы имеют «начало» в начале,
  #+ и "окончание" около конца.
  if [ "$search1" -gt 0 ]
  then
    if [ "$search2" -gt 0 ]
    then
      echo "uudecoding - $File -"
      uudecode $File
    fi
  fi
done

#  Обратите внимание, что этот сценарий обманывает себя,
#+ считая, что это файл uuencode, потому что он содержит
#+ сразу "начало" и "окончание".

#  Упражнение:
#  --------
#  Измените этот сценарий для проверки заголовка каждого файла группы
#+ новостей и перехода к следующему, если заголовок не найден.

exit 0
```

Команда **fold -s** может быть полезна (возможно конвейеру) для обработки декодированных **uudecode** больших текстовых сообщений, загруженных из группы новостей Usenet.

mimencode, mmencode

Команды **mimencode** и **mmencode** обрабатывают вложенное в электронную почту кодированное мультимедиа. Хотя *почтовые агенты пользователя* (например, pine или kmail) обычно обрабатывают их автоматически, эти утилиты, в частности, позволяют манипулировать такими вложениями вручную, из командной строки, или в режиме *пакетной* обработки с помощью сценария оболочки.

crypt

В одно время это была стандартная утилита шифрования файлов UNIX. [5] Политически мотивированные правительственные постановления, запрещающие экспорт криптографического программного обеспечения, привели к исчезновению **crypt** из огромного мира UNIX, и она до сих пор отсутствует в большинстве дистрибутивов Linux. К счастью, программисты создали ряд достойных альтернатив, среди них собственный **cruft** автора (см. Пример A-4).

openssl

Является *Open Source* реализацией шифрования ***S**ecure **S**ockets **L**ayer* с открытым исходным кодом.

```
# Чтобы зашифровать файл:
openssl aes-128-ecb -salt -in file.txt -out file.encrypted \
```

```
-pass pass:мой_пароль
#          ^^^^^^^^^^    Пароль пользователя.
#       aes-128-ecb      это выбор метода шифрования.

# Расшифровка файла:
openssl aes-128-ecb -d -salt -in file.encrypted -out file.txt \
-pass pass:мой_пароль
#          ^^^^^^^^^^    Пароль пользователя.
```

Туннелирование `openssl` в/из **`tar`** делает возможным шифрование дерева директории целиком.

```
# Шифрование директории:

sourcedir="/home/bozo/testfiles"
encrfile="encr-dir.tar.gz"
password=my_secret_password

tar czvf - "$sourcedir" |
openssl des3 -salt -out "$encrfile" -pass pass:"$password"
#       ^^^^   Используется шифрование des3.
# Запись зашифрованного файла "encr-dir.tar.gz" в текущую рабочую
# директорию.

# Расшифровка итогового тарболла:
openssl des3 -d -salt -in "$encrfile" -pass pass:"$password" |
tar -xzv
# Расшифровывается и распаковывается в текущую рабочую директорию.
```

Конечно, **`openssl`** имеет много других применений, например получение подписанных сертификатов для веб-сайтов. См. справочные страницы.

`shred`

Надежное стирание файла, путем перезаписи его, перед удалением, несколько раз случайными битами. Эта команда имеет тот же эффект, как Пример 16-61, но делает это более тщательно и элегантно.

Это одна из *файловых утилит* GNU.

Передовые технологии могут иметь возможность восстановления содержимого файла, даже после применения **`shred`**.

Разные

`mktemp`

Создает *временный* файл [6] с «уникальным» именем. При вызове из командной строки без дополнительных аргументов создает файл нулевой длины в директории `/tmp`.

```
bash$ mktemp
/tmp/tmp.zzsvql3154
```

```
PREFIX=filename
tempfile=`mktemp $PREFIX.XXXXXX`
#                         ^^^^^^ Нужны, по крайней мере, 6 заполнителей
#+                               в имени временного файла.
#   Если нет шаблона файла, то
#+ "tmp.XXXXXXXXXX" по умолчанию.

echo "tempfile name = $tempfile"
# tempfile name = filename.QA2ZpY
#                 или что-то подобное...

#  Создает файл в текущей рабочей директории с этим именем и с
#+ правами доступа к файлу 600.
#  "umask 177" поэтому необязателен, но это, тем не менее,
#+ хороший стиль программирования.
```

make

Утилита для создания и компиляции бинарных пакетов. Также может использоваться с любым набором операций, вызывающих изменения при добавлении в исходные файлы.

Команда *make* проверяет Makefile, список зависимостей файлов и операций, которые будут осуществляться.

Утилита *make* является, по сути, мощным языком сценария, подобно Bash, но с возможностью определения зависимостей. Для глубокого охвата этого полезного инструмента, см. сайт документации программного обеспечения GNU.

install

Команда копирования файла специального назначения, похожа на **cp**, но позволяет устанавливать права доступа и атрибуты скопированных файлов. Эта команда возможна для индивидуальной установки пакетов программного обеспечения, и как таковая она часто встречается в Makefiles (в части *make install :*). Он может оказаться полезной в сценариях установки.

dos2unix

Утилита, написанная и сопровождаемая Benjamin Lin, конвертирует текстовые файлы в формате DOS (строки заканчиваются CR-LF) в формат UNIX (строки заканчиваются только LF), и наоборот.

ptx

Команда **ptx [targetfile]** выводит перестановленный индекс (список ссылок) указанного файла. Далее она может фильтроваться и форматироваться в конвейер.

more, less

Постраничный вывод текстового файла или потока в stdout, страница за раз. Могут

использоваться в фильтре вывода `stdout` . . . или сценария.

Интересным применением ***more*** является «испытание» последовательности команд, для предотвращения потенциально неприятных последствий.

```
ls /home/bozo | awk '{print "rm -rf " $1}' | more
#                                           ^^^^

# Проверка работы следующей (катастрофической) командной строки:
#      ls /home/bozo | awk '{print "rm -rf " $1}' | sh
#      Руки прочь от выполнения в оболочке ...       ^^
```

Постраничный вывод ***less*** имеет интересное свойство делать форматированный вывод справочных страниц источника. См. Пример A-39.

Примечания

[1] **Архив**, в обсуждаемом здесь смысле, это просто набор связанных файлов, хранящихся в одном месте.

[2] А *tar czvf ArchiveName.tar.gz* * будет включать в себя файлы с точкой в поддиректориях ниже текущей рабочей директории. Это незарегистрированная GNU «функция» **tar** .

[3] Контрольная сумма может быть выражена, как *шестнадцатеричное* число или по другим основаниям.

[4] Для еще большей безопасности используйте команды *sha256sum*, *sha512* и *sha1pass*.

[5] Это симметричный блочный шифр, используемый для шифрования файлов на одной системе или локальной сети, в отличие от класса шифрования *общественного ключа*, из которого хорошо известным примером является *pgp*.

[6] При вызове с параметром `-d` создает временную директорию.

16.6. Команды соединения

Некоторые из следующих команд находят применение в передачи данных и анализе сети, а также в вычислении спамеров.

Информация и статистика

`host`

Поиск информации об Интернет-узле по имени или IP-адресу при помощи DNS.

```
bash$ host surfacemail.com
surfacemail.com. Имеет адрес 202.92.42.236
```

ipcalc

Отображает информацию IP хоста. С опцией `-h` **ipcalc** выполняет обратный поиск по DNS, поиск имени хоста (сервера) по IP-адресу.

```
bash$ ipcalc -h 202.92.42.236
HOSTNAME=surfacemail.com
```

nslookup

Производит в интернете "поиск имени хоста сервера" по IP адресу. Она, по существу, эквивалентна **ipcalc -h** или **dig -x.** Команда может быть запущена интерактивно или не интерактивно, т.е. из сценария.

Команда **nslookup** «устаревшая», но все равно полезная.

```
bash$ nslookup -sil 66.97.104.180
nslookup kuhleersparnis.ch
 Server:         135.116.137.2
 Address:        135.116.137.2#53

 Non-authoritative answer:
 Name:   kuhleersparnis.ch
```

dig

Domain **I**nformation **G**roper. Подобно `nslookup`, **dig** производит поиск имени хоста сервера. Может запускаться из командной строки или из сценария.

Наиболее интересные опции *dig* это `+time=N` создает таймаут между запросами в *N* секунд, `+nofail` для продолжения запроса серверам, пока не будет получен ответ, а `-x` производит обратный поиск адреса.

Сравните вывод **dig -x** с **ipcalc -h** и **nslookup**.

```
bash$ dig -x 81.9.6.2
;; Got answer:
 ;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 11649
 ;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0
```

```
;; QUESTION SECTION:
;2.6.9.81.in-addr.arpa.         IN      PTR

;; AUTHORITY SECTION:
6.9.81.in-addr.arpa.    3600    IN      SOA     ns.eltel.net. noc.eltel.net.
2002031705 900 600 86400 3600

;; Query time: 537 msec
;; SERVER: 135.116.137.2#53(135.116.137.2)
;; WHEN: Wed Jun 26 08:35:24 2002
;; MSG SIZE  rcvd: 91
```

Пример 16-40. Узнаем откуда спамер

```
#!/bin/bash
# spam-lookup.sh: Обнаружение спаммера злоупотребляющего соединением.
# Спасибо, Michael Zick.

# Проверка аргумента командной строки.
ARGCOUNT=1
E_WRONGARGS=85
if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` доменное имя"
  exit $E_WRONGARGS
fi


dig +short $1.contacts.abuse.net -c in -t txt
# Так же пробуем:
#     dig +nssearch $1
#     Пытаемся найти "солидные имена серверов" и выводим отчеты SOA.

# Это так же будет работать:
#     whois -h whois.abuse.net $1
#           ^^ ^^^^^^^^^^^^^^^^  Заданный хост.
#     Можем поискать даже несколько спамеров, т.е."
#     whois -h whois.abuse.net $spamdomain1 $spamdomain2 . . .


#  Упражнение:
#  --------
#  Расширьте функциональность этого сценария, таким образом,
#+ чтобы он автоматически отправлял по электронной почте
#+ уведомление контактным адресам ISP.
#  Подсказка: с помощью команды "mail".

exit $?

# spam-lookup.sh chinatietong.com
#                Известный спам домен.

# "crnet_mgr@chinatietong.com"
# "crnet_tec@chinatietong.com"
# "postmaster@chinatietong.com"
```

```
#  Более сложную версию этого сценария
#+ смотри на странице SpamViz, http://www.spamviz.net/index.html.
```

Пример 16-41. Анализ спам домена

```
#! /bin/bash
# is-spammer.sh: Выявление спам доменов

# $Id: is-spammer, v 1.4 2004/09/01 19:37:52 mszick Exp $
# Строка выше это информация RCS ID.
#
#  Это упрощенная версия сценария "is_spammer.bash
#+ в приложении Contributed Scripts.

# is-spammer <domain.name>

# С помощью внешней программы: 'dig'
# Проверена версия: 9.2.4rc5

# Функциональность.
# Использует IFS для анализа строк назначаемых в массивы.
# И даже делает что-то полезное: Проверяет черный список электронной почты.

# Использует доменные.имена из текста:
# http://www.good_stuff.spammer.biz/just_ignore_everything_else
#                       ^^^^^^^^^^^
# Или domain.name(s) из других адресов e-mail:
# Really_Good_Offer@spammer.biz
#
# только в качестве аргумента в этом сценарии.
#(PS: Запустите ваше соединение Inet)
#
# Поэтому, вызываем этот сценарий двумя способами выше:
#       is-spammer.sh spammer.biz


# Пробелы == :Пропуск:Табуляция:Перевод строки:Возврат каретки:
WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'

# Без пробелов == Перевод строки:Возврат каретки
No_WSP=$'\x0A'$'\x0D'

# В десятичных ip адресах поля разделяются точкой.
ADR_IFS=${No_WSP}'.'

# Получение источника текстовой записи.
# get_txt <error_code> <list_query>
get_txt() {

    # Проверка назначения точек $1.
    local -a dns
    IFS=$ADR_IFS
    dns=( $1 )
    IFS=$WSP_IFS
    if [ "${dns[0]}" == '127' ]
    then
        # Видно, что if здесь необходимо.
```

```
        echo $(dig +short $2 -t txt)
    fi
}

# Получение начальной записи dns адреса.
# chk_adr <rev_dns> <list_server>
chk_adr() {
    local reply
    local server
    local reason

    server=${1}${2}
    reply=$( dig +short ${server} )

    # Если reply может быть кодом ошибки
    if [ ${#reply} -gt 6 ]
    then
        reason=$(get_txt ${reply} ${server} )
        reason=${reason:-${reply}}
    fi
    echo ${reason:-' не в черном списке.'}
}

# Нужно получить IP адрес из имени.
echo 'Получение адреса: '$1
ip_adr=$(dig +short $1)
dns_reply=${ip_adr:-' нет ответа '}
echo ' Адрес: '${dns_reply}

# Допустимый ответ это, по крайней мере, 4 цифры плюс 3 точки.
if [ ${#ip_adr} -gt 6 ]
then
    echo
    declare query

    # Проверяем назначения точек.
    declare -a dns
    IFS=$ADR_IFS
    dns=( ${ip_adr} )
    IFS=$WSP_IFS

    # Изменение порядка октетов в порядке запросов dns.
    rev_dns="${dns[3]}"'.'"${dns[2]}"'.'"${dns[1]}"'.'"${dns[0]}"'.'

# Смотрите: http://www.spamhaus.org (Консервативный, хорошо поддерживаемый)
    echo -n 'spamhaus.org выдает: '
    echo $(chk_adr ${rev_dns} 'sbl-xbl.spamhaus.org')

# Смотрите: http://ordb.org (Открытая рассылка почты)
    echo -n '   ordb.org  выдает: '
    echo $(chk_adr ${rev_dns} 'relays.ordb.org')

# Смотрите: http://www.spamcop.net/ (Сюда Вы можете сообщать о спамерах)
    echo -n ' spamcop.net выдает: '
    echo $(chk_adr ${rev_dns} 'bl.spamcop.net')

# # # другие детали черного списка # # #

# Смотрите: http://cbl.abuseat.org.
    echo -n ' abuseat.org выдает: '
    echo $(chk_adr ${rev_dns} 'cbl.abuseat.org')
```

```
# Смотрите: http://dsbl.org/usage (Различная рассылка почты)
    echo
    echo 'Distributed Server Listings'
    echo -n '       list.dsbl.org выдает: '
    echo $(chk_adr ${rev_dns} 'list.dsbl.org')

    echo -n '   multihop.dsbl.org выдает: '
    echo $(chk_adr ${rev_dns} 'multihop.dsbl.org')

    echo -n 'unconfirmed.dsbl.org выдает: '
    echo $(chk_adr ${rev_dns} 'unconfirmed.dsbl.org')

else
    echo
    echo 'Не используйте этот адрес.'
fi

exit 0

# Упражнение:
# --------

# 1) Проверьте аргументы в сценарии
#    и выйдите, при необходимости, с соответствующим сообщением об ошибке.

# 2) Проверьте, вызывается ли сценарий он-лайн,
#    и выйдите, при необходимости, с соответствующим сообщением об ошибке.

# 3) Замените общие переменные на «жестко кодированные» домены BHL.

# 4) Установите тайм аут для сценария с помощью опции "+time="
     команды 'dig'.
```

Гораздо более сложная версия сценария выше - Пример A-28.

`traceroute`

Трассировка маршрута принятых пакетов, отправленных на удаленный хост. Эта команда работает с LAN, WAN, или иным Интернетом. Удаленный хост может быть указан IP адресом. Вывод команды может быть отфильтрован **`grep`** или **sed** через конвейер.

```
bash$ traceroute 81.9.6.2
traceroute to 81.9.6.2 (81.9.6.2), 30 hops max, 38 byte packets
 1  tc43.xjbnnbrb.com (136.30.178.8)  191.303 ms  179.400 ms  179.767 ms
 2  or0.xjbnnbrb.com (136.30.178.1)  179.536 ms  179.534 ms  169.685 ms
 3  192.168.11.101 (192.168.11.101)  189.471 ms  189.556 ms *
 ...
```

`ping`

Передача пакета *ICMP ECHO_REQUEST* другой машине, в локальной или удаленной сети. Это диагностический инструмент для проверки сетевого соединения и должен использоваться с осторожностью.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255 time=709
usec
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255 time=286
usec

 --- localhost.localdomain ping statistics ---
 2 packets transmitted, 2 packets received, 0% packet loss
 round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

Успешный **ping** возвращает статус выхода 0. Его можно проверить сценарием.

```
  HNAME=news-15.net  # Пресловутый спамер.
# HNAME=$HOST        # Отладка: проверка локальной сети.
  count=2            # Отсылаются только два пинга.

if [[ `ping -c $count "$HNAME"` ]]
then
  echo ""$HNAME" все еще работает и рассылает спам."
else
  echo ""$HNAME" Кажется выключен. Жаль."
fi
```

whois

Выполняет поиск DNS (**D**omain **N**ame **S**ystem). Опция `-h` позволяет задать для запроса конкретный ***whois***-сервер. См. Пример 4-6 и Пример 16-40.

finger

Получает информацию о пользователях сети. При необходимости, эта команда выводит пользовательские файлы `~/.plan`, `~/.project` и `~/.forward`, если они существуют.

```
bash$ finger
Login  Name           Tty      Idle  Login Time   Office     Office Phone
 bozo   Bozo Bozeman   tty1        8  Jun 25 16:59              (:0)
 bozo   Bozo Bozeman   ttyp0          Jun 25 16:59              (:0.0)
 bozo   Bozo Bozeman   ttyp1          Jun 25 17:07              (:0.0)



bash$ finger bozo
Login: bozo                             Name: Bozo Bozeman
```

```
Directory: /home/bozo                  Shell: /bin/bash
Office: 2355 Clown St., 543-1234
On since Fri Aug 31 20:13 (MST) on tty1    1 hour 38 minutes idle
On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
On since Fri Aug 31 20:13 (MST) on pts/1
On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
Mail last read Tue Jul  3 10:08 2007 (MST)
No Plan.
```

Из соображений безопасности многие сети отключают **finger** и ассоциированные с ней демоны. [1]

chfn

Изменяет информацию, раскрываемую командой **finger**.

vrfy

Проверяет адреса электронной почты.

Эта команда может отсутствовать в новых дистрибутивах Linux.

Удаленный доступ к хосту

sx, **rx**

Набор команд **sx** и **rx** служит для передачи файлов на/с удаленный хост с помощью протокола **xmodem**. Они являются частью коммуникационных пакетов, таких как **minicom**.

sz, **rz**

Набор команд **sz** и **rz** служит для передачи файлов на/с удаленный хост с использованием протокола **zmodem**. *Zmodem* имеет определенные преимущества перед *xmodem*, такие как большая скорость передачи и возобновление прерванной передачи файлов. Как ***sx*** и ***rx***, они являются частью коммуникационных пакетов.

ftp

Утилита и протокол для загрузки/скачивания файлов на/с удаленный хост. FTP-сеанс может быть автоматизирован сценарием (см. Пример 19-6 и Пример A-4).

uucp, **uux**, **cu**

uucp: *копирование с UNIX на UNIX*. Это коммуникационный пакет для обмена файлами между серверами UNIX. Сценарий оболочки является эффективным способом обработки последовательности команд **uucp**.

С появлением Интернета и электронной почты, **uucp**, кажется, канула в безвестность, но она по-прежнему существует и остается вполне работоспособной в ситуациях, когда подключение к интернету не доступно или отсутствует. Преимуществом **uucp** является отказоустойчивость, то есть, при восстановлении соединения после сбоя, операция копирования будет возобновлена с места, где была прервана.

---

uux: *выполнение с UNIX на UNIX*. Выполнение команд на удаленной системе. Эта команда является частью пакета **uucp**.

---

cu: вызов удаленной системы и подключение, как простого терминала. Это своего рода упрощенная версия **telnet**. Эта команда так же является частью пакета **uucp**.

telnet

Утилита и протокол для подключения к удаленному хосту.

Протокол *telnet* имеет дыры в безопасности и поэтому, его надо избегать. Использование его в сценариях оболочки *не рекомендуется*.

wget

Утилита **wget** *не интерактивно* извлекает или загружает файлы с Интернет или FTP-сайта. Она прекрасно работает в сценариях.

```
wget -p http://www.xyz23.com/file01.html
#  Опция -p или --page-requisite  заставляет wget извлекать все файлы,
#+ необходимые для вывода указанной страницы
wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -O $SAVEFILE
#  Опция -r  рекурсивно двигается и извлекает все ссылки на указанном сайте.
wget -c ftp://ftp.xyz25.net/bozofiles/filename.tar.bz2
#  Опция -c  позволяет wget возобновить прерванные закачки.
#  Она работает с серверами ftp и множеством сайтов HTTP.
```

Пример 16-42. Получение котировок акций

```
#!/bin/bash
# quote-fetch.sh: Загрузка котировок акций.


E_NOPARAMS=86

if [ -z "$1" ]  # Необходимо указать количество (символов) для получения.
  then echo "Usage: `basename $0` stock-symbol"
  exit $E_NOPARAMS
```

```
fi

stock_symbol=$1

file_suffix=.html
# Извлечение HTML файла, а так же соответствующего ему имени.
URL='http://finance.yahoo.com/q?s='
# Финансовый таблоид Yahoo, с суффиксами запросов акций.

# ------------------------------------------------------------
wget -O ${stock_symbol}${file_suffix} "${URL}${stock_symbol}"
# ------------------------------------------------------------


# Поиск на http://search.yahoo.com:
# ------------------------------------------------------------
# URL="http://search.yahoo.com/search?fr=ush-news&p=${query}"
# wget -O "$savefilename" "${URL}"
# ------------------------------------------------------------
# Сохранение списка URL-адресов.

exit $?

# Упражнения:
# ---------
#
# 1) Добавьте проверку для запустившего сценарий пользователя,
№+   что он работает он-лайн. (Подсказка: Анализируйте вывод
#+   из 'ps-ax' для «ppp» или «connect».
# 2) Измените этот сценарий для получения прогноза погоды,
#+   беря, в качестве аргумента, почтовый индекс пользователя.
```

Так же см. Пример A-30 и Пример A-31.

lynx

Web и файловый браузер **lynx** может использоваться внутри сценария (с опцией `-dump`) для не интерактивного извлечения файлов с ***ftp*** или веб-сайта.

```
lynx -dump http://www.xyz23.com/file01.html >$SAVEFILE
```

С опцией `-traversal`, **lynx** стартует с указанным HTTP URL в качестве аргумента, затем «проходит» через все ссылки, расположенные на этом конкретном сервере. С помощью опции `-crawl`, выводит текстовую страницу в лог-файл.

rlogin

Удаленный вход, инициация сеанса на удаленном хосте. Эта команда имеет дыры в безопасности, так что вместо нее используют **ssh**.

rsh

***R**emote **sh**ell*, выполняет команды на удаленном хосте. Она имеет дыры в безопасности, поэтому вместо нее используют **ssh**.

rcp

***R**emote **cop**y*, копирует файлы между двумя различными сетевыми машинами.

rsync

***R**emote **sync**hronize*, обновляет (синхронизирует) файлы между двумя различными сетевыми машинами.

```
bash$ rsync -a ~/sourcedir/*txt /node1/subdirectory/
```

Пример 16-43. Обновление FC4

```
#!/bin/bash
# fc4upd.sh

# Автор сценария: Frank Wang.
# Стилистика немного подправлена автором ABS Guide.
# В ABS Guide приводится с разрешения.


#  Загрузка обновлений Fedora Core 4 с зеркала сайта с помощью rsync.
#  Будет работать и с Fedora Cores -- 5, 6 ...
#  Загружает только свежий пакет, если существует несколько версий.

URL=rsync://distro.ibiblio.org/fedora-linux-core/updates/
# URL=rsync://ftp.kddilabs.jp/fedora/core/updates/
# URL=rsync://rsync.planetmirror.com/fedora-linux-core/updates/

DEST=${1:-/var/www/html/fedora/updates/}
LOG=/tmp/repo-update-$(/bin/date +%Y-%m-%d).txt
PID_FILE=/var/run/${0##*/}.pid

E_RETURN=85        # На всякий случай.


# Основные опции rsync
# -r: рекурсивная загрузка
# -t: запас времени
# -v: полный

OPTS="-rtv --delete-excluded --delete-after --partial"

# шаблон включений rsync
# Ведущий слэш вызывает абсолютный путь соответствующего имени.
INCLUDE=(
    "/4/i386/kde-i18n-Chinese*"
#   ^                       ^
# Кавычки необходимы для предотвращения подстановки шаблона.
)
```

```
# шаблон исключений rsync
# Временно закомментируйте нежелательные пакеты, используя "#" ...
EXCLUDE=(
    /1
    /2
    /3
    /testing
    /4/SRPMS
    /4/ppc
    /4/x86_64
    /4/i386/debug
   "/4/i386/kde-i18n-*"
   "/4/i386/openoffice.org-langpack-*"
   "/4/i386/*i586.rpm"
   "/4/i386/GFS-*"
   "/4/i386/cman-*"
   "/4/i386/dlm-*"
   "/4/i386/gnbd-*"
   "/4/i386/kernel-smp*"
#  "/4/i386/kernel-xen*"
#  "/4/i386/xen-*"
)

init () {
    #  Пусть команда pipe возвращает возможную ошибку rsync, например,
    #+ падение скорости сети.
    set -o pipefail                  # Нововведение в Bash, версии 3.

    TMP=${TMPDIR:-/tmp}/${0##*/}.$$  # Сохранение списка загрузки.
    trap "{
        rm -f $TMP 2>/dev/null
    }" EXIT                          # Очистка временного файла при выходе.
}

check_pid () {
# Проверка существования процесса.
    if [ -s "$PID_FILE" ]; then
        echo "Существование PID файла. Проверка ..."
        PID=$(/bin/egrep -o "^[[:digit:]]+" $PID_FILE)
        if /bin/ps --pid $PID &>/dev/null; then
            echo "Процесс $PID существует. ${0##*/} возможно работает!"
           /usr/bin/logger -t ${0##*/} \
                 "Процесс $PID существует. ${0##*/} возможно работает!"
            exit $E_RETURN
        fi
        echo "Процесс $PID не существует. Запуск нового процесса ..."
    fi
}

#  Задается файл диапазона обновлений, начиная с корня или $URL,
#+ согласно шаблонам выше.
set_range () {
    include=
    exclude=
    for p in "${INCLUDE[@]}"; do
        include="$include --include \"$p\""
    done

    for p in "${EXCLUDE[@]}"; do
        exclude="$exclude --exclude \"$p\""
```

```
    done
}


# Извлечение и уточнение списка обновлений rsync.
get_list () {
    echo $$ > $PID_FILE || {          # ИЛИ
        echo "Не удается записать в pid файл $PID_FILE"
        exit $E_RETURN
    }

    echo -n "Извлекается и уточняется список обновлений ..."

    # Извлекаемый список -- 'eval' нужен для запуска rsync одной командой.
    # $3 и $4 это дата и время создания файла.
    # $5 это полное имя пакета.
    previous=
    pre_file=
    pre_date=0
    eval /bin/nice /usr/bin/rsync \
        -r $include $exclude $URL | \
        egrep '^dr.x|^-r' | \
        awk '{print $3, $4, $5}' | \
        sort -k3 | \
        { while read line; do
            #  Получение секунд от начала эпохи, чтобы отфильтровать
            #+ устаревшие пакеты.
            cur_date=$(date -d "$(echo $line | awk '{print $1, $2}')" +%s)
            #  echo $cur_date

            # Получение имени файла.
            cur_file=$(echo $line | awk '{print $3}')
            #  echo $cur_file

            # Получение имен пакетов rpm из файла, если возможно.
            if [[ $cur_file == *rpm ]]; then
                pkg_name=$(echo $cur_file | sed -r -e \
                    's/(^([^_-]+[_-])+)[[:digit:]]+\..*[_-].*$/\1/')
            else
                pkg_name=
            fi
            # echo $pkg_name

            if [ -z "$pkg_name" ]; then   #  Если не rpm файл,
                echo $cur_file >> $TMP    #+ то добавляем в список загрузки.
            elif [ "$pkg_name" != "$previous" ]; then   # Найден новый пакет.
                echo $pre_file >> $TMP          # Вывод последней версии файла.
                previous=$pkg_name              # Сохранение текущего.
                pre_date=$cur_date
                pre_file=$cur_file
            elif [ "$cur_date" -gt "$pre_date" ]; then
                                                #  Если какой-то пакет новее,
                pre_date=$cur_date              #+ то обновляется до него.
                pre_file=$cur_file
            fi
            done
            echo $pre_file >> $TMP              #  Теперь TMP содержит ВЕСЬ
                                                #+ список.
            # echo "subshell=$BASH_SUBSHELL"

    }       # Здесь нужна скобка для завершения "echo $pre_file >> $TMP"
```

```
            # Находились в той же subshell (1) во время всего цикла.

    RET=$?  # Получение кода возврата команд конвейера.

    [ "$RET" -ne 0 ] && {
        echo "Получение списка неудачных кодов $RET"
        exit $E_RETURN
    }

    echo "сделано"; echo
}

# Основная часть загрузки rsync.
get_file () {

    echo "Загрузка..."
    /bin/nice /usr/bin/rsync \
        $OPTS \
        --filter "merge,+/ $TMP" \
        --exclude '*'  \
        $URL $DEST     \
        | /usr/bin/tee $LOG

    RET=$?

   #  --filter merge,+/ это решающее целевое значение.
   #  + означает изменение содержимого, а / означает абсолютный путь.
   #  Затем отсортированный список $TMP будет содержать названия
   #+ имен директорий по восходящей, а --exclude перед '*' "сокращенный цикл."

    echo "Сделано"

    rm -f $PID_FILE 2>/dev/null

    return $RET
}

# -------
# Main
init
check_pid
set_range
get_list
get_file
RET=$?
# -------

if [ "$RET" -eq 0 ]; then
    /usr/bin/logger -t ${0##*/} "Обновление Fedora прошло успешно."
else
    /usr/bin/logger -t ${0##*/} \
    "Обновление Fedora завершено с кодом ошибки: $RET"
fi

exit $RET
```

См. также Пример A-32.

Использование **rcp**, **rsync** и аналогичных утилит в сценарий оболочки не может быть целесообразно из-за низкой безопасности. Рассмотрите

вместо них возможность использования **ssh**, **scp** или сценария **expect**.

ssh

***S**ecure **sh**ell*, регистрирация на удаленном хосте и выполнение на нем команд. Это безопасная замена для **telnet**, **rlogin**, **rcp**, и **rsh**, использующая идентификацию и шифрование. См. ее manpage

Пример 16-44. Использование *ssh*

```
#!/bin/bash
# remote.bash: Использование ssh.

# Пример Michael Zick.
# Используется с разрешения.


#   Предположения:
#   ------------
#   fd-2 не захвачено ( '2>/dev/null' ).
#   ssh/sshd предполагает, что stderr ('2') будет выводиться пользователю.
#
#   sshd запускается на вашей машине.
#   Для любого «обычного» дистрибутива это должно быть так,
#+  и делается без каких-либо самодельных генераторов ключей ssh.

#   Попробуй запустить ssh на своей машине из командной строки:
#
# $ ssh $HOSTNAME
#   Вам будет предложено ввести пароль.
#   введите пароль
#   когда сделано,  $ exit
#
# Работает? Коли так, дальше будет интересней.

# Попробуйте запустить ssh на своей машине из-под 'root':
#
# $ ssh -l root $HOSTNAME
#  Когда запросит пароль, введите пароль root-а, а не свой.
#  Последняя регистрация: Tue Aug 10 20:25:49 2004 из localhost.localdomain
#  Введите 'exit' когда сделано.

#  Выше, вы работали в интерактивной оболочке.
#  Но возможно sshd будет установлен в режиме «single command»,
#+ а это выходит за рамки этого примера.
#  Единственное, что нужно отметить, что следующее будет работать в
#+ режиме 'single command'.

# Основная команда записи stdout (локального).

ls -l

# Теперь же основные команды на удаленной машине.
# Передача других 'USERNAME' 'HOSTNAME', при желании:
USER=${USERNAME:-$(whoami)}
HOST=${HOSTNAME:-$(hostname)}
```

```
#  Теперь запустите написанное выше в командной строке на удаленном хосте,
#+ с зашифрованной передачей.

ssh -l ${USER} ${HOST} " ls -l "

#  Ожидаемым результатом будет список домашней директории
#+ пользователя  на удаленном компьютере.
#  Чтобы увидеть разницу, запустите этот сценарий из чего-то другого,
#+ кроме вашей домашней директории.

#  Другими словами, команда Bash передается  удаленной оболочке как строка
#+ в кавычках, которая выполняется на удаленном компьютере.
#  В этом случае, sshd выполняет ' bash -c "ls -l" '  от вашего имени.

#  Для получения информации по такой теме, как не вводить пароль/фразу
#+ для каждой командной строки, см.
#+    man ssh
#+    man ssh-keygen
#+    man sshd_config.

exit 0
```

В цикле, **ssh** может вести себя неожиданно. Согласно Usenet архивов оболочки comp.unix, **ssh** наследует `stdin` цикла. Чтобы исправить это, передайте **ssh** опцию `-n` или `-f`.

Спасибо Jason Bechtel за разъяснение.

scp

S**ecure **co**p**y, функционально подобна **rcp**, копирует файлы между двумя различными сетевыми компьютерами, но делает это с помощью проверки подлинности и уровня безопасности, подобно **ssh**.

Локальная сеть

write

Это утилита для связи типа терминал-терминал. Она позволяет отправлять строки с вашего терминала (консоли или *xterm*) на терминал другого пользователя. Конечно, для отключения доступа записи в терминале может быть использована команда **mesg**.

Команда **write** является интерактивной и обычно не находит применения в сценариях.

netconfig

Утилита командной строки для настройки сетевой адаптера (с помощью *DHCP*). Эта команда является родной для дистрибутивов Red Hat Linux.

Почта

`mail`

Отправка или прием сообщений e-mail.

Этот урезанный почтовый клиент для командной строки прекрасно работает в качестве встраиваемой в сценарий команды.

Пример 16-45. Сценарий, самостоятельно рассылающий почту

```
#!/bin/sh
# self-mailer.sh: Сценарий самостоятельной рассылки

adr=${1:-`whoami`}     # По умолчанию текущий пользователь,
                       # если не указно иное.
#  Выводит 'self-mailer.sh wiseguy@superdupergenius.com'
#+ отсылаемое этим сценарием в этот адрес.
#  Сценарий 'self-mailer.sh' (без аргумента) отправляет только
#+ человеку вызвавшему его, например, bozo@localhost.localdomain.
#
#  О конструкции ${parameter:-default},
#+ см. раздел "Подстановка параметров"
#+ в главе "Возвращаемые переменные".

# ============================================================================
  cat $0 | mail -s "Сценарий \"`basename $0`\" сам пошлет вам." "$adr"
# ============================================================================

# ---------------------------------------
#  Поздравление от сценария самостоятельной рассылки.
#  Озорник запустит этот сценарий и он сам пошлет вам сообщение.
# ---------------------------------------

echo "Эти `date`, сценарий \"`basename $0`\" перешлет в "$adr"."

exit 0

#  Обратите внимание, что команда "mailx" (в режиме "send") может быть
#+ заменена на "mail" ... но с другими опциями.
```

`mailto`

Подобно команде **`mail`**, **`mailto`** отсылает e-mail сообщения из командной строки или сценария. Однако **`mailto`** также может отправлять сообщения MIME (мультимедиа).

`mailstats`

Показывает *почтовую статистику*. Эта команда может быть вызвана только *root*.

```
root# mailstats
Statistics from Tue Jan  1 20:32:08 2008
  M   msgsfr  bytes_from   msgsto    bytes_to  msgsrej msgsdis msgsqur  Mailer
  4     1682      24118K        0          0K        0       0       0  esmtp
  9      212        640K     1894      25131K        0       0       0  local
 =============================================================================
  T     1894      24758K     1894      25131K        0       0       0
  C      414                   0
```

vacation

Эта утилита автоматически отвечает на сообщения электронной почты, когда получатель находится в отпуске и временно недоступен. Она работает в сети, в сочетании с **sendmail**, и не применима к коммутируемой учетной записи POPmail.

Примечания

[1] *Демон* (*daemon)* фоновый процесс не относящийся к терминальной сессии. Демоны либо запускают назначенные службы в определенное время или явно вызываются определенными событиями.

16.7. Команды управления терминалом

Команды, влияющие на консоль или терминал

tput

Инициализация терминала или загрузка информации о нем из данных **terminfo**. Допускаются различные опции: **tput clear** является эквивалентом **clear**; **tput reset** это эквивалент **reset**.

```
bash$ tput longname
xterm terminal emulator (X Window System)
```

Ввод **tput cup X Y** выведет координаты курсора (X,Y) текущего терминала. **clear** очистит экран терминала от уже введенной или выведенной информации.

Некоторые опции *tput*:

- bold, жирный текст
- smul, подчеркивание текста в терминале

- `smso`, обратное отображение текста
- `sgr0`, сброс настроек терминала (к обычным), без очистки экрана

Примеры, в которых используется *`tput`*:

1. Пример 36-15
2. Пример 36-13
3. Пример A-44
4. Пример A-42
5. Пример 27-2

Обратите внимание, что **`stty`** предлагает более мощный набор команд для управления терминалом.

`infocmp`

Эта команда выводит подробную информацию о текущем терминале. Она ссылается на базу данных *terminfo*.

```
bash$ infocmp
#       Reconstructed via infocmp from file:
 /usr/share/terminfo/r/rxvt
 rxvt|rxvt terminal emulator (X Window System),
         am, bce, eo, km, mir, msgr, xenl, xon,
         colors#8, cols#80, it#8, lines#24, pairs#64,
         acsc=``aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
         bel=^G, blink=\E[5m, bold=\E[1m,
         civis=\E[?25l,
         clear=\E[H\E[2J, cnorm=\E[?25h, cr=^M,
         ...
```

`reset`

Сброс настроек терминала и очистка экрана от текста. Как с **`clear`**, курсор и строка приглашения появляются в верхней левой части терминала.

`clear`

Команда **`clear`** просто очищает экран консоли или *xterm* от текста. В углу верхней левой части экрана или окна *xterm* появляются строка приглашения и курсор. Эта команда может использоваться в командной строке или в сценарии. См. Пример 11-26.

`resize`

Выводит команды, необходимые для задания $TERM и $TERMCAP для дублирования размера текущего терминала (размеры).

```
bash$ resize
set noglob;
 setenv COLUMNS '80';
 setenv LINES '24';
 unset noglob;
```

script

Утилита записи (сохраняет в файл) всех нажатий клавиш командной строки пользователем в консоли или в окне *xterm*. Это, по сути, создание записи сессии.

16.8. Математические команды

"Перечисления"

factor

Разложение целого числа на простые множители.

```
bash$ factor 27417
27417: 3 13 19 37
```

Пример 16-46. Создание простых чисел

```
#!/bin/bash
# primes2.sh

#  Создание простых чисел быстрым и легким способом,
#+ не прибегая к алгоритмам.

CEILING=10000   # От 1 до 10000
PRIME=0
E_NOTPRIME=

is_prime ()
{
  local factors
  factors=( $(factor $1) )  # Загрузка вывода `factor` в массив.

if [ -z "${factors[2]}" ]
#  Третий элемент массива "factors":
#+ ${factors[2]} это аргумент 2-го фактора.
#  Если он пуст, то есть нет 2-го фактора,
#+ то аргумент является простым.
then
  return $PRIME             # 0
else
  return $E_NOTPRIME        # null
fi
```

```
}

echo
for n in $(seq $CEILING)
do
  if is_prime $n
  then
    printf %5d $n
  fi   #    ^  Пяти позиций на число достаточно.
done   #       Для увеличения $CEILING, увеличьте потолок.

echo

exit
```

bc

Bash не работает с вычислениями с плавающей точкой, а ее не хватает операторам для некоторых важных математических вычислений. К счастью на помощь приходит **bc**.

Поскольку это довольно хорошо зарекомендовавшая себя утилита UNIX, то она может быть использована в конвейере, получая данные из сценария.

Вот простой шаблон для вычисления с помощью **bc** переменной сценария. Здесь используется подстановка команд.

```
  variable=$(echo "ОПЦИИ; ОПЕРАЦИИ" | bc)
```

Пример 16-47. Ежемесячный платеж по ипотечному кредиту

```
#!/bin/bash
# monthlypmt.sh: Подсчет ежемесячного платежа по ипотечному кредиту.


#  Это измененный код
#+ пакета "mcalc" (mortgage calculator),
#+ написанного Jeff Schmidt
#+ и
#+ Mendel Cooper.
#   http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz

echo
echo "Учитывается основная, процентная ставка и срок ипотеки,"
echo "подсчитывается ежемесячный платеж."

bottom=1.0

echo
echo -n "Введите основную ставку (без запятых) "
read principal
echo -n "Введите процентную ставку (проценты) "  # Если 12%, введите "12", а не
```

```
".12".
read interest_r
echo -n "Введите срок (месяцы) "
read term


 interest_r=$(echo "scale=9; $interest_r/100.0" | bc) # Перевод в десятичные.
                #          ^^^^^^^^^^^^^^^^^  Делится на 100.
                # "scale" определяет, сколько десятичных знаков.

 interest_rate=$(echo "scale=9; $interest_r/12 + 1.0" | bc)


 top=$(echo "scale=9; $principal*$interest_rate^$term" | bc)
          #           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
          #           Стандартная формула для определения процентов.

 echo; echo "Пожалуйста, подождите. Это займет некоторое время."

 let "months = $term - 1"
# ============================================================================
 for ((x=$months; x > 0; x--))
 do
   bot=$(echo "scale=9; $interest_rate^$x" | bc)
   bottom=$(echo "scale=9; $bottom+$bot" | bc)
#  bottom = $(($bottom + $bot"))
 done
# ============================================================================

# ----------------------------------------------------------------------------
#  Rick Boivie указал на более эффективную реализацию вышеуказанного цикла,
#+ что уменьшает время вычислений на 2/3.

# for ((x=1; x <= $months; x++))
# do
#   bottom=$(echo "scale=9; $bottom * $interest_rate + 1" | bc)
# done


#  После он придумал еще более эффективную альтернативу,
#+ сокращающую время выполнения примерно на 95%!

# bottom=`{
#     echo "scale=9; bottom=$bottom; interest_rate=$interest_rate"
#     for ((x=1; x <= $months; x++))
#     do
#          echo 'bottom = bottom * interest_rate + 1'
#     done
#     echo 'bottom'
#     } | bc`       # Встраивая 'цикл for ' в подстановку команд.
# ----------------------------------------------------------------------------
#  С другой стороны Frank Wang предлагает:
#  bottom=$(echo "scale=9; ($interest_rate^$term-1)/($interest_rate-1)" | bc)

#  Потому что ...
#  Алгоритм цикла является, на самом деле,
#+ суммой серии геометрической прогрессии.
#  Формула суммы e0(1-q^n)/(1-q),
#+ где e0 это первый элемент, q=e(n+1)/e(n)
#+ а n это количество элементов.
# ----------------------------------------------------------------------------
```

```
# let "payment = $top/$bottom"
payment=$(echo "scale=2; $top/$bottom" | bc)
# Используется два десятичных знака для долларов и центов.

echo
echo "monthly payment = \$$payment"  # Выводится перед суммой знак доллара.
echo


exit 0


# Упражнения:
#   1) Отфильтруйте разрешение ввода запятых в основной сумме.
#   2) Отфильтруйте разрешение ввода процентов как в процентной
#      или как в десятичной форме.
#   3) Если вы действительно амбициозны, разверните этот сценарий
#      для вывода таблицы полной амортизации.
```

Пример 16-48. Преобразование основания

```
#!/bin/bash
###############################################################################
# Shellscript:  base.sh — вывод чисел по различным основаниям (Bourne Shell)
# Автор     :  Heiner Steven (heiner.steven@odn.de)
# Date      :  07-03-95
# Category  :  Desktop
# $Id: base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
# ==> Above line is RCS ID info.
###############################################################################
# Описание
#
# Изменения
# 21-03-95 stv  fixed error occuring with 0xb as input (0.2)
###############################################################################

# ==> Используется в ABS Guide с разрешения автора сценария.
# ==> Комментарии, добавленые автором ABS Guide.

NOARGS=85
PN=`basename "$0"`                              # Имя программы
VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2`  # ==> VER=1.2

Usage () {
    echo "$PN — вывод числа по различным основаниям, $VER (stv '95)
usage: $PN [число ...]*****

Если число не указано, то числа читаются из стандартного ввода.
Число может быть
    binary (основание 2)             начинается с 0b (т.е. 0b1100)
    octal (основание 8)              начинается с 0  (т.е. 014)
    hexadecimal (основание 16)       начинается с 0x (т.е. 0xc)
    decimal                          в противном случае (т.е. 12)" >&2
    exit $NOARGS
}   # ==> Вывод сообщения об использовании.
```

```
Msg () {
    for i   # ==> отсутствует in [list]. Почему?
    do echo "$PN: $i" >&2
    done
}bcbc

Fatal () { Msg "$@"; exit 66; }

PrintBases () {
    # Определение основания числа
    for i      # ==> отсутствует in [list]...
    do         # ==> работает с аргументами командной строки.
        case "$i" in
            0b*)                ibase=2;;       # бинарное
            0x*|[a-f]*|[A-F]*)  ibase=16;;      # шестнадцатеричное
            0*)                 ibase=8;;       # восьмеричное
            [1-9]*)             ibase=10;;      # десятичное
            *)
                Msg "недопустимое число $i - пропускается"
                continue;;
        esac

        #  Удаляется префикс, шестнадцатеричные числа преобразуются
        #+ в верхний регистр (это необходимо bc).
        number=`echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]'`
        # ==> Используется разделитель sed ":", отличный от "/".

        # Преобразование числа в десятичное
        dec=`echo "ibase=$ibase; $number" | bc`  # ==> 'bc' - утилита для
                                                 #+ ==> подсчета.
        case "$dec" in
            [0-9]*)     ;;                      # правильное число
            *)          continue;;              # ошибка: пропускается
        esac

        # Вывод всех преобразований одной строкой.
        # ==> 'here document' передает список команд в 'bc'.
        echo `bc <<!
            obase=16; "hex="; $dec
            obase=10; "dec="; $dec
            obase=8;  "oct="; $dec
            obase=2;  "bin="; $dec
!
    ` | sed -e 's: :    :g'

    done
}

while [ $# -gt 0 ]
# ==>  Этот "цикл while" здесь действительно необходим,
# ==>+ так как все case выводят из цикла
# ==>+ или завершают сценарий.
# ==> (Комментарий выше Paulo Marcel Coelho Aragao.)
do
    case "$1" in
        --)     shift; break;;
        -h)     Usage;;                 # ==> Сообщение Справки.
        -*)     Usage;;
         *)     break;;                 # Первое число
```

```
    esac   # ==> Возможна соответствующая проверка ошибок неправильного ввода.
    shift
done

if [ $# -gt 0 ]
then
    PrintBases "$@"
else                                    # Чтение из stdin.
    while read line
    do
        PrintBases $line
    done
fi


exit
```

Альтернативный способ вызова **bc** включает в себя использование ***here document***, встроенный в блок подстановки команд. Это особенно уместно, когда сценарию необходимо передать список команд и параметров в **bc**.

```
variable=`bc << LIMIT_STRING
опции
объявления
операции
LIMIT_STRING
`

...или...


variable=$(bc << LIMIT_STRING
опции
объявления
операции
LIMIT_STRING
)
```

Пример 16-49. Вызов *bc* с помощью *here document*

```
#!/bin/bash
# Вызов 'bc' с помощью подстановки команд
# в комбинации с 'here document'.


var1=`bc << EOF
18.33 * 19.78
EOF
`
echo $var1       # 362.56


#  $( ... ) эта нотация так же работает.
```

```
v1=23.53
v2=17.881
v3=83.501
v4=171.63

var2=$(bc << EOF
scale = 4
a = ( $v1 + $v2 )
b = ( $v3 * $v4 )
a * b + 15.35
EOF
)
echo $var2       # 593487.8452


var3=$(bc -l << EOF
scale = 9
s ( 1.7 )
EOF
)
# Возвращает синус 1.7 радиана.
# Опция "-l" вызывает математическую библиотеку 'bc'.
echo $var3       # .991664810


# Теперь попробуем в функции...
hypotenuse ()    # Вычисление гипотенузы правильного треугольника
{                # c = sqrt( a^2 + b^2 )
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# Bash напрямую не может возвращать из функции значения с плавающей запятой.
# Но возможно echo-и-захват:
echo "$hyp"
}

hyp=$(hypotenuse 3.68 7.31)
echo "hypotenuse = $hyp"    # 8.184039344


exit 0
```

Пример 16-50. Подсчет числа Пи

```
#!/bin/bash
# cannon.sh: Приближение PI при стрельбе ядрами.

# Автор: Mendel Cooper
# License: Public Domain
# Version 2.2, reldate 13oct08.

#  Это очень упрощенный экземпляр «Monte Carlo» :
#+ математической модели событий реальной жизни, эмулирующий
#+ случайности с помощью псевдослучайных чисел.

#  Рассмотрим совершенно квадратный участок земли,
```

```
#+ со стороной в 10000 единиц.
#  Эта площадь имеет в центре совершенно круглое озеро,
#+ диаметром в 10000 единиц.
#  На самом деле, участок занят в основном водой,
#+ за исключением земли по углам.
#  (Представьте себе квадрат с вписанной в него окружностью).

#  Мы будем стрелять железными пушечными ядрами из старой пушки по всей
#+ площади.
#+ Все выстрелы будут попадать по всей площади, в озеро или на сухие углы.
#+ Так как озеро занимает большую часть площади, то большинство выстрелов
#+ будут являться всплеском  воды (SPLASH!).
#+ И только некоторые выстрелы будут попадать по твердой основе в
#+ четырех углах площади (THUD!)

#  Если мы возьмем достаточно случайные, не прицельные,
#+ выстрелы по всей площади, то, соотношение SPLASH и всех выстрелов
#+ будет приблизительным значением PI/4.

#  Объясню проще - орудие фактически стреляет только в
#+ верхний правый сектор квадрата, то есть, Сектор I
#+ Декартовой системы координат.

#  Теоретически, чем больше выстрелов, тем больше точность. Но сценарий,
#+ в отличии от компилируемых языков рассчитывающих с плавающей точкой, требует
#+ определенных компромиссов. А это снижает точность моделирования.


DIMENSION=10000  #  Величина каждой стороны квадрата.
                 #  Также устанавливает потолок для случайных
                 #+ генерируемых чисел.

MAXSHOTS=1000    # Количество выстрелов.
                 # 10000 или чем больше — тем лучше.
PMULTIPLIER=4.0  # Коэффициент масштабирования.

declare -r M_PI=3.141592654
                 # 9-значное значение PI, для сравнения.

get_random ()
{
SEED=$(head -n 1 /dev/urandom | od -N 1 | awk '{ print $2 }')
RANDOM=$SEED                                    #  Из примера сценария
                                                #+ "seeding-random.sh".
let "rnum = $RANDOM % $DIMENSION"               #  Диапазон меньше 10000.
echo $rnum
}

distance=        # Объявление глобальной переменной.
hypotenuse ()    # Вычисление гипотенузы правильного треугольника.
{                # Из примера "alt-bc.sh".
distance=$(bc -l << EOF
scale = 0
sqrt ( $1 * $1 + $2 * $2 )
EOFbcbc
)
#  Присваиваем "scale" нуль для округления результата до целого
#+ значения, необходимый компромисс этого сценария.
#  Снижает точность симуляции.
}
```

```
# ============================================================
# main() {
# Блок кода "main", имитирует функцию main() языка Си.

# Инициализация переменных.
shots=0
splashes=0
thuds=0
Pi=0
error=0

while [ "$shots" -lt  "$MAXSHOTS" ]           # Основной цикл.
do

  xCoord=$(get_random)                        #  Получение случайных
                                              #+ координат X и Y.
  yCoord=$(get_random)
  hypotenuse $xCoord $yCoord                  #  Гипотенуза правильного
                                              #+ треугольника = расстояние.
  ((shots++))

  printf "#%4d   " $shots
  printf "Xc = %4d  " $xCoord
  printf "Yc = %4d  " $yCoord
  printf "Distance = %5d  " $distance         #   Расстояние до
                                              #+  центра озера
                                              #+  -- "начальные" --
                                              #+  координаты (0,0).

  if [ "$distance" -le "$DIMENSION" ]
  then
    echo -n "SPLASH!  "
    ((splashes++))
  else
    echo -n "THUD!    "
    ((thuds++))
  fi

  Pi=$(echo "scale=9; $PMULTIPLIER*$splashes/$shots" | bc)
  # Умножаем соотношение на 4.0.
  echo -n "PI ~ $Pi"
  echo

done

echo
echo "После $shots выстрелов, PI похоже на $Pi"
#  Как правило, немного больше, возможно из-за ошибок округления
#+ и несовершенной случайности $RANDOM.
#  Но все-таки обычно в пределах плюс-минус 5%...
#+ довольно точно, для грубого приближения.
error=$(echo "scale=9; $Pi - $M_PI" | bc)
pct_error=$(echo "scale=2; 100.0 * $error / $M_PI" | bc)
echo -n "Отклонение от математического значения PI =        $error"
echo " ($pct_error% error)"
echo

# Окончание блока кода "main".
# }
# ============================================================
```

```
exit 0

#  Можно задаться вопросом, является ли сценарий подходящим для применения
#+ в качестве комплексных и интенсивных вычислений симуляций.
#
#  Есть, по крайней мере, два довода.
#  1) Как доказательство концепции, показывающей, что это может быть сделано.
#  2) Для прототипирования и проверки алгоритмов, прежде чем переписывать
#+    их на компилируемом языке высокого уровня.
```

См. также Пример A-37.

dc

Стек-ориентированная утилита **dc** (**d**esk **c**alculator), использующая RPN (*Reverse Polish Notation*). Подобно **bc**, она увеличивает мощность языка программирования.

Подобно процедуре **bc**, в **dc** выводится из командной строки.

```
echo "[Выводимая строка ... ]P" | dc
# Команда P выводит строку заключенную в скобки.

# А теперь простое вычисление.
echo "7 8 * p" | dc     # 56
#  Двигает в стек 7, потом 8,
#+ умножает (оператор "*"), затем выводит результат (оператор "p").
```

Часто стараются избегать **dc**, из-за ее не интуитивного ввода и довольно загадочных операторов. Тем не менее имеется смысл ее использования.

Пример 16-51. Преобразование десятичного числа в шестнадцатеричное

```
#!/bin/bash
# hexconvert.sh: Преобразование десятичного числа в шестнадцатеричное.

E_NOARGS=85 # Отсутствие аргумента командной строки.
BASE=16     # Шестнадцатеричное основание.

if [ -z "$1" ]
then        # Необходим аргумент командной строки.
  echo "Usage: $0 number"
  exit $E_NOARGS
fi          # Упражнение: добавьте проверку правильности аргумента.


hexcvt ()
{
if [ -z "$1" ]
then
  echo 0
  return    # "Возвращает" 0, если аргумент не передан функции.
fi

echo ""$1" "$BASE" o p" | dc
#                 o    устанавливает основание (базу) выводимого числа.
```

```
#                    p  выводит вершину стека.
# Другие опции: смотри 'man dc' ...
return
}

hexcvt "$1"

exit
```

Изучение *справочной* страницы **dc** является трудным путем к пониманию ее тонкостей.

```
bash$ echo "16i[q]sa[ln0=aln100%Pln100/snlbx]sbA0D68736142snlbxq" | dc
Bash
```

```
dc <<< 10k5v1+2/p # 1.6180339887
#  ^^^             Операция передачи dc с помощью Here String.
#      ^^^         Помещает 10 и устанавливает точность (10k).
#         ^^       Помещает 5 и извлекает квадратный корень
#                  (5v, v = квадратный корень).
#           ^^     Помещает 1 и добавляет ее к общему (1+).
#             ^^   Помещает 2 и делит на нее общее (2/).
#               ^ Извлекает и выводит результат (p)
#  Результат  1.6180339887 ...
#  ... который является 10-значным Золотым Сечением Пифагора.
```

Пример 16-52. Факторинг

```
#!/bin/bash
# factr.sh: Множитель числа

MIN=2       # Не будет работать с числами меньше указанного.
E_NOARGS=85
E_TOOSMALL=86

if [ -z $1 ]
then
  echo "Usage: $0 число"
  exit $E_NOARGS
fi

if [ "$1" -lt "$MIN" ]
then
  echo "Число множителя должно быть $MIN или больше."
  exit $E_TOOSMALL
fi

# Упражнение: Добавьте проверку типа (отклоняющую не целый аргумент).

echo "Множители $1:"
# ------------------------------------------------------
echo  "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=\
1lrli2+dsi!>.]ds.xd1<2" | dc
```

```
# ------------------------------------------------------
#  Код выше написан Michel Charpentier <charpov@cs.unh.edu>
#  (однострочник, разбитый здесь на две строки).
#  Используется в ABS Guide с разрешения (спасибо!).

 exit

 # $ sh factr.sh 270138
 # 2
 # 3
 # 11
 # 4093
```

awk

Еще один способ вычислений с плавающей точкой в сценарии - использование встроенных математических функций **awk** в обертке оболочки.

Пример 16-53. Вычисление гипотенузы треугольника

```
#!/bin/bash
# hypotenuse.sh: Возвращает "гипотенузу" правильного треугольника.
#                (квадратный корень суммы квадратов двух катетов)

ARGS=2                # Сценарию должны быть переданы стороны треугольника.
E_BADARGS=85          # Неправильное число аргументов.

if [ $# -ne "$ARGS" ] # Проверка количества аргументов сценария.
then
  echo "Usage: `basename $0` сторона_1 сторона_2"
  exit $E_BADARGS
fi


AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
#             команда(ы) / параметры передаваемые в awk


# Теперь параметры передаются конвейером в awk.
    echo -n "Гипотенуза $1 и $2 = "
    echo $1 $2 | awk "$AWKSCRIPT"
#   ^^^^^^^^^^^^
# echo-и-конвейер это простой способ передачи параметров оболочки в awk.

exit

# Упражнение: Перепишите этот сценарий для использования 'bc' вместо awk.
#             Какой способ более интуитивен?
```

16.9. Другие команды

Команды, которые не относятся ни к какой категории

jot, **seq**

Эти утилиты выдают последовательности целых чисел, с шагом в выбираемым пользователем.

По умолчанию разделителем между каждым целым числом является символ перевода строки, но он может быть изменен параметром `-s`.

```
bash$ seq 5
1
 2
 3
 4
 5



bash$ seq -s : 5
1:2:3:4:5
```

Оба, **jot** и **seq** применяются в цикле *for*

Пример 16-54. Создание аргументов цикла с помощью *seq*

```
#!/bin/bash
# Использование "seq"

echo

for a in `seq 80`  # или   for a in $( seq 80 )
# То же, что и   for a in 1 2 3 4 5 ... 80.
# Можно использовать 'jot' (если есть в системе).
do
  echo -n "$a "
done      # 1 2 3 4 5 ... 80
# Пример использования вывода команды для создания
# in [list] цикла "for".

echo; echo


COUNT=80  # Да, 'seq' также принимает изменяемый параметр.

for a in `seq $COUNT`  # или   for a in $( seq $COUNT )
do
  echo -n "$a "
done      # 1 2 3 4 5 ... 80

echo; echo

BEGIN=75
END=80
```

```
for a in `seq $BEGIN $END`
#  Дает «seq» два аргумента, первый начальный, а второй - завершающий отсчет.
do
  echo -n "$a "
done      # 75 76 77 78 79 80

echo; echo

BEGIN=45
INTERVAL=5
END=80

for a in `seq $BEGIN $INTERVAL $END`
#  Дает «seq» три аргумента, первый - начало отсчета,
#+ второй - шаг интервала и третий - завершение отсчета
do
  echo -n "$a "
done      # 45 50 55 60 65 70 75 80

echo; echo

exit 0
```

Простой пример:

```
#  Создание построения 10 файлов,
#+ именуемых file.1, file.2 ... file.10.
COUNT=10
PREFIX=file

for filename in `seq $COUNT`
do
  touch $PREFIX.$filename
  #  Или может производить другие действия,
  #+ такие как rm, grep, и т.д.
done
```

Пример 16-55. Подсчет букв

```
#!/bin/bash
# letter-count.sh: Подсчет букв в текстовом файле.
# Написан Stefano Palmeri.
# Используется в ABS Guide с разрешения.
# Слегка изменен автором.

MINARGS=2          # Сценарию нужны, хотя бы, два аргумента.
E_BADARGS=65
FILE=$1

let LETTERS=$#-1   # Количество указываемых букв (аргументов командной строки).
                   # (Вычитание 1 от числа аргументов командной строки.)


show_help(){
```

```
            echo
            echo Usage: `basename $0` буквы файла
            echo Примечание: `basename $0` аргументы чувствительны к регистру.
            echo Пример: `basename $0` foobar.txt G n U L i N U x.
            echo
}

# Проверка количества аргументов.
if [ $# -lt $MINARGS ]; then
   echo
   echo "Не достаточно аргументов."
   echo
   show_help
   exit $E_BADARGS
fi


# Проверка существования файла.
if [ ! -f $FILE ]; then
    echo "Файл \"$FILE\" не существует."
    exit $E_BADARGS
fi



# Подсчет содержащихся букв.
for n in `seq $LETTERS`; do
      shift
      if [[ `echo -n "$1" | wc -c` -eq 1 ]]; then      # Проверка аргументов.
            echo "$1" -\> `cat $FILE | tr -cd  "$1" | wc -c` #  Подсчет.
      else
            echo "$1 не является единичным символом."
      fi
done

exit $?

#  Этот сценарий имеет точно такую же функциональность как letter-count2.sh,
#+ но выполняется быстрее.
#  Почему?
```

Несколько более мощная, чем *seq*, **jot** является классической утилитой UNIX, которая обычно не включена в стандартные дистрибутивы Linux. Тем не менее, исходники rpm доступны для скачивания из репозитория MIT.

В отличии от *seq*, **jot**, с помощью опции `-r`, может генерировать последовательность случайных чисел.

```
bash$ jot -r 3 999
1069
 1272
 1428
```

getopt

Команда **getopt** анализирует параметры командной строки после дефиса. Это внешняя команда соответствующая встроенной команде **getopts** в Bash. С помощью **getopt** возможна обработка длинных опций используя флаг `-l`, а также допускается перестановка параметров.

Пример 16-56. Анализ опций командной строки с помощью *getopt*

```
#!/bin/bash
# Использование getopt

# Попробуйте вызвать этот сценарий следующим:
#   sh ex33a.sh -a
#   sh ex33a.sh -abc
#   sh ex33a.sh -a -b -c
#   sh ex33a.sh -d
#   sh ex33a.sh -dXYZ
#   sh ex33a.sh -d XYZ
#   sh ex33a.sh -abcd
#   sh ex33a.sh -abcdZ
#   sh ex33a.sh -z
#   sh ex33a.sh a
# Объясните результаты каждой попытки.

E_OPTERR=65

if [ "$#" -eq 0 ]
then   # Сценарию необходим хотя бы один аргумент командной строки.
  echo "Usage $0 -[опции a,b,c]"
  exit $E_OPTERR
fi

set -- `getopt "abcd:" "$@"`
# Задает позиционные параметры для аргументов командной строки.
# Что произойдет, если использовать "$*" вместо "$@"?

while [ ! -z "$1" ]
do
  case "$1" in
    -a) echo "Опция \"a\"";;
    -b) echo "Опция \"b\"";;
    -c) echo "Опция \"c\"";;
    -d) echo "Опция \"d\" $2";;
     *) сброс;;
  esac

  shift
done

#  Лучше использовать 'getopts' встроенную в сценарий.
#  См. "ex33.sh."

exit 0
```

Peggy Russell поясняет:

Для корректной обработки пробелов и кавычек часто бывает необходимо включать **eval**.

```
args=$(getopt -o a:bc:d -- "$@")
eval set -- "$args"
```

См. Пример 10-5 упрощенной эмуляции **getopt**.

run-parts

Команда **run-parts** [1] выполняет последовательно все сценарии в целевой директории, в порядке сортировки имен файлов ASCII. Конечно, сценарии должны иметь права на выполнение.

Демон ***cron*** вызывает **run-parts** для запуска сценариев в директориях `/etc/cron.*` .

yes

Сама по умолчанию, команда **yes** передает непрерывную строку символов *y* с последующим передачей строки на `stdout`. **Control-C** прекращает выдачу. Для иного вывода, может быть указана строка типа **yes different string**, которая будет постоянно выдавать *different string* на `stdout`.

Можно задаться вопросом о необходимости этого. Из командной строки или в сценарии, вывод **yes** может быть перенаправлен в файл или в программу, ожидая ввода данных пользователем. По сути это своего рода обедненная версия ***expect***.

yes | fsck /dev/hda1 запускает ***fsck*** не интерактивно (опасно!).

yes | rm -r dirname имеет тот же эффект, что и **rm -rf dirname** (опасно!).

Будьте осторожны, когда передаете конвейером команду ***yes*** потенциально опасным системным командам, таким как **fsck** или **fdisk**. Это может иметь непредвиденные последствия.

Команда **yes** анализирует переменные, или точнее, отголоски проанализированных переменных. Например:

```
bash$ yes $BASH_VERSION
3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 ...
```

Эта «особенность» может использоваться для создания на лету *очень большого* файла ASCII:

```
bash$ yes $PATH > huge_file.txt
Ctl-C
```

Нажимайте **Ctl-C** очень быстро, или вы получите больше, чем вы рассчитывали...

Команда ***yes*** может *эмулирована* очень простой **функцией** сценария.

```
yes ()
{ # Тривиальная эмуляция "yes" ...
  local DEFAULT_TEXT="y"
  while [ true ]   # Бесконечный цикл.
  do
    if [ -z "$1" ]
    then
      echo "$DEFAULT_TEXT"
    else           # Если аргумент ...
      echo "$1"    # ... расширяется и выводит его.
    fi
  done             #  Единственными отсутствующими вещами являются
}                  #+ опции --help и --version.
```

banner

Вывод аргументов на `stdout` в виде большого вертикального баннера, используя символ ASCII (по умолчанию '#'). Вывод может быть перенаправлен на принтер для печати.

Обратите внимание, что *banner* был исключен из многих дистрибутивов Linux, по-видимому, потому что он больше не нужен.

printenv

Показывает все переменные окружения конкретного пользователя.

```
bash$ printenv | grep HOME
HOME=/home/bozo
```

lp

Команды **lp** и **lpr** отправляют файл (ы) для очередь для печати на принтере. [2] Происхождение этих команд ясно из их названий **l**ine **p**rinters. [3]

bash$ **lp file1.txt** или bash **lp <file1.txt**

Они часто используются при передаче конвейером форматированного вывода из **pr** в **lp**.

```
bash$ pr -options file1.txt | lp
```

Пакеты форматирования, такие как ***groff*** и ***Ghostscript*** могут направлять свою продукцию непосредственно в **lp**.

```
bash$ groff -Tascii file.tr | lp

bash$ gs -options | lp file.ps
```

Связанные команды - **lpq**, для просмотра очереди печати, и **lprm**, для удаления заданий из очереди на печать.

tee

[Идея позаимствованна в UNIX]

Это оператор перенаправления, но совсем другой. Как и заимствованный **tee** он позволяет аккумулировать вывод команды *file* или команды в конвейере, без изменения результата. Это полезно для вывода текущего процесса в файл или в документ, возможно для целей отладки.

```
                            (перенаправление)
                           |----> в файл
                           |
  =========================|====================
  команда ---> команда ---> |tee ---> команда ---> ---> вывод конвейера
  ================================================
```

```
cat listfile* | sort | tee check.file | uniq > result.file
#                      ^^^^^^^^^^^^^^   ^^^^
#  Файл "check.file" содержит объединенные отсортированные "listfiles,"
#+ перед удалением повторяющихся строк 'uniq.'
```

mkfifo

Эта скрытая команда создает *именованный* канал, *первоочередно обслуживающий временный буфер*, для передачи данных между процессами. [4] Как правило, один процесс записывает FIFO, а другой читает из него. См. Пример A-14.

```
#!/bin/bash
# Короткий сценарий написанный Omair Eshkenazi.
# Используется в ABS Guide с разрешения (спасибо!).
```

```
mkfifo pipe1   # Да, каналу может быть присвоено имя.
mkfifo pipe2   # Таким образом обозначается «именованный канал.»

(cut -d' ' -f1 | tr "a-z" "A-Z") >pipe2 <pipe1 &
ls -l | tr -s ' ' | cut -d' ' -f3,9- | tee pipe1 |
cut -d' ' -f2 | paste - pipe2

rm -f pipe1
rm -f pipe2

#  Нет необходимости останавливать фоновые процессы, при завершении сценария
#+ (почему нет?).

exit $?

Теперь, вызовите сценарий и объясните вывод:
sh mkfifo-example.sh

4830.tar.gz          BOZO
pipe1   BOZO
pipe2   BOZO
mkfifo-example.sh    BOZO
Mixed.msg BOZO
```

pathchk

Эта команда проверяет правильность имен файлов. Если имя файла превышает максимально допустимую длину (255 символов), а так же путь поиска одной или нескольких директорий, то выводится сообщение об ошибке.

К сожалению **pathchk** не возвращает информативный код ошибки и именно поэтому бесполезна в сценарии. Рассмотрите вместо нее ***операторы проверки файла***.

dd

Эта команда, **d**ata **d**uplicator, несколько туманна и очень страшна, возникла как утилита для обмена данными на магнитных лентах между микрокомпьютерами UNIX и мэйнфреймами IBM, она до сих пор используется. Команда **dd** просто копирует файл (или `stdin`/`stdout`), но с некоторыми преобразованиями. Возможные преобразования включают ASCII/EBCDIC, [5] верхний/нижний регистр, замену пары байтов между входом и выходом, и пропуск и/или усечения заголовка или окончания входного файла.

```
# Преобразование файла в верхний регистр

dd if=$filename conv=ucase > $filename.uppercase
#                  lcase   # Для преобразования в нижний регистр
```

Некоторые основные опции **dd**:

- `if=`INFILE

  INFILE это *исходный* файл.

- `of=`OUTFILE

  OUTFILE это заданный файл, файл в который будут записываться данные.

- `bs=`BLOCKSIZE

  Размер каждого читаемого или записываемого блока данных, обычно 2.

- `skip=`BLOCKS

  Количество пропускаемых блоков данных в INFILE до начала копирования. Полезно, когда в INFILE имеется "мусор" или искаженные данные в начале или когда нужно скопировать только часть INFILE.

- `seek=`BLOCKS

  Количество пропускаемых блоков данных в OUTFILE прежде, чем начать туда копировать, оставляя пустые данные в начале OUTFILE.

- `count=`BLOCKS

  Копировать только указанное количество блоков данных, а не весь INFILE.

- `conv=`CONVERSION

  Тип преобразования, который будет применен к данным INFILE перед операцией копирования.

`dd --help` список всех опций этой мощной утилиты.

Пример 16-57. Сценарий, который копирует себя

```
#!/bin/bash
# self-copy.sh

# Этот сценарий копирует себя.

file_subscript=copy

dd if=$0 of=$0.$file_subscript 2>/dev/null
# Подавление сообщений из dd:  ^^^^^^^^^^^

exit $?

#  Программа, выводом которой является только ее собственный
#+ исходный код, называется «quine» Willard Quine.
```

Пример 16-58. Работа *dd*

```
#!/bin/bash
# exercising-dd.sh

# Сценарий Stephane Chazelas.
# Небольшие изменения автора ABS Guide.

infile=$0           # Этот сценарий.
outfile=log.txt     # Задаваемый выходной файл.
n=8
p=11

dd if=$infile of=$outfile bs=1 skip=$((n-1)) count=$((p-n+1)) 2> /dev/null
# Извлекает символы с n по p (с 8 по 11) из этого сценария ("bash").

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

echo -n "hello vertical world" | dd cbs=1 conv=unblock 2> /dev/null
# Выводится "hello vertical world" вертикально вниз.
# Почему? dd выводит каждый символ на новую строку.

exit $?
```

Для демонстрации возможностей **dd** давайте с ее помощью сделаем захват нажатия клавиш.

Пример 16-59. Захват нажатий клавиш

```
#!/bin/bash
# dd-keypress.sh: Захват нажатий клавиш без необходимости нажатия ENTER.


keypresses=4                      # Число захватываемых клавиш.


old_tty_setting=$(stty -g)        # Сохранение старых настроек терминала.

echo "Нажмите клавиши $keypresses."
stty -icanon -echo                # Отключение канонического режима.
                                  # Отключение локального echo.
keys=$(dd bs=1 count=$keypresses 2> /dev/null)
# Если не указан "if" (исходный файл), то 'dd' использует stdin.

stty "$old_tty_setting"           # Восстановление старых настроек терминала.

echo "Вы нажали клавишу \"$keys\"."

# Спасибо Stephane Chazelas, за указание на этот способ.
exit 0
```

Команда **dd** может производить произвольный доступ к данным потока.

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc
#  Опция "conv=notrunc" означает, что выходной файл
#+ не будет усечен.
```

```
# Спасибо S.C.
```

Командой **dd** можно копировать необработанные данные и образы дисков на/с устройств, таких как дискеты и накопители на магнитной ленте (Пример A-5). Частым использованием является создание загрузочных дискет.

```
dd if=kernel-image of=/dev/fd0H1440
```

Аналогичным образом **dd** может скопировать содержимое дискеты, даже отформатированной в «чужой» ОС, на жесткий диск в виде файла образа.

```
dd if=/dev/fd0 of=/home/bozo/projects/floppy.img
```

Аналогичным образом **dd** может создавать загрузочные флэш-накопители и SD-карты.

```
dd if=image.iso of=/dev/sdb
```

Пример 16-60. Подготовка загрузочной SD карты для *Raspberry Pi*

```
#!/bin/bash
# rp.sdcard.sh
# Подготовка карты SD с загрузочным образом для Raspberry Pi.

# $1 = Имя файла образа
# $2 = sdcard (файл оборудования)
# В противном случае все не указанное - по умолчанию, см. ниже.

DEFAULTbs=4M                                    # Размер блока, по умолчанию 4 mb.
DEFAULTif="2013-07-26-wheezy-raspbian.img"      # Часто используемый дистрибутив.
DEFAULTsdcard="/dev/mmcblk0"                    # Может отличаться. Проверьте!
ROOTUSER_NAME=root                              # Запускается из под root!
E_NOTROOT=81
E_NOIMAGE=82

username=$(id -nu)                              # Кто запустил этот сценарий?
if [ "$username" != "$ROOTUSER_NAME" ]
then
  echo "Этот сценарий может запускать только root или с правами root."
  exit $E_NOTROOT
fi

if [ -n "$1" ]
then
  imagefile="$1"
else
  imagefile="$DEFAULTif"
fi

if [ -n "$2" ]
then
  sdcard="$2"
else
  sdcard="$DEFAULTsdcard"
```

```
fi

if [ ! -e $imagefile ]
then
  echo "Файл образа \"$imagefile\" не найден!"
  exit $E_NOIMAGE
fi

echo "Последний шанс, чтобы передумать!"; echo
read -s -n1 -p "Нажмите кнопку для записи $imagefile на $sdcard [Ctl-c для
выхода]."
echo; echo

echo "Запись $imagefile на $sdcard ..."
dd bs=$DEFAULTbs if=$imagefile of=$sdcard

exit $?

# Упражнения:
# ---------
# 1) Обеспечьте дополнительную проверку ошибок.
# 2) Создайте сценарий автоопределения устройства для SD-карты (сложно!).
# 3) Создайте сценарий автоопределения файла образа (*img) в $PWD.
```

Другие приложения с помощью **dd** создают временный своп (Пример 31-2) и RAM диск, (Пример 31-3). Она даже может сделать копию на низком уровне целых разделов жесткого диска, хотя это не рекомендуется.

Пример 16-61. Безопасное удаление файла

```
#!/bin/bash
# blot-out.sh: Удаление "всех" следов файла.

#  Этот сценарий перезаписывает указанный файл случайными байтами,
#+ затем нулями, прежде чем окончательно удалить его.
#  После этого, даже анализ дисковых секторов обычными методами,
#+ не обнаружит исходный файл.

PASSES=7          #  Количество проходов.
                  #  Увеличение замедлит выполнение сценария, особенно
                  #+ для больших исходных файлов.
BLOCKSIZE=1       #  I/O с /dev/urandom нужен размер единичного блока,
                  #+ в противном случае вы можете
                  #+ получить непредсказуемый результат.
E_BADARGS=70      #  Коды выхода различных ошибок.
E_NOT_FOUND=71
E_CHANGED_MIND=72

if [ -z "$1" ]   # Не указан файл.
then
  echo "Usage: `basename $0` файл"
  exit $E_BADARGS
fi

file=$1
```

```
if [ ! -e "$file" ]
then
  echo "Файл \"$file\" не найден."
  exit $E_NOT_FOUND
fi

echo; echo -n "Вы абсолютно уверены, что хотите уничтожить \"$file\" (y/n)? "
read answer
case "$answer" in
[nN]) echo "Передумали, да?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "Уничтожение файла \"$file\".";;
esac


flength=$(ls -l "$file" | awk '{print $5}')  # Поле 5 является размером поля.
pass_count=1

chmod u+w "$file"   # Разрешает перезапись/удаление файла.

echo

while [ "$pass_count" -le "$PASSES" ]
do
  echo "Pass #$pass_count"
  sync         # Очистка буфера.
  dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
               # Заполнение случайными байтами.
  sync         # Снова очистка буфера.
  dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
               # Заполнение нулями.
  sync         # И снова очистка буфера.
  let "pass_count += 1"
  echo
done


rm -f $file    # Наконец удаляется сам файл.
sync           # В последний раз очищается буфер.

echo "Файл \"$file\" удален."; echo


exit 0

#  Это достаточно безопасный, хотя неэффективный и медленный,
#+ способ тщательного "измельчения" файла.
#  Команда "shred", часть пакета "файловых утилит" GNU,
#+ делает то же самое, но более эффективно.

#  Этот файл нельзя «возвратить» или восстановить обычными способами.
#  Однако ...
#+ этот простой метод  *не* сможет выдержать сложный криминалистический анализ.

#  Этот сценарий не может хорошо сработать с журналируемой файловой системой.
#  Упражнение (сложное): Исправьте этот момент.

#  Пакет удаления файлов Tom Vier "wipe" производит гораздо тщательное
#+ измельчение файлов, чем этот простой сценарий.
#     http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2
```

```
#  Углубленный анализ по теме удаления файлов и безопасности,
#+ см. статью  Peter Gutmann,
#+     "Secure Deletion of Data From Magnetic and Solid-State Memory".
#       http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html
```

od

Фильтр **od** или **o**ctal **d**ump, преобразует ввод (или файлы) в восьмеричное (по основанию 8) или другое основание. Он полезен для просмотра или обработки файлов двоичных данных или иных, нечитаемых, системных файлов устройств, таких как `/dev/urandom`, а так же как фильтр для двоичных данных.

```
head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# Пример вывода: 1324725719, 3918166450, 2989231420, и т.д..

# Пример из сценария rnd.sh, написанного Stéphane Chazelas
```

Также см. Пример 9-16 и Пример A-36.

hexdump

Выполняет шестнадцатеричный, восьмеричный, десятичный или ASCII дамп двоичного файла. Эта команда является грубым эквивалентом **od**, выше, но не так полезна. Может использоваться для просмотра содержимого бинарного файла в сочетании с **dd** и **less**.

```
dd if=/bin/ls | hexdump -C | less
# Опция -C  красиво форматирует выходные данные в табличную форму.
```

objdump

Выводит информацию об объектном файле или двоичном исполняемом файле в шестнадцатеричной форме или в форме разобранного списка (с опцией `-d`).

```
bash$ objdump -d /bin/ls
/bin/ls:     формат файла elf32-i386

 Разбиение на секции .init:

 080490bc <.init>:
  80490bc:       55                      push   %ebp
  80490bd:       89 e5                   mov    %esp,%ebp
  ...
```

mcookie

Эта команда создает «**m**agi**c** **cookie**,» 128-битное (32-х символьное) псевдослучайное шестнадцатеричное число, обычно используемое X-сервером в качестве «подписи». Она также доступна для использования в сценарии для «быстрого и чернового» создания случайных чисел.

```
random000=$(mcookie)
```

Конечно, для той же цели сценарий может использовать **md5sum**.

```
# Создание md5 контрольных сумм самим сценарием.
random001=`md5sum $0 | awk '{print $1}'`
# С помощью 'awk' делим файл.
```

Команда **mcookie** является еще одним способом создания «уникального» имени файла.

Пример 16-62. Генератор имен файлов

```
#!/bin/bash
# tempfile-name.sh:  Генератор временных имен файлов

BASE_STR=`mcookie`   # 32-х символьное magic cookie.
POS=11               # Произвольная позиция в строке magic cookie.
LEN=5                # Получает $LEN последовательных символов.

prefix=temp          #  Это, в итоге, "временный" файл.
                     #  Для большей "уникальности," создается префикс
                     #+ имени файла с помощью того же способа,
                     #+ что и суффикса, ниже.

suffix=${BASE_STR:POS:LEN}
                     #  Извлекается 5-символьная строка,
                     #+ начинающаяся с позиции 11.

temp_filename=$prefix.$suffix
                     # Конструкция имени файла.

echo "Временное имя файла = "$temp_filename""

# sh tempfile-name.sh
# Временное имя файла = temp.e19ea

#  Сравните этот способ создания "уникального" имени файла
#+ со способом применения 'date' в ex51.sh.

exit 0
```

units

Эта утилита преобразует различные *единицы измерения*. Хотя обычно вызывается в

интерактивном режиме, **units** может найти применение в сценарии.

Пример 16-63. Преобразование метров в мили

```
#!/bin/bash
# unit-conversion.sh
# Нужно иметь установленную утилиту 'units'.


convert_units ()  # Принимается в качестве аргументов единицы конвертирования.
{
  cf=$(units "$1" "$2" | sed --silent -e '1p' | awk '{print $2}')
  # Снимает все, кроме коэффициентов преобразования.
  echo "$cf"
}

Unit1=мили
Unit2=метры
cfactor=`convert_units $Unit1 $Unit2`
quantity=3.73

result=$(echo $quantity*$cfactor | bc)

echo "Это $result $Unit2 в $quantity $Unit1."

#  Что произойдет, если в функцию передать несовместимые единицы,
#+ такие как «акры» и «мили»?

exit 0

# Упражнение: Отредактируйте этот сценарий, чтобы принимать параметры
#             командной строки,конечно с соответствующей проверкой ошибок.
```

m4

Скрытое сокровище, **m4** является мощным макросом[6] фильтра обработки, практически полный язык. Хотя первоначально написанная, как препроцессор для *RatFor*, **m4** оказалась полезной в качестве отдельной утилиты. В самом деле, **m4** сочетает в себе некоторые функции **eval**, **tr** и **awk**, в дополнение к своим обширным расширениям макроса.

Пример 16-64. Использование *m4*

```
#!/bin/bash
# m4.sh: Использование процессора m4

# Строки
string=abcdA01
echo "len($string)" | m4                           #   7
echo "substr($string,4)" | m4                      # A01
echo "regexp($string,[0-1][0-1],\&Z)" | m4      # 01Z

# Вычисления
var=99
```

```
echo "incr($var)" | m4                                  #  100
echo "eval($var / 3)" | m4                              #   33

exit
```

xmessage

Это вариант вывода, на основе X, всплывающего окна запроса/сообщения на рабочем столе.

```
xmessage Left click to continue -button okay
```

zenity

Утилита **zenity** адаптирует отображение диалоговых виджетов GTK и очень подходит для сценариев.

doexec

Команда **doexec** позволяет передавать произвольный список аргументов *бинарному исполняемому* файлу. В частности, передача `argv[0]` (что соответствует `$0` в сценарии) позволяет ссылаться на различные имена исполняемых файлов, а затем выполнять различные последовательности действий, соответствующие имени, которым оно было вызвано. Это равносильно передаче параметров исполняемому файлу окольным путям.

Например, директория `/usr/local/bin` может содержать двоичный файл под названием «**aaa**». Вызывая **doexec /usr/local/bin/aaa list** *перечисляются* все файлы в текущей рабочей директории, начиная с «**a**», при вызове (то же самое исполняемое) **doexec /usr/local/bin/aaa delete** эти файлы будут *удалены*.

Различные варианты поведения исполняемого файла должны быть определены внутри кода самого исполняемого файла, подобно чему-то вроде следующего сценария оболочки:

```
case `basename $0` in
"name1" ) сделать_что-то;;
"name2" ) еще_сделать_что-то;;
"name3" ) еще_сделать_другое;;
*       ) последнее_сделать;;
esac
```

dialog

Семейство инструментов **dialog** предоставляет способ вызова интерактивных «диалоговых» окон из сценариев. Более сложные вариации **dialog** -- **gdialog**, **Xdialog**, и **kdialog** -- на самом деле вызывают виджеты X-Windows .

sox

Команда **sox**, или "**so**und e**x**change" воспроизводит и преобразует звуковые файлы. На самом деле, исполняемый файл `/usr/bin/play` (теперь устарел) является только оболочкой оболочки для `sox`.

Например, **sox soundfile.wav soundfile.au** изменяет звуковой файл WAV в звуковой файл (Sun audio format) AU.

Примечания

[1] Этот сценарий адаптирован из дистрибутива Debian Linux.

[2] *Очередь печати* — это группа заданий «в строке ожидания» печати.

[3] Большие механические струйные принтеры одновременно печатают одну строку на соединяемые листов бумаги greenbar, в сопровождении большого шума. Печатные копии выглядят так, как будут переданы в печать.

[4] Прекрасный обзор этой темы в статье Andy Vaught, Introduction to Named Pipes, сентябрь 1997 года, Linux Journal.

[5] EBCDIC это акроним Extended Binary Coded Decimal Interchange Code, формата данных IBM. Не совсем обычное применение опции **dd** `conv=ebcdic` это быстрая и легкая, но не очень безопасная кодировка текстового файла.

```
cat $file | dd conv=swab,ebcdic > $file_encrypted
# Кодирование (выглядит как бред).
# Можно переставить байты (swab), для дополнительного запутывания.

cat $file_encrypted | dd conv=swab,ascii > $file_plaintext
# Раскодировка.
```

[6] *Макрос* — целочисленная константа, которая расширяется командную строкой или набором параметров операций. Проще говоря, это ярлык или аббревиатура.

Часть 5. Дополнительные темы

На данный момент, мы уже готовы вникать в некоторые трудные и необычные аспекты сценариев. Для этого мы попробуем различными способами 'выйти за рамки' и изучить отдельные условия (что случится, когда мы зайдем на неизведанную территорию?).

Содержание

Глава 18. Регулярные выражения

... мыслительная деятельность, связанная с разработкой программного обеспечения, гораздо глубже.

--Stowe Boyd

Содержание

Чтобы полностью использовать силу языка сценариев командной оболочки, необходимо освоить регулярные выражения. Некоторые команды и утилиты, обычно используемые в сценариях, такие как ***grep***, ***expr***, ***sed*** и ***awk***, интерпретируют и используют регулярные выражения. Начиная с версии 3, Bash приобрел свой собственный оператор соответствия регулярного выражения: **=~**.

18.1. Краткое введение в регулярные выражения

Выражение — это строка символов. Символы, имеющие интерпретацию шире и глубже их буквального смысла, называются ***метасимволами***. Символы в кавычках, например, могут обозначать или речь человека или ***метасмысл*** [1] символов, которые заключены в них (в кавычках). Регулярные выражения представляют собой наборы символов и/или

метасимволов, которые соответствуют (или указываются) шаблону.

Регулярное выражение должно содержать хотя бы одно из следующего:

- *Набор символов*. Это символы сохраняющие свои буквальные значения. Простейший тип регулярного выражения состоит *только* из набора символов, без метасимволов.
- *Якорь*. Обозначение (привязка) положения в текстовой строке, которое соответствует регулярному выражению. Например, **^** и **$** являются якорями (привязками).
- *Модификаторы*. Расширяют, либо сокращают (*изменяют*), текстовый фрагмент соответствующий регулярному выражению. Модификаторами являются ***звездочки***, ***скобки*** и ***обратный слэш***.

Основное использование Регулярных Выражений (**РВ**) - поиск в тексте и операции над строками. РВ с*оответствует* одному символу или набору символов -- строке или части строки.

- ***Звездочка*** --*****-- соответствие любому количеству повторов символов в строке или регулярном выражении, стоящих перед ней, в том числе и *нуля*.

  "**1133***" соответствует *11 + одна или более 3*: *113*, *1133*, *1133333*, и так далее.

- ***Точка*** -- **.** -- соответствие любому символу, кроме перевода строки. [2]

  "**13.**" соответствует *13 + по крайней мере, один любой символ (включая пробел)*: *1133*, *11333*, но не *13* (дополнительный символ отсутствует).

  См. Пример 16-18 демонстрирующий соответствие *точки одному знаку*.

- ***Курсор*** – **^** -- соответствие ***началу строки***, но иногда, в зависимости от контекста, отрицание значений набора символов регулярных выражений.

- ***Знак доллара*** -- **$** -- в конце регулярного выражения соответствует ***концу строки***.

  "**XXX$**" соответствует окончанию строки **XXX**.

  "**^$**" соответствует ***пустой*** строке.

- ***Скобки*** -- **[...]** -- заключенный в них набор символов соответствует ***одному*** *регулярному выражению*.

  "**[xyz]**" соответствие одному *любому* из символов *x*, *y* *или* *z*.

  "**[c-n]**" соответствие одному *любому* символу в диапазоне от *c* до *n*.

  "**[B-Pk-y]**" соответствие одному *любому* символу в диапазоне от *B* до *P* и от *k* до *y*.

  "**[a-z0-9]**" соответствие *любой* одной буквы нижнего регистра или *любой* одной цифры.

"**[^b-d]**" соответствие *любому* символу, за исключением находящихся в диапазоне *b-d*. В данном случае **^** - это ***отрицание*** или ***инвертирование*** значения следующего регулярного выражения (играет ту же роль, что и - **!**, но в другом контексте).

Комбинированные последовательности символов, заключенных в скобки, соответствуют общему шаблону слова. "**[Yy][Ee][Ss]**" соответствуют *yes*, *Yes*, *YES*, *yEs*, и так далее. "**[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]**" соответствует любое число Social Security.

- ***Обратный слэш*** --****-- пропуск - специальный символ, который означает, что *символ интерпретируется* ***буквально*** (и поэтому больше не является *специальным*).
- "**\$**" возвращает обратно буквальное значение «**$**», а не значение окончания строки регулярного выражения. Подобно "****", имеющему буквальное значение "\".
- *Заэкранированные* «угловые скобки» -- **\<...\>** -- границы слова.

  Угловые скобки должны быть заэкранированы, так как они имеют другой ***буквальный*** смысл.

  "**\<the\>**" соответствует слову "the," но не словам "them," "there," "other," и т.д.

```
bash$ cat textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.
 This is line 4.


bash$ grep 'the' textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.


bash$ grep '\<the\>' textfile
This is the only instance of line 2.
```

Единственный способ быть уверенным, что регулярные выражения работают - это проверять их.

```
TEST FILE: tstfile                        # Не соответствует.
                                          # Не соответствует.
Запускаем в этом файле grep "1133*".      # Соответствует.
                                          # Не соответствует.
                                          # Не соответствует.
Эта строка содержит число 113.            # Соответствует.
Эта строка содержит число 13.             # Не соответствует.
Эта строка содержит число 133.            # Не соответствует.
Эта строка содержит число 1133.           # Соответствует.
Эта строка содержит число 113312.         # Соответствует.
Эта строка содержит число 1112.           # Не соответствует.
Эта строка содержит число 113312312.      # Соответствует.
Эта строка не содержит вообще чисел.      # Не соответствует.


bash$ grep "1133*" tstfile
Запускаем в этом файле grep "1133*".      # Соответствует.
 Эта строка содержит число 113.           # Соответствует.
 Эта строка содержит число 1133.          # Соответствует.
 Эта строка содержит число 113312.        # Соответствует.
 Эта строка содержит число 113312312.     # Соответствует.
```

- **Расширенные регулярные выражения.** Дополнительные метасимволы, добавляемые в базовый комплект. Используются в **egrep**, **awk** и **Perl**.
- ***Знак вопроса*** -- **?** -- соответствие нулю или одному из предыдущих регулярных выражений. Обычно он используется для *сопоставления отдельных символов*.
- ***Плюс*** -- **+** -- соответствие одному или нескольким предыдущим регулярным выражениям. Подобно *****, но не соответствует *нулевым* вхождениям.

```
#  GNU версии  sed и awk могут использовать '+',
#  но он должна быть заэкранирован.

echo a111b | sed -ne '/a1\+b/p'
echo a111b | grep 'a1\+b'
echo a111b | gawk '/a1+b/'
# Все эти примеры эквивалентны.

# Спасибо, S.C.
```

- ***Закранированные*** «***фигурные скобки***» -- **\{ \}** -- указывают количество соответствий предыдущих РВ.

  Необходимо экранировать фигурные скобки, так как ***буквально*** они означают иное. Их использование, технически, не является частью базового набора регулярных выражений.

  '**[0-9]\{5\}**' соответствие ровно пяти цифрам (знакам в диапазоне от **0** до **9**).

Фигурные скобки не доступны, как регулярные выражения, в 'классической' (POSIX не совместимой) версии **awk**. Однако, расширенная версия **awk**, **gawk** GNU, имеет опцию `--re-interval`, которая позволяет им быть регулярными выражениями (без экранирования).

```
bash$ echo 2222 | gawk --re-interval '/2{3}/'
2222
```

Perl и некоторые версии **`egrep`** не требуют экранирования фигурных скобок.

- ***Круглые скобки*** -- **`(  )`** -- содержат (заключают в себя) ***группу*** *регулярных выражений*. Они полезны с последующим оператором '**`|`**' , а для *извлечения содержимого строки* используется **`expr`**.
- ***Оператор регулярного выражения*** --**`|`**-- "ИЛИ" - это соответствие ***любому из набора*** заданных символов.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
 The clarinet produces sound by the vibration of its reed.
```

Некоторые версии ***sed***, ***ed*** и ***ex*** поддерживают заэкранированные версии расширенных регулярных выражений, описанных выше, как это делают утилиты GNU.

- **Классы символов POSIX. `[:class:]`**

  Это альтернативный способ указания диапазона соответствия символов.

- **`[:alnum:]`** соответствие всем буквенным и числовым символам. Эквивалентно **`A-Za-z0-9`**.
- **`[:alpha:]`** соответствие всем буквенным символам. Эквивалентно **`A-Za-z`**.
- **`[:blank:]`** соответствие пропуску или табуляции.
- **`[:cntrl:]`** соответствие управляющим символам.
- **`[:digit:]`** соответствие всем (десятичным) цифрам. Эквивалентно **`0-9`**.
- **`[:graph:]`** (графические печатаемые символы). Соответствие всем символам ASCII в диапазоне от 33 - 126. То же, что и **`[:print:]`**, ниже, но за исключением символа пробела.
- **`[:lower:]`** соответствие всем буквенным символам нижнего регистра. Эквивалентно

a-z.

- **[:print:]** (печатаемые символы). Соответствие символам ASCII в диапазоне от 32 - 126. То же, что и **[:graph:]**, выше, но добавлен символ пропуска.
- **[:space:]** соответствие символу пробела (пропуск и горизонтальная табуляция).
- **[:upper:]** соответствие всем буквенным символам верхнего регистра. Эквивалентно **A-Z**.
- **[:xdigit:]** соответствие всем шестнадцатеричным цифрам. Эквивалентно **0-9A-Fa-f**.

Символьные классы POSIX, обычно, *требуют кавычек* или *двойных скобок* **[[ ]]**.

```
bash$ grep [[:digit:]] test.file
abc=723
```

```
# ...
if [[ $arow =~ [[:digit:]] ]]   #  Введено число?
then       #  символьный класс POSIX
  if [[ $acol =~ [[:alpha:]] ]] # Число, а затем буква? Не правильно!
# ...
# Пример из сценария ktour.sh.
```

- Эти символьные классы могут использоваться даже в ***подстановке***, но ограниченно.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r--    1 bozo  bozo         0 Aug 21 14:47 a33b
```

Символьные классы POSIX используются в Примере 16-21 и Примере 16-22.

Sed, ***awk***, и ***Perl***, используемые в качестве фильтров в сценариях, принимают регулярные выражения в качестве аргументов, при «просеивании» или преобразовании файлов, или в качестве потоков ввода/вывода. См. Пример A-12 и Пример A-16.

Стандартной ссылкой по этому сложному вопросу является *Mastering Regular Expressions* Friedl'a. *Sed & Awk*, Dougherty и Robbins'a, который дает очень ясную разъяснение регулярных выражений. См. ***Библиографию***, для получения дополнительной информации об этих книгах.

Примечания

[1] *Метасмысл* - это смысл термина или выражения на более высоком уровне абстракции. Например, ***буквальное*** значение *регулярного выражения* - это обычное выражение,

которое соответствует принятому значению. *Метасмысл* радикально отличается, как подробно описано в этой главе.

[2] Поскольку **sed**, **awk** и **grep** все делают одной строкой, то новая строка не будет применяться в соответствии. Точка будет соответствовать новой строке в тех случаях, когда необходим символ новой строки в многострочном выражении.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF   # Here Document
строка1
строка2
EOF
# ВЫВОД:
# [строка1
# строка2]


echo

awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
line 1                        # ^
line 2
EOF
# ВЫВОД:
# line
# 1


# Спасибо, S.C.

exit 0
```

18.2. Подстановка

Сам Bash не признает регулярные выражения. Внутри сценариев регулярные выражения интерпретируют такие команды и утилиты, как **sed** и **awk.**

Bash осуществляет расширения имени файла [1] процессом, известным как ***globbing*** (подстановка шаблона) - но он не использует стандартный набор регулярных выражений. Вместо этого ***globbing*** распознает и расширяет символы шаблонов подстановки. ***Globbing*** интерпретирует стандартные символы шаблонов подстановки [2] -- ***** и**?**, символы в ***квадратных*** скобках и некоторые другие специальные символы (например, **^** для отрицания смысла соответствия). Есть серьезные ограничения символов шаблонов подстановки ***globbing***. Строкам, содержащим *****, не будут соответствовать имена файлов, *которые начинаются с точки*, как, например, `.bashrc`. [3] Кроме того,**?**, в ***globbing***, имеет отличный

смысл от смысла в регулярных выражений.

```
bash$ ls -l
total 2
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt
```

Bash использует расширения имен файлов без кавычек, как аргументы командной строки. Команда **echo** демонстрирует это.

```
bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
t2.sh test1.txt

bash$ echo t?.sh
t2.sh
```

Можно изменять распознавание Bash специальных символов ***globbing***. Команда **set -f** отключает ***globbing***, а опции **shopt** - `nocaseglob` и `nullglob` изменяют поведение ***globbing***.

См. Пример 11-5.

Имена файлов, в которых имеются ***пробелы***, могут экранировать ***globbing***. David Wheeler показывает, как избежать таких ошибок.

```
IFS="$(printf '\n\t')"   # Удаляется пустое место (пробел).

#  Правильное использование подстановки:
#  Всегда используется цикл for, префикс подстановки, проверка
# существования файла.
for file in ./* ; do          #  Используем ./* … НИКОГДА *
                              #+ (звездочка) не может быть голой
  if [ -e "$file" ] ; then    # Проверка, существует ли файл.
     КОМАНДА ... "$file" ...
  fi
done

# Этот пример, взят с сайта David Wheeler, с разрешения автора.
```

Примечания

[1] Расширение имени файла означает расширение шаблонов имен файлов или шаблонов, содержащих специальные символы. К примеру, `example.???` может быть расширено как `example.001` и/или `example.txt`.

[2] Символы подстановки, аналогичны ***wild card*** в покере, могут представлять любой(почти) символ.

[3] Расширению имени файла может соответствовать файл с точкой, но только если шаблон ***явно*** содержит точку, как буквальный символ.

```
~/[.]bashrc    #  Не будет расширено до ~/.bashrc
~/?bashrc      #  Не получится.
               #  Wild card и метасимволы НЕ
               #+ расширяют точку в подстановке.

~/.[b]ashrc    #  Будет расширено до ~/.bashrc
~/.ba?hrc      #  Аналогично.
~/.bashr*      #  Аналогично.

# Опция «dotglob» это отключает.

# Спасибо, S.C.
```

Глава 19. Here Documents

Здесь и теперь, парни.

--Aldous Huxley, Island

Here document представляет собой *специальный блок кода*. Он использует форму **перенаправления I/O** для передачи списка команд в интерактивную программу или команду, такую как **`ftp`**, **`cat`** или текстовый редактор ***ex***.

```
КОМАНДА <<Вводится(перенаправляется)ОТСЮДА
...
...
...
Вводится(перенаправляется)СЮДА
```

Limit string *(ограничение строки)* устанавливает размеры (кадры) списка команд. Специальный символ **`<<`** предшествует ограничению строки. Он имеет эффект перенаправления вывода блока команд в `stdin` программы или команды. Подобно **`интерактивная-программа < командный-файл`**, где `командный-файл` содержит

```
команда #1
команда #2
...
```

Here document выглядит, как этот эквивалент :

```
интерактивная-программа << LimitString
команда #1
команда #2
...
LimitString
```

Выбирайте ***limit string*** достаточно необычным, что он не возник где-нибудь в списке команд и не запутал.

Обратите внимание, что ***here documents*** иногда может быть полезен для усиления эффекта не интерактивных утилит и команд, таких как, например, **`wall`**.

Пример 19-1. *broadcast*: Отправка сообщения всем зарегистрированным пользователям

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
Ваш e-mail полуденного заказа на пиццу системному администратору.
    (Добавьте еще доллар на анчоусы или дополнение из грибов.)
# Это идет текст дополнительного сообщения.
# Примечание: 'wall' выводит строку комментария.
zzz23EndOfMessagezzz23

# Можно было бы сделать более эффективно
```

```
#         wall <message-file
#  Однако, внедрение шаблона сообщения в сценарий является
#+ неоднозначным единовременным решением.

exit
```

Даже такие маловероятные кандидаты, как текстовый редактор **vi**, прибегают к ***here documents***.

Пример 19-2. *dummyfile*: Создание пустого 2-строчного файла

```
#!/bin/bash

# Не интерактивное использование 'vi' для редактирования файла.
# Эмуляция 'sed'.

E_BADARGS=85

if [ -z "$1" ]
then
  echo "Используем: `basename $0` имя_файла"
  exit $E_BADARGS
fi

TARGETFILE=$1

# Вставляются 2 строки в файл, затем сохраняются.
#--------Начало here document-----------#
vi $TARGETFILE <<x23LimitStringx23
i
Это строка 1 файла примера.
Это строка 2 файла примера.
^[
ZZ
x23LimitStringx23
#----------Конец here document-----------#

#  Обратите внимание, что ^[ , выше, является буквальным экранированием
#+ введенного Control-V <Esc>.

#  Bram Moolenaar указывает на то, что это может не сработать в «vim»,
#+ из-за возможных проблем взаимодействия с терминалом.

exit
```

Приведенный выше сценарий может так же эффективно быть выполнен с **ex**, а не с **vi**. ***Here documents*** может включать в себя список общих команд **ex,** формируя свою собственную категорию, известную как ***сценарии ex***.

```
#!/bin/bash
#  Заменяет все вхождения "Smith" на "Jones"
#+ в файлах с именами содержащими суффикс ".txt".

ORIGINAL=Smith
REPLACEMENT=Jones
```

```
for word in $(fgrep -l $ORIGINAL *.txt)
do
  # ------------------------------------
  ex $word <<EOF
  :%s/$ORIGINAL/$REPLACEMENT/g
  :wq
EOF
  # :%s это подстановка команд "ex".
  # :wq это «записать-и-выйти».
  # ------------------------------------
done
```

Аналогичны «сценариям **ex**» и ***сценарии `cat`***.

Пример 19-3. Многострочные сообщения с использованием *cat*

```
#!/bin/bash

#  'echo'  хороша для вывода одной строки сообщения,
#+  но проблематична для вывода блоков сообщений.
#   here document с 'cat' снимает это ограничение.

cat <<End-of-message
-------------------------------------
Это строка 1 сообщения.
Это строка 2 сообщения.
Это строка 3 сообщения.
Это строка 4 сообщения.
Это последняя строка сообщения.
-------------------------------------
End-of-message

#  Помещаем строку 7, выше, с помощью
#+   cat > $Newfile <<End-of-message
#+       ^^^^^^^^^^
#+ вывод записывается в файл $Newfile, а не в stdout.

exit 0


#-----------------------------------------
# Код ниже отключен, из-за "exit 0" выше.

# S.C. поясняет, что это так же будет работать.
echo "-------------------------------------
Это строка 1 сообщения.
Это строка 2 сообщения.
Это строка 3 сообщения.
Это строка 4 сообщения.
Это последняя строка сообщения.
-------------------------------------"
#  Однако текст должен быть заключен в заэкранированные двойные кавычки.
```

Опция «**-**» отмечает ограничение строки ***here document*** (**`<<-LimitString`**) подавляя

ведущие табуляции (но не пропуски места) на выходе. Это может быть полезно для создания более читаемого сценария.

Пример 19-4. Многострочные сообщения с подавлением табуляций

```
#!/bin/bash
# Почти, как предыдущий пример, но...

#  Опция - в here document <<-
#+ подавляет ведущие табуляции в теле документа,
#+ но *не* пробелы.

cat <<-ENDOFMESSAGE
        Это строка 1 сообщения.
        Это строка 2 сообщения.
        Это строка 3 сообщения.
        Это строка 4 сообщения.
        Это последняя строка сообщения.
ENDOFMESSAGE
# Сценария будет выводить левому краю.
# Ведущие табуляции в каждой строке не будут видны.

#  5 строкам «сообщения» предшествует табуляция, а не пробел.
#  Пробелы не зависят от  <<- .

# Обратите внимание, что эта опция не влияет на *встроенную* табуляцию.

exit 0
```

Here document поддерживает подстановку параметров и команд. Именно поэтому можно передавать различные параметры в тело ***here document***, изменяя, соответственно, его вывод.

Пример 19-5. Here document с изменяемыми параметрами

```
#!/bin/bash
# 'cat' с любым here document, используя подстановку параметров.

# Попробуйте без параметров командной строки,  ./scriptname
# Попробуйте с одним параметром командной строки,  ./scriptname Mortimer
# Попробуйте с одним параметром командной строки состоящим из двух слов
# заключенных в кавычки  ./scriptname "Mortimer Jones"

CMDLINEPARAM=1     #  Ожидается 1 параметр командной строки.

if [ $# -ge $CMDLINEPARAM ]
then
  NAME=$1          #  Если больше, чем один параметр командной строки,
                   #+ то принимается только первый.
else
  NAME="John Doe"  #  Умолчание, если нет параметра командной строки.
fi

RESPONDENT="автор этого прекрасного сценария"


cat <<Endofmessage
```

```
Привет, $NAME.
Рад встрече, $NAME, от $RESPONDENT.

# Этот комментарий будет показан в выводе (почему?).

Endofmessage

# Обратите внимание, пустые строки так же отображаются в выводе.
# То же и в комментарии.

exit
```

А вот полезный сценарий, содержащий ***here document*** с подстановкой параметров.

Пример 19-6. Загрузка пары файлов в директорию находящихся на Sunsite

```
#!/bin/bash
# upload.sh

#  Загрузка пары файлов (Filename.lsm, Filename.tar.gz)
#+ в директорию из Sunsite/UNC (ibiblio.org).
#  Filename.tar.gz это тарбол.
#  Filename.lsm это файл-дескриптор.
#  Sunsite требуется файл "lsm", иначе случится неожиданная неприятность.


E_ARGERROR=85

if [ -z "$1" ]
then
  echo "Использовать: `basename $0` Имя_файла_для_загрузки"
  exit $E_ARGERROR
fi


Filename=`basename $1`           # Путь к имени файла из файла.

Server="ibiblio.org"
Directory="/incoming/Linux"
#  Они не должны быть жестко закодированы в сценарии,
#+ что бы могли изменяться аргументом командной строки.

Password="your.e-mail.address"   # Изменить на нужный.

ftp -n $Server <<End-Of-Session
# опция -n отключает автоматический вход систему

user anonymous "$Password"     #  Если это не сработает, то
                               #  возьмите в кавычки user anonymous "$Password"
binary
bell                           # Звонок 'динь' после каждого трансфера файла.
cd $Directory
put "$Filename.lsm"
put "$Filename.tar.gz"
bye
End-Of-Session
```

```
exit 0
```

Заключение в кавычки или экранирование «***limit string***», заголовка ***here document,*** отключает подстановку параметров в его тело. Причина в том, что *заключение* ***limit string*** *в кавычки/экранирование* эффективно экранирует специальные символы **$**, **'**, и **** , и заставляет их толковать ***буквально***. (Спасибо, Allen Halsey, за это разъяснение.)

Пример 19-7. Выключение подстановки параметров

```
#!/bin/bash
#  'cat' с here-document, но с отключенной подстановкой параметров.

NAME="John Doe"
RESPONDENT="автор этого прекрасного сценария"

cat <<'Endofmessage'

Привет, $NAME.
Рад встрече, $NAME, от $RESPONDENT.

Endofmessage

#   Подстановка параметров отсутствует когда "limit string" в кавычках
#   или заэкранирована.
#   В любом из следующих заголовков here document будет тот же эффект.
#   cat <<"Endofmessage"
#   cat <<\Endofmessage



#   И, кроме того:

cat <<"SpecialCharTest"

 Если limit string не заключено в кавычки,то выводится список директорий.
`ls -l`

 Если limit string не заключено в кавычки, то возможно арифметическое
расширение
$((5 + 3))

Если limit string не заключено в кавычки, то на экран выводится единственный
обратный слэш.
\\

SpecialCharTest


exit
```

Отключение подстановки параметра разрешает ***буквальный*** вывод текста. Одной из возможностей этого является создание сценариев или даже программного кода.

Пример 19-8. Сценарий, который создает другой сценарий

```
#!/bin/bash
# generate-script.sh
# Основан на идее Albert Reiner.

OUTFILE=generated.sh         # Создаваемый файл.


# ----------------------------------------------------------
# 'Here document содержит тело создаваемого сценария.
(
cat <<'EOF'
#!/bin/bash

echo "Это создаваемый сценарий shell."
#  Обратите внимание, что поскольку мы находимся внутри subshell,
#+ то мы не можем получить доступ к переменным «снаружи» сценария.

echo "Созданный файл будет назван: $OUTFILE"
#  Строки выше не будут работать, как ожидалось, потому что
#+ расширение параметра  было отключено.
#  Вместо этого результатом является буквальный вывод.

a=7
b=3

let "c = $a * $b"
echo "c = $c"

exit 0
EOF
) > $OUTFILE
# ----------------------------------------------------------

#  Заключение в кавычки 'limit string' предотвращает расширение переменных в
#+ теле 'here document' выше.
#  Что позволяет выводить строки буквально в выходной файл.

if [ -f "$OUTFILE" ]
then
  chmod 755 $OUTFILE     # Созданный файл исполняемый.
else
  echo "Проблема при создании файла: \"$OUTFILE\""
fi

#  Этот способ также работает при создании
#+ Си программ, Perl программ, Python программ,
#+ файлов Makefile и т.п.

exit 0
```

Возможно присваивать переменную из выходных данных ***here document***. На самом деле это хитрая форма подстановки команд.

```
variable=$(cat <<SETVAR
Эта переменная
работает на нескольких строках
SETVAR
)
```

```
echo "$variable"
```

Here document может передавать ввод в функцию в этом же сценарии.

Пример 19-9. Here document и функции

```
#!/bin/bash
# here-function.sh

GetPersonalData ()
{
  read firstname
  read lastname
  read address
  read city
  read state
  read zipcode
} # Это, по-видимому, интерактивная функция, но...


# Передаем ввод в функцию выше.
GetPersonalData <<RECORD001
Bozo
Bozeman
2726 Nondescript Dr.
Bozeman
MT
21226
RECORD001


echo
echo "$firstname $lastname"
echo "$address"
echo "$city, $state $zipcode"
echo

exit 0
```

Можно использовать как фиктивную команду, принимающую вывод из ***here document***. Создавая, по сути, 'анонимный' ***here document***.

Пример 19-10. "Анонимный" Here Document

```
#!/bin/bash

: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?}       #  Выводит сообщение об ошибке, если одна
                                   #+ из переменных не установлена.
TESTVARIABLES

exit $?
```

Разновидностью данного способа является возможность «комментировать» блоки кода.

Пример 19-11. Комментирования блока кода

```
#!/bin/bash
# commentblock.sh

: <<COMMENTBLOCK
echo "Эта строка не будет выведена на экран."
В этой строке комментария отсутствует префикс "#".
Это другая строка комментария с отсутствующим префиксом "#".

&*@!!++=
Строка выше не вызовет сообщения об ошибке,
потому что интерпретатор Bash будет игнорировать ее.
COMMENTBLOCK

echo "Выходное значение \"COMMENTBLOCK\" выше это $?."   # 0
# Ошибки не показываются.
echo


#  Приведенная выше техника полезна для комментирования блока
#+ рабочего  кода при отладке.
#  И позволяет обойтись без «#» в начале каждой строки,
#+ что бы не возвращаться и не удалять позже каждый "#".
#  Обратите внимание, что использование двоеточия, выше, является
#+ необязательным.

echo "Только перед закомментированным блоком кода."
#  Строки кода между двойными штриховыми линиями не будут выполняться.
#  ==================================================================
: <<DEBUGXXX
for file in *
do
 cat "$file"
done
DEBUGXXX
#  ==================================================================
echo "Только после закомментированного блока кода."

exit 0


#####################################################################
#  Отметим, однако, что если в комментированном блоке кода содержится
#+ переменная со скобками, то это может вызвать проблемы.
#  например:


#/!/bin/bash
```

```
  : <<COMMENTBLOCK
  echo "Эта строка не будет выведена на экран."
  &*@!!++=
  ${foo_bar_bazz?}
  $(rm -rf /tmp/foobar/)
  $(touch my_build_directory/cups/Makefile)
COMMENTBLOCK


$ sh commented-bad.sh
commented-bad.sh: строка 3: foo_bar_bazz: параметр null или не присвоена

# В этом случае помогают ***сильные*** кавычки 'COMMENTBLOCK' строки 49, выше.

  : <<'COMMENTBLOCK'

# Спасибо, Kurt Pfeifle, за это объяснение.
```

Еще одна сторона этого ловкого трюка позволяет создавать 'самодокументируемые' сценарии.

Пример 19-12. Самодокументируемый сценарий

```
#!/bin/bash
# self-document.sh: самодокументируемый сценарий
# Модификация "colm.sh".

DOC_REQUEST=70

if [ "$1" = "-h"  -o "$1" = "--help" ]     # Запрос справки.
then
  echo; echo "Используйте: $0 [имя-директории]"; echo
  sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATIONXX$/p' "$0" |
  sed -e '/DOCUMENTATIONXX$/d'; exit $DOC_REQUEST; fi


: <<DOCUMENTATIONXX
Список статистических данных заданной директории в табличном формате.
---------------------------------------------------------------
Параметром командной строки задается директория для перечисления.
Если директория не задана или заданная директория не может быть прочитана,
то перечисляется текущая рабочая директория.

DOCUMENTATIONXX

if [ -z "$1" -o ! -r "$1" ]
then
  directory=.
else
  directory="$1"
fi

echo "Перечисление "$directory":"; echo
(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l "$directory" | sed 1d) | column -t
```

```
exit 0
```

Использование ***сценария cat*** является альтернативным способом выполнения этой задачи.

```
DOC_REQUEST=70

if [ "$1" = "-h"  -o "$1" = "--help" ]     # Запрос справки.
then                                       # Применяется "сценарий cat" ...
  cat <<DOCUMENTATIONXX
Список статистических данных заданной директории в табличном формате.
-----------------------------------------------------------------
Параметром командной строки задается директория для перечисления.
Если директория не задана или заданная директория не может быть прочитана,
то перечисляется текущая рабочая директория.

DOCUMENTATIONXX
exit $DOC_REQUEST
fi
```

Также смотрите Пример A-28, Пример A-40, Пример A-4, Пример A-42 и другие примеры самодокументируемых сценариев.

Here documents создает временные файлы, но эти файлы, после открытия, удаляются и не доступны для любого другого процесса.

```
bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> EOF
lsof    1213 bozo    0r   REG    3,5    0 30386 /tmp/t1213-0-sh
(deleted)
```

Некоторые утилиты не работают внутри *here document*.

Закрывающая ***limit string***, в конце строки ***here document***, должна начинаться *с символа на первой* позиции. Здесь не может быть ***ведущих пробелов***. ***Конечные пробелы*** после ***limit string***, также приводят к неожиданному поведению. Пробелы, в начале и в конце, не дают возможности определить ***limit string***. [1]

```
#!/bin/bash

echo
"-------------------------------------------------------------------
------"

cat <<LimitString
echo "Это строка 1 сообщения внутри here document."
echo "Это строка 2 сообщения внутри here document."
echo "Это конечная строка сообщения внутри here document."
     LimitString
```

```
#^^^^ Отступ limit string. Ошибка! Этот сценарий не будет вести
#+ себя так, как ожидается.

echo
"----------------------------------------------------------------
------"

#  Эти комментарии вне 'here document',
#+ и не будут выведены на экран.

echo "Вне here document."

exit 0

echo "Эту строку лучше не выводить на экран."

# Следует команда «exit».
```

Некоторые пользователи очень умело используют одиночный **!,** как ***limit string***. Но, это не совсем хорошая идея.

```
# Это работает.
cat <<!
Привет!
! Еще три восклицательных знака!!!
!


# Но ...
cat <<!
Привет!
Следует один восклицательный знак!
!
!
# Сбой с сообщением об ошибке.


# Тем не менее, следующее будет работать.
cat <<EOF
Привет!
Следует один восклицательный знак!
!
EOF
# Это безопаснее для использования многосимвольной  limit string.
```

Выполнение подобных задач слишком сложно с ***here document***, рассмотрите возможность использования языка сценариев ***expect***, который был специально разработан для вывода в интерактивные программы.

Примечания

[1] Исключением является, как пояснил Dennis Benzinger, использование **<<-** для

подавления табуляций.

19.1. Here Strings

> ***Here string*** может рассматриваться, как урезанный вид ***here document***. Она состоит из **КОМАНДА <<< $СЛОВО**, где $СЛОВО расширяется и подается на stdin КОМАНДА.

В качестве простого примера рассмотрим эту альтернативу конструкции **echo-grep**.

```
# Вместо:
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
# и т.д.

# Попробуем:
if grep -q "txt" <<< "$VAR"
then         #   ^^^
   echo "$VAR содержит в строках последовательность \"txt\""
fi
# Спасибо Sebastian Kaminski за предложение.
```

Или, в комбинации с **read**:

```
String="Это строка из слов."

read -r -a Words <<< "$String"
#  Опция -a в "read" последовательно присваивает
#+ членам массива результирующие значения.

echo "Первое слово в String:      ${Words[0]}"   # Это
echo "Второе слово в String:      ${Words[1]}"   # строка
echo "Третье сово в String:       ${Words[2]}"   # из
echo "Четвертое слово в String:   ${Words[3]}"   # слов
echo "Седьмое слово в String:     ${Words[4]}"   # (null)
                                                 # Конец $String.

# Спасибо Francisco Lobo за предложение.
```

Конечно, можно передавать вывод ***here string*** в stdin ***цикла***.

```
# Как поясняет Seamus...

ArrayVar=( element0 element1 element2 {A..D} )

while read element ; do
```

```
  echo "$element" 1>&2
done <<< $(echo ${ArrayVar[*]})

# element0 element1 element2 A B C D
```

Пример 19-13. Добавление строки в файл

```
#!/bin/bash
# prepend.sh: Добавление текста в начало файла.
#
#  Пример предоставлен Kenny Stauffer,
#+ но слегка изменен автором документа.


E_NOSUCHFILE=85

read -p "File: " file   # Аргумент -p в 'read' выводит приглашение.
if [ ! -e "$file" ]
then                    # Выручает, если нет такого файла.
  echo "Файл $file не найден."
  exit $E_NOSUCHFILE
fi

read -p "Title: " title
cat - $file <<<$title > $file.new

echo "Измененный файл $file.new"

exit  # Окончание выполнения сценария.

  Из 'man bash':
  Here Strings
        Вариант here documents, в формате:

                <<<word

        word расширяется и передается команде на ее стандартный ввод.


  Конечно, так же работает и:
   sed -e '1i\
   Title: ' $file
```

Пример 19-14. Разборка почтового ящика

```
#!/bin/bash
#  Сценарий Francisco Lobo,
#+ слегка изменен и прокомментирован автором ABS Guide.
#  Используется в ABS Guide с разрешения. (Спасибо!)

# Этот сценарий не запускается в версии Bash ниже 3.0.


E_MISSING_ARG=87
if [ -z "$1" ]
then
```

```
  echo "Используйте: $0 файл_почтового_ящика"
  exit $E_MISSING_ARG
fi

mbox_grep()  # Проверяем файл_почтового_ящика.
{
    declare -i body=0 match=0
    declare -a date sender
    declare mail header value


    while IFS= read -r mail
#         ^^^^                  Сброс $IFS.
#  В противном случае "read" удалит начальное и конечное место из ввода в нее.

   do
       if [[ $mail =~ ^From  ]]   # Соответствие полю "From" в сообщении.
       then
          (( body  = 0 ))           # "Обнуление" переменных.
          (( match = 0 ))
          unset date

       elif (( body ))
       then
            (( match ))
            #  echo "$mail"
            #  Раскомментируйте строку выше, если хотите вывести тело
            #+ сообщения на экран.

   elif [[ $mail ]]; then
      IFS=: read -r header value <<< "$mail"
      #                          ^^^  "here string"

      case "$header" in
      [Ff][Rr][Oo][Mm] ) [[ $value =~ "$2" ]] && (( match++ )) ;;
      # Соответствие строке "From".
      [Dd][Aa][Tt][Ee] ) read -r -a date <<< "$value" ;;
      #                                  ^^^
      # Соответствие строке "Date".
      [Rr][Ee][Cc][Ee][Ii][Vv][Ee][Dd] ) read -r -a sender <<< "$value" ;;
      #                                                    ^^^
      # Соответствие IP адресу (может быть фальшивым).
      esac

       else
          (( body++ ))
          (( match  )) &&
          echo "MESSAGE ${date:+of: ${date[*]} }"
       #    Весь массив $date              ^
          echo "IP адрес отправителя: ${sender[1]}"
       #    Второе поле строка "Received"      ^

       fi


    done < "$1" # Перенаправление stdout файла в цикл.
}


mbox_grep "$1"  # Файл_почтового_ящика отправлен в функцию.
```

```
exit $?

# Упражнения:
# ---------
# 1) Разбейте одну функцию, выше, на несколько,
#+   для лучшей читабельности.
# 2) Добавьте в сценарий анализ для проверки различных ключевых слов.



$ mailbox_grep.sh scam_mail
  MESSAGE of Thu, 5 Jan 2006 08:00:56 -0500 (EST)
  IP address of sender: 196.3.62.4
```

Упражнение: Найдите другое применение ***here strings***, такое как, например, передача ввода в **dc**.

Глава 20. Перенаправление I/O

Содержание

По умолчанию, всегда существует три открытых файла [1], `stdin` (***клавиатура***), `stdout` (***экран***) и `stderr` (***вывод сообщения об ошибках на экран***). Эти и любые другие открытые файлы могут быть перенаправлены. Перенаправление означает простой захват вывода файла, команды, программы, сценария или даже блока кода сценария (см. Пример 3-1 и Пример 3-2) и его направления, в качестве входных данных, другому файлу, команде, программе или сценарию.

Каждый открытый файл получает определенный дескриптор файла.[2] Файловые дескрипторы `stdin`, `stdout` и `stderr` имеют значения `0`, `1` и `2`, соответственно. Для дополнительных открытых файлов, остаются дескрипторы `3-9`. Иногда полезно назначать один из этих дополнительных файловых дескрипторов `stdin`, `stdout` или `stderr`, как временную дублирующую ссылку. [3] Это упрощает восстановление нормальной работы после сложных перенаправлений и перестановок (см. Пример 20-1).

```
   ВЫВОД_КОМАНДЫ >
# Перенаправляет stdout в файл.
# Если файла нет – он создается, если существует, то он перезаписывается.
```

```
   ls -lR > dir-tree.list
# Создание файла содержащего листинг дерева директории.

   : > filename
# > уменьшает файл "filename" до нулевого размера.
# Если файла нет, то создается файл нулевого размера, (подобно 'touch').
# : служит фиктивным заполнителем, не производя никакого вывода.

   > filename
# > уменьшает файл "filename" до нулевого размера.
# Если файла нет, то создается файл нулевого размера, (подобно 'touch').
# (Результат подобен ": >", выше, но не работает в некоторых оболочках.)

   ВЫВОД_КОМАНДЫ >>
# Перенаправляет stdout в файл.
# Создает файл, если его нет, в противном случае — добавляет в него.


# Однострочные команды перенаправления (влияют только на ту строку, где
#+ находятся):
# ----------------------------------------------------------------------

   1>filename
# Перенаправляет stdout в файл "filename."

   1>>filename
# Перенаправляет и добавляет stdout в файл "filename."

   2>filename
# Перенаправляет stderr в файл "filename."

   2>>filename
# Перенаправляет и добавляет stderr в файл "filename."

   &>filename
# Перенаправляет оба stdout и stderr в файл "filename."
# Этот оператор работает, начиная с Bash 4, final release.

   M>N
# "M" - файловый дескриптор, по умолчанию 1, если не указан иной.
# "N" — имя файла.
# Файловый дескриптор "M" перенаправляет в файл "N."

   M>&N
# "M" — файловый дескриптор, по умолчанию 1, если не указан иной.
# "N" — другой файловый дескриптор.


#=============================================================================

# Перенаправление stdout, одна строка за раз.
   LOGFILE=script.log

   echo "Это запись направляется в файл журнала, \"$LOGFILE\"." 1>$LOGFILE
   echo "Эта запись добавляется в \"$LOGFILE\"." 1>>$LOGFILE
   echo "Эта запись тоже добавляется в \"$LOGFILE\"." 1>>$LOGFILE
   echo эта запись выводится в stdout, но не записывается в \"$LOGFILE\"."
#  Эти команды перенаправления автоматически "сбрасываются" после
#+ каждой строки.
```

```
# Перенаправление stderr, одна строка за раз.
   ERRORFILE=script.errors

   bad_command1 2>$ERRORFILE       #  Сообщение об ошибке направляется в
                                   #+  $ERRORFILE.
   bad_command2 2>>$ERRORFILE      #  Сообщение об ошибке добавляется в
                                   #+ $ERRORFILE.
   bad_command3                    #  Сообщение об ошибке выводится в
                                   #+ stderr, а не пишется в $ERRORFILE.
      # Эти команды перенаправления автоматически "сбрасываются" после
      #+ каждой строки.
      #============================================================================
```

```
   2>&1
# Перенаправление stderr в stdout.
# Сообщения об ошибках передаются туда же, куда и стандартный вывод.

   >>filename 2>&1
   bad_command >>filename 2>&1
# Присоединяет оба stdout и stderr в файл "filename" ...

   2>&1 | [команда(ы)]
   bad_command 2>&1 | awk '{print $5}'
# Направляет stderr через туннель.
# |& было добавлено в Bash 4 как аббревиатура для 2>&1 |.

   i>&j
# Перенаправление файлового дескриптора i в j.
# Все выходные данные файла указанные i отправляются в файл j.

   >&j
# Перенаправление, по умолчанию, файловый дескриптор 1 (stdout) в j.
# Весь вывод направляется в j.
```

```
   0< FILENAME
    < FILENAME
# Доступ ввода из файла.
# Команда компаньон ">",  часто используется в сочетании.
# grep искомое слово <имя_файла


   [j]<>filename
#  Открывает файл "filename" для чтения и записи,
#+ и присваивает ему дескриптор файла "j".
#  Если "filename" не существует, создает его.
#  Если дескриптор файла "j" не указан, то по умолчанию 0, stdin.
#  Приложение для записи в указанное место  в файле.

   echo 1234567890 > File    # Записывает строку в "File".
   exec 3<> File             # Открывает "File" и присваивает ему файловый
                             #+ дескриптор 3.
   read -n 4 <&3             # Считывает только 4 символа.
   echo -n . >&3             # Записывает в него десятичную точку.
   exec 3>&-                 # Закрывает дескриптор 3.
   cat File                  # ==> 1234.67890
#  Произвольный доступ, ей-богу.

# Конвейер (канал, туннель).
# Процесс общего назначения и инструмент связки команд.
```

```
# Подобен ">", но с более общим эффектом.
# Полезно для соединения (связки) вместе команд, сценариев, файлов и
#+ программ.
      cat *.txt | sort | uniq > result-file
# Сортирует все выводимые файлы .txt и удаляет повторяющиеся строки,
# в заключении сохраняет результат в "result-file".
```

Несколько перенаправлений ввода и вывода и/или конвейер могут быть объединены в одну командную строку.

```
команда < входной_файл > выходной_файл
# Или эквивалент:
< входной_файл команда > выходной_файл   # Хотя это необычно.

команда1 | команда2 | команда3 > выходной_файл
```

См. Пример 16-31 и Пример A-14.

Несколько выходных потоков могут быть перенаправлены в один файл.

```
ls -yz >> command.log 2>&1
#  Захват результата неправильных опций 'yz' в файле "command.log."
#  Поэтому stderr перенаправляет в файл
#+ все сообщения о происходящих здесь ошибках.
#  Отметим, однако, что следующее не дает тот же результат.
ls -yz 2>&1 >> command.log
#  Выводит сообщение об ошибке, но не пишет его в файл.
#  Точнее, вывод команды (в данном случае, null) записывается в файл, но
#+ сообщение об ошибке идет только в stdout.

#  Если перенаправляются оба stdout и stderr, то порядок команд имеет большое
#+ значение.
```

Закрытие дескрипторов файлов

n<&- Закрытие дескриптора входного файла *n*.

0<&-, **<&-** Закрытие `stdin`.

n>&- Закрытие дескриптора выходного файла *n*.

1>&-, **>&-** Закрытие `stdout`.

Дочерние процессы наследуют открытые файловые дескрипторы. Именно поэтому каналы работают. Чтобы запретить наследование файлового дескриптора, закройте его.

```
# Перенаправление конвейером только stderr.

exec 3>&1                             # Сохранение текущего "значения" stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&- # Закрываем ФД 3 для 'grep'
                                      #+ (но не для 'ls').
#                    ^^^^   ^^^^
exec 3>&-                              # Теперь закрываем его для оставшейся
                                       #+ с части сценария.
```

```
# Спасибо, S.C.
```

Более подробное объяснение перенаправления I/O см. в Приложении F.

Примечания

[1] По соглашению в UNIX и Linux потоки данных периферийных устройств (***файлов устройств***), рассматриваются как файлы, являющиеся аналогами обычных файлов.

[2] *Дескриптор файла (ФД)* это просто число, которое операционная система присваивает открытому файлу для его отслеживания. Рассматривайте его как упрощенный тип указателя на файл. Аналогично дескриптору файла в Си.

[3] Использование `file descriptor 5` может вызвать проблемы. Когда Bash создает дочерний процесс, такой как ***exec***, потомок наследует файловый дескриптор `5` (см. Chet Ramey's archived e-mail, SUBJECT: RE: File descriptor 5 is held open). Лучше всего этот файловый дескриптор не использовать.

20.1. Использование *exec*

Команда **`exec <filename`** перенаправляет `stdin` в файл. С этого момента весь ввод приходит из этого файла, а не из обычного источника (обычного ввода с клавиатуры). Этот способ обеспечивает чтение файла построчно и возможность анализа каждой введенной строки с помощью ***sed*** и/или ***awk***.

Пример 20-1. Перенаправление stdin с помощью *exec*

```
#!/bin/bash
# Перенаправление stdin с помощью 'exec'.


exec 6<&0          # Ссылка файлового дескриптора 6 на stdin.
                   # Сохраняется для stdin.

exec < data-file   # stdin заменен файлом "data-file"

read a1            # Чтение первой строки файла "data-file".
read a2            # Чтение второй строки файла "data-file."

echo
echo "Из файла читаются следующие строки."
echo "-------------------------------"
echo $a1
echo $a2

echo; echo; echo
```

```
exec 0<&6 6<&-
#  Теперь восстановим stdin из файлового дескриптора 6, где он был сохранен,
#+ и закроем дескриптор 6 ( 6<&- ), чтобы освободить его для использования
#+ другими процессами.
#
# <&6 6<&-    работает так же.

echo -n "Введите данные  "
read b1  # Теперь функция «read», как и ожидалось, читает из обычного stdin.
echo "Читается ввод из stdin."
echo "----------------------"
echo "b1 = $b1"

echo

exit 0
```

Аналогичным образом, команда **`exec >filename`** перенаправляет `stdout` указанному файлу. Она посылает весь вывод команды, который бы обычно шел в `stdout`, в этот файл.

`exec N > filename` влияет на весь сценарий или *текущую оболочку*. **PID** перенаправления сценария или оболочки с этого момента меняется. Хотя . . .

`N > filename` влияет только на новый форк процесса, а не на весь сценарий или оболочку.

Спасибо Ahmed Darwishза указание на это.

Пример 20-2. Перенаправление stdout с помощью *exec*

```
#!/bin/bash
# reassign-stdout.sh

LOGFILE=logfile.txt

exec 6>&1           # Ссылка дескриптора файла 6 на stdout.
                    # Сохраняем для stdout.

exec > $LOGFILE     # stdout заменяем файлом "logfile.txt".

# ----------------------------------------------------------- #
# Весь вывод команд в этом блоке передается в файл $LOGFILE.

echo -n "Logfile: "
date
echo "-------------------------------------"
echo

echo "Вывод команды \"ls -al\""
echo
ls -al
echo; echo
echo "Вывод команды \"df\""
```

```
echo
df

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

exec 1>&6 6>&-      # Восстанавливаем stdout и закрываем дескриптор 6.

echo
echo "== stdout восстановлен по умолчанию == "
echo
ls -al
echo

exit 0
```

Пример 20-3. Перенаправление вместе stdin и stdout в тот же сценарий *с exec*

```
#!/bin/bash
# upperconv.sh
# Преобразование заданного входного файла в верхний регистр.

E_FILE_ACCESS=70
E_WRONG_ARGS=71

if [ ! -r "$1" ]     # Читается ли заданный входной файл?
then
  echo "Не читается из введенного файла!"
  echo "Используйте: $0 входной_файл выходной_файл"
  exit $E_FILE_ACCESS
fi                   #  Будет выход с такой же ошибкой
                     #+ даже если входной файл ($1) не указан (почему?).

if [ -z "$2" ]
then
  echo "Нужно указать выходной файл."
  echo "Используйте: $0 входной_файл выходной_файл"
  exit $E_WRONG_ARGS
fi


exec 4<&0
exec < $1            # Чтение из входного файла.

exec 7>&1
exec > $2            # Запись в выходной файл.
                     # Принимаем выходной файл для записи (нужна проверка?).

# ------------------------------------------------
    cat - | tr a-z A-Z   # Преобразуем в верхний регистр.
#   ^^^^^               # Читаем из stdin.
#           ^^^^^^^^^^  # Пишем в stdout.
# Однако stdin и stdout были перенаправлены.
# Обратите внимание, что «cat» может быть пропущено.
# ------------------------------------------------

exec 1>&7 7>&-       # Восстанавливаем stout.
exec 0<&4 4<&-       # Восстанавливаем stdin.
```

```
# После восстановления, следующая строка выводится в stdout как обычно.
echo "Файл \"$1\" записан в \"$2\" преобразованным в верхний регистр."

exit 0
```

Перенаправление ввода-вывода является хорошим способом избежать проблем с недоступными переменными в *subshell*.

Пример 20-4. Избегаем subshell

```
#!/bin/bash
# avoid-subshell.sh
# Предложено Matthew Walker.

Lines=0

echo

cat myfile.txt | while read line;
                 do {
                   echo $line
                   (( Lines++ ));  #  Увеличение значения этой переменной,
                                   #+ недоступной за пределами цикла.
                                   #  Проблемы с Subshell.
                 }
                 done

echo "Число читаемых строк = $Lines"     # 0
                                         # Ошибка!

echo "-------------------------"


exec 3<> myfile.txt
while read line <&3
do {
  echo "$line"
  (( Lines++ ));                 #  Увеличение значения этой переменной,
                                 #+ недоступной за пределами цикла.
                                 #  Нет subshell, нет проблем.
}
done
exec 3>&-

echo "Число читаемых строк = $Lines"     # 8

echo

exit 0

# Строки ниже не видны сценарию.

$ cat myfile.txt

Строка 1.
Строка 2.
Строка 3.
Строка 4.
```

```
Строка 5.
Строка 6.
Строка 7.
Строка 8.
```

20.2. Перенаправление блоков кода

Блоки кода, такие как ***while***, ***until,*** циклы ***for***, даже блоки сравнения ***if/then,*** могут объединяться для перенаправления в `stdin`. Даже функции могут использовать эту форму перенаправления (см. Пример 24-11). Эту задачу решает оператор `<` в конце кодового блока.

Пример 20-5. Перенаправление цикла *while*

```
#!/bin/bash
# redir2.sh

if [ -z "$1" ]
then
  Filename=names.data       # Умолчание, если не указано имя файла.
else
  Filename=$1
fi
#  Filename=${1:-names.data}
#+ может заменить проверку выше (подстановка параметра).

count=0

echo

while [ "$name" != Smith ]  # Почему переменная $name в кавычках?
do
  read name                 # Чтение из $Filename, а не из stdin.
  echo $name
  let "count += 1"
done <"$Filename"           # Перенаправление stdin в файл $Filename.
#    ^^^^^^^^^^^^

echo; echo "$count names read"; echo

exit 0

#  Обратите внимание, что в некоторых старых языках сценариев оболочки,
#+ перенаправление цикла запускается как subshell.
#  Поэтому, $count возвращает 0, инициализируя значение вне цикла.
#  Bash и ksh избегают запуска subshell *всегда, когда возможно*,
#+ поэтому, к примеру, этот сценарий правильно работает.
#  (Спасибо Heiner Steven за разъяснение.)

#  Однако ...
#  Bash *может* иногда запускать subshell в КАНАЛЬНОМ цикле «while-read»,
#+ в отличие от ПЕРЕНАПРАВЛЕННОГО цикла 'while'.

abc=hi
echo -e "1\n2\n3" | while read l
```

```
     do abc="$l"
        echo $abc
     done
echo $abc

#  Спасибо, Bruno de Oliveira Schneider, за демонстрацию
#+ фрагмента кода выше.
#  И благодарность, Brian Onn, за исправление ошибок.
```

Пример 20-6. Другая форма перенаправления цикла *while*

```
#!/bin/bash

# Это альтернативная форма предыдущего сценария.

#  Предоставлена Heiner Steven,
#+ как временное решение для тех ситуаций, когда перенаправление
#+ цикла выполняется как subshell, и поэтому переменные, внутри цикла,
#+ не сохраняют свои значения по его завершению.


if [ -z "$1" ]
then
  Filename=names.data     # Умолчание, если имя файла не указано.
else
  Filename=$1
fi


exec 3<&0                 # Сохраняется stdin с файловым дескриптором 3.
exec 0<"$Filename"        # Перенаправление стандартного ввода.

count=0
echo


while [ "$name" != Smith ]
do
  read name               # Чтение из перенаправленного stdin ($Filename).
  echo $name
  let "count += 1"
done                      #  Цикл считывает из файла $Filename
                          #+ из-за строки 20.

#  Оригинальная версия данного сценария прекращает цикл "while"
#+ исполнением <"$Filename"
#  Упражнение:
#  Почему это необычно?


exec 0<&3                 # Восстанавливаем старый stdin.
exec 3<&-                 # Закрываем временный файловый дескриптор 3.

echo; echo "$count names read"; echo

exit 0
```

Пример 20-7. Перенаправление цикла *until*

```
#!/bin/bash
# Подобно примеру выше, но с циклом "until".

if [ -z "$1" ]
then
  Filename=names.data         # Умолчание, если имя файла не задано.
else
  Filename=$1
fi

# while [ "$name" != Smith ]
until [ "$name" = Smith ]     # Заменим  !=  на =.
do
  read name                   # Считывается из $Filename, а не из stdin.
  echo $name
done <"$Filename"             # Перенаправление stdin в файл $Filename.
#    ^^^^^^^^^^^^

# Тот же результат, как у цикла «while» в предыдущем примере.

exit 0
```

Пример 20-8. Перенаправление цикла *for*

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data          # Умолчание, если имя файла не задано.
else
  Filename=$1
fi

line_count=`wc $Filename | awk '{ print $1 }'`
#           Число строк в целевом файле.
#
#  Очень надуманно и запутано, тем не менее видно, что
#+ можно перенаправлять stdin в пределах цикла for...,
#+ если сообразите как.
#
# Короче     line_count=$(wc -l < "$Filename")


for name in `seq $line_count`    # Напомним, что «seq» выводит
                                 #+ последовательность чисел.
# while [ "$name" != Smith ]    -- более сложно, чем с циклом "while" --
do
  read name                    # Считывание из $Filename, а не из stdin.
  echo $name
  if [ "$name" = Smith ]       # Здесь все нужное дополнительное содержимое.
  then
    break
  fi
done <"$Filename"              # Перенаправление stdin в файл $Filename.
#    ^^^^^^^^^^^^
```

```
exit 0
```

Можно изменить предыдущий пример так, чтобы перенаправить вывод цикла.

Пример 20-9. Перенаправление цикла *for* (перенаправление обоих stdin и stdout)

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data          # Умолчание, если файл не указан.
else
  Filename=$1
fi

Savefile=$Filename.new         # Файл для сохранения результата.
FinalName=Jonah                # Имя, на котором прекращается «чтение».

line_count=`wc $Filename | awk '{ print $1 }'` # Число строк целевого файла.


for name in `seq $line_count`
do
  read name
  echo "$name"
  if [ "$name" = "$FinalName" ]
  then
    break
  fi
done < "$Filename" > "$Savefile"    # Перенаправление stdin в файл $Filename,
#    ^^^^^^^^^^^^^^^^^^^^^^^^^^^      и сохранение резервной копии файла.

exit 0
```

Пример 20-10. Перенаправление сравнения *if/then*

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data   # Умолчание, если файл не задан.
else
  Filename=$1
fi

TRUE=1

if [ "$TRUE" ]          # if true и if: то же сработает.
then
 read name
 echo $name
fi <"$Filename"
#  ^^^^^^^^^^^^

# Считывается только первая строка файла.
```

```
# Сравнение "if/then" не имеет возможности прохода, если не встроено в цикл.

exit 0
```

Пример 20-11. Данные файла *names.data* для примеров выше

```
Aristotle
Arrhenius
Belisarius
Capablanca
Dickens
Euler
Goethe
Hegel
Jonah
Laplace
Maroczy
Purcell
Schmidt
Schopenhauer
Semmelweiss
Smith
Steinmetz
Tukhashevsky
Turing
Venn
Warshawski
Znosko-Borowski

#  Это файл данных для
#+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

Перенаправление `stdout` блока кода имеет эффект сохранения выходных данных в файл. См. Пример 3-2.

Here documents представляет собой особый случай перенаправления блоков кода. В этом случае, можно направлять вывод ***here document*** на `stdin` *цикла* ***while***.

```
# Пример Albert Siersema
# Используется с разрешения (благодарность!).

function doesOutput()
 # Могут быть и внешние команды, конечно.
 # Здесь мы демонстрируем, что можно использовать также и функцию.
{
  ls -al *.jpg | awk '{print $5,$9}'
}


nr=0          #  Мы хотим, чтобы цикл while имел возможность управления
totalSize=0   #+ и был в состоянии увидеть изменения после его окончания.

while read fileSize fileName ; do
  echo "$fileName это $fileSize байт"
```

```
  let nr++
  totalSize=$((totalSize+fileSize))   # Или: "let totalSize+=fileSize"
done<<EOF
$(doesOutput)
EOF

echo "$nr файлов составляет $totalSize байт"
```

20.3. Приложения

Умелое использование перенаправления ввода-вывода позволяет разделять и соединять фрагменты вывода команды (см. Пример 15-7). Что, в свою очередь, позволяет создавать отчеты и лог-файлы.

Пример 20-12. Регистрация событий

```
#!/bin/bash
# logevents.sh
# Автор: Stephane Chazelas.
# Используется в ABS Guide с разрешения.

# Регистрация событий в файл.
# Запускается из-под root (для возможности записи в /var/log).

ROOT_UID=0     # Только пользователь с $UID 0 имеет права root.
E_NOTROOT=67   # Не-root, выводится как ошибка.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Этот сценарий может запускать root."
  exit $E_NOTROOT
fi


FD_DEBUG1=3
FD_DEBUG2=4
FD_DEBUG3=5

# == Раскомментируйте одну из двух строк ниже, чтобы активировать сценарий.
==
# LOG_EVENTS=1
# LOG_VARS=1


log()  # Запись даты и времени в log-файл.
{
echo "$(date)  $*" >&7     # *Добавление* даты в файл.
#     ^^^^^^^  подстановка команды
                           # См. ниже.
}


case $LOG_LEVEL in
 1) exec 3>&2         4> /dev/null 5> /dev/null;;
 2) exec 3>&2         4>&2         5> /dev/null;;
```

```
 3) exec 3>&2         4>&2         5>&2;;
 *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
esac

FD_LOGVARS=6
if [[ $LOG_VARS ]]
then exec 6>> /var/log/vars.log
else exec 6> /dev/null                     # Получаем вывод.
fi

FD_LOGEVENTS=7
if [[ $LOG_EVENTS ]]
then
  # exec 7>(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
  # Строка выше не работает в версиях Bash более поздних, чем 2,04. Почему?
  exec 7>> /var/log/event.log              # Добавляется в "event.log".
  log                                      # Запись времени и даты.
else exec 7> /dev/null                     # Получаем вывод.
fi

echo "DEBUG3: beginning" >&${FD_DEBUG3}

ls -l >&5 2>&4                             # команда1 >&5 2>&4

echo "Done"                                # команда2

echo "sending mail" >&${FD_LOGEVENTS}
# Запись "отправленной почты" в файловый дескриптор 7.


exit 0
```

Глава 21. Subshells (Подоболочки)

Subshell *это* ***новый*** запускаемый процесс уже запущенным сценарием оболочки (***дочерний*** процесс или процесс ***потомок***).

> **Определение:** *subshell* - это **дочерний процесс** запущенный *оболочкой* (или *сценарием оболочки*).

Подоболочка является отдельным экземпляром командного процессора - *оболочки*, которая дает вам приглашение в консоли или в окне XTERM. Так же, как командной строкой интерпретируются ваши команды, так же делает и сценарий процесса пакетного вывода команд. Каждый запускаемый сценарий является, по сути, подпроцессом (*subshell* - дочерним процессом) родительской оболочки.

Сценарий оболочки может сам запускать подпроцессы. Эти подоболочки позволяют сценарию производить эффективную параллельную обработку выполняющихся одновременно нескольких подзадач.

```
#!/bin/bash
# subshell-test.sh

(
# Внутри скобок, а поэтому subshell...
while [ 1 ]   # Бесконечный цикл.
do
  echo "Запущенна subshell..."
done
)

#  Сценарий будет работать вечно
#+ или до тех пор, пока не будет прекращен комбинацией Ctl-C.

exit $?  # Конец сценария (никогда здесь не закончится).



Теперь запускаем сценарий:
sh subshell-test.sh

И в это время запускаем сценарий из другого xterm:
ps -ef | grep subshell-test.sh

UID        PID   PPID  C STIME TTY      TIME     CMD
500       2698  2502  0 14:26 pts/4    00:00:00 sh subshell-test.sh
500       2699  2698 21 14:26 pts/4    00:00:24 sh subshell-test.sh
          ^^^^

Анализ:
PID 2698, сценария, запустившего PID 2699, подоболочку.
```

```
Примечание: Строка "UID ..." отфильтрована командой «grep»,
а здесь показана для примера.
```

В общем, ***внешняя*** команда, в форках сценария, ***отключает*** подпроцесс, [1], тогда как ***встроенная*** в Bash - нет. По этой причине, встроенные команды выполняются быстрее и используют меньше системных ресурсов, чем эквивалентные им внешние команды.

Список команд в круглых скобках

```
( команда1; команда2; команда3; ... )
```

Список команд ***заключенный в скобки*** работает как **подоболочка** (***subshell***).

Переменные подоболочки *не видны* вне блока кода подоболочки. Они не являются доступными для родительского процесса, т.е. для оболочки запустившей подоболочку. Это, по сути, ***локальные*** переменные дочернего процесса.

Пример 21-1. Область видимости переменной в подоболочке

```
#!/bin/bash
# subshell.sh

echo

echo "Мы вне subshell."
echo "Значение subshell ВНЕ subshell = $BASH_SUBSHELL"
# В Bash, версии 3, добавлена новая переменная  $BASH_SUBSHELL.
echo; echo

outer_variable=Outer
global_variable=
#  Объявление глобальной переменной для хранения значения переменной
subshell.

(
echo "Мы внутри subshell."
echo "Значение subshell ВНУТРИ subshell = $BASH_SUBSHELL"
inner_variable=Inner

echo "Из находящегося в subshell, \"inner_variable\" = $inner_variable"
echo "Из находящегося в subshell, \"outer\" = $outer_variable"

global_variable="$inner_variable"   #  Можно ли 'экспортировать'
                                    #+ переменную подоболочки?
)

echo; echo
echo "Мы вне subshell."
echo "Значение subshell ВНЕ subshell = $BASH_SUBSHELL"
echo

if [ -z "$inner_variable" ]
then
  echo "inner_variable не определена основной частью оболочки"
else
```

```
  echo "inner_variable определена основной частью оболочки"
fi

echo "Из основной части оболочки, \"inner_variable\" = $inner_variable"
#  $inner_variable будет пустой (не инициализированной)
#+ потому что переменные определяемые в subshell это "локальные переменные".
#  Есть средство от этого?
echo "global_variable = "$global_variable""  # Почему это не работает?

echo

# ============================================================================

# Дополнительно ...

echo "-----------------"; echo

var=41                                          # Глобальная переменная.

( let "var+=1"; echo "\$var ВНУТРИ subshell = $var" )  # 42

echo "\$var ВНЕ subshell = $var"                  # 41
#  Операции с переменной  внутри subshell, даже ГЛОБАЛЬНОЙ переменной, не
#+ влияют на значение переменной за пределами subshell!


exit 0

#  Вопросы:
#  --------
#  Однажды выйдя из подоболочки, есть ли способ, чтобы повторно, в той же
#+ подоболочке, изменять или иметь доступ к переменным подоболочки ?
```

См. также **$ BASHPID** и Пример 34-2.

Определение: Область действия переменной - это область, в которой она имеет смысл, в которой *она имеет значение на которое может ссылаться*. Например, видимость локальной переменной находится лишь в функции, блоке кода или подоболочке, *в которой она определена*, в то время как глобальная переменная видима всему сценарию, в котором она появляется.

В то время как внутренняя переменная $BASH_SUBSHELL указывает на уровень вложенности подоболочки, переменная $SHLVL не показывает никаких изменений в подоболочке.

```
echo " \$BASH_SUBSHELL вне subshell       = $BASH_SUBSHELL"             # 0
  ( echo " \$BASH_SUBSHELL внутри subshell        = $BASH_SUBSHELL" )   # 1
  ( ( echo " \$BASH_SUBSHELL вложено в subshell = $BASH_SUBSHELL" ) )   # 2
# ^ ^                    *** вложено ***                          ^ ^

echo

echo " \$SHLVL вне subshell = $SHLVL"       # 3
( echo " \$SHLVL внутри subshell  = $SHLVL" )   # 3 (без изменений!)
```

Изменения в директориях, выполненные в подоболочке, не переносятся в родительскую оболочку.

Пример 21-2. Список профилей пользователей

```
#!/bin/bash
# allprofs.sh: Вывод всех профилей пользователей.

# Сценарий написан Heiner Steven и изменен автором этого документа.

FILE=.bashrc  #  Файл, содержащий профили пользователя,
              #+ в оригинале сценария был ".profile".

for home in `awk -F: '{print $6}' /etc/passwd`
do
  [ -d "$home" ] || continue    # Если директория не домашняя,то переход к
                                #+ следующей.
  [ -r "$home" ] || continue    # Если не читается,то переход к следующей.
  ( cd $home; [ -e $FILE ] && less $FILE )
done

#  Когда сценарий завершается, нет необходимости в «переходе» обратно в
#+ исходный каталог, потому что в subshell имеет место 'cd $home'.

exit 0
```

Subshell может использоваться для настройки «специальной среды» для группы команд.

```
КОМАНДА1
КОМАНДА2
КОМАНДА3
(
  IFS=:
  PATH=/bin
  unset TERMINFO
  set -C
  shift 5
  КОМАНДА4
  КОМАНДА5
  exit 3 # Только выход из subshell!
)
# Родительская оболочка не пострадала и сохранилась окружающая среда.
КОМАНДА6
КОМАНДА7
```

Команда `exit` завершает только ***subshell***, в которой она выполняется, а не родительскую оболочку или сценарий.

Одним из применений таких «специальных сред» является проверка объявления переменной.

```
If ( set -u; : $variable ) 2> /dev/null
then
  echo "Переменная объявлена."
fi     #  Переменная задается в текущем сценарии,
       #+ или как внутренняя переменная Bash,
       #+ или присутствующая в окружающей среде (была экспортирована).

# Может быть записано [[ ${variable-x} != x || ${variable-y} != y ]]
```

```
# или                    [[ ${variable-x} != x$variable ]]
# или                    [[ ${variable+x} = x ]]
# или                    [[ ${variable-x} != x ]]
```

Другое приложение проверяет файл блокировки

```
If ( set -C; : > lock_file ) 2> /dev/null
then
  :   # lock_file не существует: ни один пользователь, не запустил сценарий.
else
  echo "Какой-то пользователь уже запустил этот сценарий."
exit 65
fi

#  Фрагмент кода Stéphane Chazelas,
#+ измененный Paulo Marcel Coelho Aragao.
```

В разных подоболочках процессы могут выполняться параллельно. Это позволяет разбивать сложную задачу на части, одновременно обрабатывая их.

Пример 21-3. Запуск параллельных процессов в subshells

```
        ( cat список1 список2 список3 | sort | uniq > список123 ) &
        ( cat список4 список5 список6 | sort | uniq > список456 ) &
        # Объединяет и одновременно сортирует оба набора списков.
        # Запущено в фоновом режиме обеспечивая параллельное выполнение.
        #
        # Такой же эффект как
        #   cat список1 список2 список3 | sort | uniq > список123 &
        #   cat список4 список5 список6 | sort | uniq > список456 &

        wait   # Следующая команда не выполняется  до тех пор, пока
               #+ подоболочка не закончит работу.

        diff список123 список456
```

Для перенаправления I/O в ***subshell*** используется оператор "**|**" *конвейер*, например так **`ls -al | (команда)`**.

Блок кода в *фигурных скобках* ***не*** запускает ***subshell***.

```
{ команда1; команда2; команда3; . . . командаN; }
```

```
var1=23
echo "$var1"   # 23

{ var1=76; }
echo "$var1"   # 76
```

Примечания

[1] Внешняя команда вызываемая **exec**, (как правило) не является форком подпроцесса/подоболочки.

Глава 22. Ограниченные оболочки

Отключенные команды в ограниченных оболочках

Запуск сценария или части сценария в ***ограниченном*** *режиме* отключает определенные команды, которые в обычном режиме были бы доступны. Эта мера безопасности специально ограничивает привилегии пользователя сценария и сводит к минимуму возможный ущерб от выполнения сценария.

Отключаются следующие команды и действия:

- Использование *cd* для изменения рабочей директории.
- Изменение значений переменных сред *$PATH*, *$SHELL*, *$BASH_ENV* или *$ENV*.
- Чтение или изменение опций среды оболочки *$SHELLOPTS*.
- Вывод перенаправления.
- Вызов команды, содержащей один или более / (слэш).
- Вызов ***exec*** для подстановки другого процесса оболочки.
- Различные другие команды, которые позволили бы направить сценарий к непредусмотренной цели.
- Выход из режима ограничения в сценарии.

Пример 22-1. Запуск сценария в ограниченном режиме

```
#!/bin/bash

#  Запуск сценария с "#!/bin/bash -r"
#+ заставляет работать  сценарий в ограниченном режиме.

echo

echo "Изменяем директорию."
cd /usr/local
echo "Теперь в `pwd`"
echo "Возвращаемся обратно."
cd
```

```
echo "Теперь в `pwd`"
echo

# Все здесь в нормальном, неограниченном режиме.

set -r
# установлен --restricted  имеющий другой эффект.
echo "==> Теперь в ограниченном режиме. <=="

echo
echo

echo "Пытаемся изменить директорию в ограниченном режиме."
cd ..
echo "Все еще в `pwd`"

echo
echo

echo "\$SHELL = $SHELL"
echo "Пытаемся изменить оболочку в ограниченном режиме."
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo

echo "Пытаемся перенаправить вывод в ограниченном режиме."
ls -l /usr/bin > bin.files
ls -l bin.files    # Попробуем принудительно вывести файл.

echo

exit 0
```

Глава 23. Подстановка процесса

Конвейер `stdout` команды в `stdin` другой команды – мощное средство. Но, что делать, если вам нужно передать `stdout` *нескольких* команд? Это возможно сделать *`подстановкой процесса`*.

Операция подстановки процессов передает вывод процесса (или процессов) в `stdin` другого процесса.

Шаблон

Список команд заключается в скобки

>(список_команд)

<(список_команд)

Процесс подстановки использует файлы `/dev/fd/<n>` для передачи результатов процесса(ов) внутри скобок другому процессу. [1]

Между "**<**" или "**>**" и скобками ***отсутствует*** пробел или пустое место. Использование пробела или пустого места вызовет сообщение об ошибке.

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/dev/fd/63

bash$ echo >(true) <(true)
/dev/fd/63 /dev/fd/62



bash$ wc <(cat /usr/share/dict/linux.words)
 483523  483523 4992010 /dev/fd/63

bash$ grep script /usr/share/dict/linux.words | wc
    262     262    3601

bash$ wc <(grep script /usr/share/dict/linux.words)
    262     262    3601 /dev/fd/63
```

Bash создает канал с двумя ***файловыми дескрипторами***, `--fIn` и `fOut--`. `Stdin` **true** соединяется с `fOut` (dup2(fOut, 0)), затем Bash передает аргумент `/dev/fd/fIn` в **echo**. В системах, где отсутствуют файлы `/dev/fd/<n>` , Bash может использовать временные файлы (Спасибо, S.C.)

Процесс подстановки может сравнивать выводы двух разных команд или даже выводы различных вариантов одной и той же команды.

```
bash$ comm <(ls -l) <(ls -al)
total 12
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
        total 20
        drwxrwxrwx    2 bozo bozo     4096 Mar 10 18:10 .
        drwx------   72 bozo bozo     4096 Mar 10 17:58 ..
        -rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
        -rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
```

```
        -rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh

```

Процессом подстановки можно сравнивать содержимое двух директорий – чтобы видеть какие файлы одной директории отсутствуют в другой.

```
diff <(ls $первая_директория) <(ls $вторая_директория)
```

Другие случаи использования подстановки процесса :

```
read -a list < <(od -Ad -w24 -t u2 /dev/urandom)
#  Чтение списка случайных чисел из /dev/urandom,
#+ обработка «od»
#+ и передача stdin в 'read'...

#  Пример из сценария "insertion-sort.bash".
#  Благодарность JuanJo Ciarlante.
```

```
PORT=6881   # bittorrent (Битторент)

# Сканирование порта, для проверки, что ничего плохого на нем не происходит.
netcat -l $PORT | tee >(md5sum ->mydata-orig.md5) |
gzip | tee >(md5sum - | sed 's/-$/mydata.lz2/'>mydata-gz.md5)> mydata.gz

# Проверка распаковки:
  gzip -d <(mydata.gz | md5sum -c mydata-orig.md5)
# md5sum оригинала проверяет stdin и обнаруживает проблемы сжатия.

#  Этот пример предоставлен Bill Davidsen
#+ (с небольшими изменениями автора ABS Guide).


cat <(ls -l)
# То же, что и  ls -l | cat

sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
# Список всех файлов в 3 основных директориях «bin» и сортировка файлов по
#+ имени.
# Обратите внимание, что три отдельные команды передаются в «sort».


diff <(команда1) <(команда2)    # Дает разницу в выводе команд.

tar cf >(bzip2 -c > file.tar.bz2) $directory_name
# Вызов "tar cf /dev/fd/?? $directory_name", и "bzip2 -c > file.tar.bz2".
#
# Из-за функционирования системы /dev/fd/<n>,
# конвейер между командами не должен быть именованным.
#
# Но может быть эмулирован.
#
bzip2 -c < pipe > file.tar.bz2&
tar cf pipe $directory_name
rm pipe
#       или
exec 3>&1
```

```
tar cf /dev/fd/4 $directory_name 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2 3>&-
exec 3>&-


# Спасибо Stéphane Chazelas
```

Есть способ обойти проблему передачи ***echo*** при помощи *цикла* ***while-read***, запущенного в подоболочке.

Пример 23-1. Перенаправление блока кода без порождения потомка (форка)

```
#!/bin/bash
# wr-ps.bash: цикл while-read с подстановкой процесса.

# Пример предоставлен Tomas Pospisek.
# (Сильно отредактирован автором ABS Guide.)

echo

echo "случайный ввод" | while read i
do
  global=3D": Не доступна за пределами цикла."
  # ... потому что запущена в subshell.
done

echo "\$global (вне подпроцесса) = $global"
# $global (вне подпроцесса) =

echo; echo "--"; echo

while read i
do
  echo $i
  global=3D": Доступна за пределами цикла."
  # ... потому что она НЕ запущена в subshell.
done < <(echo "случайный ввод")
#    ^ ^

echo "\$global (используется подстановка процесса) = $global"
# Случайный ввод
# $global (используется подстановка процесса) = 3D: Доступна за пределами
цикла.


echo; echo "##########"; echo



# А также ...

declare -a inloop
index=0
cat $0 | while read line
do
  inloop[$index]="$line"
  ((index++))
  # Запущено в subshell, далее ...
done
```

```
echo "OUTPUT = "
echo ${inloop[*]}           # ... ничего не выводится на экран.


echo; echo "--"; echo


declare -a outloop
index=0
while read line
do
  outloop[$index]="$line"
  ((index++))
  # НЕ запущено в subshell, далее ...
done < <( cat $0 )
echo "OUTPUT = "
echo ${outloop[*]}          # ... выводится на экран весь сценарий.

exit $?
```

Вот еще подобный пример.

Пример 23-2. Перенаправление вывода *подстановки процесса* в цикл.

```
#!/bin/bash
# psub.bash

# Вдохновлено Diego Molina (спасибо!).

declare -a array0
while read
do
  array0[${#array0[@]}]="$REPLY"
done < <(sed -e 's/bash/CRASH-BANG!/' $0 | grep bin | awk '{print $1}')
#  Устанавливает в значение по умолчанию переменную 'read', $REPLY,
#+ подстановка процессов, а затем копирование в массив.
echo "${array0[@]}"

exit $?

# ===================================== #

bash psub.bash

#!/bin/CRASH-BANG! done #!/bin/CRASH-BANG!
```

Один из итателей прислал следующий интересный пример процесса подстановки.

```
# Фрагмент сценария взятый из дистрибутива SuSE:

# --------------------------------------------------------------#
while read  des what mask iface; do
# Некоторые команды   ...
done < <(route -n)
#    ^ ^  Первое < это перенаправление, второе- подстановка процесса.
```

```
# Чтобы проверить это, давайте проделаем что-то.
while read  des what mask iface; do
  echo $des $what $mask $iface
done < <(route -n)

# Вывод:
# Kernel IP routing table
# Destination Gateway Genmask Flags Metric Ref Use Iface
# 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
# ------------------------------------------------------------#

#  Как пояснил это Stéphane Chazelas,
#+ для понимания проще эквивалент:
route -n |
  while read des what mask iface; do   # Переменные присваиваются из
                                       #+ вывода канала.
    echo $des $what $mask $iface
  done  #  Это дает такой же результат, как и выше.
        #  Однако, как поясняет Ulrich Gayer ...
        #+ это упрощенный эквивалент цикла while, использующий  subshell,
        #+ а поэтому, переменные исчезают, когда исчезает канал.

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

#  Filip Moritz поясняет, что существует маленькое различие между двумя
#+ примерами выше, что и увидим далее.

(
route -n | while read x; do ((y++)); done
echo $y # $y все еще не присвоено

while read x; do ((y++)); done < <(route -n)
echo $y # $y количество строк вывода route -n
)

Вообще говоря
(
: | x=x
# кажется, что запускается subshell
: | ( x=x )
# в то время как
x=x < <(:)
# на самом деле - нет
)

# Это полезно для синтаксического анализа csv и т.п.
# То есть, по сути, вот что делает исходный фрагмент кода SuSE.
```

Примечания

[1] Имеет тот же эффект, что и именованный канал (временный файл), а именованные каналы, на самом деле, в одно время использовались в подстановке процесса.

Глава 24. Функции

Содержание

24.2. Локальные переменные
24.3. Рекурсия без локальных переменных

Как и «настоящие» языки программирования, Bash имеет функции, хотя и осуществляемые в несколько ограниченном виде. Функция — это подпрограмма, блок кода, который реализует набор операций, «черный ящик», который выполняет указанную задачу. Везде, где есть повторяющийся код, когда задача повторяется лишь с незначительными изменениями процедуры, рекомендуется использовать функции.

`function`

```
имя_функции {
команда...
}
```

или

```
имя_функции () {
команда...
}
```

Вторая форма будет по сердцу программистам Си (и более переносима).

Как и в Си, открывающая скобка функции, при необходимости, может появляться на второй строке.

```
имя_функции ()
{
команда...
}
```

Функция может «сжаться» в одну строку.

```
fun () { echo "Это функция"; echo; }
#                          ^     ^
```

В этом случае, *точка с запятой* должны завершать последнюю команду в функции.

```
fun () { echo "Это функция"; echo } # Ошибка!
#                                ^

fun2 () { echo "Даже функция из одной команды? Да!"; }
#                                                   ^
```

Функции вызываются, начинают *работать*, просто вызвав их по имени. *Вызов функции*

эквивалентен ***команде***.

Пример 24-1. Простые функции

```
#!/bin/bash
# ex59.sh: Выполнение функций (простых).

JUST_A_SECOND=1

funky ()
{ # Это, примерно, так же просто, как присваивание функции.
  echo "Это функция funky."
  echo "Теперь выйдем из функции funky."
} # Объявление функции должно осуществляться до ее вызова.


fun ()
{ # Несколько более сложная функция.
  i=0
  REPEATS=30

  echo
  echo "И теперь, действительно, начинается самое интересное."
  echo

  sleep $JUST_A_SECOND    # Эй, подожди секунду!
  while [ $i -lt $REPEATS ]
  do
    echo "----------FUNCTIONS---------->"
    echo "<------------ARE------------"
    echo "<------------FUN------------>"
    echo
    let "i+=1"
  done
}

  # Теперь вызываем функции.

funky
fun

exit $?
```

Объявление функции должно осуществляться ***до первого ее вызова***. Не существует способа «объявления» функции, как, например, в Си.

```
f1
# Даст сообщение об ошибке, поскольку функция «f1» пока еще не объявлена.

declare -f f1      # Это тоже не поможет.
f1                 # Все еще сообщение об ошибке.

# Однако...


f1 ()
{
  echo "Вызывается функция \"f2\" из функции \"f1\"."
  f2
```

```
}

f2 ()
{
  echo "Функция \"f2\"."
}

f1  #  Функция «f2», до этого момента, на самом деле не вызывается, хотя
    #+ на нее и делается ссылка до ее объявления.
    #  Это допустимо.

    # Спасибо, S.C.
```

Функция не может быть ***пустой***!

```
#!/bin/bash
# empty-function.sh

empty ()
{
}

exit 0  # Здесь нет выхода!

# $ sh empty-function.sh
# empty-function.sh: строка 6: Синтаксическая ошибка или
# непредвиденная лексема `}'
# empty-function.sh: строка 6: `}'

# $ echo $?
# 2


# Обратите внимание, эта функция содержит только пустой комментарий.

func ()
{
  # Комментарий 1.
  # Комментарий 2.
  # Это, по-прежнему, пустая функция.
  # Спасибо, Mark Bova, за разъяснение.
}
# В результате те же сообщения об ошибке, как и выше.


# Однако ...

not_quite_empty ()
{
  неправильная_команда
} #  Сценарий, содержащий эту функцию, *не* взорвется тех пор, пока
  #+ функция не вызвана.

not_empty ()
{
  :
```

```
} # Содержит : (команду null), и это - нормально.


# Благодарность Dominick Geyer и Thiemo Kellner.
```

Возможно даже вкладывать функцию внутрь другой функции, хотя это не очень хорошо.

```
f1 ()
{

  f2 () # Вложена
  {
    echo "Функция \"f2\", внутри \"f1\"."
  }

}

f2  #  Выдаст сообщение об ошибке.
    #  Даже "declare -f f2" перед этим - не поможет.

echo

f1  #  Ничего не происходит, так как вызов «f1» не вызывает автоматически
    #+ «f2».
f2  #  Теперь, правильный вызов «f2»,
    #+ поскольку ее объявление было сделано явно, путем вызова «f1».

    # Спасибо, S.C.
```

Объявление функции может производиться в самых неожиданных местах, даже там, где не работают команды.

```
ls -l | foo() { echo "foo"; }  # Допустимо, но бесполезно.



if [ "$USER" = bozo ]
then
  bozo_greet ()   # Объявление функции встроенное в конструкцию if/then.
  {
    echo "Привет, Bozo."
  }
fi

bozo_greet        # Сработает только для Bozo, для других пользователей
                  #+ выводит ошибку.


# Что-то подобное может оказаться полезным в других контекстах.

NO_EXIT=1   # Дает возможность объявить функцию позже.

[[ $NO_EXIT -eq 1 ]] && exit() { true; }     #  Объявление функции
                                             #+ в "and-list".
# Если $NO_EXIT это 1, объявляется "exit ()".
# Это отключает встроенный «exit», сглаживая его значением «true».
```

```
exit  # Вызываемая функция «exit()», это не встроенный 'exit'.



# Или, подобное:
filename=file1

[ -f "$filename" ] &&
foo () { rm -f "$filename"; echo "Файл "$filename" удален."; } ||
foo () { echo "Файл "$filename" не существует."; сенсорная панель; }

foo

# Спасибо S.C. и Christopher Head
```

Имена функций могут иногда принимать странные формы.

```
  _(){ for i in {1..10}; do echo -n "$FUNCNAME"; done; echo; }
# ^^^         Нет пропуска между именем функции и скобками.
#             Это не всегда работает. Почему?

# Теперь, давайте вызовем функцию.
  _         # __________
#             ^^^^^^^^^^   10 подчеркиваний (10 x имя функции)!
# «Голое» подчеркивание является приемлемым для имени функции.


# Двоеточие также является приемлемым именем для функции.

:(){ echo ":"; }; :

# И зачем это?
# Это хитрый способ запутывания кода в сценарии.
```

См. Пример A-56.

Что произойдет, если в сценарии появятся различные версии одной и той же функции?

```
#  Как поясняет Yan Chen,
#  когда функция объявляется несколько раз,
#  то вызваться будет последний вариант.
#  Однако, это не очень хорошо.

func ()
{
  echo "Первая версия функции ()."
}

func ()
{
  echo "Вторая версия функции ()."
}

func   # Вторая версия функции ().

exit $?
```

```
#  Возможно даже использовать функции для переопределения или
#+ упреждения системных команд.
#  Конечно, это *не* желательно.
```

24.1. Сложные функции и функциональные сложности

Функции могут обрабатывать аргументы, переданные им, и возвращать статус выхода в сценарий для дальнейшей обработки.

```
имя_функции $аргумент1 $аргумент2
```

Функция обращается к позиции переданных аргументов (как к позиционным параметрам), то есть, $1, $2 и так далее.

Пример 24-2. Функция принимающая параметры

```
#!/bin/bash
# Функции и параметры

DEFAULT=default                                 # Значение параметра по
                                                #+ умолчанию.
func2 () {
   if [ -z "$1" ]                               # Параметр 1 не нулевого размера?
   then
     echo "-Параметр 1 нулевого размера.-"  # Или параметр не присвоен.
   else
     echo "-Параметр 1 это \"$1\".-"
   fi

   variable=${1-$DEFAULT}                       #  Что покажет
   echo "variable = $variable"                  #+ замещение параметра?
                                                #  ---------------------------
                                                #  Различается отсутствие
                                                #+ параметра и параметр null.

   if [ "$2" ]
   then
     echo "-Параметр 2 это \"$2\".-"
   fi

   return 0
}

echo

echo "Ничего не присваивается."
func2                           # Вызов без параметров.
echo
```

```
echo "Присвоен параметр нулевого размера."
func2 ""                      # Вызов с параметром нулевого размера.
echo

echo " Присвоен параметр Null."
func2 "$uninitialized_param"   # Вызов с не объявленным параметром
echo

echo "Присвоен один параметр."
func2 first           # Вызов с одним параметром
echo

echo "Присвоены два параметра."
func2 first second    # Вызов с двумя параметрами
echo

echo Присвоены "\"\" \"second\"."
func2 "" second       # Вызов с первым параметром нулевой длины
echo                  # и строки ASCII, как один второй параметр.

exit 0
```

Команда **shift** влияет на аргументы, передаваемые функциям

(см. Пример 36-18).

А как насчет аргументов командной строки передаваемых в сценарий? Видит ли их функция? Давайте это выясним.

Пример 24-3. Функции и аргументы командной строки передающиеся в сценарий

```
#!/bin/bash
# func-cmdlinearg.sh
#  Вызываем этот сценарий с таким аргументом командной строки,
#+ как $0 arg1.

func ()

{
echo "$1"   # Вывод на экран первого аргумента переданного функции.
}           # Аргумент определен командной строкой?

echo "Сначала вызов функции: без переданного аргумента."
echo "Посмотрите, виден ли аргумент командной строки."
func
# Нет! Аргумент командной строки не виден.

echo "============================================================="
echo
echo "Затем вызов функции: явно передан аргумент командной строки."
func $1
# Теперь виден!
```

```
exit 0
```

В отличие от некоторых других языков программирования, сценарии shell обычно передают только ***значения*** параметров функций. Имена переменных (являющиеся на самом деле *указателями*), передаваемые функциям в качестве параметров, рассматриваются буквально, как строки. Функции интерпретируют их как *аргументы, буквально*.

Косвенные ссылки на переменные (см. Пример 37-2) плохо обеспечивают механизм передачи указателя переменной функции.

Пример 24-4. Передача косвенной ссылки функции

```
#!/bin/bash
# ind-func.sh: Передача косвенной ссылки функции.

echo_var ()
{
echo "$1"
}

message=Hello
Hello=Goodbye

echo_var "$message"        # Hello
# Теперь давайте передадим косвенную ссылку функции.
echo_var "${!message}"     # Goodbye

echo "-------------"
# Что случится, если мы изменим содержимое переменной "hello"?
Hello="Hello, again!"
echo_var "$message"        # Hello
echo_var "${!message}"     # Hello, again!

exit 0
```

Следующий логичный вопрос, могут ли быть разыменованы находящиеся параметры после передачи функции.

Пример 24-5. Разыменование параметра переданного функции

```
#!/bin/bash
# dereference.sh
# Разыменование параметра переданного функции.
# Сценарий Bruce W. Clare.

dereference ()
{
     y=\$"$1"   # Имя переменной (не значение!).
     echo $y    # $Junk

     x=`eval "expr \"$y\" "`
     echo $1=$x
     eval "$1=\"Какой-то текст \""  # Присваиваем новое значение.
```

```
}

Junk="Какой-то текст"
echo $Junk "до"    # Какой-то текст до

dereference Junk
echo $Junk "после"     # Какой-то другой текст после

exit 0
```

Пример 24-6. Снова, разыменование параметра переданного функции

```
#!/bin/bash
# ref-params.sh: Разыменование параметра переданного функции.
#                (Сложный пример)

ITERATIONS=3  # Промежуток времени для получения ввода.
icount=1

my_read () {
  #  Вызывается именем переменной my_read,
  #+ выводит предыдущее значение в квадратных скобках, как значение по
  #+ умолчанию, затем запрашивает новое значение.

  local local_var

  echo -n "Enter a value "
  eval 'echo -n "[$'$1'] "'  #  Начальное значение.
# eval echo -n "[\$$1] "     #  Легче для понимания,
                             #+ но теряет конечные пробелы в приглашении
                             #+ пользователя.
  read local_var
  [ -n "$local_var" ] && eval $1=\$local_var

  # "And-list": если "local_var" устанавливает "$1" в это значение.
}

echo

while [ "$icount" -le "$ITERATIONS" ]
do
  my_read var
  echo "Записывается #$icount = $var"
  let "icount += 1"
  echo
done


#  Благодарность Stephane Chazelas за предоставление этого поучительного
#+ примера.

exit 0
```

Выход и возвращение

Статус выхода

Функция возвращает значение называемое ***exit status*** *(статус выхода)*. Аналогичный статусу выхода возвращаемого командой. Статус выхода должен быть явно указан оператором ***`return`***, в противном случае это будет статус выхода последней команды в функции (`0` в случае успеха и не нулевой код в случае ошибки). Этот *статус выхода* может использован в сценарии, ссылаясь на него, как на **`$?`**. Этот механизм позволяет эффективно использовать «возвращаемое значение» функции сценария, подобно функциям Си.

`return`

Завершает функцию. Команда **`return`** [1] необязательно принимает значение *целого* числа, возвращаемого вызвавшему ее сценарию, как '*статус выхода*' функции, и этот статус выхода присваивается переменной **`$?`**.

Пример 24-7. Большее из двух чисел

```
#!/bin/bash
# max.sh: Большее из двух целых чисел.

E_PARAM_ERR=250    # Если функции передано меньше 2 параметров.
EQUAL=251          # Возвращаемое значение, если оба параметра
                   #+ равны.
#  Значения ошибок из диапазона любых параметров,
#+ которые могут быть присвоены функции.

max2 ()            # Возвращает большее из двух чисел.
{                  # Примечание: сравниваемые числа должны быть
                         #+ меньше 250.
if [ -z "$2" ]
then
  return $E_PARAM_ERR
fi

if [ "$1" -eq "$2" ]
then
  return $EQUAL
else
  if [ "$1" -gt "$2" ]
  then
    return $1
  else
    return $2
  fi
fi
}

max2 33 34
```

```
return_val=$?

if [ "$return_val" -eq $E_PARAM_ERR ]
then
  echo "Необходимо передать функции два параметра."
elif [ "$return_val" -eq $EQUAL ]
  then
    echo "Оба числа равны."
else
    echo "Большее из двух чисел это $return_val."
fi


exit 0

#  Упражнение (легкое):
#  --------------
#  Конвертируйте в интерактивный сценарий, то есть, сценарий
#+ запрашивающий ввод (двух чисел).
```

Для функции, возвращающей *строку* или *массив*, используется специальная переменная.

```
подсчет_строк_в_etc_passwd()
{
  [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
  #  Если /etc/passwd читается, то REPLY устанавливается в
  #+ строку count.
  #  Возвращаются оба значения параметра и информация о
  #+ состоянии.
  #  'echo' кажется ненужным, но ...
  #+ оно удаляет избыточные пробелы из вывода.
}

if count_lines_in_etc_passwd
then
  echo "Есть строка $REPLY в /etc/passwd."
else
  echo "Строки в /etc/passwd не подсчитаны."
fi

# Спасибо, S.C.
```

Пример 24-8. Преобразование чисел в римские цифры

```
#!/bin/bash

# Преобразование арабских чисел в римские цифры
# Диапазон: 0 - 200
# Это черновик, но рабочий.

#  Расширение диапазона и иное улучшение сценария остается в качестве
#+ упражнения.
```

```
# Используем: латинская цифра-для-преобразования

LIMIT=200
E_ARG_ERR=65
E_OUT_OF_RANGE=66

if [ -z "$1" ]
then
  echo "Используем: `basename $0` число-для-преобразования"
  exit $E_ARG_ERR
fi

num=$1
if [ "$num" -gt $LIMIT ]
then
  echo "Вне диапазона!"
  exit $E_OUT_OF_RANGE
fi

to_roman ()   # Необходимо объявлять функцию до ее вызова.
{
number=$1
factor=$2
rchar=$3
let "remainder = number - factor"
while [ "$remainder" -ge 0 ]
do
  echo -n $rchar
  let "number -= factor"
  let "remainder = number - factor"
done

return $number

       # Упражнения:
       # ---------
       # 1) Объясните, как работает эта функция.
       #    Подсказка: деление с последующим вычитанием.
       # 2) Расширьте диапазон функции.
       #    Подсказка: Используйте "echo" и захват подстановки команд.
}


to_roman $num 100 C
num=$?
to_roman $num 90 LXXXX
num=$?
to_roman $num 50 L
num=$?
to_roman $num 40 XL
num=$?
to_roman $num 10 X
num=$?
to_roman $num 9 IX
num=$?
to_roman $num 5 V
num=$?
to_roman $num 4 IV
num=$?
to_roman $num 1 I
# Последовательные вызовы функции преобразования!
```

```
# Это так необходимо??? Можно упростить?

echo

exit
```

Также см. Пример 11-29.

Наибольшее положительное целое число, которое может вернуть функция - 255. Команда **return** тесно связана с понятием ***статус выхода***, на который приходится это конкретное ограничение. К счастью, существуют различные обходные пути для ситуаций, когда необходимо большее значение возвращаемого функцией числа.

Пример 24-9. Проверка наибольшего возвращаемого значения функции

```
#!/bin/bash
# return-test.sh

# Наибольшее возвращаемое положительное значение функцией - 255.

return_test ()          # Возвращение любых значений, передаваемых ей.
{
  return $1
}

return_test 27          # o.k.
echo $?                 # Возвращено 27.

return_test 255         # Опять o.k.
echo $?                 # Возвращено 255.

return_test 257         # Ошибка!
echo $?                 # Возвращено 1 (код возврата для различных
                        #+ ошибок).

# ============================================================
return_test -151896     # Работает ли большое отрицательное число?
echo $?                 # Возвратится -151896?
                        # Нет! Возвратится 168.
#  Версии Bash до 2.05b  поддерживают большие отрицательные целые числа
#+ возвращаемых значений.
#  Это было полезное свойство.
#  Новые версии Bash, к сожалению, отключили эту лазейку.
#  Они могут сбрасывать старые версии сценариев.
#  Внимание!
# ============================================================

exit 0
```

Обходным путем для получения больших целых «возвращаемых значений» является простое назначение «возвращаемого значения» глобальной переменной.

```
Return_Val=   # Глобальная переменная  сохраняет большие
              #+ возвращаемые значения функции.

alt_return_test ()
{
```

```
  fvar=$1
  Return_Val=$fvar
  return   # Возвращает 0 (успешно).
}

alt_return_test 1
echo $?                                 # 0
echo "возвращаемое значение = $Return_Val"    # 1

alt_return_test 256
echo "возвращаемое значение = $Return_Val"    # 256

alt_return_test 257
echo "возвращаемое значение = $Return_Val"    # 257

alt_return_test 25701
echo "возвращаемое значение = $Return_Val"    #25701
```

Более элегантным способом является получение 'возвращаемого значения `stdout`' функции ***echo***, а затем его захват путем подстановки команды. См. обсуждение этого в Разделе 36.7.

Пример 24-10. Сравнение двух больших целых чисел

```
#!/bin/bash
# max2.sh: Большая из двух БОЛЬШИХ целых чисел.

#  Это первичный пример "max.sh",
#+ измененный для сравнения больших целых чисел.

EQUAL=0             # Возвращаемое значение, если оба параметра равны.
E_PARAM_ERR=-99999  # Не хватает параметров переданных функции.
#           ^^^^^^    Может быть передан любой параметр Из этого
                    #+ диапазона.

max2 ()             # "Возвращение" большего из двух чисел.
{
if [ -z "$2" ]
then
  echo $E_PARAM_ERR
  return
fi

if [ "$1" -eq "$2" ]
then
  echo $EQUAL
  return
else
  if [ "$1" -gt "$2" ]
  then
    retval=$1
  else
    retval=$2
  fi
```

```
fi

echo $retval        # Выводит на экран (в stdout), вместо возвращения
                    #+ значения.
                    # Почему?
}


return_val=$(max2 33001 33997)
#            ^^^^             Имя функции
#                 ^^^^^ ^^^^^ Передаваемые параметры
#  На самом деле это форма подстановки команд:
#+ трактовка функции, как если бы она была командой,
#+ и присвоение stdout функции, переменной "return_val."


# ======================== ВЫВОД ========================
if [ "$return_val" -eq "$E_PARAM_ERR" ]
  then
  echo "Ошибка параметров, передаваемых функции сравнения!"
elif [ "$return_val" -eq "$EQUAL" ]
  then
    echo "Два числа равны."
else
    echo "Наибольшее из двух чисел это $return_val."
fi
# ============================================================

exit 0

#  Упражнения:
#  ---------
#  1) Найдите более элегантный способ проверки
#+    параметров, передаваемых функции.
#  2) Упростите структуру "OUTPUT" if/then.
#  3) Перепишите сценарий, чтобы принимать ввод параметров из командной #+
строки.
```

Вот еще один пример захвата "возвращаемого значения" функции. Понимание его требует некоторых знаний ***awk***.

```
month_length ()  # Число месяцев принимается как аргумент.
{                # Возвращает число дней в месяце.
monthD="31 28 31 30 31 30 31 31 30 31 30 31"  # Объявляется как локальная?
echo "$monthD" | awk '{ print $'"${1}"' }'    # Хитро.
#                          ^^^^^^^^^
# Функции передается параметр ($1 – номер месяца),а затем awk.
# Awk видит его как "print $1... print $12" (зависит от номера месяца)
# Шаблон для передачи параметра встроенному сценарию awk:
#                 $'"${параметр_сценария}"'

#    Вот несколько упрощенная конструкция awk:
#    echo $monthD | awk -v month=$1 '{print $(month)}'
#    Используя опцию awk -v, присваиваем значение переменной
#+   перед выполнением блока программы awk.
#    Спасибо, Rich.

#  Необходима проверка ошибок правильности диапазона параметра (1-12)
```

```
#+ и для февраля в високосный год.
}

# ---------------------------------------------
# Пример использования:
month=4        # Апрель, для примера (4-й месяц).
days_in=$(month_length $month)
echo $days_in  # 30
# ---------------------------------------------
```

Также см. Пример A-7 и Пример A-37.

Упражнение: Используя изученное, расширьте предыдущий пример латинских цифр для большего произвольного ввода.

Перенаправление

Перенаправление stdin функции

Функция, по существу, является блоком кода, а это означает, что ее `stdin` может быть перенаправлен (как в Примере 3-1).

Пример 24-11. Настоящее имя из имени пользователя

```
#!/bin/bash
# realname.sh
#
# Получение "настоящего имени" из имени пользователя в /etc/passwd.


ARGCOUNT=1       # Необходим один аргумент.
E_WRONGARGS=85

file=/etc/passwd
pattern=$1

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Используйте: `basename $0` USERNAME"
  exit $E_WRONGARGS
fi

file_excerpt ()    #  Сканируем файл по шаблону,
{                  #+ затем выводим соответствующую часть строки.
  while read line  #  В "while" [условие] не обязательно
  do
    echo "$line" | grep $1 | awk -F":" '{ print $5 }'
    # Используем разделитель awk ":".
  done
} <$file  # Перенаправляем stdin функции.

file_excerpt $pattern

# Да, весь этот сценарий может быть сокращен до
#       grep PATTERN /etc/passwd | awk -F":" '{ print $5 }'
```

```
# или
#       awk -F: '/PATTERN/ {print $5}'
# или
#       awk -F: '($1 == "username") { print $5 }' # настоящее имя из
                                                   #+ username
# Однако, они не так поучительны.

exit 0
```

Существует другой и, возможно, менее запутанный способ перенаправления `stdin` функции. Он включает в себя перенаправление `stdin` в блок встроенного кода внутри функции заключенного в скобки,.

```
# Вместо:
Функция ()
{
 ...
 } < файл

# Попробуйте:
Функция ()
{
  {
    ...
   } < файл
}

# Аналогично,

Функция ()  # Это работает.
{
  {
   echo $*
  } | tr a b
}

Function ()  # Это не работает.
{
  echo $*
} | tr a b   # Здесь блок вложенного кода является обязательным.


# Спасибо, S.C.
```

Пример `bashrc` файла Emmanuel Rouat содержит некоторые поучительные примеры функций.

Примечания

[1] Команда ***`return`*** является встроенной командой Bash.

24.2. Локальные переменные

Как переменная становится *локальной* ?

Локальные переменные

Переменная, объявленная как ***локальная***, это переменная, которая является видимой только в пределах **блока кода**, в котором она появилась. Имеет локальную (местную) сферу деятельности. В функции, *локальная переменная*, имеет смысл только в пределах блока функции. [1]

Пример 24-12. Видимость локальной переменной

```
#!/bin/bash
# ex62.sh: Глобальная и локальная переменные внутри функции.

func ()
{
  local loc_var=23       # Объявлена, как локальная переменная.
  echo                   # Используя встроенное 'local'.
  echo "\"loc_var\" внутри функции = $loc_var"
  global_var=999         # Если не объявлена, как локальная,
                         # то, по умолчанию - глобальная.
  echo "\"global_var\" внутри функции = $global_var"
}

func

# Теперь посмотрим, видна ли локальная переменная «loc_var»  за пределами
#+ функции.

echo
echo "\"loc_var\" вне функции = $loc_var"
                                     # $loc_var вне функции =
                                     # Нет, $loc_var не видна.
echo "\"global_var\" вне функции = $global_var"
                                     # $global_var вне функции = 999
                                     # $global_var видна везде.
echo

exit 0
#  В отличие от Си, переменная Bash, объявленная внутри функции, является
#+ локальной, ТОЛЬКО если объявлена как таковая.
```

До вызова функции, вне тела функции не видны ***все*** переменные, объявленные в функции, а не только те, которые объявлены явно, как ***локальные***.

```
#!/bin/bash

func ()
{
global_var=37    #  До вызова функции видима только
                 #+ в блоке функции.
```

```
}                 #  ОКОНЧАНИЕ ФУНКЦИИ

echo "global_var = $global_var"  # global_var =
                                 #  Функция "func" пока не
                                 #+ вызвана, поэтому
                                 #+ $global_var не видна.

func
echo "global_var = $global_var"  # global_var = 37
                                 # Была создана вызовом функции.
```

Как указывает Евгений Иванов, при объявлении и присваивании локальной переменной одной командой, порядок операций будет таким: *сначала* - ***ограничение ее сферы локальности*** *и, только потом,* ***присваивание*** *переменной.* Это отражается на ***возвращаемом значении***.

```
#!/bin/bash

echo "==ВНЕШНЯЯ функция (глобальная)=="
t=$(exit 1)
echo $?      # 1
             # Как и ожидалось.
echo

function0 ()
{

echo "==ВНУТРЕННЯЯ функция=="
echo "Global"
t0=$(exit 1)
echo $?      # 1
             # Как и ожидалось.

echo
echo "Объявление локальной и назначение одной команде."
local t1=$(exit 1)
echo $?      # 0
             # Неожиданно!
#  По-видимому, присваивание переменной
#+ произошло до объявления ее локальной.
#+ Выводится возвращаемое значение последнего действия.

echo
echo " Объявление локальной, а затем присвоение (отдельные
команды)."
local t2
t2=$(exit 1)
echo $?      # 1
             # Как и ожидалось.

}

function0
```

24.2.1. Локальные переменные и рекурсии.

Рекурсия - это интересная, а иногда и полезная форма *самовоспроизведения*. Herbert Mayer определяет ее как «... выражение алгоритма, использующего более простой вариант этого же алгоритма...»

Рассматриваемое определение определяется с точки зрения самого себя, [2] выражение его собственным выражением, [3] *змея глотает свой собственный хвост*, [4] или... ***функция, вызывающая сама себя***. [5]

Пример 24-13. Демонстрация простой рекурсивной функции

```
#!/bin/bash
# recursion-demo.sh
# Демонстрация рекурсии.

RECURSIONS=9   # Время рекурсии.
r_count=0      # Должна быть глобальной. Почему?

recurse ()
{
  var="$1"

  while [ "$var" -ge 0 ]
  do
    echo "Отсчет рекурсии = "$r_count"  +-+  \$var = "$var""
    (( var-- )); (( r_count++ ))
    recurse "$var"  #  Функция вызывает сама себя (рекурсия)
  done              #+ до тех пор, пока выполняется условие?
}
while[-ge]dodone
recurse $RECURSIONS

exit $?
```

Пример 24-14. Еще одна простая демонстрация

```
#!/bin/bash
# recursion-def.sh
# Сценарий, который определяет «рекурсию» графическим способом.

RECURSIONS=10
r_count=0
sp=" "

define_recursion ()
{
  ((r_count++))
```

```
  sp="$sp"" "
  echo -n "$sp"
  echo "\"The act of recurring ... \""   # Из словаря 1913 Webster's.

  while [ $r_count -le $RECURSIONS ]
  do
    define_recursion
  done
}

echo
echo "Рекурсия: "
define_recursion
echo

exit $?
```

Локальные переменные являются полезным инструментом для записи рекурсивного кода, но обычно забирают на себя значительную часть вычислительной нагрузки и определенно не рекомендуются в сценариях оболочки. [6]

Пример 24-15. Рекурсия, использующая локальную переменную

```
#!/bin/bash

#               факториал
#               ---------


# Возможны ли в bash рекурсии?
# Да, но...
# Они очень медленны.


MAX_ARG=5
E_WRONG_ARGS=85
E_RANGE_ERR=86


if [ -z "$1" ]
then
  echo "Usage: `basename $0` number"
  exit $E_WRONG_ARGS
fi

if [ "$1" -gt $MAX_ARG ]
then
  echo "Вне диапазона ($MAX_ARG это максимум)."
  #  Давайте теперь реально.
  #  Если вы хотите увеличить диапазон, то, перепишите
  #+ это на языке настоящего программирования.
  exit $E_RANGE_ERR
fi

fact ()
{
```

```
  local number=$1
  #  Переменная "number" должна быть объявлена как локальная,
  #+ иначе работать не будет.
  if [ "$number" -eq 0 ]
  then
    factorial=1    # Факториал 0 = 1.
  else
    let "decrnum = number - 1"
    fact $decrnum  # Вызов рекурсивной функции (вызывающей саму себя).
    let "factorial = $number * $?"
  fi

  return $factorial
}

fact $1
echo "Факториал $1 это $?."

exit 0
```

Также см. Пример A-15, пример рекурсии в сценарии. Имейте в виду, что рекурсия является ресурсоёмкой операцией и выполняется медленно, и поэтому, как правило, не подходит для сценариев.

Примечания

[1] Однако, как поясняет Thomas Braunberger, локальная переменная, объявленная в родительской функции, также *видна для функций, вызванных родительской функцией.*

```
#!/bin/bash

function1 ()
{
  local func1var=20

  echo "В function1, \$func1var = $func1var."

  function2
}

function2 ()
{
  echo "В function2, \$func1var = $func1var."
}

function1

exit 0


# Вывод сценария:

# В function1, $func1var = 20.
# В function2, $func1var = 20.
```

Это документировано в мануале Bash:

«*Локальность* может использоваться только в пределах функции; это делает видимым имя переменной в области, ограниченной этой функцией, и ее потомков.» [подчеркивание добавлено] *Автор ABS Guide считает такое поведение багом.*

[2] Известно, как *избыточность*.

[3] Известно, как *тафтология*.

[4] Известно, как *метафора*.

[5] Известно, как *рекурсивная функция*.

[6] Слишком много уровней рекурсии может обрушить сценарий *сегментацией*.

```
#!/bin/bash

#  Внимание: Работа этого сценария может заблокировать вашу систему!
#  Если вам повезет, произойдет сегментация всей  использующейся
#+ доступной памяти.

recursive_function ()
{
echo "$1"     #  Заставляет функцию что-то делать, и ускоряет
              #+ сегментацию.
(( $1 < $2 )) && recursive_function $(( $1 + 1 )) $2;
#  Пока 1-й параметр меньше 2-го,
#+ приращиваем 1-й и запускаем рекурсию.
}

recursive_function 1 50000  # Значение рекурсии 50,000!
#  Наиболее вероятна ошибка сегментации (зависит от размера стека,
#+ предельное значение устанавливается -m).

#  Рекурсия может привести к глубокой сегментации даже программу Си,
#+ используя всю, выделенную стеку, память.


echo "Вероятно ничего не будет выведено."
exit 0  # Сценарий нормально не завершится.
#  Спасибо, Stéphane Chazelas.
```

24.3. Рекурсия без локальных переменных

Функция может рекурсивно вызывать сама себя даже не используя локальные переменные.

Пример 24-16. ***Последовательность Фибоначчи***

```
#!/bin/bash
# fibo.sh : Последовательность Фибоначчи (рекурсивно)
# Author: M. Cooper
# License: GPL3

# ----------алгоритм--------------
# Fibo(0) = 0
# Fibo(1) = 1
# else
#   Fibo(j) = Fibo(j-1) + Fibo(j-2)
# --------------------------------

MAXTERM=15       # Количество создаваемых условий (+1).
MINIDX=2         # Если idx меньше, чем 2, то Fibo(idx) = idx.

Fibonacci ()
{
  idx=$1   # Нет необходимости в локальной переменной. Почему?
  if [ "$idx" -lt "$MINIDX" ]
  then
    echo "$idx"  # Первые два условия 0 1 ... см.выше.
  else
    (( --idx ))  # j-1
    term1=$( Fibonacci $idx )   #  Fibo(j-1)

    (( --idx ))  # j-2
    term2=$( Fibonacci $idx )   #  Fibo(j-2)

    echo $(( term1 + term2 ))
  fi
  #  Уродливый, уродливый ляп.
  #  Более элегантным осуществлением рекурсии fibo, в Си, является простой
  #+ перевод алгоритма в строках 7-10.
}

for i in $(seq 0 $MAXTERM)
do                               # Подсчет условий $MAXTERM+1.
  FIBO=$(Fibonacci $i)
  echo -n "$FIBO "
done
# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
# Занимает некоторое время, не так ли? В сценариях рекурсия очень медленна.

echo

exit 0
```

Пример 24-17. ***Ханойская пагода***

```
#! /bin/bash
#
# Ханойская пагода
# Сценарий Bash
# Copyright (C) 2000 Amit Singh. All Rights Reserved.
# http://hanoi.kernelthread.com
#
# Проверено на Bash version 2.05b.0(13)-release.
# Так же работает на Bash version 3.x.
```

```
#
#  Используется в "Advanced Bash Scripting Guide"
#+ с разрешения автора.
#  Небольшие изменения и комментарии автора ABS.

#==========================================================#
#  Ханойская пагода это математическая головоломка приписываемая Edouard
#+ Lucas, французскому математику 19-го века.
#
#  Есть три вертикальные стойки, установленные на основании.
#  Первая стойка имеет, уложенный на нее, набор дисков.
#  В центре дисков - отверстия, поэтому они могут двигаться стойкам и лежат
#+ друг на друге.
#  Диски имеют различные диаметры и сложены в порядке возрастания
#+ ( от большего диаметра внизу, до меньшего - вверху).
#
#  Задача состоит в перекладке стопки дисков на одну из стоек.
#  Можно перемещать только один диск на другую стойку.
#  Разрешается перемещать диски обратно на исходную стойку.
#  Можно размещать меньший диск на большем, но не наоборот.
#  Или, запрещено размещать больший диск на меньшем.
#
#  Для небольшого числа дисков требуется только несколько ходов.
#+ Для каждого дополнительного диска,
#+ требуемое количество ходов приблизительно удваивается,
#+ и «стратегия» становится все более сложной.
#
#  Подробнее на: http://hanoi.kernelthread.com
#+ или стр. 186-92  _The Armchair Universe_   A.K. Dewdney.
#
#
#          ...                    ...                    ...
#          | |                    | |                    | |
#         _|_|_                   | |                    | |
#        |_____|                  | |                    | |
#       |_______|                 | |                    | |
#      |_________|                | |                    | |
#     |___________|               | |                    | |
#    |_____________|              | |                    | |
#   |               |             | |                    | |
# .-------------------------------------------------------------.
# |*************************************************************|
#          #1                     #2                     #3
#
#==========================================================#


E_NOPARAM=66  # Нет параметров переданных сценарию.
E_BADPARAM=67 # Неправильное количество дисков, передаваемых сценарию.
Moves=        # Глобальная переменная, содержащая количество ходов,
              # изменяющая исходный сценарий.

dohanoi() {   # Рекурсивная функция.
    case $1 in
    0)
        ;;
    *)
        dohanoi "$(($1-1))" $2 $4 $3
        echo move $2 "-->" $3
        ((Moves++))          # Изменение исходного сценария.
        dohanoi "$(($1-1))" $4 $3 $2
        ;;
```

```
    esac
}

case $# in
    1) case $(($1>0)) in     # Нужно иметь, хотя бы, один диск.
       1)  # Вложенные инструкции case.
           dohanoi $1 1 3 2
           echo "Всего перемещено = $Moves"   # 2^n - 1, где n = № дисков.
           exit 0;
           ;;
       *)
           echo "$0: Неправильное значение количества дисков";
           exit $E_BADPARAM;
           ;;
       esac
    ;;
    *)
       echo "usage: $0 N"
       echo "       Где \"N\" это количество дисков."
       exit $E_NOPARAM;
       ;;
esac

# Упражнения:
# ---------
# 1) Будут ли когда-либо выполнены команды за этой точкой ?
#    Почему? (Легко)
# 2) Объясните работу делающуюся функцией «dohanoi».
#    (Трудно — см. ссылку на Dewdney, выше.)
```

Глава 25. Псевдонимы (*alias*)

Псевдоним (alias) Bash - это, по сути, не более чем сочетание клавиш, аббревиатура, средство для сокращенного ввода длинной последовательности команд. Если, к примеру, мы вставим ***alias lm="ls -l | more"*** в файл `~/.bashrc`, то каждый **`lm`** [1], вводимый в командной строке, будет автоматически заменяться на **`ls -l | more`**. Это помогает уменьшить ввод в командную строку и избегать необходимости запоминания сложных комбинаций команд и параметров. Установка ***alias rm="rm -i"*** (интерактивный режим удаления) помогает избежать многих неприятностей, предотвращая случайное удаление важных файлов.

В сценарии ***алиасы*** не всегда полезны. Было бы неплохо, если бы ***алиасы*** могли выполнять некоторые функции препроцессора Си, например расширение макроса, но, к сожалению, Bash не расширяет аргументы в теле ***алиаса***. [2] Кроме того, сценарию не удается в "сложных конструкциях" расширить сам ***алиас***, например в ***if/then***, циклах и функциях. Ограничение добавления заключается в том, что ***алиас*** не расширяется рекурсивно. Почти все, что мы можем делать ***алиасом***, гораздо более эффективно делает ***функция***.

Пример 25-1. Псевдонимы в сценарии

```
#!/bin/bash
# alias.sh

shopt -s expand_aliases
# Необходимо задать этот параметр, иначе сценарий не будет расширен алиасами.

# Во первых.
alias Jesse_James='echo "\"Alias Jesse James\" это была комедия в 1959 с
участием Bob Hope."'
Jesse_James

echo; echo; echo;

alias ll="ls -l"
#  Для определения алиаса можно использовать либо одиночные (')
#+ либо двойные (") кавычки .

echo "Попробуем \"ll\":"
ll /usr/X11R6/bin/mk*   #* Псевдоним работает.

echo

directory=/usr/X11R6/bin/
prefix=mk*  # См. спецсимволы вызывающие проблемы.
echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
echo

alias lll="ls -l $directory$prefix"

echo "Пробуем \"lll\":"
lll         # Большой список всех файлов в /usr/X11R6/bin начинающихся с mk.
# Алиас может обрабатывать связанные переменные содержащие спецсимволы -- o.k.

TRUE=1
```

```
echo

if [ TRUE ]
then
  alias rr="ls -l"
  echo "Пробуем \"rr\" объявленное в if/then:"
  rr /usr/X11R6/bin/mk*   #* В результате сообщение об ошибке!
  # Алиасы не расширяются в составных объявлениях.
  echo "Однако, ранее расширенный алиас по-прежнему признается:"
  ll /usr/X11R6/bin/mk*
fi

echo

count=0
while [ $count -lt 3 ]
do
  alias rrr="ls -l"
  echo "Пробуем \"rrr\" в цикле \"while\":"
  rrr /usr/X11R6/bin/mk*   #* Алиас так же не расширяется.
                           #  alias.sh: строка 57: rrr: не найдена
  let count+=1
done

echo; echo

alias xyz='cat $0'   # Листинг самого сценария.
                     # Но в строгих кавычках.
xyz
#  Как видим работает,
#+ хотя документация Bash свидетельствует о том, что не должно.
#
#  Однако, как поясняет Steve Jacobson,
#+ параметр «$0» расширяется сразу при объявлении алиаса.

exit 0
```

Команда **`unalias`** удаляет первично установленный ***alias***.

Пример 25-2. *unalias*: Установка и сброс alias

```
#!/bin/bash
# unalias.sh

shopt -s expand_aliases  # Разрешено расширение алиаса.

alias llm='ls -al | more'
llm

echo

unalias llm              # Сброс alias.
llm
# Сообщение об ошибке, поскольку «llm» больше не определяется.

exit 0
```

```
bash$ ./unalias.sh
total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.sh

./unalias.sh: llm: команда не найдена
```

Примечания

[1] ..., как первое слово из командной строки. Очевидно что ***алиас*** имеет смысл только в *начале* команды.

[2] Однако, ***алиасами***, похоже, расширяются позиционные параметры.

Глава 26. Конструкции list

Конструкции ***and list*** и ***or list*** обеспечивают метод последовательной обработки ряда команд. Они могут эффективно заменить составные вложения ***if/then*** или даже объявления ***case***.

Объединение команд в цепочку

`and list`

```
команда-1 && команда-2 && команда-3 && ... команда-n
```

Каждая команда выполняется в свою очередь, при том условии, что предыдущая команда возвращает значение ***`true`*** (ноль). При первом возвращении *`false`* (не ноль) цепочка команд завершается (первая команда возвратившая *`false`* является последней выполняемой).

Интересно использование двоякого состояния ***and list*** в более ранней версии сценария YongYe игры тетрис :

```
equation()

{  # основной алгоритм, используемый для удвоения и деления пополам
   #+ координат
   [[ ${cdx} ]] && ((y=cy+(ccy-cdy)${2}2))
   eval ${1}+=\"${x} ${y} \"
}
```

Пример 26-1. Использование *and list* для проверки аргументов командной строки

```
#!/bin/bash
# and list

if [ ! -z "$1" ] && echo "Аргументу №1 = $1" && [ ! -z "$2" ] && \
#                ^^                           ^^               ^^
echo "Аргумент №2 = $2"
then
  echo "Последние 2 аргумента переданы в сценарий."
  # Все связанные команды возвратили true.
else
  echo "В сценарий передано меньше 2 аргументов."
  # Первая из связанных команд в if возвратит false.
fi
# Обратите внимание, что "if [ ! -z $1 ]" работает, а ее
# эквивалент "if [ -n $1 ]" как будто - нет.
# Хотя, если поместить его в кавычки
#  "if [ -n "$1" ]" сработает.
#           ^  ^    Внимательнее!
# При проверке лучше помещать переменные в КАВЫЧКИ.


# Выполним то же самое, используя «чистое» объявление if/then.
```

```
if [ ! -z "$1" ]
then
  echo "Аргумент #1 = $1"
fi
if [ ! -z "$2" ]
then
  echo "Аргумент №2 = $2"
  echo "Последние 2 аргумента переданы в сценарий."
else
  echo "В сценарий передано меньше 2 аргументов."
fi
# Это длиннее и более сложнее, чем с помощью "and list".


exit $?
```

Пример 26-2. Еще одна проверка аргументов командной строки с помощью *and list*

```
#!/bin/bash

ARGS=1        # Число ожидаемых аргументов.
E_BADARGS=85  # Значение статуса выхода, если  передано неправильное число
              #+ аргументов.

test $# -ne $ARGS && \
#    ^^^^^^^^^^^^ условие №1
echo "Usage: `basename $0` $ARGS argument(s)" && exit $E_BADARGS
#                                               ^^
#  Если проверяемое условие 1 имеет значение true (неправильное
#+ количество переданных сценарию аргументов), то выполняется
#+ оставшаяся часть строки, и сценарий завершается.

#  Нижняя строка выполняется только в том случае, если проверка выше не
#+ пройдена.
echo "Сценарию передано правильное количество аргументов."

exit 0

# Проверьте значение выхода, сделав "echo $?" после завершения сценария.
```

Конечно, ***and list*** так же может присваивать переменным значение по умолчанию.

```
arg1=$@ && [ -z "$arg1" ] && arg1=DEFAULT

#  Присваивает любому $arg1 аргумент указанный командной
#+ строкой.
#  Но ... присваивает DEFAULT, если в командной строке
#+ ничего не указано.
```

or list

```
команда-1 || команда-2 || команда-3 || ... команда-n
```

Каждая команда выполняется по очереди до тех пор, пока предыдущая команда возвращает значение ***false***. Первый возвращенный результат *true* завершает цепочку команд (первая команда возвратившая *true* является последней выполненной). Это очевидная противоположность "***and list***".

Пример 26-3. Использование *or lists* в комбинации с *and list*

```
#!/bin/bash

#  delete.sh, не хитрая утилита удаления файла.
#  Usage: удаление файла

E_BADARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS  # Отсутствует аргумент? Выручает.
else
  file=$1          # Объявляем файл.
fi


[ ! -f "$file" ] && echo "Файл \"$file\" не найден. \
Трусливый отказ удалить несуществующий файл."
# AND LIST, выдаст сообщение об ошибке, если файла нет.
# Обратите внимание, что вывод echo продолжается на второй
#+ строке после пробела.

[ ! -f "$file" ] || (rm -f $file; echo "Файл \"$file\" удален.")
# OR LIST, удаляет файл, если он существует.

# Обратите внимание на инверсионную логику выше.
# AND LIST выполняется при true, OR LIST при false.

exit $?
```

Если первая команда в ***or list*** возвращает значение `true`, она *будет* выполняться.

```
# ==> Следующие фрагменты из /etc/rc.d/init.d/single,
#+==> сценария Miquel van Smoorenburg,
#+==> иллюстрирующие действие "and" и "or" lists.
# ==> "Стрелки" комментариев добавлены автором документа.

[ -x /usr/bin/clear ] && /usr/bin/clear
  # ==> Если /usr/bin/clear существует, то он вызывается.
  # ==> Проверка существования команды перед вызовом помогает избежать
  #+==> сообщений об ошибках и других неудобных последствий.
```

```
  # ==> ...

# Если нужно запустить в однопользовательском режиме, то он также
#+ запустится...
for i in /etc/rc1.d/S[0-9][0-9]* ; do
        # Проверка существования сценария.
        [ -x "$i" ] || continue
  # ==> Если соответствующий файл в $PWD *не* найден,
  #+==> то "continue" перепрыгивает на начало цикла.

        # Отбрасываем резервные файлы и файлы созданные rpm.
        case "$1" in
                *.rpmsave|*.rpmorig|*.rpmnew|*~|*.orig)
                        continue;;
        esac
        [ "$i" = "/etc/rc1.d/S00single" ] && continue
  # ==> Задаем имя сценарию, но еще не выполняем его.
        $i start
done

  # ==> …
```

Статусом выхода **`and list`** или **`or list -`** является статус выхода последней выполненной команды.

Возможны сложные «заумные» комбинации ***and*** и ***or lists***, но в их логике легко запутаться, поэтому и требуется тщательное соблюдение ***правил приоритета операторов***, а так же возможность многочисленных отладок.

```
false && true || echo false         # false

# Тот же результат
( false && true ) || echo false     # false
# Но НЕ
false && ( true || echo false )     # (ничего не выводится на экран)

#  Обратите внимание, слева направо, на группировку и оценку объявлений.

#  Но лучше всего избегать таких сложностей.

#  Спасибо, S.C.
```

См. Пример A-7 и Пример 7-4 для иллюстрации использования конструкций ***and/or list*** при проверке переменных.

Глава 27. Массивы

Более новые версии Bash поддерживают одномерные массивы. Элементы массива могут быть инициализированы формой записи переменной **[xx]**. Кроме того, сценарий может ввести весь массив, явно объявляя его **declare -a переменная**. Для разыменования (извлечения содержимого) элемента массива, используются фигурные скобки, то есть **${элемент[xx]}**.

Пример 27-1. Простое использование массива

```
#!/bin/bash


area[11]=23
area[13]=37
area[51]=UFOs

#  Составляющие массива не обязательно должны быть последовательными или
#+ непрерывными.

#  Некоторые составляющие массива могут быть не инициализированными.
#  Пробелы в массиве в порядке вещей.
#  На самом деле, массивы с разряженными данными ('разряжённые массивы')
#+ полезны в программном обеспечении для обработки электронных таблиц.


echo -n "area[11] = "
echo ${area[11]}    #  {фигурные скобки} необходимы.

echo -n "area[13] = "
echo ${area[13]}

echo "Содержимое[51] это ${area[51]}."

#  Переменная не инициализированного массива содержащая пустой вывод (null
#+ переменная).
echo -n "area[43] = "
echo ${area[43]}
echo "(area[43] не объявлена)"

echo

# Сумма двух переменных массивов присваивается третьей переменной массива
```

```
area[5]=`expr ${area[11]} + ${area[13]}`
echo "area[5] = area[11] + area[13]"
echo -n "area[5] = "
echo ${area[5]}

area[6]=`expr ${area[11]} + ${area[51]}`
echo "area[6] = area[11] + area[51]"
echo -n "area[6] = "
echo ${area[6]}
# Не получится, потому что добавление целого числа к строке не допускается.

echo; echo; echo

# --------------------------------------------------------------
# Другой массив, "area2".
# Другой способ присвоения массиву переменных...
# array_name=( XXX YYY ZZZ ... )

area2=( ноль один два три четыре )

echo -n "area2[0] = "
echo ${area2[0]}
# Да, индексация с нуля (первый элемент массива [0], а не [1]).

echo -n "area2[1] = "
echo ${area2[1]}    # [1] это второй элемент массива.
# --------------------------------------------------------------

echo; echo; echo

# ------------------------------------------------
# Опять другой массив, "area3".
# И другой способ присвоения массива переменных...
# array_name=([xx]=XXX [yy]=YYY ...)

area3=([17]=семнадцать [24]=двадцать_четыре)

echo -n "area3[17] = "
echo ${area3[17]}

echo -n "area3[24] = "
echo ${area3[24]}
# ------------------------------------------------

exit 0
```

Как видим, самый удобный способ инициализации всего массива - это запись **`array=( элемент1 элемент2 ... элементN )`**.

```
base64_charset=( {A..Z} {a..z} {0..9} + / = )
               #  Для инициализации элементов массива
               #+ используются круглые скобки расширения.
               #  Выдержка из сценария vladz "base64.sh"
               #+ в приложении"Внесенные сценарии".
```

Bash допускает операции над массивами переменных, даже если переменные не объявлены

явно, как массивы.

```
string=abcABC123ABCabc
echo ${string[@]}               # abcABC123ABCabc
echo ${string[*]}               # abcABC123ABCabc
echo ${string[0]}               # abcABC123ABCabc
echo ${string[1]}               # Не выводится!
                                # Почему?
echo ${#string[@]}              # 1
                                # В массиве один элемент.
                                # Сама строка.

# За разъяснение спасибо Michael Zick.
```

Еще раз убедились, что переменные Bash ***не имеют типа***.

Пример 27-2. Форматирование стихотворения

```
#!/bin/bash
# poem.sh: Красивые строки одного из любимых стихотворений автора ABS Guide.

# Строки поэмы (по одной строфе).
Line[1]="I do not know which to prefer,"
Line[2]="The beauty of inflections"
Line[3]="Or the beauty of innuendoes,"
Line[4]="The blackbird whistling"
Line[5]="Or just after."
# Обратите внимание, что кавычки дают возможность вставлять пробелы.

# Аттрибуты.
Attrib[1]=" Wallace Stevens"
Attrib[2]="\"Thirteen Ways of Looking at a Blackbird\""
# Эта поэма в публичном доступе (копирайт истек).

echo

tput bold   # Вывод шрифтом Bold.

for index in 1 2 3 4 5    # Пять строк.
do
  printf "     %s\n" "${Line[index]}"
done

for index in 1 2          # Две строки аттрибутов.
do
  printf "     %s\n" "${Attrib[index]}"
done

tput sgr0   # Сброс терминала.
            # См. документацию 'tput'.

echo

exit 0

# Упражнение:
# --------
```

```
# Измените этот сценарий для строк стихотворения из текстового файла данных.
```

Переменные массивов имеют собственный синтаксис, и даже стандартные команды Bash и операторы имеют специальные опции, адаптированные для работы с массивом.

Пример 27-3. Различные операции с массивами

```
#!/bin/bash
# array-ops.sh: Развлечение с массивами.


array=( ноль один два три четыре пять )
#Элемент 0    1    2   3    4      5

echo ${array[0]}       #  ноль
echo ${array:0}        #  ноль
                       #  Параметр расширения первого элемента,
                       #+ начальная позиция #0 (1й символ).
echo ${array:1}        #  ноль
                       #  Параметр расширения первого элемента,
                       #+ начинающегося с позиции #1 (2й символ).

echo "--------------"

echo ${#array[0]}      #  4
                       #  Величина (число букв) первого элемента массива.
echo ${#array}         #  4
                       #  Величина первого элемента массива.
                       #  (Альтернативный способ записи)

echo ${#array[1]}      #  4
                       #  Величина второго элемента массива.
                       #  Индексация массивов в Bash начинается с нуля.

echo ${#array[*]}      #  6
                       #  Число элементов в массиве.
echo ${#array[@]}      #  6
                       #  Число элементов в массиве.

echo "--------------"

array2=( [0]="первый элемент" [1]="второй элемент" [3]="четвертый элемент" )
#           ^      ^       ^     ^      ^       ^     ^          ^       ^
# Кавычки допускают пробелы в пределах индивидуальных элементов массива.

echo ${array2[0]}      # первый элемент
echo ${array2[1]}      # второй элемент
echo ${array2[2]}      #
                       # Пропущена инициализация, и поэтому null.
echo ${array2[3]}      # четвертый элемент
echo ${#array2[0]}     # 14    (величина (с пробелом) первого элемента)
echo ${#array2[*]}     # 3     (число (с пробелом) элементов в массиве)

exit
```

Многие из стандартных ***операций string*** работают с массивами.

Пример 27-4. Операции string массива

```
#!/bin/bash
# array-strops.sh: string операции массива.

# Сценарий Michael Zick.
# Используется в ABS Guide с разрешения.
# Исправления: 05 May 08, 04 Aug 08.

#  В основном, строковые операции используют запись ${название ... },
#+ применяющуюся для всех элементов строки в массиве,
#+ а так же нотацию ${название[@] ... } или ${название[*] ...}.


arrayZ=( one two three four five five )

echo

# Извлечение содержимого строки до конца
echo ${arrayZ[@]:0}     # one two three four five five
#                ^        Все элементы.


echo ${arrayZ[@]:1}     # two three four five five
#                ^        Все элементы после элемента [0].

echo ${arrayZ[@]:1:2}   # two three
#                  ^      Только два элемента после элемента [0].

echo "---------"


# Удаление содержимого строки

# Удаление коротких соответствий от передней части строк(и).

echo ${arrayZ[@]#f*r}   # one two three  five five
#                ^       # Применяется ко всем элементам массива.
                         # Соответствует "four" и удаляется.

# Длинных соответствий от передней части строк(и)
echo ${arrayZ[@]##t*e}  # one two four  five five
#                ^^      # Применяется ко всем элементам массива.
                         # Соответствует "three" и удаляется.

# Коротких соответствий от конца строк(и)
echo ${arrayZ[@]%h*e}   # one two four  five five
#                ^       # Применяется ко всем элементам массива.
                         # Соответствует "three" и удаляется.

# Длинных соответствий от конца строк(и)
echo ${arrayZ[@]%%t*e}  # one two four  five five
#                ^^      # Применяется ко всем элементам массива.
                         # Соответствует "three" и удаляется.

echo "----------------------"


# Замена содержимого строки
```

```
# Замена первого вхождения содержимого строки с заменой.
echo ${arrayZ[@]/fiv/XYZ}   # one two three four XYZe XYZe
#               ^           # Применяется ко всем элементам массива.

# Замена всех вхождений содержимого строки.
echo ${arrayZ[@]//iv/YY}    # one two three four fYYe fYYe
                            # Применяется ко всем элементам массива.

# Удаление всех вхождений содержимого строки.
# По умолчанию не задается замена для 'delete' ...
echo ${arrayZ[@]//fi/}      # one two three four ve ve
#               ^^          # Применяется ко всем элементам массива.

# Замена переднего плана вхождения содержимого строки.
echo ${arrayZ[@]/#fi/XY}    # one two three four XYve XYve
#               ^           # Применяется ко всем элементам массива.

# Замена заднего плана вхождения содержимого строки.
echo ${arrayZ[@]/%ve/ZZ}    # one two three four fiZZ fiZZ
#               ^           # Применяется ко всем элементам массива.

echo ${arrayZ[@]/%o/XX}     # one twXX three four five five
#               ^           # Почему?

echo "-----------------------------"


replacement() {
    echo -n "!!!"
}

echo ${arrayZ[@]/%e/$(replacement)}
#               ^  ^^^^^^^^^^^^^^
# on!!! two thre!!! four fiv!!! fiv!!!
# Вывод replacement() это строка замены.
# Q.E.D: Действие замены является, по сути, «назначением».

echo "-----------------------------------"

#  Доступ к "for-each":
echo ${arrayZ[@]//*/$(replacement необязательные_аргументы)}
#               ^^ ^^^^^^^^^^^^^^
# !!! !!! !!! !!! !!! !!!

#  Теперь, если Bash будет передавать только совпадающую строку
#+ в вызываемую функцию...

echo

exit 0

# Прежде чем попасть под большой молоток - Perl, Python, или других ЯП -
# сравни:
#    $( ... ) это подстановка команды.
#    Функция запускается, как подпроцесс (потомок).
#    Функция записывает свой вывод (если выводится на экран) в stdout.
#    Объявление, в сочетании с «echo» и подстановкой команды,
#+   может читать stdout функции.
#    Запись имени[@] указывает (эквивалент) на операцию "for-each".
```

```
#    Bash более мощный, чем ты думал!
```

Подстановка команд может создавать отдельные элементы массива.

Пример 27-5. Загрузка содержимого сценария в массив

```
#!/bin/bash
# script-array.sh: Загрузка этого сценария в массив.
# Вдохновлен e-mail от Chris Martin (спасибо!).

script_contents=( $(cat "$0") )  #  Сохранение содержимого этого сценария
                                 #+ ($0) в массив.

for element in $(seq 0 $((${#script_contents[@]} - 1)))
  do                #  ${#script_contents[@]}
                    #+ задает число элементов в массиве.
                    #
                    #  Вопрос:
                    #  Почему необходимо равенство 0?
                    #  Попробуйте изменить на 1.
  echo -n "${script_contents[$element]}"
                    # Листинг каждого поля сценария построчно.
# echo -n "${script_contents[element]}" то же работает, т.к. ${ ... }.
  echo -n " -- "    # Используется " -- " как разделитель полей.
done

echo

exit 0

# Упражнение:
# --------
#  Измените этот сценарий, что бы его листинг был в первозданном виде,
#+ вместе с пробелами, разрывами строк и т.д.
```

В контексте массива, некоторые встроенные команды Bash имеют несколько измененное значение. Например, ***unset*** удаляет элементы массива, или даже весь массив.

Пример 27-6. Некоторые специальные свойства массивов

```
#!/bin/bash

declare -a colors
#  Все последующие команды в этом сценарии будут рассматриваться как массив  #+
переменной «colors».

echo "Введите ваши любимые цвета (разделяя их пропуском)."

read -a colors    #  Введите, по крайней мере, 3 цвета (для демонстрации
                  #+ возможностей ниже).
#  Специальная опция команды 'read',
#+ позволяет назначать элементы в массив.

echo

element_count=${#colors[@]}
```

```
# Специальный синтаксис для извлечения количества элементов массива.
#     element_count=${#colors[*]} то же работает.
#
#  Переменная "@"  позволяет разделять слово в кавычках
#+ (извлекает переменные, разделенные пробелами).
#
#  Что соответствует поведению "$@" и "$*"
#+ в позиционных параметрах.

index=0

while [ "$index" -lt "$element_count" ]
do    # Список всех элементов массива.
  echo ${colors[$index]}
  #    ${colors[index]} то же работает, потому что ${ ... } в скобках.
  let "index = $index + 1"
  # Или:
  #    ((index++))
done
# Каждый элемент массива перечисляется на отдельной строке.
# Если это не требуется, используйте  echo -n "${colors[$index]}"
#
# Сделаем это с помощью цикла "for":
#   for i in "${colors[@]}"
#   do
#     echo "$i"
#   done
# (Спасибо, S.C.)

echo

# Снова, список всех элементов массива, но используя более изящный метод.
  echo ${colors[@]}          # echo ${colors[*]} то же работает.

echo

# Команда "unset" удаляет элементы массива, или сам массив.
unset colors[1]              # Удаляем 2й элемент массива.
                             # Эффект подобен   colors[1]=
echo  ${colors[@]}           # Снова список массива, отсутствует 2й элемент.

unset colors                 # Удаляем весь массив.
                             #  unset colors[*] и
                             #+ unset colors[@] то же работают.
echo; echo -n "Colors gone."
echo ${colors[@]}            # Снова список массива, теперь пустой.

exit 0
```

Как видно в предыдущем примере, **${array_name[@]}** или **${array_name[*]}** ссылаются на ***все*** элементы массива. Аналогично, для получения количества элементов в массиве, используется либо **${#array_name[@]}** либо **${#array_name[*]}**. **${#array_name}** это размер (количество символов) **${array_name[0]}**, первого элемента массива.

Пример 27-7. Пустые массивы и пустые элементы

```
#!/bin/bash
# empty-array.sh

#  Спасибо за этот пример Stephane Chazelas,
#+ и Michael Zick и Omair Eshkenazi, за его расширение.
#  A Nathan Coulter за уточнения и исправления.


# Пустой массив это не то же самое, что массив из пустых элементов.

  array0=( first second third )
  array1=( '' )   # "array1" содержит один пустой элемент.
  array2=( )      # Нет элементов ... "array2" пустой.
  array3=(   )    # А это что за массив?


echo
ListArray()
{
echo
echo "Элементы в array0:  ${array0[@]}"
echo "Элементы в array1:  ${array1[@]}"
echo "Элементы в array2:  ${array2[@]}"
echo "Элементы в array3:  ${array3[@]}"
echo
echo "Размер первого элемента в array0 = ${#array0}"
echo "Размер первого элемента в array1 = ${#array1}"
echo "Размер первого элемента в array2 = ${#array2}"
echo "Размер первого элемента в array3 = ${#array3}"
echo
echo "Количество элементов в array0 = ${#array0[*]}"  # 3
echo "Количество элементов в array1 = ${#array1[*]}"  # 1  (Сюрприз!)
echo "Количество элементов в array2 = ${#array2[*]}"  # 0
echo "Количество элементов в array3 = ${#array3[*]}"  # 0
}

# ==============================================================

ListArray

# Попробуем расширить эти массивы.

# Добавление элемента в массив.
array0=( "${array0[@]}" "новый1" )
array1=( "${array1[@]}" "новый1" )
array2=( "${array2[@]}" "новый1" )
array3=( "${array3[@]}" "новый1" )

ListArray

# или
array0[${#array0[*]}]="новый2"
array1[${#array1[*]}]="новый2"
array2[${#array2[*]}]="новый2"
array3[${#array3[*]}]="новый2"

ListArray

# При расширении, как показано выше, массивы становятся «стеками» ...
# Выше - это 'push' ...
# 'Вершиной' стека  является:
```

```
height=${#array2[@]}
echo
echo "Вершина стека для array2 = $height"

# 'pop' это:
unset array2[${#array2[@]}-1]   #  Массивы начинаются с нуля,
height=${#array2[@]}            #+ Это значит, что первый элемент имеет
                                #+ индекс 0.
echo
echo "POP"
echo "Новая вершина стека для array2 = $height"

ListArray

# Список только 2-го и 3-его элементов array0.
from=1              # Отсчет начинается с нуля.
to=2
array3=( ${array0[@]:1:2} )
echo
echo "Элементы в array3:  ${array3[@]}"

# Работает как строка (массив символов).
# Попробуем другую форму "строки".

# Изменим:
array4=( ${array0[@]/second/2nd} )
echo
echo "Элементы в array4:  ${array4[@]}"

# Заменим все строки совпадающие с подстанавливаемыми символами.
array5=( ${array0[@]//new?/old} )
echo
echo "Элементы в array5:  ${array5[@]}"

# Просто, когда вы прочувствуете это ...
array6=( ${array0[@]#*new} )
echo # Этот может вас удивить.
echo "Элементы в array6:  ${array6[@]}"

array7=( ${array0[@]#new1} )
echo # После array6 это не удивляет.
echo "Элементы в array7:  ${array7[@]}"

# Выглядит огромным, как ...
array8=( ${array0[@]/new1/} )
echo
echo "Элементы в array8:  ${array8[@]}"

#  Что можно сказать об этом?

#  Строковые операции в var[@] выполняются с каждым элементом подряд.
#  Отсюда: Bash поддерживает вектор строковых  операций.
#  Если результатом является строка нулевой длины,
#+ то в результате присвоения элемент исчезает.
#  Однако, если расширение находится в кавычках, нулевые элементы остаются.

#  Michael Zick: Вопрос, а если эти строки находятся в жестких или мягких
#+ кавычках?
#  Nathan Coulter: Здесь нет такого понятия, как "мягкие кавычки."
#!    Что происходит на самом деле: это происходит настройка
#!+   шаблона после всех других расширений [word], в данном случае
```

```
#!+   как ${ parameter#word}.


zap='new*'
array9=( ${array0[@]/$zap/} )
echo
echo "Количество элементов в array9:  ${#array9[@]}"
array9=( "${array0[@]/$zap/}" )
echo "Элементы в array9:  ${array9[@]}"
# На этот раз нулевые элементы остались.
echo "Количество элементов в array9:  ${#array9[@]}"


# Просто, когда вы думали, вы были все еще в Канзасе ...
array10=( ${array0[@]#$zap} )
echo
echo "Элементы в array10:  ${array10[@]}"
# Но, звездочка zap, в кавычках, не будет интерпретироваться.
array10=( ${array0[@]#"$zap"} )
echo
echo "Элементы в array10:  ${array10[@]}"
# Хорошо, может быть мы_все_еще в Канзасе ...
# (Изменения в блоке кода выше Nathan Coulter.)


#  Сравните array7 с array10.
#  Сравните array8 с array9.

#  Подтверждение: Нет такой вещи, как мягкие кавычки!
#  Nathan Coulter объясняет:
#  Настройка шаблона 'word' в ${parameter#word} происходит после
#+ расширения параметра и *до* удаления кавычек.
#  В обычном случае, настройка шаблона происходит *после* удаления кавычек.

exit
```

Отношение **${array_name[@]}** и **${array_name[*]}** аналогично, отношению между **$@** и **$***. Эта мощная нотация массива имеет ряд применений.

```
# Копирование массива.
array2=( "${array1[@]}" )
# или
array2="${array1[@]}"
#
#  Тем не менее, как поясняет Jochen DeSmet, это не получится с
#+ «разреженными» массивами, массивами
#+ с дырами (недостающими элементами).
# ----------------------------------------
  array1[0]=0
# array1[1] не назначен
  array1[2]=2
  array2=( "${array1[@]}" )       # Скопирует его?

echo ${array2[0]}      # 0
echo ${array2[2]}      # (null), должно быть 2
# ----------------------------------------
```

```
# Добавление элемента в массив.
array=( "${array[@]}" "new element" )
# или
array[${#array[*]}]="new element"

# Спасибо, S.C.
```

Операция инициализации **array=(элемент1 элемент2 ... элементN)**, использующая подстановку команд, дает возможность загрузить содержимое текстового файла в массив.

```
#!/bin/bash

filename=sample_file

#            cat sample_file
#
#            1 a b c
#            2 d e fg


declare -a array1

array1=( `cat "$filename"` )               #  Загрузка содержимого
                                           #+ $filename в array1.
# Содержимое файла в stdout
#  array1=( `cat "$filename" | tr '\n' ' '` )
#                 символы перевода строки в файле заменяются на пробелы.
#  Изменять перевод строки на пробелы не обязательно, потому
#+ что Bash разделяет слово.

echo ${array1[@]}            # Содержимое массива.
#                              1 a b c 2 d e fg
#
#  Каждое разделенное пробелами 'слово' в файле
#+ было назначено, как элемент массива.

element_count=${#array1[*]}
echo $element_count          # 8
```

Умный сценарий позволяет добавлять операции с массивами.

Пример 27-8. Инициализация массивов

```
#! /bin/bash
# array-assign.bash

#  Операции с массивами характерными для Bash, и, следовательно, имеющими
#+ '.bash 'в имени сценария.
# Copyright (c) Michael S. Zick, 2003, All rights reserved.
# License: Unrestricted reuse in any form, for any purpose.
```

```
# Version: $ID$
#
# Разъяснения и дополнительные комментарии William Park.

#  На основании примера, предоставленного Stephane Chazelas,
#+ который появился в более ранней версии
#+ Advanced Bash Scripting Guide.

#  Вывод формата команды 'times':
#  Процесс Пользователя <пробел> Системный Процесс
#  Мертвый потомок Пользовательского Процесса <пробел> мертвые потомки
#+ Системного Процесса

#  Bash имеет два способа присвоения всех элементов массива
#+ новой переменной массива.
#  Оба опускают элементы 'пустой ссылки'
#+ в версии Bash 2.04 и поздних.
#  Дополнительные назначения массива, который сохраняет отношения
#+ [подсценарий]=значение для массива, могут добавляться в более новых
#+ версиях.

#  Создается большой массив с помощью внутренней команды,
#+ и массив из нескольких тысяч элементов будет прекрасно создан.

declare -a bigOne=( /dev/* )  # Все файлы в /dev ...
echo
echo 'Условие: Без кавычек, IFS по умолчанию, Включены все элементы'
echo "Число элементов массива - ${#bigOne[@]}"

# задаем -vx



echo
echo '- - testing: =( ${array[@]} ) - -'
times
declare -a bigTwo=( ${bigOne[@]} )
# Внимание:       ^              ^   круглые скобки
times


echo
echo '- - testing: =${array[@]} - -'
times
declare -a bigThree=${bigOne[@]}
# Теперь без круглых скобок.
times

#  Сравнение показывает, что вторая форма, по
#+ мнению Stephane Chazelas, быстрее.
#
#  Как объяснил William Park:
#+ Массив bigTwo назначает элементы по очереди (круглые скобки!),
#+ тогда как bigThree назначает как единую строку.
#  Так, по сути, имеем:
#                   bigTwo=( [0]="..." [1]="..." [2]="..." ... )
#                   bigThree=( [0]="... ... ..." )
#
#  Убедитесь в этом:  echo ${bigTwo[0]}
#                     echo ${bigThree[0]}
```

```
#  Я буду продолжать использовать первую форму в моем примере, потому
#+ что мне кажется, что это лучше иллюстрирует то, что происходит.

#  Во многих частях моих примеров будет содержаться вторая форма,
#+ соответственно из-за скорости.


#  Обратите внимание:
#  ----
#  Объявления "declare -a" в строках 32 и 44
#+ не являются строго необходимыми, так как это
#+ не явная форма объявления в Array=( ... ).
#  Однако, устранение этих объявлений замедляет выполнение
#+ следующих разделов сценария.
#  Попробуй и увидишь.

exit 0
```

Добавление лишнего объявления **`declare -a`** в массив может ускорить выполнение последующих операций в массиве.

Пример 27-9. Копирование и объединение массивов

```
#! /bin/bash
# CopyArray.sh
#
# Сценарий написан Michael Zick.
# Используется с разрешения.

#  How-To "Pass by Name & Return by Name"
#+ или "Создание собственного оператора присваивания".


CpArray_Mac() {

# Создание команды присваивания значения

    echo -n 'eval '
    echo -n "$2"                    # Назначаемое имя
    echo -n '=( ${'
    echo -n "$1"                    # Исходное имя
    echo -n '[@]} )'

# Это все можно сделать одной командой.
# Вопрос  стиля.
}

declare -f CopyArray                # Функция "Указатель"
CopyArray=CpArray_Mac               # Создатель значения

Hype()
{

# Hype — имя массива $1.
```

```
# (Объединяется вместе с массивом, содержащим "Really Rocks".)
# Возвращается в массив с именем $2.

    local -a TMP
    local -a hype=( Really Rocks )

    $($CopyArray $1 TMP)
    TMP=( ${TMP[@]} ${hype[@]} )
    $($CopyArray TMP $2)
}

declare -a before=( Advanced Bash Scripting )
declare -a after

echo "Array Before = ${before[@]}"

Hype before after

echo "Array After = ${after[@]}"

# Слишком много hype?

echo "Какое ${after[@]:3:2}?"

declare -a modest=( ${after[@]:2:1} ${after[@]:3:2} )
#                    ---- извлечение substring ----

echo "Массив Modest = ${modest[@]}"

# Что случилось с 'before' ?

echo "Массив Before = ${before[@]}"

exit 0
```

Пример 27-10. Подробнее об объединении массивов

```
#! /bin/bash
# array-append.bash

# Copyright (c) Michael S. Zick, 2003, All rights reserved.
# License: Unrestricted reuse in any form, for any purpose.
# Version: $ID$
#
# Слегка изменен М.С.


# Операции с массивами являющиеся специфичными для Bash.
# Устаревший эквивалент UNIX /bin/sh.


#  Передавайте вывод этого сценария в 'more'
#+ так как в терминале отключен скролл.
#  Или перенаправьте вывод в файл.


declare -a array1=( zero1 one1 two1 )
# Составляющие упакованы.
```

```
declare -a array2=( [0]=zero2 [2]=two2 [3]=three2 )
# Разреженная составляющая -- [1] не определена.

echo
echo '-Убедитесь, что массив имеет действительно разреженные составляющие.-'
echo "Количество элементов: 4"        # Жесткий код для иллюстрации.
for (( i = 0 ; i < 4 ; i++ ))
do
    echo "Элемент [$i]: ${array2[$i]}"
done
# См.  более общий пример кода в basics-reviewed.bash.


declare -a dest

# Комбинируем (объединяем) два массива в третий массив.
echo
echo 'Условия: Без кавычек, IFS по умолчанию, Включены все операторы элементов'
echo '- Нет не определенных элементов, составляющие не поддерживаются. -'
# Не определенных элементов нет; они не потеряны.

dest=( ${array1[@]} ${array2[@]} )
# dest=${array1[@]}${array2[@]}     # Странные результаты, возможно баг.

# Теперь, список результатов.
echo
echo '- - Сравнение добавленных массивов - -'
cnt=${#dest[@]}

echo "Число элементов: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Элемент [$i]: ${dest[$i]}"
done

# Присваиваем  массив  одному элементу массива (дважды).
dest[0]=${array1[@]}
dest[1]=${array2[@]}

# Список результатов.
echo
echo '- - Проверка измененного массива - -'
cnt=${#dest[@]}

echo "Число элементов: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Элемент [$i]: ${dest[$i]}"
done

# Проверяем изменение второго элемента.
echo
echo '- - Переназначение и список второго элемента - -'

declare -a subArray=${dest[1]}
cnt=${#subArray[@]}

echo "Число элементов: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Элемент [$i]: ${subArray[$i]}"
```

```
done

#  Назначение всего массива в элемент другого массива с помощью назначения
массива '${...}' переводит присваиваемый массив в строку, с элементами
разделяемыми пробелом (первый символ IFS).

# Если исходные элементы не содержат пробелы ...
# Если исходный массив не имеет разреженных составляющих ...
# То мы можем снова получить структуру исходного массива.

# Восстановление из измененного второго элемента.
echo
echo '- - Список восстановленных элементов - -'

declare -a subArray=( ${dest[1]} )
cnt=${#subArray[@]}

echo "Число элементов: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Элемент [$i]: ${subArray[$i]}"
done
echo '- - Поведение не зависит. - -'
echo '- - Это поведение меняется в - -'
echo '- - версиях Bash новее, чем 2.05b - -'


exit 0
```

--

Массивы позволяют развертывать старые знакомые алгоритмы в виде сценариев оболочки. Является ли это хорошей идеей - решать читателям.

Пример 27-11. Сортировка пузырей

```
#!/bin/bash
# bubble.sh: Сортировка пузырей sorts.

# Вспомним алгоритм сортировки пузырьков. В данном конкретном варианте...

#  С каждым последующим проходом по массиву, чтобы отсортировать, сравниваем
#  два соседних элемента, и меняем их, если требуется.
#  В конце первого прохода, 'тяжелый' элемент опустился на дно.
#  В конце второго прохода, следующий 'тяжелый'  затонул рядом с нижним.
#  И так далее.
#  Это означает, что каждый последующий проход должен уменьшать массив.
#  Поэтому вы заметите ускорении в выводе более поздних проходов.


exchange()
{
  # Замена двух элементов массива.
  local temp=${Countries[$1]} #  Временное хранилище
                              #+ для элемента, после замены.
  Countries[$1]=${Countries[$2]}
  Countries[$2]=$temp

  return
}
```

```
declare -a Countries  #  Объявление массива,здесь не обязательно,
                      #+ поскольку он инициализируется ниже.

#  Допустимо разделить массив переменных на несколько строк
#+ с помощью обратного слэша (\)?
#  Да.

Countries=(Netherlands Ukraine Zaire Turkey Russia Yemen Syria \
Brazil Argentina Nicaragua Japan Mexico Venezuela Greece England \
Israel Peru Canada Oman Denmark Wales France Kenya \
Xanadu Qatar Liechtenstein Hungary)

# "Xanadu" это мифическое место где, по словам Coleridge,
#+ Kubla Khan дал указ воздвигнуть шатер наслаждения.


clear                     # Очищаем экран.

echo "0: ${Countries[*]}"  # Список всего массива передаваемого 0.

number_of_elements=${#Countries[@]}
let "comparisons = $number_of_elements - 1"

count=1 # Переданное число.

while [ "$comparisons" -gt 0 ]          # Начало внешнего цикла
do

  index=0  # Сбрасываем индекс при начале массива после каждого прохода.

  while [ "$index" -lt "$comparisons" ] # Начало внутреннего цикла
  do
    if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`]} ]
    #  Если не получилось...
    #  Напомню, что \> в одинарных скобках является
    #+ оператором сравнения ASCII.

    #  if [[ ${Countries[$index]} > ${Countries[`expr $index + 1`]} ]]
    #+ то же работает.
    then
      exchange $index `expr $index + 1`  # Замена.
    fi
    let "index += 1"  # Или,   index+=1   в Bash, вер. 3.1 или новее.
  done # Конец внутреннего цикла

# ----------------------------------------------------------------------
# Paulo Marcel Coelho Aragao предложил циклы  for, как более простую
альтернативу.
#
# for (( last = $number_of_elements - 1 ; last > 0 ; last-- ))
##                   Исправил C.Y. Hunt          ^   (Спасибо!)
# do
#     for (( i = 0 ; i < last ; i++ ))
#     do
#         [[ "${Countries[$i]}" > "${Countries[$((i+1))]}" ]] \
#             && exchange $i $((i+1))
#     done
# done
# ----------------------------------------------------------------------
```

```
let "comparisons -= 1" #  Как только "тяжелые" элементы пузырьков опускаются,
                       #+ в каждом проходе нужно делать на одно сравнение
                       #+ меньше.

echo
echo "$count: ${Countries[@]}"  #  Вывод полученного массива в конце каждого
                                #+ прохода.
echo
let "count += 1"                # Увеличиваем отсчет передач.

done                            # Конец внешнего цикла
                                # Все сделано.

exit 0
```

--

Возможны ли в массивах вложенные массивы?

```
#!/bin/bash
# "Вложенный" массив.

#  Этот пример предоставлен Michael Zick,
#+ исправлен и уточнен William Park.

AnArray=( $(ls --inode --ignore-backups --almost-all \
        --directory --full-time --color=none --time=status \
        --sort=time -l ${PWD} ) )  # Команды и опции.

# Объем значительный ... и не в кавычках.

SubArray=( ${AnArray[@]:11:1}  ${AnArray[@]:6:5} )
#  Этот массив имеет 6 элементов:
#+     SubArray=( [0]=${AnArray[11]} [1]=${AnArray[6]} [2]=${AnArray[7]}
#      [3]=${AnArray[8]} [4]=${AnArray[9]} [5]=${AnArray[10]} )
#
#  Массивы в Bash это (по кругу) списки связанных типов
#+ строк (char *).
#  Так что это, фактически, не является вложенным массивом,
#+ но по функциональности - аналогичен.

echo "Текущая директория и дата последнего изменения состояния:  "
echo "${SubArray[@]}"

exit 0
```

--

Встроенные массивы в сочетании с косвенными ссылками создают увлекательные возможности

Пример 27-12. Встроенные массивы и косвенные ссылки

```
#!/bin/bash
# embedded-arrays.sh
# Встроенные массивы и косвенные ссылки.
```

```
# Сценарий Dennis Leeuw.
# Используется с разрешения.
# Изменен автором документа.


ARRAY1=(
        VAR1_1=value11
        VAR1_2=value12
        VAR1_3=value13
)

ARRAY2=(
        VARIABLE="test"
        STRING="VAR1=value1 VAR2=value2 VAR3=value3"
        ARRAY21=${ARRAY1[*]}
)       # Встроили ARRAY1 во второй массив.

function print () {
        OLD_IFS="$IFS"
        IFS=$'\n'       #  Вывод каждого элемента массива
                        #+ на отдельной строке.
        TEST1="ARRAY2[*]"
        local ${!TEST1} # Смотрим, что случится, если мы удалим эту строку.
        #  Косвенная ссылка.
        #  Она делает компоненты $TEST1
        #+ доступными данной функции.


        #  Давайте посмотрим, что мы пока имеем.
        echo
        echo "\$TEST1 = $TEST1"       #  Только имя переменной.
        echo; echo
        echo "{\$TEST1} = ${!TEST1}"  #  Содержимое переменной.
                                      #  Это действие косвенной ссылки.
        echo
        echo "-----------------------------------------"; echo
        echo


        # Вывод переменной
        echo "Variable VARIABLE: $VARIABLE"

        # Вывод элементов строки
        IFS="$OLD_IFS"
        TEST2="STRING[*]"
        local ${!TEST2}      # косвенная ссылка (как и выше).
        echo "Элемент строки VAR2: $VAR2 из STRING"

        # Вывод элементов массива
        TEST2="ARRAY21[*]"
        local ${!TEST2}      # Косвенная ссылка (как и выше).
        echo "Элемент массива VAR1_1: $VAR1_1 из ARRAY21"
}

print
echo

exit 0

#  Как поясняет автор:
#+ "Вы легко можете расширить массив для создания хэшей имен в bash."
```

```
#  (Трудное) упражнение для читателей: реализуйте это.
```

--

Массивы позволяют реализовывать различные задачи сценарием оболочки *Сито Эратосфена*. Конечно, ресурсоемкие приложения такого рода, в действительности, должны быть написаны на компилируемом языке, таком как Си. Как сценарий, он работает мучительно медленно.

Пример 27-13. Сито Эратосфена

```
#!/bin/bash
# sieve.sh (ex68.sh)

# Сито Эратосфена
# Древний алгоритм нахождения простых чисел.

#  Он работает на несколько порядков медленнее,
#+ чем эквивалентная программа написанная на Си.

LOWER_LIMIT=1       # Начинается с 1.
UPPER_LIMIT=1000    # И заканчивается 1000.
# (Можете установить другой потолок ... если есть много свободного времени)

PRIME=1
NON_PRIME=0

let SPLIT=UPPER_LIMIT/2
# Оптимизация:
# Нужно проверить только половину чисел до верхнего предела. Почему?


declare -a Primes
# Простые числа Primes[] являются массивом.


initialize ()
{
# Инициализируем массив.

i=$LOWER_LIMIT
until [ "$i" -gt "$UPPER_LIMIT" ]
do
  Primes[i]=$PRIME
  let "i += 1"
done
#  Полагаем, что все члены массива виновны (простые),
#+ пока не доказана невиновность.
}

print_primes ()
{
# Вывод членов массива Primes [], помеченных как простые.

i=$LOWER_LIMIT

until [ "$i" -gt "$UPPER_LIMIT" ]
do
```

```
  if [ "${Primes[i]}" -eq "$PRIME" ]
  then
    printf "%8d" $i
    # 8 пробелов на число делают столбцы симпатичными.
  fi

  let "i += 1"

done

}

sift () # Отсеиваем числа не относящиеся к простым.
{

let i=$LOWER_LIMIT+1
# Давайте начнем с 2.

until [ "$i" -gt "$UPPER_LIMIT" ]
do

  if [ "${Primes[i]}" -eq "$PRIME" ]
# Уже просеянные числа не просеиваются (помечаются как не простые).
  then

  t=$i

     while [ "$t" -le "$UPPER_LIMIT" ]
     do
     let "t += $i "
     Primes[t]=$NON_PRIME
# Пометим все кратные не простым числам.
     done

  fi

  let "i += 1"
done


}


# ===============================================
# main ()
# Вызов функций последовательно.
initialize
sift
print_primes
# Это то, что называют структурным программированием.
# ===============================================

echo

exit 0



# ------------------------------------------------------- #
# Код строк ниже не будет выполняться, из-за «exit.»
```

```
#  Это улучшенная версия просеивания Stephane Chazelas,
#+ исполняемая несколько быстрее.

# Должна вызваться аргументом из командной строки (в пределе простых чисел).

UPPER_LIMIT=$1                  # Из командной строки.
let SPLIT=UPPER_LIMIT/2         # Половина от максимального количества.

Primes=( '' $(seq $UPPER_LIMIT) )

i=1
until (( ( i += 1 ) > SPLIT ))  # Нужно проверить только половину.
do
  if [[ -n ${Primes[i]} ]]
  then
    t=$i
    until (( ( t += i ) > UPPER_LIMIT ))
    do
      Primes[t]=
    done
  fi
done
echo ${Primes[*]}

exit $?
```

Пример 27-14. Сито Эратосфена, оптимизированое

```
#!/bin/bash
# Оптимизированное Сито Эратосфена
# Сценарий Jared Martin, с небольшими изменениями автором ABS Guide.
# Используется в ABS Guide с разрешения (спасибо!).

# Основан на сценарии Advanced Bash Scripting Guide.
# http://tldp.org/LDP/abs/html/arrays.html#PRIMES0 (ex68.sh).

# http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf (reference)
# Проверка результатов http://primes.utm.edu/lists/small/1000.txt

# Необходимо, но будет не достаточным, например,
#     ((($(sieve 7919 | wc -w) == 1000)) && echo "7919 это 1000-е простое"

UPPER_LIMIT=${1:?"Необходим верхний предел поиска простых чисел."}

Primes=( '' $(seq ${UPPER_LIMIT}) )

typeset -i i t
Primes[i=1]='' # 1 это не простое число.
until (( ( i += 1 ) > (${UPPER_LIMIT}/i) ))  # Нужно проверить только i.
  do                                         # Почему?
    if ((${Primes[t=i*(i-1), i]}))
    # Незаметно, но поучительно, использование арифметического расширения
    #+ для составляющих.
    then
      until (( ( t += i ) > ${UPPER_LIMIT} ))
        do Primes[t]=; done
    fi
  done
```

```
# echo ${Primes[*]}
echo   # Измените исходный сценарий для вывода (80-столбцового вывода).
printf "%8d" ${Primes[*]}
echo; echo

exit $?
```

Сравните эти массивы на основе простых чисел, с созданными альтернативно, без использования массивов, Пример A-15 и Пример 16-46.

--

Массивы предоставляют собой, в некоторой степени, имитацию структур данных, для которых в Bash нет никакой поддержки.

Пример 27-15. Эмуляция стека push-down

```
#!/bin/bash
# stack.sh: моделирование стека push-down

#  Подобно стеку процессора, стек push-down хранит элементы данных
#+ последовательно, но выпускает их в обратном порядке, последний-первым.


BP=100            #  Базовый указатель стека массива.
                  #  Начинается с элемента 100.

SP=$BP            #  Указатель стека.
                  #  Инициализация «базы» (низа) стека.

Data=             #  Содержимое, находящееся в стеке.
                  #  Должна использоваться глобальная переменная,
                  #+ потому что ограниченная функция возвращает диапазон.


                  # 100     Базовый указатель       <-- Базовый указатель
                  #  99     Первые данные в нем
                  #  98     Вторые данные в нем
                  # ...     Другие данные
                  #         Последние данные в нем  <-- Указатель стека


declare -a stack


push()                    # Из стека.
{
if [ -z "$1" ]            # Ничего не извлекается?
then
  return
fi

let "SP -= 1"             # Вылет указателя стека.
stack[$SP]=$1

return-=+=
}
```

```
pop()                     # Извлечение элемента стека.
{
Data=                     # Опустошаем данные.

if [ "$SP" -eq "$BP" ]    # Стек пустой?
then
  return
fi                        #  Это сохраняет SP от получения последних 100,
                          #+ т.е., прекращает перемещение в стеке.

Data=${stack[$SP]}
let "SP += 1"             # Вылет указателя стека.
return
}

status_report()           # Узнаем, что происходит.
{
echo "-----------------------------------"
echo "REPORT"
echo "Stack Pointer = $SP"
echo "Просто выскочила \'' $Data'\ ' из стека."
echo "-----------------------------------"
echo
}


# =======================================================
# Теперь немного приятного.

echo

# Посмотрим, можем ли вынуть что - нибудь из пустого стека.
pop
status_report

echo

push garbage
pop
status_report     # Мусор в, мусор из.

value1=23;        push $value1
value2=skidoo;    push $value2
value3=LAST;      push $value3

pop               # ПОСЛЕДНИЙ
status_report
pop               # убежало
status_report
pop               # 23
status_report     # Последний в, первый из!

#  Обратите внимание, как стек указателя уменьшается с каждым push и
#+ увеличивается с каждым pop.

echo

exit 0

# =======================================================
```

```
# Упражнения:
# ---------

# 1)  Измените функцию «push()», что бы разрешить извлекать несколько
#+    элементов стека одним вызовом функции.

# 2)  Измените функцию «pop()», что бы разрешить вкладывать несколько
#+    элементов из стека одним вызовом функции.

# 3)  Добавьте проверку ошибок в критических функциях.
#     То есть, возвращаемого кода ошибки, в зависимости от успешного или
#+    неудачного завершения операции, и принятия соответствующих мер.

# 4)  С помощью этого сценария, в качестве отправной точки, напишите 4-х
#+    разрядный калькулятор на основе стека.
```

--

Необычные манипуляции с содержимым «массива» могут потребовать промежуточных переменных. Для таких проектов все-таки рассмотрите вопрос об использовании более мощного языка программирования, такого как Perl или Си.

Пример 27-16. Комплексное применение массива: *Изучение странных математических серий*

```
#!/bin/bash

# Пресловутые "Q-серии" Douglas Hofstadter:

# Q(1) = Q(2) = 1
# Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), for n>2

#  Это серия «хаотичных» целых чисел со странным
#+ и непредсказуемым поведением.
#  Первые 20 членов этого ряда:
#  1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12

#  См учебник Hofstadter, _Goedel, Escher, Bach: An Eternal Golden Braid_,
#+ стр.. 137.


LIMIT=100     # Количество вычисляемых членов.
LINEWIDTH=20  # Число членов выводимых в строке.

Q[1]=1        # Первые два члена серии это 1.
Q[2]=1

echo
echo "Q-series [$LIMIT terms]:"
echo -n "${Q[1]} "             # Вывод первых двух членов.
echo -n "${Q[2]} "

for ((n=3; n <= $LIMIT; n++))  # Выражение цикла аналогично Си.
do   # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]]  для n>2
#    Нужно разбить выражение на промежуточные члены,
#+   так как Bash хорошо не обрабатывает сложные арифметические массивы.

  let "n1 = $n - 1"        # n-1
  let "n2 = $n - 2"        # n-2
```

```
  t0=`expr $n - ${Q[n1]}`  # n - Q[n-1]
  t1=`expr $n - ${Q[n2]}`  # n - Q[n-2]

  T0=${Q[t0]}              # Q[n - Q[n-1]]
  T1=${Q[t1]}              # Q[n - Q[n-2]]

Q[n]=`expr $T0 + $T1`      # Q[n - Q[n-1]] + Q[n - Q[n-2]]
echo -n "${Q[n]} "


   if [ `expr $n % $LINEWIDTH` -eq 0 ]    # Формат вывода.
   then   #      ^ модуль
   echo # Делим строки на аккуратные части.
   fi

done

echo

exit 0

#  Это итеративная реализация Q-серии.
#  Более интуитивная, рекурсивная, реализация останется в качестве
#+ упражнения.
#  Предупреждение:  С помощью сценария расчет этой серии рекурсивно, занимает
#+ очень много времени. На Си/C++ будет порядком быстрее.
```

--

Bash поддерживает только одномерные массивы, хотя - небольшая хитрость позволяет создавать из них многомерные.

Пример 27-17. Имитация двумерного массива

```
#!/bin/bash
# twodim.sh: Имитация двумерного массива.

# Одномерный массив состоит из одной строки.
# Двумерный массив сохраняет строки последовательно.

Rows=5
Columns=5
# Массив 5 X 5.

declare -a alpha     # Символ alpha [Rows] [Column];
                     # Объявлять не нужно. Почему?

load_alpha ()
{
local rc=0
local index

for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
do     # Используем различные символы, какие нравятся.
  local row=`expr $rc / $Columns`
  local column=`expr $rc % $Rows`
  let "index = $row * $Rows + $column"
  alpha[$index]=$i
# alpha[$row][$column]
```

```
  let "rc += 1"
done

#  Проще было бы
#+ declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T U V W X Y )
#+ но как-то не хватает «аромата» двумерного массива.
}

print_alpha ()
{
local row=0
local index

echo

while [ "$row" -lt "$Rows" ]   #  Порядок вывода «больших строк»:
do                             #+ колонки изменяются,
                               #+ в то время как строка (внешний контур)
                               #+ остается такой же.
  local column=0

  echo -n "       "            #  Верхняя строка «площади» массива меняет
                               #+ место.

  while [ "$column" -lt "$Columns" ]
  do
    let "index = $row * $Rows + $column"
    echo -n "${alpha[index]} "
#             alpha[$row][$column]
    let "column += 1"
  done

  let "row += 1"
  echo

done

# Более простой вариант
#     echo ${alpha[*]} | xargs -n $Columns

echo
}

filter ()     # Отфильтровываются отрицательные индексы.
{

echo -n "  "  # Обеспечивается наклон.
             # Объясните как.

if [[ "$1" -ge 0 &&  "$1" -lt "$Rows" && "$2" -ge 0 && "$2" -lt "$Columns" ]]
then
    let "index = $1 * $Rows + $2"
#         Теперь выводим это наоборот.
    echo -n " ${alpha[index]}"
#              alpha[$row][$column]
fi

}


rotate ()  #  Разворот массива на 45 градусов --
```

```
{          #+ «балансируем» на его нижнем левом углу.
local row
local column

for (( row = Rows; row > -Rows; row-- ))
 do       # Шаг назад через массив обратно. Почему?

   for (( column = 0; column < Columns; column++ ))
   do

     if [ "$row" -ge 0 ]
     then
       let "t1 = $column - $row"
       let "t2 = $column"
     else
       let "t1 = $column"
       let "t2 = $column + $row"
     fi

    filter $t1 $t2   # Отфильтровываются отрицательные индексы.
                     # Что случится, если этого не сделать?
  done

  echo; echo

done

#  Вращение массива вдохновлено примерами (стр. 143-146) в
#+ "Advanced C Programming on the IBM PC," Herbert Mayer
#+ (см. библиографию).
#  Он просто подходит для демонстрации, что многое из того, что может
#+ быть сделано в Си - может быть также сделано и в сценариях shell.

}


#--------------- Теперь позвольте начать шоу. ------------#
load_alpha     # Грузим массив.
print_alpha    # Выводим вывод.
rotate         # Поворачиваем его на 45 градусов против часовой стрелки.
#---------------------------------------------------------#

exit 0

# Довольно надуманное, не говоря уже о том, что это безвкусное,
# моделирование.

# Упражнения:
# ---------
# 1)  Перепишите загрузку массива и функции вывода в более интуитивном и
#     менее запутанном стиле.
#
# 2)  Выясните, как работают функции поворота массива.
#     Подсказка: подумайте о последствиях обратного индексирования массива.
#
# 3)  Перепишите этот сценарий для обработки не квадратного массива,
#     например 6 X 4.
#     Попробуйте свести к минимуму «искажения», когда массив повернут.
```

Двумерный массив, по существу, эквивалентен одномерному, но с дополнительными

режимами адресации для ссылок и управления отдельными элементами положения строки и столбца.

Еще более сложный пример имитации двумерного массива - Пример A-10.

Наиболее интересные сценарии с использованием массивов:

- Пример 12-3
- Пример 16-46
- Пример A-22
- Пример A-44
- Пример A-41
- Пример A-42

Глава 28. Косвенные ссылки

Мы видели, что ссылка на переменную, `$var` извлекает ее *значение*. Но, как насчет *значения значения*? Как насчет `$$var`?

Обычно она записывается `\$$var` и ей предшествует **`eval`** (а иногда и **`echo`**). Это называется *косвенной ссылкой*.

Пример 28-1. Косвенные ссылки на переменные

```
#!/bin/bash
# ind-ref.sh: Косвенные ссылки на переменные
# Доступ к содержимому содержимого переменной.

# Во-первых, давайте немного подурачимся.

var=23

echo "\$var   = $var"           # $var   = 23
# Пока все, как ожидается. Но ...

echo "\$\$var  = $$var"         # $$var  = 4570var
#  Не пойдет ...
#  \$\$ расширяет PID сценария
#  -- обращается на вход переменной $$ --
#+ и "var" выводится на экран как основной текст.
#  (Спасибо за разъяснение Jakob Bohm.)

echo "\\\$\$var = \$$var"       # \$$var = $23
#  Как и ожидалось. Первый $ экранируется и подставляется к
#+ значению var ($var = 23 ).
#  Значимо, но до сих пор не полезно.

# Теперь давайте начнем снова и пойдем по правильному пути.

# ============================================== #
```

```
a=letter_of_alphabet   # Переменная "a" содержит имя другой переменной.
letter_of_alphabet=z

echo

# Прямая ссылка.
echo "a = $a"          # a = letter_of_alphabet

# Косвенная ссылка.
  eval a=\$$a
# ^^^        Принудительная eval(оценка), и ...
#       ^    экранируемый первый $ ...
# ---------------------------------------------------------------------
# «eval» принуждает обновится $a, подстанавливая ее на обновленное
# значение \$$a.
# Таким образом, мы видим, причину частого появления «eval» в нотации
# косвенной ссылки.
# ---------------------------------------------------------------------
  echo "Теперь a = $a"    # Теперь a = z

echo


# Теперь давайте попробуем изменить ссылку второго порядка.

t=table_cell_3
table_cell_3=24
echo "\"table_cell_3\" = $table_cell_3"            # "table_cell_3" = 24
echo -n "dereferenced \"t\" = "; eval echo \$$t    # разыменованное "t" = 24
# Это простой случай, следующий работает также (почему?).
#         eval t=\$$t; echo "\"t\" = $t"

echo

t=table_cell_3
NEW_VAL=387
table_cell_3=$NEW_VAL
echo "Меняем значение \"table_cell_3\" на $NEW_VAL."
echo "\"table_cell_3\" теперь $table_cell_3"
echo -n "теперь \"t\" разыменованное "; eval echo \$$t
# "eval" принимает два аргумента "echo" и "\$$t"
#+ (устанавливается равным $table_cell_3)

echo

# (Спасибо за объяснения поведения выше Stephane Chazelas.)

#   Более простой способ - это нотация ${!t}, обсуждаемая в разделе 'Bash,
#+  версии 2'.
#   CM. ex78.sh.

Exit 0
```

Косвенные ссылки в Bash являются многоэтапным процессом. Сначала берется имя переменной: `varname`. Затем, ссылка на нее: `$varname`. Потом, ссылка на ссылку: `$$varname`. Затем, *экранируем* первый `$`: `\$$varname`. В конце, заставляем переоценить

выражение и присвоить его: **`eval newvar=\$$varname`**.

Какая практическая польза от косвенной ссылки на переменные? Она придает Bash немного функциональности указателей Си, например в таблице подстановки. А они имеют некоторые очень интересные приложения...

Nils Radtke демонстрирует, как создавать «динамические» имена переменной и оценивать их содержимое. Это может пригодиться в исходных файлах конфигурации.

```
#!/bin/bash


# ----------------------------------------------
# Это может быть «исходный код» из отдельного файла.
isdnMyProviderRemoteNet=172.16.0.100
isdnYourProviderRemoteNet=10.0.0.10
isdnOnlineService="MyProvider"
# ----------------------------------------------


remoteNet=$(eval "echo \$$(echo isdn${isdnOnlineService}RemoteNet)")
remoteNet=$(eval "echo \$$(echo isdnMyProviderRemoteNet)")
remoteNet=$(eval "echo \$isdnMyProviderRemoteNet")
remoteNet=$(eval "echo $isdnMyProviderRemoteNet")

echo "$remoteNet"    # 172.16.0.100

# ================================================================

#  И получается даже лучше.

#  Рассмотрим следующий фрагмент, задана переменная с именем getSparc,
#+ а переменная getIa64 - нет:

chkMirrorArchs () {
  arch="$1";
  if [ "$(eval "echo \${$(echo get$(echo -ne $arch |
       sed 's/^\(.\).*/\1/g' | tr 'a-z' 'A-Z'; echo $arch |
       sed 's/^.\(.*\)/\1/g')):-false}")" = true ]
  then
     return 0;
  else
     return 1;
  fi;
}

getSparc="true"
unset getIa64
chkMirrorArchs sparc
echo $?        # 0
               # Истинно

chkMirrorArchs Ia64
echo $?        # 1
```

```
                # Ложно

# Примечания:
# -----
# Явно создается даже замещаемая часть имени переменной.
# Все параметры для вызова chkMirrorArchs в нижнем регистре.
# Имя переменной состоит из двух частей: "get" и "Sparc" ...
```

Пример 28-2. Передача косвенных ссылок *в awk*

```
#!/bin/bash

#  Еще одна версия сценария «column totaler», который добавляет указанную
#+ колонку (номер) в целевой файл.
#  Используя косвенные ссылки.

ARGS=2
E_WRONGARGS=85

if [ $# -ne "$ARGS" ] #  Проверка наличия нужного числа аргументов
                      #+ командной строки.
then
   echo "Usage: `basename $0` номер колонки файла"
   exit $E_WRONGARGS
fi

filename=$1         # Имя обрабатываемого файла.
column_number=$2    # Какую колонку вставить.

#===== До этого момента - аналог оригинального сценария  =====#


# Вызывается многострочный сценарий awk
#   awk "
#   ...
#   ...
#   ...
#   "


# Начало сценария awk.
# ------------------------------------------------
awk "

{ total += \$${column_number} # Косвенная ссылка
}
END {
     print total
     }

     " "$filename"
# Обратите внимание, что для awk не нужен eval перед \$.
# ------------------------------------------------
# Окончание сценария awk.

#  Косвенная ссылка на переменную позволяет избегать неприятностей при ссылке
#+ на переменную оболочки во встроенном сценарии awk.
#  Спасибо Stephane Chazelas.
```

```
exit $?
```

Метод косвенных ссылок является немного сложным. Если переменная второго порядка меняет свое значение, то переменная первого порядка должна быть надлежащим образом разыменована (как в приведенном выше примере). К счастью нотация *${!variable}* введена в версии 2 Bash (см. Пример 37-2 и Пример-22), что делает косвенные ссылки более интуитивными.

Bash не поддерживает арифметические указатели, а это серьезно ограничивает полезность косвенных ссылок. На самом деле, косвенные ссылки в языке сценариев являются, в лучшем случае, чем-то второстепенным.

Глава 29. /dev и /proc

Содержание

Файловые системы Linux или UNIX, как правило, имеют специальные директории /dev и /proc.

29.1. /dev

Директория /dev содержит записи *физических устройств*, которые могут или не могут присутствовать в оборудовании. [1] Обычно они называются *файлами устройств*. В качестве примера - разделы привода жесткого диска, содержащие смонтированную файловую систему(ы) имеют записи в /dev, и выводятся командой **df**.

```
bash$ df
Filesystem           1k-blocks      Used Available Use%
 Mounted on
 /dev/hda6               495876    222748    247527  48% /
 /dev/hda1                50755      3887     44248   9% /boot
 /dev/hda8               367013     13262    334803   4% /home
 /dev/hda5              1714416   1123624    503704  70% /usr
```

Среди прочего, директория `/dev` содержит *петлевые устройства*, такие как `/dev/loop0`. Устройство ***loopback*** — трюк, который позволяет обращаться к обычному файлу, как к блочному устройству. [2] Это позволяет монтировать всю файловую систему одним большим файлом. См. Пример 17-8 и Пример 17-7.

Некоторые устройства в `/dev` имеют другие, специализированные, назначения, и называются *псевдо-устройствами*, такие как `/dev/null`, `/ dev/zero`, `/dev/urandom`, `/dev/sda1` (раздел жесткого диска), `/dev/udp` (порт пакетов пользовательских дейтаграмм) и `/dev/tcp`.

Например:

Чтобы вручную примонтировать флэш-накопитель USB, добавьте следующую строку в `/etc/fstab`. [3]

```
/dev/sda1    /mnt/flashdrive    auto    noauto,user,noatime    0 0
```

(См. также Пример A-23).

Проверка, присутствия диска в CD-приводе (мягкая ссылка на `/dev/hdc`):

```
head -1 /dev/hdc


#  head: cannot open '/dev/hdc' for reading: No medium found
#  (Нет диска в приводе)

#  head: error reading '/dev/hdc': Input/output error
#  (Диск в дисководе, ошибка чтения;
#+  возможно это не записываемый CDR диск.)

#  Поток символов и смешанный бред
#  (Предварительно записанный диск находится в приводе,
#+ выводятся необработанные данные --поток ASCII и бинарные данные.)
#  Здесь мы видим мудрое использование «head» для ограничения выводимого
#+ управляемыми пропорциями, в отличии от «cat» или чего-то подобного.

#  Теперь это просто вопрос проверки и анализа выходных данных и принятия
#+ соответствующих мер.
```

При выполнении командой файла псевдо-устройства `/dev/tcp/$host/$port`, Bash открывает соединение TCP связанного *сокета*.

Сокет — это коммуникационный узел связанный с определенным портом I/O. (Это аналог *аппаратного сокета*, или *гнезда* для соединительного кабеля). Позволяет передавать данные между устройствами на самой машине, между машинами в той же сети, между машинами в разных сетях и, конечно же, между машинами в разных местах в Интернете.

В следующих примерах предполагается, что подключение к Интернету активно.

Получение точного время от `nist.gov`:

```
bash$ cat </dev/tcp/time.nist.gov/13
53082 04-03-18 04:26:54 68 0 0 502.3 UTC(NIST) *
```

[Этот пример внес Mark.]

Обобщение вышеупомянутого в сценарии:

```
#!/bin/bash
# Этот сценарий запускается с правами root.

URL="time.nist.gov/13"

Time=$(cat </dev/tcp/"$URL")
UTC=$(echo "$Time" | awk '{print$3}')   # Третье поле - временя UTC (GMT).
# Упражнение: Измените сценарий для различных часовых поясов.

echo "UTC Time = "$UTC""
```

Загрузка URL:

```
bash$ exec 5<>/dev/tcp/www.net.cn/80
bash$ echo -e "GET / HTTP/1.0\n" >&5
bash$ cat <&5
```

[Благодарность Mark и Mihai Maties.]

Пример 29-1. Использование `/dev/tcp` для устранения неполадок

```
#!/bin/bash
# dev-tcp.sh: перенаправление /dev/tcp, для проверки подключения к Интернету.

# Сценарий Troy Engel.
# Используется с разрешения.

TCP_HOST=news-15.net       # Известный spam-friendly ISP.
TCP_PORT=80                # Порт 80 это http.

# Попытка соединения. (Напоминает 'ping' ...)
echo "HEAD / HTTP/1.0" >/dev/tcp/${TCP_HOST}/${TCP_PORT}
MYEXIT=$?

: <<ОБЪЯСНЕНИЕ
```

```
Если Баш был скомпилирован с --enable-net-redirections, он может использовать
специальный символ устройства для перенаправления TCP и UDP. Эти
перенаправления одинаково используются STDIN/STDOUT/STDERR. Устройствами записи
для /dev/tcp являются 30,36:

  mknod /dev/tcp c 30 36

>Из ссылки bash:
/dev/tcp/host/port
    Если host это допустимое имя хоста или Интернет-адреса, а port это целое
число порта или имени сервиса, Bash пытается открыть TCP соединение через
соответствующий сокет.
ОБЪЯСНЕНИЕ

if [ "X$MYEXIT" = "X0" ]; then
  echo "Соединение удалось. Код выхода: $MYEXIT"
else
  echo "Соединение не удалось. Код выхода: $MYEXIT"
fi

exit $MYEXIT
```

Пример 29-2. Проигрывание музыки

```
#!/bin/bash
# music.sh

# Музыка без внешних файлов

# Автор: Antonio Macchi
# Используется в ABS Guide с разрешения.


#  /dev/dsp по умолчанию = 8000 фреймов в секунду, 8 бит на фрейм (1 байт),
#+ 1 канал (mono)

duration=2000       # Если 8000 байт = 1 секунде, то 2000 = 1/4 секунды.
volume=$'\xc0'      # Макс. значение = \xff (или \x00).
mute=$'\x80'        # Без значения   = \x80 (среднее).

function mknote ()  # $1=Высота ноты в байтах (т.е. A (Ля) = 440Hz ::
{                   #+ 8000 fps / 440 = 16 :: A = 16 байт в секунду)
  for t in `seq 0 $duration`
  do
    test $(( $t % $1 )) = 0 && echo -n $volume || echo -n $mute
  done
}

e=`mknote 49`
g=`mknote 41`
a=`mknote 36`
b=`mknote 32`
c=`mknote 30`
cis=`mknote 29`
d=`mknote 27`
e2=`mknote 24`
```

```
n=`mknote 32767`
# Европейское обозначение.

echo -n "$g$e2$d$c$d$c$a$g$n$g$e$n$g$e2$d$c$c$b$c$cis$n$cis$d \
$n$g$e2$d$c$d$c$a$g$n$g$e$n$g$a$d$c$b$a$b$c" > /dev/dsp
# dsp = Digital Signal Processor (Цифровой сигнальный процессор)

exit      # «bonny» элегантный пример сценария оболочки!
```

Примечания

[1] Записи в `/dev` предоставляют точки монтирования для физических и виртуальных устройств. Эти записи занимают очень мало места на диске.

Некоторые устройства, такие как `/dev/null`, `/dev/zero` и `/dev/urandom` являются виртуальными. Они не являются фактическими физическими устройствами и существуют только в программном обеспечении.

[2] *Блочное устройство* считывает/записывает данные в блоки или *блоками*, в отличие от *символьных устройств*, данные которых доступны единичными *символами*. Примерами блочных устройств являются жесткие диски, компакт-диски и флэш-накопители. Примеры символьных устройств: клавиатуры, модемы, звуковые карты.

[3] Конечно, точка монтирования `/mnt/flashdrive` должна уже существовать. Если нет, то из-под *root* создайте ее, **`mkdir /mnt/flashdrive`**.

На самом деле диск монтируется с использованием команды: **`mount /mnt/flashdrive`**

Новые дистрибутивы Linux автоматически монтируют флэш-накопители к директории `/media` без вмешательства пользователя.

29.2. `/proc`

Директория `/proc` это псевдо-файловая система. Файлы `/proc` отражают запущенные процессы системы и ядра, а так же содержат информацию и статистические данные о них.

```
bash$ cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
128 ptm
136 pts
```

```
 162 raw
 254 pcmcia

 Block devices:
   1 ramdisk
   2 fd
   3 ide0
   9 md



bash$ cat /proc/interrupts
           CPU0
   0:      84505          XT-PIC  timer
   1:       3375          XT-PIC  keyboard
   2:          0          XT-PIC  cascade
   5:          1          XT-PIC  soundblaster
   8:          1          XT-PIC  rtc
  12:       4231          XT-PIC  PS/2 Mouse
  14:     109373          XT-PIC  ide0
 NMI:          0
 ERR:          0


bash$ cat /proc/partitions
major minor  #blocks  name     rio rmerge rsect ruse wio wmerge wsect wuse
running use aveq

    3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0
111550 644030
    3     1      52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
    3     2          1 hda2 0 0 0 0 0 0 0 0 0 0 0
    3     4     165280 hda4 10 0 20 210 0 0 0 0 0 210 210
    ...



bash$ cat /proc/loadavg
0.13 0.42 0.27 2/44 1119



bash$ cat /proc/apm
1.16 1.2 0x03 0x01 0xff 0x80 -1% -1 ?



bash$ cat /proc/acpi/battery/BAT0/info
present:                 yes
 design capacity:         43200 mWh
 last full capacity:      36640 mWh
 battery technology:      rechargeable
 design voltage:          10800 mV
 design capacity warning: 1832 mWh
 design capacity low:     200 mWh
 capacity granularity 1:  1 mWh
 capacity granularity 2:  1 mWh
 model number:            IBM-02K6897
 serial number:            1133
 battery type:            LION
 OEM info:                Panasonic
```

```
bash$ fgrep Mem /proc/meminfo
MemTotal:       515216 kB
 MemFree:        266248 kB
```

Сценарии оболочки могут извлекать из `/proc` данные некоторых файлов. [1]

```
FS=iso                          # Файловая система ISO поддерживается ядром?

grep $FS /proc/filesystems   # iso9660

kernel_version=$( awk '{ print $3 }' /proc/version )

CPU=$( awk '/model name/ {print $5}' < /proc/cpuinfo )

if [ "$CPU" = "Pentium(R)" ]
then
  запуск_каких-то_команд
  ...
else
  запуск_других_команд
  ...
fi



cpu_speed=$( fgrep "cpu MHz" /proc/cpuinfo | awk '{print $4}' )
#  Текущая скорость ЦП на вашем компьютере  (в МГц).
#  На ноутбуке может отличаться, в зависимости от питания - от батареи
#+ или от сети.


#!/bin/bash
# get-commandline.sh
# Получение параметров процесса в командной строке.

OPTION=cmdline

# Определяем PID.
pid=$( echo $(pidof "$1") | awk '{ print $1 }' )
# Получаем только первое    ^^^^^^^^^^^^^^^^^^  из нескольких параметров.

echo
echo "ID процесса (первое из нескольких) "$1" = $pid"
echo -n "Аргументы командной строки: "

cat /proc/"$pid"/"$OPTION" | xargs -0 echo
#   Выводимые форматы:     ^^^^^^^^^^^^^^^
#   (Спасибо Han Holl за исправление!)

echo; echo


# Например:
```

```
# sh get-commandline.sh xterm
```

```
+
```

```
devfile="/proc/bus/usb/devices"
text="Spd"
USB1="Spd=12"
USB2="Spd=480"


bus_speed=$(fgrep -m 1 "$text" $devfile | awk '{print $9}')
#                 ^^^^ Остановка после нахождения первого соответствия.

if [ "$bus_speed" = "$USB1" ]
then
  echo "Порт USB 1.1."
  # Соответствует USB 1.1.
fi
```

Возможен даже контроль некоторых периферийных устройств с помощью команд направляемых в директорию `/proc`.

```
root# echo on > /proc/acpi/ibm/light
```

Это обращение к *Thinklight* в некоторых моделях Thinkpads IBM/Lenovo. (Работает не на всех дистрибутивах Linux.)

Конечно, необходима осторожность при записи в `/proc`.

Директория `/proc` содержит поддиректории с необычными числовыми именами. Каждое из этих *имен* отображает ***ID*** запущенного текущего процесса. В каждой из этих поддиректорий есть несколько файлов содержащих информацию о соответствующем процессе. Файлы `stat` и `status` хранят *статистику* запущенного процесса, файл `cmdline` содержит аргументы командной строки, которыми был вызван процесс, а файл `exe` является символической ссылкой на полный путь вызванного процесса. Есть несколько таких файлов, вот они-то и являются наиболее интересными с точки зрения сценариев.

Пример 29-3. Поиск процесса, связанного с заданным PID

```
#!/bin/bash
# pid-identifier.sh:
# Дает полный путь процесса связанного с заданным pid.

ARGNO=1  # Число аргументов ожидаемых сценарием.
E_WRONGARGS=65
E_BADPID=66
E_NOSUCHPROCESS=67
E_NOPERMISSION=68
PROCFILE=exe
```

```
if [ $# -ne $ARGNO ]
then
  echo "Usage: `basename $0` номер PID" >&2  # Сообщение об ошибке >stderr.
  exit $E_WRONGARGS
fi

pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
# Проверяем pid в листинге "ps", поле #1.
# Затем удостоверимся, что это реальный процесс, а не процесс вызванный этим
сценарием.
# Последний "grep $1" отфильтровывает эту вероятность.
#
#    pidno=$( ps ax | awk '{ print $1 }' | grep $1 )
#    Как пояснил Teemu Huovila, это так же будет работать.

if [ -z "$pidno" ]  #  Если результатом, после всех фильтраций, является строка
then                #+ нулевой длины, то процесс, соответствующий заданному
                    #+ pid,отсутствует.
  echo "Нет такого запущенного процесса."
  exit $E_NOSUCHPROCESS
fi

# Альтернативно:
#   if ! ps $1 > /dev/null 2>&1
#   then                # не запущен процесс соответствующий заданному pid.
#     echo "Нет такого запущенного процесса."
#     exit $E_NOSUCHPROCESS
#    fi

# Чтобы упростить весь процесс, используйте "pidof".


if [ ! -r "/proc/$1/$PROCFILE" ]  # Проверка прав на чтение.
then
  echo "Процесс $1 запущен, но..."
  echo "Нет прав на чтение в /proc/$1/$PROCFILE."
  exit $E_NOPERMISSION  #  Обычный пользователь не имеет доступа к некоторым
                        #+ файлам в/proc.
fi

# Последние две проверки могут быть заменены:
#    if ! kill -0 $1 > /dev/null 2>&1 # '0' это не сигнал, а
                                      # проверка возможности
                                      # посылать сигнал процессу.
#    затем выводится echo "PID не существует или вы не владелец" >&2
#    exit $E_BADPID
#    fi


exe_file=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
# Или      exe_file=$( ls -l /proc/$1/exe | awk '{print $11}' )
#
#  /proc/pid-number/exe это символическая ссылка
#+ на полный путь имени вызываемого процесса.

if [ -e "$exe_file" ]  #  Если /proc/pid-number/exe существует,
then                   #+ то существует и соответствующий процесс.
  echo "Процесс #$1 вызываемый $exe_file."
else
  echo "Нет такого запущенного процесса."
```

```
fi

#  Разработан сценарий, который *почти* заменяет
#       ps ax | grep $1 | awk '{ print $5 }'
#  Однако, он не работает...
#+ потому что пятое поле 'ps' это argv[0] процесса,
#+ а не путь исполняемого файла.
#
# Однако, одно из следующего будет работать.
#       find /proc/$1/exe -printf '%l\n'
#       lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'

# Комментарии добавлены Stephane Chazelas.

exit 0
```

Пример 29-4. Статус он-лайн подключения

```
#!/bin/bash
# connect-stat.sh
#  Обратите внимание, что этот сценарий может быть изменен
#+ для работы с беспроводными подключениями.

PROCNAME=pppd        # демон (служба) ppp
PROCFILENAME=status  # Статус.
NOTCONNECTED=85
INTERVAL=2           # Обновление каждые 2 секунды.

pidno=$( ps ax | grep -v "ps ax" | grep -v grep | grep $PROCNAME |
awk '{ print $1 }' )

# Ищем номер процесса 'pppd', т.е. 'демона ppp'.
# Нужно отфильтровать строки процессов создаваемых самим поиском.
#
#  Однако, как объясняет Oleg Philon,
#+ это можно значительно упростить используя «pidof».
#  pidno=$( pidof $PROCNAME )
#
#  Мораль:
#+ Когда последовательность команд становится слишком сложной, поищите
#+ ярлык.


if [ -z "$pidno" ]   # Если pid отсутствует, то процесс не запущен.
then
  echo "Нет соединения."
# exit $NOTCONNECTED
else
  echo "Соединение."; echo
fi

while [ true ]       # Бесконечный цикл, здесь сценарий может быть улучшен.
do

  if [ ! -e "/proc/$pidno/$PROCFILENAME" ]
  # Когда процесс запущен, то существует и "статус" файла.
  then
    echo "Соединение разорвано."
```

```
#   exit $NOTCONNECTED
  fi

netstat -s | grep "packets received"  # Какая-то статистика соединения.
netstat -s | grep "packets delivered"


  sleep $INTERVAL
  echo; echo

done

exit 0

# Как можно заметить, этот сценарий должен завершаться нажатием Control-C.

#    Упражнения:
#    ---------
#    Улучшите сценарий, что бы он завершался нажатием клавиши «q».
#    Сделайте сценарий более удобным для пользователей иными способами.
#    Исправить сценарий для работы с беспроводными/DSL соединениями.
```

В общем, опасно *записывать* в файлы находящиеся в `/proc`, так как это может повредить файловой системе или привести к аварии машины.

Примечания

[1] Некоторые системные команды, такие как ***procinfo***, ***free***, ***vmstat***, ***lsdev*** и ***uptime*** сделают это не хуже.

Глава 30. Сетевое программирование

Сеть это среднее между проданным слоном и белым слоном: никогда не забывается и всегда загажена.

--Nemo

Linux имеет множество инструментов для доступа, управления и поиска неисправностей сетевых соединений. Мы можем включать некоторые из этих инструментов в сценарии -- сценарии, которые расширят наши знания о сетях и облегчат администрирование сетей.

Вот простой сценарий CGI, который демонстрирует подключение к удаленному серверу.

Пример 30-1. Распечатка серверной среды

```
#!/bin/bash
# test-cgi.sh
# Michael Zick
# Используется с разрешения

# Возможно, придется изменить место расположения.
# (На серверах провайдера, Bash может быть расположен в необычном месте.)
# Другие места: /usr/bin или /usr/local/bin
# Можно даже попробовать sha-bang без какого-либо пути.

# Отключение создания имен файлов с помощью символов подстановки.
set -f

# Заголовок говорит браузеру чего ожидать.
echo Content-type: text/plain      # Тип содержимого: текст/обычное
echo

echo CGI/1.0 test script report:   # тестовый отчет сценария
echo

echo environment settings:         # параметры среды
set
echo

echo whereis bash?                 # где находится Bash?
whereis bash
echo


echo who are we?                   # кто мы?
echo ${BASH_VERSINFO[*]}
echo

echo argc это $#. argv это "$*".
echo

# CGI/1.0 ожидаемые переменные среды.
```

```
echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = "$PATH_INFO"
echo PATH_TRANSLATED = "$PATH_TRANSLATED"
echo SCRIPT_NAME = "$SCRIPT_NAME"
echo QUERY_STRING = "$QUERY_STRING"
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo AUTH_TYPE = $AUTH_TYPE
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH

exit 0

# Here document даст короткие инструкции.
:<<-'_test_CGI_'

1) Поместите это в директорию http://domain.name/cgi-bin.
2) Затем, откройте http://domain.name/cgi-bin/test-cgi.sh.

_test_CGI_
```

Может быть полезным для определения IP-адресов, которые пытались получить доступ к компьютеру.

Пример 30-2. IP адреса

```
#!/bin/bash
# ip-addresses.sh
# Список IP-адресов, с которыми соединялся ваш компьютер.

#  Вдохновлено сценарием Greg Bledsoe ddos.sh,
#  Linux Journal, 09 March 2011.
#    URL:
#  http://www.linuxjournal.com/content/back-dead-simple-bash-complex-ddos
#  Greg лицензировал этот сценарий под GPL2,
#+ и как производный, этот сценарий также под GPL2.

connection_type=TCP      # Также попробуйте UDP.
field=2           # Интересующее нас поле вывода.
no_match=LISTEN   # Фильтр записей, содержащих его. Почему?
lsof_args=-ni     # -i список Интернет-ассоциированных файлов.
                  # -n сохраняет числовые IP-адреса.
                  # Что будет без опции -n? Попробуй.
router="[0-9][0-9][0-9][0-9][0-9]->"
#       Удаляем информацию маршрутеризатора.

lsof "$lsof_args" | grep $connection_type | grep -v "$no_match" |
      awk '{print $9}' | cut -d : -f $field | sort | uniq |
      sed s/"^$router"//
```

```
#  Сценарий Bledsoe присваивает вывод отфильтрованного списка IP (аналогично
#+ строкам 19-22, выше) переменной.
#  Он проверяет нескольких подключений с одного IP-адреса,
#  затем использует:
#
#    iptables -I INPUT -s $ip -p tcp -j REJECT --reject-with tcp-reset
#
# ... в пределах цикличной 60-секундной задержки для отброса пакетов DDOS атак.


#  Упражнение:
#  --------
#  Используя команду «iptables» расширьте этот сценарий, чтобы пресекать
#+ попытки подключения от IP известных спамерских  доменов.
```

Примеры сетевого программирования:

1. Пример 16-41
2. Пример A-28
3. Пример A-29
4. Пример 29-1

См. также «Сетевые команды» в главе «Системные и административные команды», а так же коммуникационные команды во Внешних фильтрах,

Глава 31. O Zero и Nulls

Безупречная ошибочность, холодное
постоянство, прекрасное ничто...
Мертвое совершенство; не более.

--Alfred Lord Tennyson

`/dev/zero` … `/dev/null`

Использование `/dev/null`

Думайте об `/dev/null` как о *черной дыре*. По существу это эквивалент файла доступного только для записи. Все, что записано в него - исчезает. Попытки чтения или вывод из него не дадут результата. И все же `/dev/null` может быть весьма полезным в командной строке и в сценариях.

Подавление `stdout`.

```
cat $filename >/dev/null
# Содержимое файла не выводится в stdout.
```

Подавление `stderr` (из Примера 16-3).

```
rm $badname 2>/dev/null
#           Так отбрасывается сообщение об ошибке [stderr].
```

Подавление вывода из *обоих* `stdout` и `stderr`.

```
cat $filename 2>/dev/null >/dev/null
# Если "$filename" не существует, то не будет и вывода сообщения об ошибке.
# Если "$filename" существует, то содержимое файла не будет выведено в stdout.
# Т.е., не будет вывода из приведенных выше строк кода.
#
#  Это полезно в ситуациях, когда вывод не требуется, а возвращаемый код
#+ команды должен быть проверен.
#
# cat $filename &>/dev/null
# Как поясняет Baris Cicek - так же работает.
```

Удаление содержимого файла, но сохранение самого файла, со всеми правами (из Примера 2-1 и Примера 2-3):

```
cat /dev/null > /var/log/messages
#  : > /var/log/messages   имеет тот же эффект, но
#+ не порождает новый процесс.

cat /dev/null > /var/log/wtmp
```

Автоматически удаляет содержимое лог файла (особенно хорошо для борьбы с этими противными «cookies», посылаемыми коммерческими веб-сайтами):

Пример 31-1. Скрытие cookie jar

```
# Устаревший браузер Netscape.
# Аналогичный принцип применим и к новым браузерам.

if [ -f ~/.netscape/cookies ]  # Удаляются, если существуют.
then
  rm -f ~/.netscape/cookies
fi

ln -s /dev/null ~/.netscape/cookies
#  Все cookie теперь направляются в черную дыру раньше, чем сохраняются
#+ на диске.
```

Использование `/dev/zero`

Как и `/dev/null`, `/dev/zero` это файл псевдо-устройства, но производит поток нулей

(***двоичные нули***, а не вида ASCII). Вывод записываемый в `/dev/zero` исчезает, и, действительно, довольно трудно прочитать нули направляемые в него, хотя это можно сделать при помощи **`od`** или hex редактора. Основное использование `/dev/zero` - это создание инициализированного фиктивного файла заданной длины, используемого в качестве временного файла подкачки.

Пример 31-2. Настройка файла подкачки с помощью `/dev/zero`

```
#!/bin/bash
# Создание swap файла.

#  Файл подкачки предоставляет кэш для временного хранения данных
#+ и помогает ускорить определенные операции файловой системы.

ROOT_UID=0         # Root имеет $UID 0.
E_WRONG_USER=85    # Не root?

FILE=/swap
BLOCKSIZE=1024
MINBLOCKS=40
SUCCESS=0


# Этот сценарий должен запускаться из-под root.
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo; echo "Для запуска этого сценария нужно иметь права root."; echo
  exit $E_WRONG_USER
fi


blocks=${1:-$MINBLOCKS}          #  Устанавливается по умолчанию 40 блоков,
                                 #+ если иное не указано в командной
                                 #+ строке.
# Эквивалентно блоку команд ниже.
# -------------------------------------------------
# if [ -n "$1" ]
# then
#   blocks=$1
# else
#   blocks=$MINBLOCKS
# fi
# -------------------------------------------------


if [ "$blocks" -lt $MINBLOCKS ]
then
  blocks=$MINBLOCKS              # Должно быть не менее 40 блоков.
fi


###############################################################################
echo "Создание swap файла размером $blocks блоков (KB)."
dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks  # Обнуление файла.
mkswap $FILE $blocks             # Назначение его swap файлом.
swapon $FILE                     # Активация swap файла.
retcode=$?                       # Все работает?
#  Обратите внимание, что если одна или несколько из этих команд не
#+ выполняются, то возникают проблемы.
```

```
###########################################################################

#  Упражнение:
#  Перепишите  блок кода выше, так, что если он не
#+ выполнился успешно, то:
#    1) сообщение об ошибке выводилось бы в stderr,
#    2) все временные файлы очищались, а
#    3) сценарий упорядоченно завершался с соответствующим кодом
#+      ошибки.

echo "Swap файл создан и активирован."

exit $retcode
```

Еще одним применением `/dev/zero` является «обнуление» файла указанного размера для особых целей, например, монтирования файловой системы на устройство ***loopback*** (см. Пример 17-8) или «безопасного» удаления файла (см. Пример 16-61).

Пример 31-3. Создание ramdisk

```
#!/bin/bash
# ramdisk.sh

#  "ramdisk" является элементом системного ОЗУ
#+ который работает так, как если бы это была файловая система.
#  Его преимуществом является очень быстрый доступ (время чтения/записи).
#  Недостатком: нестабильность, потеря данных при перезагрузке или выключении,
#+             уменьшение доступной для системы оперативной памяти.
#
#  Какая польза от ramdisk?
#  Хранение большого набора данных, например, таблицы или словаря на
#+ ramdisk, ускоряет поиск данных, поскольку доступ к памяти осуществляется
#+ гораздо быстрее, чем доступ к диску.


E_NON_ROOT_USER=70             # Запускается с правами root.
ROOTUSER_NAME=root

MOUNTPT=/mnt/ramdisk           # Ранее, при помощи mkdir, создана /mnt/ramdisk.
SIZE=2000                      # Блоки по 2K (при необходимости измените)
BLOCKSIZE=1024                 # Размер блока 1K (1024 байт)
DEVICE=/dev/ram0               # Первое устройство ОЗУ

username=`id -nu`
if [ "$username" != "$ROOTUSER_NAME" ]
then
  echo "Нужно запускать с правами root \"`basename $0`\"."
  exit $E_NON_ROOT_USER
fi

if [ ! -d "$MOUNTPT" ]         #  Проверка наличия точки монтирования,
then                           #+ это не ошибка, если этот сценарий
  mkdir $MOUNTPT               #+ выполняется долго.
fi

###########################################################################
```

```
dd if=/dev/zero of=$DEVICE count=$SIZE bs=$BLOCKSIZE  # Обнуление ОЗУ.
                                                      # Почему это необходимо?
mke2fs $DEVICE                 # Создание файловой системы ext2.
mount $DEVICE $MOUNTPT         # Монтирование.
chmod 777 $MOUNTPT             # Разрешение обычным пользователям получать
                               #+ доступ к ramdisk.
                               # Но, отмонтировать его сможет только root.
###############################################################################
#  Необходимо проверять успешное завершение команд выше.
#+ Иначе возникнут проблемы.
#  Упражнение: измените этот сценарий, чтобы он был более безопасным.

echo "\"$MOUNTPT\" готово для использования."
#  Теперь ramdisk доступен  для хранения файлов даже обычным пользователям.

#  Осторожно, ramdisk нестабилен, а его содержимое исчезает при
#+ перезагрузке или выключении.
#  Копируйте все, что вы хотите сохранить, в основную директорию.

# После перезагрузки запустите этот сценарий, чтобы снова установить ramdisk.
# Перемонтировать /mnt/ramdisk без этих шагов не получится.

#  Соответствующее изменение, для автоматической установки ramdisk при
#+ загрузке, этот сценарий может вызываться из /etc/rc.d/rc.local.
#  Это может быть уместно, к примеру, на сервере баз данных.

exit 0
```

В дополнение ко всему выше, `/dev/zero` необходим двоичным файлам ***ELF*** (*Executable and Linking Format*) UNIX/Linux.

Глава 32. Отладка

> *Отладка в два раза сложнее, чем написание кода. Поэтому, если вы пишете код настолько искусный, насколько это возможно, то, по определению, вы не достаточно умны, чтобы отладить его.*
>
> *--Brian Kernighan*

Оболочка Bash не содержит встроенного отладчика, а только чистые команды и конструкции

отладки. Синтаксические ошибки или опечатки в сценариях создают сообщения об ошибках, которые часто не помогают при отладке не работающего сценария.

Пример 32-1. Сценарий с ошибками

```
#!/bin/bash
# ex74.sh

# Это сценарий с ошибками.
# Где эти ошибки?

a=37

if [$a -gt 27 ]
#  ^^            Нет пробела
then
  echo $a
fi

exit $?   # 0! Почему?
```

Вывод сценария:

```
./ex74.sh: [37: command not found
```

Что же случилось со сценарием выше? Подсказка: после оператора ***if***.

Пример 32-2.Отсутствующее ключевое слово

```
#!/bin/bash
# missing-keyword.sh
# Какое сообщение об ошибке выдаст этот сценарий? И почему?

for a in 1 2 3
do
  echo "$a"
# done     # Нужно ключевое слово «done», закомментированное в строке 8.

exit 0     # Здесь не выйдем!

# === #

# Из командной строки, после завершения сценария:
  echo $?    # 2
```

Вывод из сценария:

```
missing-keyword.sh: line 10: syntax error: unexpected end of file (неожидаемый
конец файла)
```

Обратите внимание, что сообщение об ошибке ссылается не обязательно на строку, где произошла ошибка, а на строку, на которой интерпретатор Bash, наконец, осознал ошибку.

Сообщения об ошибках могут игнорировать строки комментариев в сценарии, сообщая номер строки синтаксической ошибки.

Что делать, если сценарий выполняется, но работает не как ожидается? Это всем очень хорошо знакомая логическая ошибка.

Пример 32-3. *test24*: другой сценарий с ошибками

```
#!/bin/bash

#  Этот сценарий должен удалить все имена содержащие пробелы
#+ в текущей директории.
#  Он не работает.
#  Почему не работает?


badname=`ls | grep ' '`

# Попробуйте так:
# echo "$badname"

rm "$badname"

exit 0
```

Узнайте, что произойдет с Примером 32-3 раскомментировав строку *echo "$badname"*. *Echo* полезно для того, что бы увидеть то ли, что вы ожидали, вы получаете на самом деле.

В данном конкретном случае, **rm «$badname»** не даст желаемых результатов, потому что $badname *не должно быть заключено в кавычки*. Помещение его в кавычки гарантирует, что **rm** имеет только один аргумент, (т.е. соответствует только одному имени файла). Частично эта проблема решается удалением кавычек $badname и сбросом $IFS, что бы содержались только строки, **IFS $'\n'**. Тем не менее, есть более простые способы для этого.

```
# Правильные способы удаления файловых имен содержащих пробелы.
rm *\ *
rm *" "*
rm *' '*
# Спасибо S.C.
```

Обобщенные признаки сценария с ошибками:

1. Бомбит сообщениями «syntax error» или
2. Запускается, но не работает как ожидалось (логическая ошибка).
3. Запускается, работает как ожидалось, но есть неприятные побочные эффекты (логическая бомба).

Инструментами для отладки не работающих сценариев являются:

1. Помещение ***echo*** в критических точках сценария для отслеживания переменных, а кроме того для получения моментального снимка происходящего.

Еще лучше **-echo**, которое производит вывод на экран только во время *отладки*.

```
### debecho (отладочное-echo), Stefano Falsetto ###
### Выводит передаваемые параметры, только если  установлено значение
DEBUG. ###
debecho () {
  if [ ! -z "$DEBUG" ]; then
     echo "$1" >&2
     #         ^^^ в stderr
  fi
}

DEBUG=on
Whatever=whatnot
debecho $Whatever   # whatnot

DEBUG=
Whatever=notwhat
debecho $Whatever   # (Не будет выводить на экран.)
```

2. Использование фильтра **tee** для проверки процессов или потоков данных в критических точках.

3. Установка флагов опций `-n -v -x`

   sh -n scriptname проверяет синтаксические ошибки без фактического запуска сценария. Это равнозначно помещению в сценарий **set -n** или **set -o noexec**. Обратите внимание, что некоторые виды синтаксических ошибок могут проскользнуть мимо этой проверки.

   sh -v scriptname выводит на экран каждую команду перед ее выполнением. Эквивалентно помещению в сценарий **set -v** или **set -o verbose**.

   Флаги `-n` и `-v` отлично работают вместе. **sh -nv scriptname** производит подробную синтаксическую проверку.

   sh -x scriptname выводит на экран результат каждой команды, но в сокращенном виде. Равносильно помещению в сценарий **set -x** или **set -o xtrace**.

   Помещение в сценарий **set -u** или **set -o nounset** запускает его, но дает бессвязное сообщение об ошибке переменной и прерывает сценарий.

```
set -u   # Или   set -o nounset

# Установка переменной в null не вызывает ошибки/отмены.
# unset_var=

echo $unset_var   # Не установленная (и не объявленная) переменная.

echo "Нет вывода!"
```

```
# sh t2.sh
# t2.sh: line 6: unset_var: unbound variable (бессвязная переменная)
```

4. Проверка переменной или условия в критических точках сценария с помощью функции «***assert***». (Эта мысль пришла из Си.)

Пример 32-4. Проверка условий при помощи *assert*

```
#!/bin/bash
# assert.sh

##########################################################################
assert ()                 #  Если условие false,
{                         #+ то выход из сценария
                          #+ с выводом сообщения об ошибке.
  E_PARAM_ERR=98
  E_ASSERT_FAILED=99


  if [ -z "$2" ]          #  Не хватает передаваемых параметров
  then                    #+ функции assert().
    return $E_PARAM_ERR   #  Не будет лишним.
  fi

  lineno=$2

  if [ ! $1 ]
  then
    echo "Assertion failed:  \"$1\""
    echo "Файл \"$0\", строка $lineno"    #  Задаем имя файла
                                          #+ и номер строки.
    exit $E_ASSERT_FAILED
  # else
  #   return
  #   и продолжение исполнения сценария.
  fi
}
#  Аналогично, функцию assert() помещают в сценарий, который необходимо
#+ отладить.
##########################################################################


a=5
b=4
condition="$a -lt $b"     #  Сообщение об ошибке и выход из сценария.
                          #  Попробуйте установить «условием» что-то еще
                          #+ и посмотрим, что получится.

assert "$condition" $LINENO
#  Остальная часть сценария выполняется только в том случае, если
#+ «assert» не терпит неудачу.

# Какие-то команды.
# Какие-то другие команды ...
echo "Вывод на экран только в случае, если \"assert\" не неудачна."
# ...
# Другие команды ...
```

```
exit $?
```

5. Использование переменной **$LINENO** и встроенного ***caller***.
6. Перехват на выходе.

   Команда ***exit*** в сценарии вызывает сигнал 0, завершая процесс, то есть, непосредственно сценарий. [1] Зачастую полезно перехватывать *exit*, заставляя, к примеру, «выводить» переменные. ***trap*** должна быть первой командой в сценарии.

Перехват сигналов

trap

Указывает действие при получении сигнала; полезно для отладки.

> Сигнал - это сообщение посылаемое процессом, либо ядром или другим процессом, которое рассказывает о некотором заданном действии (обычно прекратить). Например нажатие CTRL-C отправляет сигнал INT, т.е. прерывание пользователем запущенной программы.

Простой случай:

```
trap '' 2
# Игнорирование прерывания 2 (Control-C), без определенного действия.

trap 'echo "Control-C отключен."' 2
# Сообщение при нажатии Control-C.
```

Пример 32-5. Перехват exit

```
#!/bin/bash
# Охота за переменными при помощи trap.

trap 'echo Список Переменных --- a = $a  b = $b' EXIT
#  EXIT это имя сигнала, создаваемого после выхода из сценария.
#
#  Команда, указывает «trap» не выполняться до тех пор, пока
#+ не отправлен соответствующий сигнал.

echo "Это выводится до \"trap\" --"
echo "даже несмотря на то, что сценарий сначала видит \"trap\"."
echo

a=39

b=36
```

```
exit 0
#  Обратите внимание, что даже закомментировав команду «exit» не будет никакой
#+ разницы, сценарий завершится в любом случае после запуска команд.
```

Пример 32-6. Очистка после Control-C

```
#!/bin/bash
# logon.sh: Быстрый и черновой сценарий для проверки нахождения в он-лайн.

umask 177  # Убедитесь, что временные файлы не доступны для чтения.


TRUE=1
LOGFILE=/var/log/messages
#  Обратите внимание, что $LOGFILE должен быть читаемым
#+ (для root, chmod 644 /var/log/messages).
TEMPFILE=temp.$$
#  Создаем "уникальное" имя временного файла, с помощью id процесса сценария.
#     Используйте 'mktemp', как альтернативу.
#     Например:
#     TEMPFILE=`mktemp temp.XXXXXX`
KEYWORD=address
#  При входе в систему, строка "remote IP address xxx.xxx.xxx.xxx"
#  добавляется в /var/log/messages.
ONLINE=22
USER_INTERRUPT=13
CHECK_LINES=100
#  Количество строк лог-файла для проверки.

trap 'rm -f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
#  Очистка временного файла, если сценарий прерывается Control-C.

echo

while [ $TRUE ]  # Бесконечный цикл.
do
  tail -n $CHECK_LINES $LOGFILE> $TEMPFILE
  #  Сохраняет последние 100 строк системного лог-файла как временный файл.
  #  Необходимо, так как новые ядра создают много сообщений в журнале.
  search=`grep $KEYWORD $TEMPFILE`
  #  Проверка наличия фразы "IP address",
  #+ для указания успешного входа.

  if [ ! -z "$search" ] #  Кавычки необходимы т.к. возможны пробелы.
  then
     echo "On-line"
     rm -f $TEMPFILE    #  Очистка временного файла.
     exit $ONLINE
  else
     echo -n "."        #  Опция -n в echo подавляет новую строку,
                        #+ Таким образом вы получите непрерывный ряд точек.
  fi

  sleep 1
done


#  Внимание: если вы измените переменную trap KEYWORD в "Exit",
#+ этот сценарий может использоваться для
```

```
#+ проверки неожиданного выхода из он-лайн.

# Упражнение: Измените сценарий, в соответствии с предупреждением выше,
#           и улучшите его.

exit 0


# Nick Drage предложил другой способ:

while true
  do ifconfig ppp0 | grep UP 1> /dev/null && echo "соединено" && exit 0
  echo -n "."   # Печатает точки (.....) до соединения.
  sleep 2
done

#  Проблема: Нажатия CTRL-C для прекращения этого процесса может быть
#+ недостаточно (Точки могут продолжать выводиться на экран.)
#  Упражнение: Исправьте это.


# Stephane Chazelas предложил другую альтернативу:

CHECK_INTERVAL=1

while ! tail -n 1 "$LOGFILE" | grep -q "$KEYWORD"
do echo -n .
   sleep $CHECK_INTERVAL
done
echo "On-line"

#  Упражнение: Обсудите относительные преимущества и недостатки
#+ каждого из этих различных подходов.
```

Пример 32-7. Простая реализация индикатора

```
#! /bin/bash
# progress-bar2.sh
# Автор: Graham Ewart (С переформатированием автором ABS Guide).
# Используется в ABS Guide с разрешения (спасибо!).

# Этот сценарий вызывается в bash. Не работает и в sh.

interval=1
long_interval=10

{
     trap "exit" SIGUSR1
     sleep $interval; sleep $interval
     while true
     do
       echo -n '.'     # Используются точки.
       sleep $interval
     done; } &         # Запуск индикатора в фоновом режиме.

pid=$!
trap "echo !; kill -USR1 $pid; wait $pid"  EXIT        # Для обработки ^C.

echo -n 'Длительный процесс '
sleep $long_interval
```

```
echo ' Окончено!'

kill -USR1 $pid
wait $pid               # Остановка индикатора.
trap EXIT

exit $?
```

Аргумент DEBUG в **trap** заставляет указанное действие выполняться после каждой команды в сценарии. Это, к примеру, позволяет трассировать переменные.

Пример 32-8.Трассировка переменной

```
#!/bin/bash

trap 'echo "VARIABLE-TRACE> \$variable = \"$variable\""' DEBUG
#  Вывод на экран значения переменной $variable после
#+ каждой команды.

variable=29; line=$LINENO

echo "  Просто инициализация \$variable в $variable в строке номер
$line."

let "variable *= 3"; line=$LINENO
echo "Просто умножение \$variable на 3 в строке номер $line."

exit 0

#  Конструкция «trap ' команда1... команда2...'  DEBUG» является
#+ более  уместной в контексте сложного сценария, где вставка
#+ нескольких заявлений "echo $variable"  может быть неудобной
#+ и длительной.

# Спасибо за пояснения Stephane Chazelas.


Вывод сценария:

VARIABLE-TRACE> $variable = ""
VARIABLE-TRACE> $variable = "29"
  Just initialized $variable to 29.
VARIABLE-TRACE> $variable = "29"
VARIABLE-TRACE> $variable = "87"
  Just multiplied $variable by 3.
VARIABLE-TRACE> $variable = "87"
```

Конечно, команда **trap** имеет и другие применения, помимо отладки, например отключение определенных клавиш в сценарии (см. Пример A-43).

Пример 32-9. Выполнение нескольких процессов (в SMP box)

```
#!/bin/bash
# parent.sh
# Выполнение нескольких процессов (в SMP box).
# Автор: Tedman Eng

#  Это первый из двух сценариев,
#+ оба сценария должны находиться в текущем рабочем каталоге.


LIMIT=$1         # Начальное общее число процессов
NUMPROC=4        # Количество одновременных потоков (форков?)
PROCID=1         # ID Начального процесса
echo "Мой PID это $$"

function start_thread() {
        if [ $PROCID -le $LIMIT ] ; then
                ./child.sh $PROCID&
                let "PROCID++"
        else
           echo "Достигнут предел."
           wait
           exit
        fi
}

while [ "$NUMPROC" -gt 0 ]; do
        start_thread;
        let "NUMPROC--"
done


while true
do

trap "start_thread" SIGRTMIN

done

exit 0



# ======== Второй, следующий, сценарий ========


#!/bin/bash
# child.sh
# Запускаем несколько процессов в SMP box.
# Этот сценарий называется parent.sh.
# Автор: Tedman Eng

temp=$RANDOM
index=$1
shift
let "temp %= 5"
let "temp += 4"
echo "Начало $index  Время:$temp" "$@"
sleep ${temp}
echo "Завершение $index"
kill -s SIGRTMIN $PPID
```

```
exit 0


# ======================= ПРИМЕЧАНИЯ АВТОРА СЦЕНАРИЯ ======================= #
#  Не полностью свободен от ошибок.
#  Я запускал его с лимитом =500 и после первых нескольких сотен итераций,
#+ один из параллельных потоков исчез!
#  Не уверен, что это из-за столкновения сигналов trap или чего-то еще.
#  После того, как trap получает, обработка выполняется быстро, после чего
#+ производится следующий trap. В это время возможно пропустить сигнал trap,
#+ т.е. пропустить порождение дочернего процесса.

#  Без сомнения кто-то заметит ошибку и перепишет
#+ ... в будущем.

# ========================================================================== #

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

##############################################################################
# Следующий оригинальный сценарий написанный Vernia Damiano.
# К сожалению, он так же нормально не работает.
##############################################################################

#!/bin/bash

#  Нужно вызывать сценарий хотя бы с одним целочисленным параметром
#+ (количеством одновременных процессов).
#  Все остальные параметры передаются через запущенные процессы.


INDICE=8        # Всего запускаемых процессов
TEMPO=5         # Максимальное время сна каждого процесса
E_BADARGS=65    # Если сценарию не передан аргумент(ы).

if [ $# -eq 0 ] #  Проверка наличие хотя бы одного аргумента, передаваемого
                #+ сценарию.
then
  echo "Bcgjkmpeqnt: `basename $0` число_процессов [передаваемые параметры]"
  exit $E_BADARGS
fi

NUMPROC=$1              # Количество одновременных процессов
shift
PARAMETRI=( "$@" )      # Параметры каждого процесса

function avvia() {
         local temp
         local index
         temp=$RANDOM
         index=$1
         shift
         let "temp %= $TEMPO"
         let "temp += 1"
         echo "Начало $index Время:$temp" "$@"
         sleep ${temp}
         echo "Окончание $index"
         kill -s SIGRTMIN $$
}

function parti() {
```

```
        if [ $INDICE -gt 0 ] ; then
             avvia $INDICE "${PARAMETRI[@]}" &
               let "INDICE--"
        else
               trap : SIGRTMIN
        fi
}

trap parti SIGRTMIN

while [ "$NUMPROC" -gt 0 ]; do
        parti;
        let "NUMPROC--"
done

wait
trap - SIGRTMIN

exit $?

: <<КОММЕНТАРИИ_АВТОРА_СЦЕНАРИЯ
Мне нужно было запустить программу с помощью SMP машины с заданными параметрами
в виде нескольких различных файлов. Поэтому я подумал [и сделал] держать
запущенными указанное количество процессов и запускать новый каждый раз...,
когда предыдущий заканчивается.

Инструкция «wait» не помогает, поскольку она ждет выполнения данного процесса
или *всех* процессов запущенных в фоновом режиме. Поэтому я написал [этот] bash
сценарий который может проделать эту работу используя инструкцию «trap».
  --Vernia Damiano
КОММЕНТАРИИ_АВТОРА_СЦЕНАРИЯ
```

trap '' SIGNAL (два смежных апострофа) отключает SIGNAL для оставшейся части сценария . **trap SIGNAL** повторяет действие SIGNAL еще раз. Это полезно для защиты критической части сценария от нежелательных прерываний.

```
        trap '' 2  # Сигнал 2 это Control-C, теперь отключен.
        команда
        команда
        команда
        trap 2     # Возобновляет Control-C
```

Версия 3 Bash добавляет следующие внутренние переменные используемые отладчиком.

1. $BASH_ARGC

   Количество аргументов командной строки, передаваемых сценарию, похожа на **$#**.

2. $BASH_ARGV

Последний параметр командной строки, переданный сценарию, эквивалентна **${!#}**

3. `$BASH_COMMAND`

   Выполняемая текущая команда.

4. `$BASH_EXECUTION_STRING`

   Опции строки после опции `-c` в Bash.

5. `$BASH_LINENO`

   В функции, указывает номер строки вызова функции.

6. `$BASH_REMATCH`

   Массив переменных ассоциируемый с **=~** , условным регулярным соответствием.

7. `$BASH_SOURCE`

   Это имя сценария, обычно, то же, что и **$0**.

8. `$BASH_SUBSHELL`

Примечания

[1] По соглашению, сигнал 0 присваивается выходу.

Глава 33. Опции

Варианты **set**, которые изменяют оболочку и/или поведение сценария.

Команда **set** включает опции в сценарии. В том месте сценария, где вы хотите, что бы опции вступили в силу, используйте ***set -o название-опции*** или в краткой форме,

set -сокращенная-опция. Эти две формы одинаковы.

```
#!/bin/bash

set -o verbose
# Выводит на экран все команды перед их выполнением.


#!/bin/bash

set -v
# Такой же эффект, что и выше.
```

Отключение опций в сценарии производится ***set +o имя-опции*** или

set +сокращенная-опция.

```
#!/bin/bash

set -o verbose
# Команды выводятся на экран.
команда
...
команда

set +o verbose
# Команды перестают выводиться на экран.
команда
# Не выводится.


set -v
# Команды выводятся на экран.
команда
...
команда

set +v
# Команды перестают выводиться на экран.
команда

exit 0
```

Другим способом включения опций в сценарии является указание их сразу после заголовка сценария ***#!*** .

```
#!/bin/bash -x
#
# Тело сценария.
```

Также можно включать опции сценария из командной строки. Некоторые опции, которые не работают с ***set***, будут, таким образом, доступны. Среди них *-i*, заставляющая сценарий запуститься в интерактивном режиме.

bash -v script-name

bash -o verbose script-name

Ниже приведен список некоторых полезных опций. Они могут быть указаны в любой сокращенной форме (должно предшествовать **одно тире**) или полным именем (должно предшествовать **два *тире*** или **-o**).

Таблица 33-1. Опции Bash

Аббревиатура	Имя	Эффект
`-B`	`brace expansion`	Включить *расширение в скобках* (настройка по умолчанию = включено)
`+B`	`brace expansion`	Отключение расширения в скобках
`-C`	`noclobber`	Запретить перезапись файлов перенаправлением (могут быть переопределены >|)
`-D`	(нет)	Список строк в двойных кавычках, отмеченных символом `$`, а не выполнение команды в сценарии
`-a`	`allexport`	Экспорт всех определенных переменных
`-b`	`notify`	Уведомление о прекращении заданий, запущенных в фоновом режиме (редко используется в сценариях)
`-c ...`	(нет)	Чтение команд из **...**
`checkjobs`		Информирует пользователя о любых открытых заданиях при выходе из оболочки. Введено в версии 4 Bash, и до сих пор 'экспериментальное'. *Используйте:* `shopt -s checkjobs` (*Осторожно*: зависает!)
`-e`	`errexit`	Прерывает сценарий при первой ошибке, когда команда завершается с не нулевым статусом (за исключением циклов ***until*** или ***while***, сравнений ***if***, конструкции **list**)
`-f`	`noglob`	Отключение расширения имен файлов (символами подстановки)
`globstar`	`globbing star-match`	Включает оператор подстановки ** (версия 4 Bash). *Используйте:* `shopt -s globstar`
`-i`	`interactive`	Сценарий запускается в интерактивном режиме
`-n`	`noexec`	Читать команды в сценарии, но не запускать их (проверка синтаксиса)
`-o Option-Name`	(нет)	Вызов опции *`Option-Name`*
`-o posix`	`POSIX`	Изменяет поведение Bash или вызываемого сценария в соответствии со стандартом POSIX.
`-o pipefail`	`pipe failure`	Вызывает возвращение статуса выхода последней команды в конвейере, которая возвращает не нулевое возвращаемое значение.
`-p`	`privileged`	Сценарий запускается под "**`suid`**" (Внимание!)
`-r`	`restricted`	Сценарий запускается в *ограниченном* режиме (см. Главу 22).
`-s`	`stdin`	Чтение команд из `stdin`
`-t`	(нет)	Выход после первой команды
`-u`	`nounset`	Попытка использования неопределенной переменной выведет сообщение об ошибке и принудительный выход
`-v`	`verbose`	Вывод каждой команды в `stdout` перед ее выполнением
`-x`	`xtrace`	Подобно `-v`, но более расширенно
`-`	(нет)	Флаг окончания опции. Все остальные аргументы

Аббревиатура	Имя	Эффект
		являются ***позиционными параметрами***.
`--`	(нет)	Отключенные позиционные параметры. Если заданы аргументы (`--arg1 arg2`), позиционные параметры становятся аргументами.

Глава 34. Gotchas

Turandot: Gli enigmi sono tre, la morte una!

Caleph: No, no! Gli enigmi sono tre, una la vita!

--Puccini

Вот некоторые практические приемы сценариев (**не рекомендуемые!**), которые скрашивают скучную жизнь.

- Присвоение именам переменных зарезервированных слов или символов.

```
case=value0       # Проблема.
```

```
23skidoo=value1   # Так же проблемно.
#  Имена переменных, начинающиеся цифрами, зарезервированы оболочкой.
#  Попробуем _23skidoo=value1. Начинать имя переменной с подчеркивания -
#+ нормально.

# Однако ...   использование только подчеркивания не будет работать.
_=25
echo $_           # $_ это специальная переменная присваиваемая
                  #+ последнему аргументу последней команды.
# Но ...       _ это правильное название функции!

xyz((!*=value2    # Вызовет серьезные проблемы.
# Начиная с версии 3 Bash, в именах переменных не разрешается.
```

- Использование дефиса или других зарезервированных символов в имени переменной (или имени функции).

```
var-1=23
# Вместо этого используйте 'var_1'.

function-whatever ()   # Ошибка
# Вместо этого используйте 'function_whatever ()'.


# Начиная с версии 3 Bash, в именах функций не разрешается.
function.whatever ()   # Ошибка
# Вместо этого используйте 'functionWhatever ()'.
```

- Использование одинакового имени для переменной и функции. Это делает сценарий трудно понимаемым.

```
do_something ()
{
  echo "Эта функция что-то делает с \"$1\"."
}

do_something=do_something

do_something do_something

# Все это допустимо, но весьма запутанно.
```

- Неправильное использование пробелов. В отличие от других языков программирования, Bash может быть весьма чувствителен к пробелам.

```
var1 = 23   # Правильно 'var1=23'.
# В строке выше, Bash попытается выполнить команду "var1"
# с аргументами "=" и "23".

let c = $a - $b # Замените на:  let c=$a-$b  или  let "c = $a - $b"
```

```
if [ $a -le 5]  # if [ $a -le 5 ]   будет правильно.
#           ^^    if [ "$a" -le 5 ]   даже лучше.
                # [[ $a -le 5 ]] то же работает.
```

- Не завершение последней команды блока кода в фигурных скобках ***точкой с запятой***.

```
{ ls -l; df; echo "Done." }
# bash: синтаксическая ошибка: неожиданное окончание файла

{ ls -l; df; echo "Done."; }
#                          ^     ### После последней команды нужна точка
#+ с запятой.
```

- Предположим, что не инициализированные переменные (переменные, которым до этого было присвоено значение) «обнуляются». Не инициализированная переменная имеет значение ***null***, а не ноль.

```
#!/bin/bash

echo "uninitialized_var = $uninitialized_var"
# uninitialized_var =

# Однако ...
# если  $BASH_VERSION ≥ 4.2; то

if [[ ! -v uninitialized_var ]]
then
  uninitialized_var=0   # Инициализирована нулем!
fi
```

- Смешивание **=** и ***-eq*** в сравнении. Запомните, **=** для сравнения ***строковых*** переменных, а ***-eq*** для ***целых чисел***.

```
if [ "$a" = 273 ]      # $a целое число или строка?
if [ "$a" -eq 273 ]    # Если целое число.

# Иногда можно заменять -eq и = без отрицательных последствий.
# Однако ...


a=273.0   # Не целое число.

if [ "$a" = 273 ]
then
  echo "Сравнение сработало."
else
  echo "Сравнение не работает."
fi    # Сравнение не сработало.

# То же самое с a=" 273" и a="0273".
```

```
#  Кроме того, возникают проблемы при попытке использования '-eq' с
#+ нецелым значением.

if [ "$a" -eq 273.0 ]
then
  echo "a = $a"
fi  # Прервано с сообщением об ошибке.
# test.sh: [: 273.0: Ожидается целочисленное выражение
```

- Неправильное использование операторов сравнения строк.

Пример 34-1. Сравнение чисел и строк не эквивалентны

```
#!/bin/bash
# bad-op.sh: Попытка использовать строковое сравнение целых чисел.

echo
number=1

#  Следующий цикл while содержит две ошибки:
#+ одна вопиющая, а другая коварная.

while [ "$number" < 5 ]    #  Не правильно! Должно быть:
                           #+ while [ "$number" -lt 5 ]
do
  echo -n "$number "
  let "number += 1"
done
#  Попытка запустить эту бомбу с сообщением об ошибке:
#+ bad-op.sh: line 10: 5: Нет такого файла или директории
#  В одиночных скобках для сравнения целых чисел, "<" должно быть
#+ экранировано, даже тогда, когда это ошибочно.

echo "----------------------"

while [ "$number" \< 5 ]    #  1 2 3 4
do                          #
  echo -n "$number "        #  Это *кажется*  работает, но ...
  let "number += 1"         #+ на самом деле делается сравнение ASCII,
done                        #+ а не числовое.

echo; echo "----------------------"

# Это может вызывать проблемы. Например:

lesser=5
greater=105

if [ "$greater" \< "$lesser" ]
then
  echo "$greater меньше, чем $lesser"
fi                          # 105 меньше, чем 5
#  На самом деле, в строковом сравнении (порядок сортировки ASCII),
#+ "105" на самом деле меньше, чем "5".

echo
```

```
exit 0
```

- Попытка использовать **let** для установки строковых переменных.

```
let "a = hello, you"
echo "$a"   # 0
```

- Иногда переменные внутри квадратных скобок (**[ ]**) «сравнения» должны быть в кавычках (двойных кавычках). Невыполнение этого требования может привести к непредвиденному поведению. См. Пример 7-6, Пример 20-5 и Пример 9-6.
- Заключение в кавычки переменной, содержащей пробелы, предотвращает разделение. Иногда это приводит к непредвиденным последствиям.
- Команды, запущенные из сценария могут не выполняться, потому что владельцу сценария не хватает прав на их выполнение. Если пользователь не может вызвать команду из командной строки, то не применяйте ее в сценарии. Попробуйте изменить атрибуты команды, возможно даже установив бит `suid` (как ***root***, конечно).
- Попытка использования **–** (тире), как оператора перенаправления (каковым оно не является), может привести к неприятным сюрпризом.

```
команда1 2> - | команда2
# Попытка перенаправления вывода ошибки команды1 в конвейер ...
# ... работать не будет.

команда1 2>& - | команда2  # Также бесполезно.

Спасибо S.C.
```

- Использование функциональности Bash версии 2 может помочь с сообщениями об ошибках. Старые машины Linux могут иметь версии Bash 1.xx, как установленные по умолчанию.

```
#!/bin/bash

minimum_version=2
# Так Chet Ramey постоянно добавляет новые возможности в Bash,
# вы можете установить $minimum_version в 2.XX, 3.XX, или необходимую.
E_BAD_VERSION=80

if [ "$BASH_VERSION" \< "$minimum_version" ]
then
  echo "Этот сценарий работает только с версией Bash, $minimum или выше."
  echo "Настоятельно рекомендуем обновиться."
  exit $E_BAD_VERSION
fi
```

```
...
```

- Использование функций специфичных для Bash в сценарии Bourne shell (***#!/bin/sh***) на не Linux машине может привести к непредсказуемости. Система Linux, как правило, подразумевает, что **sh** это ***bash***, но это не всегда верно для большинства UNIX машин.
- Использование недокументированных возможностей Bash оказывается опасной практикой. В предыдущих версиях этой книги было несколько сценариев, которые зависели от '*фич*', когда максимальное значение значений **exit** или **return** было 255, пределом, который не распространялся на *отрицательные* целые числа. К сожалению, в версии 2.05b и более поздних, что лазейка исчезла. Смотрите Пример 24-9.
- В некоторых контекстах, вводящих в заблуждение, ***статус выхода*** может возвращаться. Это может произойти при установке ***локальной переменной в пределах функции*** или при назначении ***арифметического значения в переменную***.
- ***Статус выхода арифметического выражения*** не равнозначен *ошибке кода*.

```
var=1 && ((--var)) && echo $var
#        ^^^^^^^^^ Здесь список заканчивается статусом выхода 1.
#                     $var не выведется на экран!
echo $?   # 1
```

- Сценарий, с новыми типами строк DOS (*\r\n*), не удастся выполнить, поскольку **#!/bin/bash\r\n** это не понимает, это *не* то же самое, ожидаемое в **#!/bin/bash\n**. Исправлением будет преобразование сценария в стиль новых строк UNIX.

```
#!/bin/bash

echo "Здесь"

unix2dos $0    # Сценарий изменяет себя в формат DOS.
chmod 755 $0   # Возвращаем обратно права на выполнение.
               # Команда 'unix2dos' удаляет права на исполнение.

./$0           # Пробуем перезапустить сценарий.
               # Но запустить как DOS файл не получается.

echo "Там"

exit 0
```

- Заголовок сценария оболочки **#!/bin/sh** не будет работать в режиме полной

совместимости Bash. Некоторые, специфичные для Bash функции, могут оказаться недоступными. Сценарий, которому требуется полный доступ ко всем специфичным для Bash расширениям, должен начинаться с **#!/bin/bash**.

- Ввод пробелов, перед ограничением завершающей строки ***here document***, вызовет неожиданное поведение сценария.
- Ввод, более чем одного, заявления **echo** в функцию, если ее вывод захватывается (передается).

```
add2 ()
{
  echo "Неважно ... "   # Удалите эту строку!
  let "retval = $1 + $2"
    echo $retval
    }

    num1=12
    num2=43
    echo "Сумма $num1 и $num2 = $(add2 $num1 $num2)"

#   Сумма 12 и 43 = Неважно ...
#   55

#   "Вывод на экран" объединился.
```

Это не работает.

- Сценарий не может ***экспортировать*** переменные в **родительский** процесс, оболочку или окружающую среду. Так же, как мы знаем из биологии: процесс потомок может наследовать от родителей, а не наоборот.

```
WHATEVER=/home/bozo
export WHATEVER
exit 0
```

```
bash$ echo $WHATEVER
Исправлением будет преобразование сценария в стиль новых строк UNIX.
bash$
```

Конечно же, возвращаясь к командной строке, $WHATEVER осталась незаданной.

- Установка переменных и управление ими в ***subshell***, которая пытается использовать те же самые переменные за пределами *subshell*, приведет к неприятным сюрпризам.

Пример 34-2. Ловушки Subshell

```
#!/bin/bash
# Ловушки переменных в subshell.

outer_variable=outer
echo
echo "outer_variable = $outer_variable"
echo

(
```

```
# Начало subshell

echo "outer_variable внутри subshell = $outer_variable"
inner_variable=inner  # Установка
echo "inner_variable внутри subshell = $inner_variable"
outer_variable=inner  # Значение изменится глобально?
echo "outer_variable внутри subshell = $outer_variable"

# Есть разница в экспортировании?
#    export inner_variable
#    export outer_variable
# Попробуй и увидишь.

# Конец subshell
)

echo
echo "inner_variable вне subshell = $inner_variable"
                                                    # Не установлено.
echo "outer_variable вне subshell = $outer_variable"  # Не изменилось.
echo

exit 0

# Что случится, если раскомментировать строки 19 и 20?
# Будет разница?
```

- Перенаправление вывода **echo** в ***read*** может привести к непредсказуемым результатам. В этом случае ***read*** работает так, как если бы она была запущена в ***subshell***. Вместо этого используйте команду ***set*** (как в Примере 15-18).

Пример 34-3. Передача вывода *echo* в *read*

```
#!/bin/bash
#  badread.sh:
#  Попытка использования 'echo и 'read'
#+ для не интерактивного присвоения переменных.

#   shopt -s lastpipe

a=aaa
b=bbb
c=ccc

echo "one two three" | read a b c
# Попробуем переназначить a, b, и c.

echo
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
# Переназначение неверно.

### Однако ...
##  Если раскомментировать строку 6:
#   shopt -s lastpipe
##+ то проблемы будут исправлены!
```

```
### Это новая фича Bash версии 4.2.

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

# Попробуем следующую альтернативу.

var=`echo "one two three"`
set -- $var
a=$1; b=$2; c=$3

echo "-------"
echo "a = $a"  # a = one
echo "b = $b"  # b = two
echo "c = $c"  # c = three
# Переназначение успешно.

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

#  Обратите внимание, что вывод echo в «read» работает в subshell.
#  Но значение переменной изменяется *только* в subshell.

a=aaa          # Начнем все сначала.
b=bbb
c=ccc

echo; echo
echo "one two three" | ( read a b c;
echo "Внутри subshell: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
# a = one
# b = two
# c = three
echo "-----------------"
echo "Вне subshell: "
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
echo

exit 0
```

На самом деле, как указывает Anthony Richardson, туннелирование в *любой* цикл может вызвать аналогичные проблемы.

```
# Неприятности с перенаправлением цикла.
#  Пример Anthony Richardson,
#+ с добавлениями Wilbert Berendsen.


foundone=false
find $HOME -type f -atime +30 -size 100k |
while true
do
   read f
   echo "$f более 100 КБ и к ней не обращались в течение 30 дней"
   echo "Рассмотрите возможность перемещения файла в архив."
   foundone=true
   # -----------------------------------
     echo "Subshell level = $BASH_SUBSHELL"
```

```
   # Уровень Subshell = 1
   # Да, мы в subshell.
   # ----------------------------------
done

#  найденное здесь всегда будет ложным, так как оно имеет значение true
#+ внутри subshell
if [ $foundone = false ]
then
   echo "Нет файлов для архивирования."
fi

# ====================Теперь правильный способ:================

foundone=false
for f in $(find $HOME -type f -atime +30 -size 100k)  # Канала нет.
do
   echo "$f более 100 КБ и к ней не обращались в течение 30 дней"
   echo "Рассмотрите возможность перемещения файла в архив."
   foundone=true
done

if [ $foundone = false ]
then
   echo "Нет файлов для архивирования."
fi

# ==================И другой вариант==================

#  Содержит часть сценария, которая считывает переменные в блоке кода, а
#+ поэтому они отделены от subshell.
#  Спасибо W.B.

find $HOME -type f -atime +30 -size 100k | {
     foundone=false
     while read f
     do
       echo "$f более 100 КБ и к ней не обращались в течение 30 дней"
       echo "Рассмотрите возможность перемещения файла в архив."
       foundone=true
     done

     if ! $foundone
     then
       echo "Нет файлов для архивирования."
     fi
}
```

- Подобная проблема возникает при попытке записи вывода **`tail -f`** конвейером в **`grep`**.

```
tail -f /var/log/messages | grep "$ERROR_MSG" >> error.log
#  В файл "error.log" ничего не будет записано.
#  Как указывает Samuli Kaipiainen, это результат буферизации

#+ вывода из grep.
#  Исправьте, добавив параметр "--line-buffered" в grep.
```

- Использование команд «**suid**» внутри сценариев является рискованным, так как может привести к нарушению безопасности. [1]
- Использование сценариев shell для программирования CGI может быть проблематичным. Переменные сценариев оболочки не 'типосохраняемые', а это может привести к нежелательным последствиям. Кроме того, это проблематично в сценариях «cracker-proof».
- Bash не справляется корректно со строкой содержащей двойной слэш (**//**) .
- Сценарии Bash, написанные для Linux или BSD, возможно, потребуют исправлений при их запуске на коммерческой UNIX машине. Такие сценарии часто используют набор команд GNU и фильтров, которые имеют большую функциональность, чем их общие с UNIX аналоги. Это особенно верно в отношении таких утилит обработки текста, как **tr**.
- К сожалению, обновления Bash сами нарушают старые сценарии, которые работали прекрасно. Давайте вспомним, как опасно использовать недокументированные функции Bash.

Опасность около тебя -

Остерегайтесь, остерегайтесь,
остерегайтесь, остерегайтесь.

Многие отважные сердца в глубине спят.

Так будьте осторожны -

Остерегайтесь.

--A.J. Lamb and H.W. Petrie

Примечания

[1] Установка бита **suid** на сам сценарий не имеет никакого эффекта в Linux и большинстве других UNIX.

Глава 35. Пишем стильные сценарии

Имейте привычку писать сценарии в структурированной и систематической манере. Даже сценарии «написанные на обратной стороне конверта» или на коленке выиграют, если вы найдете несколько минут, чтобы спланировать и организовать свои мысли, прежде чем сесть за кодирование.

Здесь несколько стилистических принципов. Они не (не обязательно) являются официальным стилем написания сценариев Shell.

35.1. Неофициальный стиль написания сценариев Shell

- Комментируйте код. Это облегчает понимание (и оценку) его другим и упрощает поддержку.

```
PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
#  Смысл был понятен, когда вы написали это в прошлом году,
#+ но теперь это полная загадка.
#  (Из сценария Antek Sawicki "pw.sh".)
```

- Добавьте описательные заголовки ваших сценариев и их функции.

```
#!/bin/bash

#***********************************************#
#                   xyz.sh                      #
#           написан Bozo Bozeman                #
#                July 05, 2001                  #
#                                               #
#           Очистка файлов проекта.             #
#***********************************************#

E_BADDIR=85                       # Нет такой директории.
projectdir=/home/bozo/projects    # Очищаемая директория.

# ------------------------------------------------------- #
# cleanup_pfiles ()                                       #
# Удаляет все файлы в указанной директории.               #
# Параметр: $целевая_директория                           #
# Возвращает: 0 - успех, $E_BADDIR если что-то не так.    #
# ------------------------------------------------------- #
cleanup_pfiles ()
{
  if [ ! -d "$1" ]  # Проверка наличия целевой директории.
  then
    echo "$1 это не директория."
    return $E_BADDIR
  fi

  rm -f "$1"/*
  return 0   # Успешно.
}
```

```
cleanup_pfiles $projectdir

exit $?
```

- Избегайте использовать «***magic numbers***», [1] являющихся «жесткими» буквенными константами. Вместо этого используйте *значимые* имена переменных. Это делает сценарий легче для понимания, а так же позволяет вносить изменения и обновления, не нарушая приложения.

```
if [ -f /var/log/messages ]
then
  ...
fi
#  Год спустя, вы решили изменить сценарий для проверки /var/log/syslog.
#  Теперь необходимо вручную изменять сценарий, шаг за шагом, и
#+ надеяться,что ничего не упустим.

# Но лучше:
LOGFILE=/var/log/messages  # Только строка, которую нужно изменить.
if [ -f "$LOGFILE" ]
then
  ...
fi
```

- Выбирайте описательные имена для переменных и функций.

```
fl=`ls -al $dirname`                  # Загадочно.
file_listing=`ls -al $dirname`        # Лучше.


MAXVAL=10   # Все верхние пределы, постоянно используемые сценарием.
while [ "$index" -le "$MAXVAL" ]
...


E_NOTFOUND=95                         #  Коды ошибок прописными буквами,
                                      #+ а имена начинаются с E_.
if [ ! -e "$filename" ]
then
  echo "File $filename не найден."
  exit $E_NOTFOUND
fi


MAIL_DIRECTORY=/var/spool/mail/bozo   #  Переменные среды прописными
export MAIL_DIRECTORY                 #+ буквами.


GetAnswer ()                          #  Смешанный регистр хорош для
{                                     #+ имен функций, особенно,
  prompt=$1                           #+ когда важна удобочитаемость.
  echo -n $prompt
  read answer
  return $answer
}
```

```
GetAnswer "Названо Ваше любимое число?"
favorite_number=$?
echo $favorite_number


_uservariable=23                    # Допустимо, но не рекомендуется.
#  Лучше, если определяемые пользователем переменные не начинаются с
#+ подчеркивания.
# Оставим это для системных переменных.
```

- Используйте *коды выхода* систематично и осмысленно.

```
E_WRONG_ARGS=95
...
...
exit $E_WRONG_ARGS
```

См. Приложение E

Ender предполагает использование кодов выхода сценариев */usr/include/sysexits.h*, хотя они предназначаются, главным образом, для программирования на C и C++ .

- Используйте стандартные параметры флагов при вызове сценария. *Ender* предлагает следующий набор флагов.

```
-a      Все: Возвращает всю информацию (включая скрытые файлы).
-b      Краткое описание: Короткая версия, как правило, для
        других сценариев.
-c      Копирование, конкатенация и т.д..
-d      Ежедневно: информация использующаяся ежедневно, а не просто
        информация конкретного экземпляра/пользователя.
-e      Расширение/разработка: (часто не включает в себя скрытые
        файлы).
-h      Помощь: Подробности использования w/descs, aux инфо,
        обсуждения, помощь.
        См. Так же -V.
-l      Журнал вывода сценария.
-m      Руководство: Поиск справочных страниц для основных команд.
-n      Числа: Только числовые данные.
-r      Рекурсивно: Все файлы в директории (и/или все поддиректории).
-s      Настройка и управление файлом: Конфигурационные файлы для
        этого сценария.
-u      Использование: Список флагов с которыми вызывается сценарий.
-v      Подробно: Читабельный вывод, более или менее
        отформатированный.
-V      Версия / Лицензия / Копи(райт|лефт) / Вложения (email).
```

См. Раздел G1.

- Разбивайте сложные сценарии на простые модули. Используйте функции, где это уместно. См. Пример 37-4.

- Не используйте сложные конструкции, где можно обойтись несколькими простыми.

```
КОМАНДА
if [ $? -eq 0 ]
...
# Избыточно и не интуитивно.

if КОМАНДА
...
# Более кратко (насколько возможно, не совсем разборчиво).
```

... читал исходный код UNIX в оболочке Bourne (/bin/sh). Я был в шоке, как много простых алгоритмов могут быть сделаны непонятными и, следовательно, бесполезными, из-за плохого выбора стиля кода. Я спросил себя: «Может ли кто-то гордиться этим кодом?

--Landon Noll

Примечания

[1] В этом контексте, «***magic numbers***» имеют совершенно иной смысл, чем ***магические числа***, используемые для обозначения типов файлов.

Глава 36. Разное

В действительности никому не известно, что является грамматикой Bourne Shell. Даже исследование исходного кода мало помогает.

--Tom Duff

Содержание

36.1. Интерактивные и не интерактивные оболочки и сценарии

Интерактивные оболочки считывают команды вводимые пользователем в `tty`. Среди прочего, такие оболочки считывают активацию запускаемых файлов, выводят на экран подсказку и позволяют управлять работой по умолчанию. Пользователь может *взаимодействовать* с оболочкой.

Запускаемая оболочка сценария всегда является *не интерактивной* оболочкой. Все же сценарий может получать доступ к `tty`. Можно даже эмулировать интерактивную оболочку в сценарии.

```
#!/bin/bash
MY_PROMPT='$ '
while :
do
  echo -n "$MY_PROMPT"
  read line
  eval "$line"
done
exit 0

# Этот пример сценария  предоставлен
# Stéphane Chazelas (снова благодарность).
```

Рассмотрим интерактивный сценарий, как правило использующий оператор **`read`**, который просит от пользователя что-то ввести (Пример 15-3). «Реальная жизнь», на самом деле, немного беспорядочнее. Теперь допустим, что интерактивный сценарий связан с `tty`, сценарий вызывается пользователем в консоли или XTerm.

Init и запуск сценария обязательно не интерактивны, так как они должны работать без вмешательства человека. Многие сценарии управления и обслуживания системы также являются не интерактивными. Неизменно повторяющиеся задачи взывают к автоматизации не-интерактивными сценариями.

Не интерактивные сценарии могут работать в *фоновом* режиме, а интерактивные в этом режиме зависают, ожидая ввода, которого нет и не будет. Обрабатывает эту трудность сценария **`expect`** или ***here document***, встроенный в интерактивный сценарий, вводимый при помощи канала, при работе в фоновом режиме. В простейшем случае файл, из которого осуществляется ввод, перенаправляют оператору **`read`** (переменной **`read <file`**). Именно эти обходные пути дают возможность сценариям общего назначения работать либо в интерактивном, либо в не интерактивном режимах.

Если необходимо проверить работает ли сценарий в интерактивной оболочке, то просто поищите переменную приглашения, задаваемую `$PS1`. (Если пользователю предлагается что-то ввести, то сценарию необходимо вывести на экран запрос.)

```
if [ -z $PS1 ] # нет приглашения?
### if [ -v PS1 ]   # Для Bash 4.2+ ...
then
  # не интерактивный
  ...
else
  # интерактивный
  ...
fi
```

Кроме того, можно проверить сценарий на наличие опции «`i`» с флагом `$-` .

```
case $- in
*i*)    # интерактивная оболочка
;;
*)      # Не интерактивная оболочка
;;
# (Предоставлено "UNIX F.A.Q.," 1993)
```

Однако John Lange описал другой способ, с использованием оператора проверки `-t`

```
# Проверка терминала!

fd=0   # stdin

#  Как мы помним, опция -t  проверяет является ли это stdin, [ -t 0 ]
#+ или stdout, [ -t 1 ] в данном запущенном в терминале сценарии.
if [ -t "$fd" ]
then
  echo интерактивный
else
  echo не интерактивный
fi


#  Но, как поясняет John:
#    if [ -t 0 ] работает... когда вы входите в отдельную систему, но сбоит
#    при вызове команды дистанционно, через ssh.
#    Таким образом, для полной проверки нужно также проверять сокет.
```

```
if [[ -t "$fd" || -p /dev/stdin ]]
then
  echo интерактивный
else
  echo не интерактивный
fi
```

Сценарии могут работать в интерактивном режиме с опцией `-i` или с заголовком **`#!/bin/bash -i`**. Но знайте, что это может привести к неустойчивой работе сценария или выводить сообщения об ошибках, даже если ошибок нет.

36.2. Обертка оболочки

Обертка (***wrapper***) - это сценарий оболочки, который внедряет системные команды или утилиты, которые принимают и передают набор параметров команды. [1] Обертка упрощает вызов сценария вызываемого сложной командной строкой. Особенно полезны ***sed*** и ***awk***.

Сценарии ***sed*** или ***awk*** обычно вызываются командной строкой **`sed -e 'команды'`** или **`awk 'команды'`**. Внедрение такого сценария в сценарий Bash позволяет вызывать его более просто и позволяет использовать его многократно. Кроме того позволяет комбинировать функциональность ***sed*** и ***awk***, например передавать вывод набора команд ***sed*** в ***awk***. Вы можете неоднократно ссылаться на него, как на сохраненный исполняемый файл в его первоначальном виде или с изменениями, без не удобства повторного ввода в командной строке.

Пример 36-1. ***Обертка оболочки***

```
#!/bin/bash

# Это простой сценарий для удаления пустых строк из файла.
# Без проверки аргумента.
#
# Вы, возможно, пожелаете добавить что-то вроде:
#
# E_NOARGS=85
# if [ -z "$1" ]
# then
#  echo "Usage: `basename $0` целевой_файл"
#  exit $E_NOARGS
# fi
```

```
sed -e /^$/d "$1"
# То же, что и
#    sed -e '/^$/d' имя_файла
# вызываемый из командной строки.

#  '-e' означает команду "редактировать" (здесь не обязательно).
#  '^' указывает на начало строки, '$' на окончание.
#  Соответствует строке ничего не содержащей между ее началом и концом --
#+ пустой строке.
#  'd' команда удаления.

#  Аргумент командной строки в кавычках допускает
#+ пробелы и специальные символы в имени файла.

#  Обратите внимание, что этот сценарий не изменяет целевой файл.
#  Если вам нужно это сделать, перенаправьте вывод.

exit
```

Пример 36-2. Слегка усложненная *обертка оболочки*

```
#!/bin/bash

#  subst.sh: сценарий, который заменяет один шаблон в файле на другой,
#+ т.е., "sh subst.sh Smith Jones letter.txt".
#                     Jones заменяется на Smith.

ARGS=3          # Сценарию необходимо 3 аргумента.
E_BADARGS=85    # Сценарию передано не правильное число аргументов.

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` старый_шаблон новый_шаблон файл"
  exit $E_BADARGS
fi

old_pattern=$1
new_pattern=$2

if [ -f "$3" ]
then
    file_name=$3
else
    echo "Файл \"$3\" не существует."
    exit $E_BADARGS
fi


# -------------------------------------------------
#  Вот где выполняется тяжелая работа.
sed -e "s/$old_pattern/$new_pattern/g" $file_name
# -------------------------------------------------

#  's' это, конечно, команда замены в sed,
#+ а /pattern/ вызывает соответствующий  адрес.
#  Флаг 'g,' или global вызывает замену для каждого вхождения $old_pattern
#+ в каждой строке, а не только в первой.
```

```
#  Читайте документацию по 'sed' для углубленного понимания.

exit $?  # Перенаправьте вывод этого сценария для записи в файл.
```

Пример 36-3. Создание *обертки оболочки*, которая пишет в логфайл

```
#!/bin/bash
#  logging-wrapper.sh
#  Создание обертки оболочки, которая выполняет операцию
#+ и регистрирует ее в журнал.

DEFAULT_LOGFILE=logfile.txt

# Объявляются следующие две  переменные.
OPERATION=
#         Может быть сложной цепочкой команд,
#+        например сценарием awk или конвейером ...

LOGFILE=
if [ -z "$LOGFILE" ]
then     # Если не присвоено, то по умолчанию ...
  LOGFILE="$DEFAULT_LOGFILE"
fi

#         Аргументы командной строки операций, если таковые имеются.
OPTIONS="$@"


# Их логирование.
echo "`date` + `whoami` + $OPERATION "$@"" >> $LOGFILE
# Теперь, сделаем.
exec $OPERATION "$@"

# Логирование необходимо делать до операции.
# Почему?
```

Пример 36-4. *Обертка оболочки* вокруг сценария awk

```
#!/bin/bash
# pr-ascii.sh: Печать таблицы символов ASCII.

START=33   # Диапазон печатаемых символов ASCII (десятичный).
END=127    # Не будет работать для непечатных символов (>127).

echo " Decimal   Hex     Character"   # Заголовки.
echo " -------   ---     ---------"

for ((i=START; i<=END; i++))
do
  echo $i | awk '{printf("%3d   %2x   %c\n", $1, $1, $1)}'
# printf, встроенное в Bash, в этом случае не работает:
#     printf "%c" "$i"
done

exit 0
```

```
#  Decimal   Hex     Character
#  -------   ---     ---------
#    33       21         !
#    34       22         "
#    35       23         #
#    36       24         $
#
#    . . .
#
#   122       7a         z
#   123       7b         {
#   124       7c         |
#   125       7d         }


#  Перенаправьте вывод этого сценария в файл
#+ или передайте в "more":  sh pr-asc.sh | more
```

Пример 36-5. ***Обертка оболочки*** **вокруг другого сценария awk**

```
#!/bin/bash

#  Добавление заданной колонки (количества) в целевой файл.
#  Числа с плавающей запятой (десятичные) — не страшно - awk может
#+ справиться с ними.

ARGS=2
E_WRONGARGS=85

if [ $# -ne "$ARGS" ] #  Проверка правильного количества аргументов
                      #+ командной строки.
then
   echo "Usage: `basename $0` filename число_колонок"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#  Небольшой хитростью является передача переменным оболочки
#+ части сценария awk.
#  Одним из методов является заключение переменной сценария Bash в жесткие
#+ кавычки в сценарии awk.
#     $'$BASH_SCRIPT_VAR'
#      ^                 ^
#  Это делается во встроенном сценарии awk ниже.
#  Подробности смотри в документации awk.

# Здесь вызывается многострочный сценарий awk
#   awk '
#   ...
#   ...
#   ...
#   '
```

```
# Начало сценария awk.
# ----------------------------
awk '

{ total += $'"${column_number}"'
}
END {
     print total
}

' "$filename"
# ----------------------------
# Окончание сценария awk.


#   Это не безопасно, передавать переменные оболочки во встроенный сценарий
#+  awk, поэтому Стефан Chazelas предлагает следующую альтернативу:
#   -------------------------------------
#   awk -v column_number="$column_number" '
#   { total += $column_number
#   }
#   END {
#       print total
#   }' "$filename"
#   -------------------------------------


exit 0
```

Для сценариев, нуждающихся в инструменте все-в-одном, ***Perl*** является швейцарским армейским ножом. ***Perl*** сочетает в себе возможности ***sed*** и ***awk***, и обрабатывает загрузку больших подмножеств ***Cu***. Он является модульным и содержит поддержку всего, начиная от объектно-ориентированного программирования и заканчивая кухонной раковиной. Короткие сценарии ***Perl*** поддаются встраиванию в сценарии оболочки, и в них могут быть вещи, которыми ***Perl*** может полностью заменить сценарии оболочки (хотя автор ABS Guide относится к этому скептически).

Пример 36-6. Perl встроенный в сценарий *Bash*

```
#!/bin/bash

# Команде оболочки может предшествовать сценарий на Perl.
echo "Это предшествует внедренному сценарию Perl в \'$0.\"."
echo "============================================================="

perl -e 'print "Эта строка выведена из встроенного сценария Perl.\n";'
# Как и sed, Perl так же использует опцию "-e".

echo "============================================================="
echo "Однако сценарий также может содержать системные команды и команды
оболочки."

exit 0
```

Можно даже комбинировать сценарий Bash и сценарий ***Perl*** в одном файле. В зависимости от того, как вызывается сценарий будет выполняться часть Bash или часть ***Perl***.

Пример 36-7. Комбинация сценариев Bash и Perl

```
#!/bin/bash
# bashandperl.sh

echo "Поздравление из части сценария Bash, $0."
# Здесь может следовать множество команд Bash.

exit
# Конец части сценария Bash.

# =============================================================

#!/usr/bin/perl
# Эта часть сценария будет вызываться
#    perl -x bashandperl.sh

print "Поздравления из части сценария Perl, $0.\n";
#      Perl не нравится "echo" ...
# Здесь может следовать множество команд Perl.

# Конец части сценария Perl.
```

```
bash$ bash bashandperl.sh
Поздравление из части сценария Bash.


bash$ perl -x bashandperl.sh
Поздравления из части сценария Perl.
```

Конечно, можно вставлять даже более экзотические языки сценариев в обертку оболочки. К примеру ***Python*** ...

Пример 36-8. Python встроенный в сценарий *Bash*

```
#!/bin/bash
# ex56py.sh

# Командам оболочки может предшествовать сценарий Python.
echo "Это предшествует внедренному сценарию Python в \'$0.\""
echo "============================================================="

python -c 'print "Эта строка выводится из внедренного сценария Python.\n";'
# В отличии от sed и perl, Python использует опцию "-c".
python -c 'k = raw_input( "Нажмите клавишу для выхода из внешнего сценария." )'

echo "============================================================="
echo "Этот сценарий может содержать системные команды и команды оболочки."

exit 0
```

Обертка сценария вокруг ***mplayer*** и сервера переводов Google, может создать то, что будет отвечать вам голосом.

Пример 36-9. Говорящий сценарий

```
#!/bin/bash
#   Любезно предоставлено:
#   http://elinux.org/RPi_Text_to_Speech_(Speech_Synthesis)

#  для работы этого сценария Вы должны быть онлайн, т.к. вам
#+ нужен доступ к серверу переводов Google.
#  Конечно, на Вашем компьютере должен быть установлен mplayer.

speak()
  {
  local IFS=+
  # Вызывается mplayer, затем соединяется с Google translation server.
  /usr/bin/mplayer -ao alsa -really-quiet -noconsolecontrols \
 "http://translate.google.com/translate_tts?tl=en&q="$*""
  # Google переводит, но может так же и произносить.
  }

LINES=4

spk=$(tail -$LINES $0) # Заключительная часть того же самого сценария!
speak "$spk"
exit
# Приятно было пообщаться.
```

Один интересный пример сложной обертки оболочки - сценарий ***undvd*** Martin Matusiak, который обеспечивает простой в использовании интерфейс командной строки для сложной утилиты ***mencoder***. Другим примером является ***Ext3Undel***, набор сценариев для восстановления удаленных файлов из файловой системы ext3, Itzchak Rehberg.

Примечания

[1] Целый ряд утилит Linux являются, по сути, обертками оболочки. Например: `/usr/bin/pdf2ps`, `/usr/bin/batch` и `/usr/bin/xmkmf`.

36.3. Проверки и сравнения: альтернативы

Для проверки, конструкция **`[[ ]]`** может оказаться более подходящей, чем **`[ ]`**.

Аналогичным образом ***арифметические*** сравнения могут воспользоваться конструкцией **(( ))**.

```
a=8

# Все сравнения, ниже, эквивалентны.
test "$a" -lt 16 && echo "да, $a < 16"    # "and list"
/bin/test "$a" -lt 16 && echo "да, $a < 16"
[ "$a" -lt 16 ] && echo "да, $a < 16"
[[ $a -lt 16 ]] && echo "да, $a < 16"     # В [[ ]] и (( )) заключение
(( a < 16 )) && echo "да, $a < 16"        # переменных в кавычки не обязательно.

city="New York"
# И снова, все сравнения, ниже, эквивалентны.
test "$city" \< Париж && echo "Да, Париж больше, чем $city"
                                  # Больше порядок ASCII.
/bin/test "$city" \< Париж && echo " Да, Париж больше, чем $city"
[ "$city" \< Париж ] && echo " Да, Париж больше, чем $city"
[[ $city < Париж ]] && echo " Да, Париж больше, чем $city"
                                  # Для $city кавычки не нужны.

# Спасибо, S.C.
```

36.4. Рекурсия: сценарий вызывающий себя

Может ли сценарий рекурсивно вызывать себя? Может.

Пример 36-10. Сценарий (полезный), который вызывает себя

```
#!/bin/bash
# recurse.sh

#  Может ли сценарий рекурсивно вызывать сам себя?
#  Да, но какая от этого польза?
#  (Увидим далее.)

RANGE=10
MAXVAL=9

i=$RANDOM
let "i %= $RANGE"  # Генерирование случайных чисел между 0 и $RANGE - 1.

if [ "$i" -lt "$MAXVAL" ]
then
  echo "i = $i"
  ./$0             #  Сценарий рекурсивно порождает свой новый экземпляр.
fi                 #  Каждый сценарий-потомок делает то же самое, до тех пор
                   #+ пока создаваемый $i равен $MAXVAL.

#  Использование цикла «while» вместо проверки 'if/then' вызывает проблемы.
#  Объясните почему.
```

```
exit 0

# Примечание:
# ----
# Для правильной работы сценарий должен иметь права на выполнение.
# Даже в том случае, если он вызывается командой «sh».
# Объясните почему.
```

Пример 36-11. Сценарий (полезный), который вызывает себя

```
#!/bin/bash
# pb.sh: телефонная книга

# Написан Rick Boivie, и используется с разрешения.
# Изменения автора ABS Guide.

MINARGS=1     #  Сценарию необходим хотя бы один аргумент.
DATAFILE=./phonebook
              #  В текущей рабочей директории должен существовать файл данных,
              #+ названный «phonebook».
PROGNAME=$0
E_NOARGS=70   #  Ошибка при отсутствии аргументов.

if [ $# -lt $MINARGS ]; then
      echo "Usage: "$PROGNAME" "данные_для_просмотра"
      exit $E_NOARGS
fi


if [ $# -eq $MINARGS ]; then
      grep $1 "$DATAFILE"
      # если отсутствует $DATAFILE, 'grep' выводит сообщение об ошибке.
else
      ( shift; "$PROGNAME" $* ) | grep $1
      # Сценарий рекурсивно вызывает сам себя.
fi

exit 0        #  Здесь выход сценария.
              #  Таким образом, это годится для вложения не отмеченных хэш
              #+ комментариев и данных после этого.

# ------------------------------------------------------------------------
Пример файла данных "phonebook":

John Doe        1555 Main St., Baltimore, MD 21228         (410) 222-3333
Mary Moe        9899 Jones Blvd., Warren, NH 03787         (603) 898-3232
Richard Roe     856 E. 7th St., New York, NY 10009         (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009         (212) 444-5678
Zoe Zenobia     4481 N. Baker St., San Francisco, SF 94338  (415) 501-1631
# ------------------------------------------------------------------------

$bash pb.sh Roe
Richard Roe     856 E. 7th St., New York, NY 10009         (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009         (212) 444-5678

$bash pb.sh Roe Sam
Sam Roe         956 E. 8th St., New York, NY 10009         (212) 444-5678
```

```
#  Когда этому сценарию передается более одного аргумента,
#+ он выводит строки, содержащие *только* эти аргументы.
```

Пример 36-12. Другой сценарий (полезный) рекурсивно вызывающий сам себя

```
#!/bin/bash
# usrmnt.sh, написан Anthony Richardson
# Используется в ABS Guide с разрешения.

# usage:       usrmnt.sh
# описание: монтируемые устройства, вызываемые пользователем должны быть
#             перечислены в группе MNTUSERS в файле /etc/sudoers.

# ------------------------------------------------------------
#  Это сценарий usermount, который возвращает сам себя с помощью sudo.
#  Пользователь с соответствующими правами может только
#   usermount /dev/fd0 /mnt/floppy
# вместо
#   sudo usermount /dev/fd0 /mnt/floppy

#  Я использую эту же технику для всех моих сценариев sudo,
#+ потому что я считаю ее удобной.
# ------------------------------------------------------------

#  Если переменная SUDO_COMMAND не задана, то через sudo не запускается,
#+ поэтому перезапускает себя. Передаются настоящие id пользователя и группы...

if [ -z "$SUDO_COMMAND" ]
then
   mntusr=$(id -u) grpusr=$(id -g) sudo $0 $*
   exit 0
fi

# мы будем здесь находиться, пока мы не запустим sudo.
/bin/mount $* -o uid=$mntusr,gid=$grpusr

exit 0

# Дополнительные примечания (автора этого сценария):
# ------------------------------------------------

# 1) Linux доступна опция «users», в /etc/fstabfile, поэтому любой
# пользователь может монтировать съемные носители. Но на сервере
# я хотел бы иметь доступ только нескольких человек к съемным носителям.
# Я считаю, что использование sudo дает мне больше контроля.

# 2) Я также нахожу sudo  более удобным, чем выполнение
# таких задач через группы.

# 3) Этот способ не дает никому соответствующих прав доступа
# с правами root для команды mount, так что будьте осторожны,
# при разрешении доступа.
# Вы можете получить более точный контроль над  правами,
# которые могут быть установлены с помощью этой техники отдельными
# сценариями mntfloppy, mntcdrom и mntsamba.
```

Слишком много уровней рекурсии могут исчерпать пространство стека сценария, вызвав

сегментацию.

36.5. 'Раскрашивание' сценариев

Управляющие последовательности ANSI [1] устанавливают такие атрибуты экрана, как ***толщина*** шрифта и ***цвет*** переднего плана и фона. Пакетные файлы DOS обычно используют управляющие коды ANSI для вывода цвета, но это возможно и в сценариях Bash.

Пример 36-13. "Раскрашивание" адресной базы данных

```
#!/bin/bash
# ex30a.sh: "Раскрашенная" версия ex30.sh.
#           Необработанная адресная база данных


clear                                   # Очистка экрана.

echo -n "          "
echo -e '\E[37;44m'"\033[1mСписок контактов\033[0m"
                                        # Белый на синем фоне
echo; echo
echo -e "\033[1mВыберите одно из следующих лиц:\033[0m"
                                        # Жирный шрифт
tput sgr0                               # Сброс атрибутов.
echo "(Enter only the first letter of name.)"
echo
echo -en '\E[47;34m'"\033[1mE\033[0m"   # Синий
tput sgr0                               # Сброс цвета в "нормальный."
echo "vans, Roland"                     # "[E]vans, Roland"
echo -en '\E[47;35m'"\033[1mJ\033[0m"   # Пурпурный
tput sgr0
echo "ambalaya, Mildred"
echo -en '\E[47;32m'"\033[1mS\033[0m"   # Зеленый
tput sgr0
echo "mith, Julie"
echo -en '\E[47;31m'"\033[1mZ\033[0m"   # Красный
tput sgr0
echo "ane, Morris"
echo

read person

case "$person" in
# Обратите внимание, что переменная в кавычках.

  "E" | "e" )
  # Верхний или нижний регистр ввода.
  echo
  echo "Roland Evans"
  echo "4321 Flash Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;
```

```
  "J" | "j" )
  echo
  echo "Mildred Jambalaya"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Добавим инфо для Smith & Zane позже.

         * )
   # Опции по умолчанию.
   # Здесь так же подходит пустой ввод (выпадение RETURN).
   echo
   echo "Пока нет в базе данных."
  ;;

esac

tput sgr0                                 # Сброс цветов в "нормальный."

echo

exit 0
```

Пример 36-14. Рисование полей

```
#!/bin/bash
# Draw-box.sh: Рисование полей с помощью символов ASCII.

# Сценарий Stefano Palmeri, с небольшими изменениями автора.
# Небольшие изменения внесены Jim Angstadt.
# В ABS Guide используется с разрешения.


######################################################################
###  Документация функции draw_box  ###

#  Функция "draw_box" позволяет пользователю в терминале рисовать поля.
#
#  Использование: ROW COLUMN HEIGHT WIDTH [COLOR]   ROW и COLUMN,
#+ которые вы собираетесь рисовать, начинают позиционирование от
#+ верхнего левого угла окна.
#  ROW и COLUMN должны быть больше, чем 0
#+ но меньше, чем текущий размер терминала.
#  HEIGHT число строк в окне, должно быть > 0.
#  HEIGHT + ROW должно быть <= чем текущая высота терминала.
#  WIDTH число колонок в окне и должно быть больше > 0.
#  WIDTH + COLUMN должно быть <= чем текущая ширина терминала.
#
# Например: Если размер Вашего терминала 20x80,
#  draw_box 2 3 10 45 это хорошо
#  draw_box 2 3 19 45 имеет плохое значение HEIGHT (19+2 > 20)
#  draw_box 2 3 18 78 имеет плохое значение WIDTH (78+3 > 80)
```

```
#
#  COLOR это цвет рамки окна.
#  Это 5-й аргумент и не обязателен.
#  0=черный 1=красный 2=зеленый 3=охристый 4=синий 5=пурпурный 6=голубой
#  7=белый.
#  Если функции переданы плохие аргументы, она просто выйдет с кодом 65,
#+ а сообщения не будут выведены в stderr.
#
#  Очистите терминал, прежде чем вы начнете рисовать поля.
#  Команда clear не включена в функцию.
#  Это позволяет пользователю рисовать несколько полей, даже перекрывающих.

###  конец документации функции draw_box  ###
#############################################################################

draw_box(){

#=============#
HORZ="-"
VERT="|"
CORNER_CHAR="+"

MINARGS=4
E_BADARGS=65
#=============#


if [ $# -lt "$MINARGS" ]; then          # Если меньше 4 аргументов, выход.
    exit $E_BADARGS
fi

# Проверка, что в аргументах символы, а не цифры.
# Вероятно, это может быть реализовано лучше (упражнение для читателей?).
if echo $@ | tr -d [:blank:] | tr -d [:digit:] | grep . &> /dev/null; then
   exit $E_BADARGS
fi

BOX_HEIGHT=`expr $3 - 1`   #  -1 коррекция, поскольку символ угла '+'
BOX_WIDTH=`expr $4 - 1`    #+ является частью обоих полей высоты и ширины.
T_ROWS=`tput lines`        #  Определение текущего размера терминала
T_COLS=`tput cols`         #+ в строках и столбцах.

if [ $1 -lt 1 ] || [ $1 -gt $T_ROWS ]; then    #  Начало проверки
   exit $E_BADARGS                             #+ правильны ли аргументы.
fi
if [ $2 -lt 1 ] || [ $2 -gt $T_COLS ]; then
   exit $E_BADARGS
fi
if [ `expr $1 + $BOX_HEIGHT + 1` -gt $T_ROWS ]; then
   exit $E_BADARGS
fi
if [ `expr $2 + $BOX_WIDTH + 1` -gt $T_COLS ]; then
   exit $E_BADARGS
fi
if [ $3 -lt 1 ] || [ $4 -lt 1 ]; then
   exit $E_BADARGS
fi                                  # Окончание проверки аргументов.

plot_char(){                        # Функция в функции.
   echo -e "\E[${1};${2}H"$3
```

```
}

echo -ne "\E[3${5}m"              # Задаем нужный цвет рамки поля.

# Начинаем рисовать

count=1                                          #  Вертикальные линии рисуем
for (( r=$1; count<=$BOX_HEIGHT; r++)); do       #+ с помощью функции plot_char.
  plot_char $r $2 $VERT
  let count=count+1
done

count=1
c=`expr $2 + $BOX_WIDTH`
for (( r=$1; count<=$BOX_HEIGHT; r++)); do
  plot_char $r $c $VERT
  let count=count+1
done

count=1                                          #  Горизонтальные линии рисуем
for (( c=$2; count<=$BOX_WIDTH; c++)); do        #+ с помощью функции plot_char.
  plot_char $1 $c $HORZ
  let count=count+1
done

count=1
r=`expr $1 + $BOX_HEIGHT`
for (( c=$2; count<=$BOX_WIDTH; c++)); do
  plot_char $r $c $HORZ
  let count=count+1
done

plot_char $1 $2 $CORNER_CHAR                     # Отрисовка углов полей.
plot_char $1 `expr $2 + $BOX_WIDTH` $CORNER_CHAR
plot_char `expr $1 + $BOX_HEIGHT` $2 $CORNER_CHAR
plot_char `expr $1 + $BOX_HEIGHT` `expr $2 + $BOX_WIDTH` $CORNER_CHAR

echo -ne "\E[0m"              #  Восстановление прежних цветов.

P_ROWS=`expr $T_ROWS - 1`     #  Помещение командной строки в нижнюю часть
                              #+ терминала.

echo -e "\E[${P_ROWS};1H"
}


# Теперь попытка нарисовать поле.
clear                        # Очистка терминала.
R=2      # Строка
C=3      # Колонка
H=10     # Высота
W=45     # Ширина
col=1    # Цвет (красный)
draw_box $R $C $H $W $col   # Рисование поля.

exit 0

# Упражнение:
# --------
```

```
# Добавьте параметр вывода текста в нарисованном поле.
```

Самая простая, и, возможно, наиболее полезная управляющая последовательность ANSI это ***полужирный*** текст, **\033[1m ... \033[0m**. \033 является экранированием '[1' включающего атрибут полужирного начертания, в то время как '[0' выключает его. "**m**" прекращает каждую управляющую последовательность.

```
bash$ echo -e "\033[1mЭто полужирный текст.\033[0m"
```

Эта последовательность переключает на подчеркивание атрибута (на *rxvt* и на *Aterm*).

```
bash$ echo -e "\033[4mЭто подчеркнутый полужирный текст.\033[0m"
```

С **echo** опция -e включает управляющие последовательности.

Другие управляющие последовательности изменяющие цвет текста или фона.

```
bash$ echo -e '\E[34;47mЭто выводится синим цветом.'; tput sgr0

bash$ echo -e '\E[33;44m'"желтый текст на синем фоне"; tput sgr0

bash$ echo -e '\E[1;33;44m'"Полужирный желтый текст на синем фоне"; tput sgr0
```

Желательно задавать атрибут полужирного шрифта для светлого цвета текста переднего плана.

tput sgr0 восстанавливает настройки терминала в обычные. Пропуск делает все последующие выводы из этого, конкретного, терминала по-прежнему синим цветом.

Поскольку **tput sgr0** не всегда удается восстановить настройки терминала при определенных обстоятельствах, **echo -ne \E[0m** может быть лучшим выбором.

Используйте следующий шаблон для написания цветного текста на цветном фоне.

```
echo -e '\E[COLOR1;COLOR2mSome text goes here.'
```

"**\E[**" начинает управляющую последовательность. Разделенные точкой с запятой COLOR1 и COLOR2 указывают цвет переднего плана и цвет фона, в соответствии с приведенной ниже таблицей. (Порядок чисел не имеет значения, так как числа переднего и заднего плана попадают в не перекрывающиеся диапазоны.) '**m**' завершает управляющую последовательность, и текст начинается сразу после нее.

Также, обратите внимание, что оставшаяся часть последовательности команд после

echo -e заключается в одинарные кавычки.

Числа в таблице, ниже, работают в терминале ***rxvt***. Для других эмуляторов терминала они могут отличаться.

Таблица 36-1. Числа представляющие цвета в управляющей последовательности

Цвет	Передний план	Задний план (фон)
черный	**30**	**40**
красный	**31**	**41**
Зеленый	**32**	**42**
желтый	**33**	**43**
синий	**34**	**44**
пурпур	**35**	**45**
голубой	**36**	**46**
белый	**37**	**47**

Пример 36-15. Вывод на экран цветного текста

```
#!/bin/bash
# color-echo.sh: Вывод текстового сообщения в цвете.

# Измените этот сценарий для собственных нужд.
# Это проще, чем ручное кодирования цвета.

black='\E[30;47m'
red='\E[31;47m'
green='\E[32;47m'
yellow='\E[33;47m'
blue='\E[34;47m'
magenta='\E[35;47m'
cyan='\E[36;47m'
white='\E[37;47m'


alias Reset="tput sgr0"      #  Устанавливаем атрибуты текста нормальными
                             #+ без очистки экрана.


cecho ()                     # Цветной вывод.
```

```
                              # Аргумент $1 = сообщение
                              # Аргумент $2 = цвет
{
local default_msg="Нет переданных сообщений."
                              # Локальная переменная, на самом деле, не нужна.

message=${1:-$default_msg}   # Умолчания в сообщении по умолчанию.
color=${2:-$black}           # По умолчанию черным, если не указано.

  echo -e "$color"
  echo "$message"
  Reset                      # Сброс на нормальный.

  return
}


# Теперь попытка.
# ------------------------------------------------
cecho "Грустный синий..." $blue
cecho "Пурпурный выглядит лучше фиолетового." $magenta
cecho "Зеленая зависть." $green
cecho "Видишь красный?" $red
cecho "Голубой, более известный как Аква." $cyan
cecho "Нет переданного цвета (по умолчанию черный)."
       # Отсутствует аргумент $color.
cecho "\"Пустой\" передан цвет (по умолчанию черный)." ""
       # Пустой аргумент $color.
cecho
       # Отсутствуют аргументы $message и $color.
cecho "" ""
       # Пустые аргументы $message и $color.
# ------------------------------------------------

echo

exit 0

# Упражнения:
# ---------
# 1) Добавьте атрибут "bold" в функцию 'cecho ()'.
# 2) Добавьте опции расцвечивания фона.
```

Упражнение 36-16. Игра "скачки"

```
#!/bin/bash
# horserace.sh: Очень простой эмулятор скачек.
# Автор: Stefano Palmeri Существует, однако, со всем этим серьезная проблема.
Управляющие последовательности ANSI являются решительно непереносимыми. Что
отлично работает на одних эмуляторах терминалов (или консоли) может работать
по-разному или вообще не работать, на других. «Раскрашенный» сценарий, который
выглядит потрясающе на машине автора, может производить нечитаемым вывод на
чужой машине. Это несколько подрывает полезность раскрашивания сценариев и
возможно относит эту технику к статусу трюков. Раскрашенные сценарии, вероятно,
будут неуместны в коммерческих условиях, т.е., ваш руководитель может  их не
одобрить.
# Используется с разрешения.
```

```
#####################################################################
#  Цели сценария:
#  игра с управляющими последовательностями и цветами терминала.
#
#  Упражнение:
#  Измените сценарий, чтобы сделать его работу менее случайной, создайте
#+ фальшивую букмекерскую контору...
#  Умм... умм... это начинает напоминать мне о фильме...
#
#  Сценарий дает одной из случайных лошадей фору.
#  Коэффициенты вычисляются по забегу лошади и выражаются в
#+ европейском(?) стиле.
#  Например, odds=3.75 означает, что если вы поставили $1 и выиграли,
#+ вы получите $3.75.
#
#  Сценарий был проверен на ОС GNU/Linux, с использованием xterm,
#+ rxvt и консольно.
#  На компьютере с процессором AMD 900 МГц среднее время гонки
#+ составляет 75 секунд.
#  На более быстрых компьютерах время будет меньше.
#  А, если вы хотите увеличить интригу, сбросьте переменную USLEEP_ARG.
#
#  Сценарий Stefano Palmeri.
#####################################################################

E_RUNERR=65

# Проверка установки md5sum и bc.
if ! which bc &> /dev/null; then
   echo bc не установлена.
   echo "Не возможно запустить... "
   exit $E_RUNERR
fi
if ! which md5sum &> /dev/null; then
   echo md5sum не установлена.
   echo "Не возможно запустить... "
   exit $E_RUNERR
fi

#  Установите переменную ниже, для замедления выполнения сценария.
#  Она передается в качестве аргумента usleep (читай мануал usleep) и
#+ выражается в микросекундах (500000 = полсекунды).
USLEEP_ARG=0

#  Очистка временной директории, восстановление курсора терминала и
#+ цвета терминала, прерывание сценария нажатием CTL-C.
trap 'echo -en "\E[?25h"; echo -en "\E[0m"; stty echo;\
tput cup 20 0; rm -fr  $HORSE_RACE_TMP_DIR'  TERM EXIT
#  См. главу объясняющую отладку 'trap.'

#  Установка уникального (параноидально) имени временной директории
#+ необходимой сценарию.
HORSE_RACE_TMP_DIR=$HOME/.horserace-`date +%s`-`head -c10 /dev/urandom \
| md5sum | head -c30`

# Создание временной директории и переход в нее.
mkdir $HORSE_RACE_TMP_DIR
cd $HORSE_RACE_TMP_DIR

#  Эта функция перемещает курсор в строку $1 колонки $2, а затем выводит $3.
#  Например: "move_and_echo 5 10 linux" эквивалентно
```

```
#+ "tput cup 4 9; echo linux", но с помощью одной команды вместо двух.
#  Примечание: "tput cup" определяет 0 0  верхний левый угол терминала, echo
#+ определяет 1 1 верхний левый угол терминала.
move_and_echo() {
          echo -ne "\E[${1};${2}H""$3"
}

# Функция создающая псевдо-случайные числа между 1 и 9.
random_1_9 ()
{
    head -c10 /dev/urandom | md5sum | tr -d [a-z] | tr -d 0 | cut -c1
}

#  Две функции имитирующие «движение», при отрисовке лошади.
draw_horse_one() {
               echo -n " "//$MOVE_HORSE//
}
draw_horse_two(){
              echo -n " "\\\\$MOVE_HORSE\\\\
}


# Определение размера текущего терминала.
N_COLS=`tput cols`
N_LINES=`tput lines`

# Требуется, хотя бы, терминал 20-СТРОК X 80-КОЛОНОК. Проверьте.
if [ $N_COLS -lt 80 ] || [ $N_LINES -lt 20 ]; then
   echo "`basename $0` необходим терминал 80-cols X 20-lines."
   echo "Ваш терминал ${N_COLS}-cols X ${N_LINES}-lines."
   exit $E_RUNERR
fi


# Начало отрисовки ипподрома.

# Требуется строка из 80 символов. Смотри ниже.
BLANK80=`seq -s "" 100 | head -c80`

clear

# Установка белого цвета переднего плана и фона.
echo -ne '\E[37;47m'

# Перемещение курсора в верхний левый угол терминала.
tput cup 0 0

# Отрисовка шести белых линий.
for n in `seq 5`; do
      echo $BLANK80   #  Используется строка из 80 символов для раскраски
                      #+ терминала.
done

# Установка черного цвета фона.
echo -ne '\E[30m'

move_and_echo 3 1 "START  1"
move_and_echo 3 75 FINISH
move_and_echo 1 5 "|"
move_and_echo 1 80 "|"
```

```
move_and_echo 2 5 "|"
move_and_echo 2 80 "|"
move_and_echo 4 5 "|  2"
move_and_echo 4 80 "|"
move_and_echo 5 5 "V  3"
move_and_echo 5 80 "V"

# Установка красного цвета переднего плана
echo -ne '\E[31m'

# Немного искусства ASCII.
move_and_echo 1 8 "..@@@..@@@@@...@@@@@.@...@..@@@@..."
move_and_echo 2 8 ".@...@...@.......@...@...@.@......."
move_and_echo 3 8 ".@@@@@...@.......@...@@@@@.@@@@...."
move_and_echo 4 8 ".@...@...@.......@...@...@.@......."
move_and_echo 5 8 ".@...@...@.......@...@...@..@@@@..."
move_and_echo 1 43 "@@@@...@@@...@@@@..@@@@..@@@@."
move_and_echo 2 43 "@...@.@...@.@.....@.....@....."
move_and_echo 3 43 "@@@@..@@@@@.@.....@@@@...@@@.."
move_and_echo 4 43 "@..@..@...@.@.....@.........@."
move_and_echo 5 43 "@...@.@...@..@@@@..@@@@.@@@@.."


# Установка зеленого цвета переднего плана и фона.
echo -ne '\E[32;42m'

# Отрисовка одиннадцати зеленых линий.
tput cup 5 0
for n in `seq 11`; do
      echo $BLANK80
done

# Установка черного цвета фона.
echo -ne '\E[30m'
tput cup 5 0

# Отрисовка барьеров.
echo "++++++++++++++++++++++++++++++++++++\
++++++++++++++++++++++++++++++++++++++++"

tput cup 15 0
echo "++++++++++++++++++++++++++++++++++++\
++++++++++++++++++++++++++++++++++++++++"

# Установка белого цвета переднего плана и фона.
echo -ne '\E[37;47m'

# Отрисовка трех белых линий.
for n in `seq 3`; do
      echo $BLANK80
done

# Установка черного цвета фона.
echo -ne '\E[30m'

# Создание 9 файлов для сохранения забегов.
for n in `seq 10 7 68`; do
      touch $n
done
```

```
# Установка первого типа сценария нарисованной «лошади».
HORSE_TYPE=2

#  Создание файла позиции и файла шансов для каждой 'лошади'.
#+ В этих файлах будет сохраняться текущее положение лошади, тип и шансы.
for HN in `seq 9`; do
      touch horse_${HN}_position
      touch odds_${HN}
      echo \-1 > horse_${HN}_position
      echo $HORSE_TYPE >>  horse_${HN}_position
      # Определение забега случайной лошади.
       HANDICAP=`random_1_9`
      # Проверка, возвращает ли функция random_1_9 «хорошее» значение.
      while ! echo $HANDICAP | grep [1-9] &> /dev/null; do
                HANDICAP=`random_1_9`
      done
      # Определение позиции забега для последней лошади.
      LHP=`expr $HANDICAP \* 7 + 3`
      for FILE in `seq 10 7 $LHP`; do
            echo $HN >> $FILE
      done

      # Рассчет шансов.
      case $HANDICAP in
              1) ODDS=`echo $HANDICAP \* 0.25 + 1.25 | bc`
                                  echo $ODDS > odds_${HN}
              ;;
              2 | 3) ODDS=`echo $HANDICAP \* 0.40 + 1.25 | bc`
                                       echo $ODDS > odds_${HN}
              ;;
              4 | 5 | 6) ODDS=`echo $HANDICAP \* 0.55 + 1.25 | bc`
                                             echo $ODDS > odds_${HN}
              ;;
              7 | 8) ODDS=`echo $HANDICAP \* 0.75 + 1.25 | bc`
                                       echo $ODDS > odds_${HN}
              ;;
              9) ODDS=`echo $HANDICAP \* 0.90 + 1.25 | bc`
                                  echo $ODDS > odds_${HN}
      esac


done


# Вывод шансов.
print_odds() {
tput cup 6 0
echo -ne '\E[30;42m'
for HN in `seq 9`; do
      echo "#$HN odds->" `cat odds_${HN}`
done
}

# Отрисовка лошадей на стартовой линии.
draw_horses() {
tput cup 6 0
echo -ne '\E[30;42m'
for HN in `seq 9`; do
```

```
      echo /\\$HN/\\"                                    "
done
}

print_odds

echo -ne '\E[47m'
# Ожидание нажатия клавиши enter для начала гонки.
# Отключение курсора управляющей последовательностью '\E[?25l' .
tput cup 17 0
echo -e '\E[?25l'Нажмите клавишу [enter] для начала гонки...
read -s

#  Отключение обычного вывода на экран терминала.
#  Это позволит избежать нажатия клавиш, которые могут «загрязнять» экран
#+ во время гонки.
stty -echo

# ----------------------------------------------------
# Начало гонки.

draw_horses
echo -ne '\E[37;47m'
move_and_echo 18 1 $BLANK80
echo -ne '\E[30m'
move_and_echo 18 1 Starting...
sleep 1

# Задается колонка финишной линии.
WINNING_POS=74

# Определение време от начала гонки.
START_TIME=`date +%s`

# Переменной COL следующей нужна конструкция "while".
COL=0

while [ $COL -lt $WINNING_POS ]; do

          MOVE_HORSE=0

          # Проверка возвращения функцией random_1_9 хорошего значения.
          while ! echo $MOVE_HORSE | grep [1-9] &> /dev/null; do
                MOVE_HORSE=`random_1_9`
          done

          # Определение прежнего типа и позиции "случайной лошади".
          HORSE_TYPE=`cat  horse_${MOVE_HORSE}_position | tail -n 1`
          COL=$(expr `cat  horse_${MOVE_HORSE}_position | head -n 1`)

          ADD_POS=1
          # Проверка, что текущая позиция является позицией забега.
          if seq 10 7 68 | grep -w $COL &> /dev/null; then
                if grep -w $MOVE_HORSE $COL &> /dev/null; then
                      ADD_POS=0
                      grep -v -w  $MOVE_HORSE $COL > ${COL}_new
                      rm -f $COL
                      mv -f ${COL}_new $COL
                      else ADD_POS=1
                fi
```

```
          else ADD_POS=1
          fi
          COL=`expr $COL + $ADD_POS`
          echo $COL >  horse_${MOVE_HORSE}_position # Сохранение новой позиции.

         # Выбор рисунка типа лошади.
          case $HORSE_TYPE in
                1) HORSE_TYPE=2; DRAW_HORSE=draw_horse_two
                ;;
                2) HORSE_TYPE=1; DRAW_HORSE=draw_horse_one
          esac
          echo $HORSE_TYPE >>  horse_${MOVE_HORSE}_position
          # Сохранение текущего типа.

          # Установка цвета переднего плана в черный, а фона в зеленый.
          echo -ne '\E[30;42m'

          # Перемещение курсора на новую позицию лошади.
          tput cup `expr $MOVE_HORSE + 5` \
          `cat  horse_${MOVE_HORSE}_position | head -n 1`

          # Отрисовка лошади.
          $DRAW_HORSE
           usleep $USLEEP_ARG

          # Когда все лошади вышли за рамки поля строки 15, перепечатка шансов.
           touch fieldline15
           if [ $COL = 15 ]; then
             echo $MOVE_HORSE >> fieldline15
           fi
           if [ `wc -l fieldline15 | cut -f1 -d " "` = 9 ]; then
               print_odds
               : > fieldline15
           fi

          # Определение лидирующей лошади.
          HIGHEST_POS=`cat *position | sort -n | tail -1`

          # Установка фона в белый цвет.
          echo -ne '\E[47m'
          tput cup 17 0
          echo -n Current leader: `grep -w $HIGHEST_POS *position | cut -c7`\
          "                              "
done

# Определение времени окончания скачки.
FINISH_TIME=`date +%s`

# Задание цвета фона зеленым и включение мигающего текста.
echo -ne '\E[30;42m'
echo -en '\E[5m'

# Делаем победившую лошадь мигающей.
tput cup `expr $MOVE_HORSE + 5` \
`cat  horse_${MOVE_HORSE}_position | head -n 1`
$DRAW_HORSE

# Отключается мигающий текст.
```

```
echo -en '\E[25m'

# Установка цвета переднего плана и фона белым.
echo -ne '\E[37;47m'
move_and_echo 18 1 $BLANK80

# Установка цвета переднего плана черным.
echo -ne '\E[30m'

# Делаем победителя мигающим.
tput cup 17 0
echo -e "\E[5mWINNER: $MOVE_HORSE\E[25m""  Odds: `cat odds_${MOVE_HORSE}`"\
"Время скачки: `expr $FINISH_TIME - $START_TIME` секунд"

# Восстановление курсора и прежних цветов.
echo -en "\E[?25h"
echo -en "\E[0m"

# Восстановление вывода на экран.
stty echo

# Удаление директории скачки temp.
rm -rf $HORSE_RACE_TMP_DIR

tput cup 19 0

exit 0
```

См. также Пример A-21, Пример A-44, Пример A-52-и Пример A-40.

Существует, однако, со всем этим серьезная проблема. *Управляющие последовательности ANSI* являются решительно непереносимыми. Что отлично работает на одних эмуляторах терминалов (или консоли) может работать по-разному, или вообще не работать, на других. «Раскрашенный» сценарий, который выглядит потрясающе на машине автора, может производить нечитаемым вывод на чужой машине. Это несколько подрывает полезность раскрашивания сценариев и возможно относит эту технику к статусу трюков. Раскрашенные сценарии, вероятно, будут неуместны в коммерческих условиях, т.е., ваш руководитель может их не одобрить.

Утилита *ansi* цветов ***Alister*** (основанная на утилите Moshe Jacobson) значительно упрощает использование управляющих последовательностей ANSI. Она заменяет чистым и логическим синтаксисом неуклюжие конструкции, которые мы только что обсудили.

Henry также создал утилиту (http://scriptechocolor.sourceforge.net/) для упрощения создания раскрашенных сценариев.

Примечания

[1] ANSI - это, конечно, акроним для American National Standards Institute. Этот высокий орган устанавливает и поддерживает различных технические и промышленные стандарты.

36.6. Оптимизация

Большинство сценариев оболочки - это быстрые и черновые решения не сложных проблем. Поэтому оптимизация их скорости не является сложной задачей. Хотя давайте рассмотрим случай, когда сценарий выполняет важную задачу, делает это хорошо, но работает слишком медленно. Переписывание его на компилируемый язык не является приемлемым вариантом. Проще исправить, переписав части сценария, которые его тормозят. Можно ли применять принципы оптимизации кода в скромном сценарии оболочки?

Проверьте циклы в сценарии. Время, затрачиваемое на повторяющиеся операции, быстро растет. Если это возможно, то удалите трудоемкие операции внутри цикла.

Используйте встроенные команды вместо системных команд. Встроенные выполняются быстрее и обычно не запускают подоболочку при вызове.

Избегайте ненужных команд, особенно в *конвейере*.

```
cat "$file" | grep "$word"

grep "$word" "$file"

#  Командные строки выше имеют одинаковый эффект, но вторая
#+ работает быстрее, так как она запускает на один процесс-потомок меньше.
```

Кажется особенно часто злоупотребляют в сценариях командой **`cat`**.

Отключение некоторых параметров Bash может ускорить выполнение сценариев.

Как поясняет Erik Brandsberg:

Если вам не нужна поддержка **Unicode**, вы сможете увеличить скорость примерно в 2 раза, просто установив переменную **`LC_ALL`**.

```
   export LC_ALL=C

   [определяет языковой стандарт как ANSI в Си,
      тем самым отключая поддержку Unicode]

[На примере сценария ...]

Без [поддержки Unicode]:
erik@erik-desktop:~/capture$ time ./cap-ngrep.sh
live2.pcap > out.txt

  real        0m20.483s
  user        1m34.470s
  sys         0m12.869s

С [поддержкой Unicode]:
```

```
erik@erik-desktop:~/capture$ time ./cap-ngrep.sh
live2.pcap > out.txt

  real        0m50.232s
  user        3m51.118s
  sys         0m11.221s

Я считаю, что большая часть затрат это оптимизация
используемых совпадений регулярных выражений [[string ~ REGEX]],
которая может помочь и другим частям кода.
Я не [видел это] отметил, что эта оптимизация помогает
в Bash, и я видел, что это помогло с «grep»,
так почему бы не попробовать?
```

Некоторые операторы, особенно **`expr`**, не очень эффективны и могут быть заменены *двойными скобками арифметического расширения*. См. Пример A-59.

```
Математические сравнения

math через $(( ))
real          0m0.294s
user          0m0.288s
sys           0m0.008s

math через expr:
real          1m17.879s   # Очень медленно!
user          0m3.600s
sys           0m8.765s

math через let:
real          0m0.364s
user          0m0.372s
sys           0m0.000s
```

Проверка состояния конструкций в сценариях заслуживает пристального внимания. Замена конструкции ***if-then*** на **`case`** и объединение проверок, когда это возможно, сводит к минимуму время выполнения сценария. Опять же обратитесь к Примеру A-59.

```
Проверка использования конструкции "case":
real          0m0.329s
user          0m0.320s
sys           0m0.000s


Проверка с if [], без кавычек:
real          0m0.438s
user          0m0.432s
sys           0m0.008s


Проверка с if [], с кавычками:
real          0m0.476s
user          0m0.452s
sys           0m0.024s
```

```
Проверка с if [], используя -eq:
real          0m0.457s
user          0m0.456s
sys           0m0.000s
```

Erik Brandsberg рекомендует использовать в большинстве случаев *ассоциативные массивы* вместо обычных цифровых индексированных массивов. При перезаписи значения ассоциативный массив имеет значительную производительность против числового массива. Сценарий проверки подтверждает это. См. Пример A-60.

```
Назначение испытаний

Назначение простой переменной
real          0m0.418s
user          0m0.416s
sys           0m0.004s

Назначение записи числового индекса массива
real          0m0.582s
user          0m0.564s
sys           0m0.016s

Перезапись записи числового индекса массива
real          0m21.931s
user          0m21.913s
sys           0m0.016s


Построчное чтение числового индекса массива
real          0m0.422s
user          0m0.416s
sys           0m0.004s

Назначение записи ассоциативного массива
real          0m1.800s
user          0m1.796s
sys           0m0.004s

Перезапись записи ассоциативного массива
real          0m1.798s
user          0m1.784s
sys           0m0.012s

Построчное чтение ассоциативного массива
real          0m0.420s
user          0m0.420s
sys           0m0.000s

Присвоение простой переменной случайного числа
real          0m0.402s
user          0m0.388s
sys           0m0.016s

Назначение случайной записи массива разреженного числового индекса
в 64k ячейки
real          0m12.678s
user          0m12.649s
sys           0m0.028s
```

```
Чтение записи разреженного числового индекса массива
real          0m0.087s
user          0m0.084s
sys           0m0.000s

Назначение случайной записи разреженного ассоциативного массива  в
64k ячейки
real          0m0.698s
user          0m0.696s
sys           0m0.004s

Чтение разреженного индекса ассоциативного массива
real          0m0.083s
user          0m0.084s
sys           0m0.000s
```

Используйте **`time`** и инструменты **`time`** в профиле интенсивных вычисляющих команд. Рассмотрите возможность перезаписи части медленного кода на ***Си*** или даже на ***Ассемблере***.

Постарайтесь свести к минимуму I/O файла. Bash не особенно эффективен при работе с файлами, так что попробуйте использовать для этого более подходящие инструменты в сценарии, например ***awk*** или ***Perl***.

Пишите ваши сценарии в модульной и последовательной форме, [1], так что бы они могли быть реорганизованы и сжаты по мере необходимости. Некоторые из способов оптимизации, применяемые в языках высокого уровня, могут работать в сценариях, а другие, такие как разворачивание цикла, в основном не имеют значения. Прежде всего используйте здравый смысл.

Отличную демонстрацию того, как оптимизация может резко сократить время выполнения сценария, см. Пример 16-47.

Примечания

[1] Обычно означает либеральное использование функций.

36.7. Разные советы

36.7.1. Идеи увеличения мощности сценариев

- У вас есть проблема, которую вы хотите решить путем написания сценария Bash. К сожалению, вы совсем не знаете с чего начать. Одним из способов является окунуться прямо в код и в те части сценария, которые решаются легко и писать постоянные части как *псевдо-код*.

```
#!/bin/bash

ARGCOUNT=1                      # Нужно имя, как аргумент.
```

```
E_WRONGARGS=65

if [ число-аргументов не-равно "$ARGCOUNT" ]
#    ^^^^^^^^^^^^^^^^ ^^^^^^^^
#  Не можете понять, как это написать кодом...
#+ ...тогда пишем псевдо-кодом

then
  echo "Usage: название название-сценария"
  #                    ^^^^^^^^^^^^^^^^^     Еще псевдо-код.
  exit $E_WRONGARGS
fi

...

exit 0


# Позже замените псевдо-код на рабочий код.

# Строка 6 станет:
if [ $# -ne "$ARGCOUNT" ]

# Строка 12 станет:
  echo "Usage: `название basename $0`"
```

Как пример использования псевдо-кода, см. упражнение *Square Root*.

- Сохраняйте запись сценария с которым пользователь работал во время сессии или на протяжении ряда сессий, добавляйте следующие строки в каждый сценарий, который вы хотите отслеживать. Они будут сохраняться в файле отчета название сценария и время запуска.

```
# Добавляйте (>>) в конце каждого отслеживаемого сценария.

whoami>> $SAVE_FILE    # Пользователь вызвавший сценарий.
echo $0>> $SAVE_FILE   # Имя сценария.
date>> $SAVE_FILE      # Дата и время.
echo>> $SAVE_FILE      # Пустые строки, как разделитель.

#  Конечно, SAVE_FILE определяется и экспортируется в качестве
#+ переменной среды в ~/.bashrc
#+ (иногда как ~/.scripts-run)
```

- Оператор **>>** ***добавляет*** строки в файл. Что делать, если вы хотите, чтобы *предварять* строку в существующем файле, то есть, чтобы вставлять ее в начале?

```
file=data.txt
title="***Это титульная строка данных текстового файла***"

echo $title | cat - $file >$file.new
# "cat -" объединяет stdout в $file.
#  Конечный результат это
#+ запись нового файла с $title, добавляемым в *начале*.
```

Это упрощенный вариант сценария Примера 19-13, данного ранее. И, конечно же, также это может сделать ***sed***.

- Сценарий оболочки может выступать в качестве встроенной команды внутри другого сценария оболочки, сценарий *Tcl* или *wish*, или даже **`Makefile`**. Он может быть вызван как внешняя команда оболочки в программе Си с помощью вызова *`system()`*, т.е., *`system('имя_сценария');`*.

- Установка переменной во встраиваемое содержимое сценария ***sed*** или ***awk*** повышает удобочитаемость обертки окружающей оболочку. См. Пример A-1 и Пример 15-20.

- Соберите вместе файлы, содержащие ваши любимые и самые полезные определения и функции. При необходимости «включайте» один или несколько этих «библиотечных файлов» в сценарии точкой (**`.`**) или командой **`source`**.

```
# БИБЛИОТЕКА СЦЕНАРИЕВ
# ------ -------

# Примечание:
# Здесь нет "#!".
# Так же нет "живого кода".


# Полезные определения переменных

ROOT_UID=0              # Root имеет $UID 0.
E_NOTROOT=101           # Ошибка, пользователь не root.
MAXRETVAL=255           # Максимальное (положительное) возвращаемое
                        #+ значение функцией.
SUCCESS=0
FAILURE=-1

# Функции

Usage ()                # "Usage:" сообщение.
{
  if [ -z "$1" ]        # Нет переданных аргументов.
  then
    msg=filename
  else
    msg=$@
  fi

  echo "Usage: `basename $0` "$msg""
}


Check_if_root ()        # Проверка запуска сценария root-ом.
{                       # Из примера "ex39.sh".
  if [ "$UID" -ne "$ROOT_UID" ]
  then
    echo "Для запуска сценария должен быть root."
    exit $E_NOTROOT
  fi
}
```

```
CreateTempfileName ()  # Создание "уникального" временного файла.
{                      # Из примера "ex51.sh".
  prefix=temp
  suffix=`eval date +%s`
  Tempfilename=$prefix.$suffix
}


isalpha2 ()            # Проверка *вся ли строка* из букв.
{                      # Из примера "isalpha.sh".
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
  *) return $SUCCESS;;
  esac                 # Thanks, S.C.
}


abs ()                             # Абсолютное значение.
{                                  # Внимание: максимальное возвращаемое
                                   #+ значение = 255.
  E_ARGERR=-999999

  if [ -z "$1" ]                   # Необходим переданный аргумент.
  then
    return $E_ARGERR               # Очевидно ошибка возвращаемого
                                   #+  значения.
  fi

  if [ "$1" -ge 0 ]                # Если не отрицательное,
  then                             # то
    absval=$1                      # остается как есть.
  else                             # Иначе,
    let "absval = (( 0 - $1 ))"    # меняем знак.
  fi

  return $absval
}


tolower ()             #  Конвертация передаваемых строк, как
{                      #+ аргумент(ов), в нижний регистр.

  if [ -z "$1" ]       #  Если нет переданных аргументов,
  then                 #+ посылается сообщение об ошибке
    echo "(null)"      #+ (Сообщение об ошибке с указателем
                       #+ void в Си-стиле)
    return             #+ и возвращение из функции.
  fi

  echo "$@" | tr A-Z a-z
  # Конвертация всех переданных аргументов ($@).

  return

# Используйте команду подстановки для задания переменной в функцию
# вывода.
# Например:
#    oldvar="A seT of miXed-caSe LEtTerS"
#    newvar=`tolower "$oldvar"`
```

```
#    echo "$newvar"    # набор смешанных букв
#
# Упражнение:Перепишите эту функцию для изменения передаваемых строчных
# аргументов в прописные ... toupper()  [легко].
}
```

- Используйте специальные символы в комментариях заголовков для повышения ясности и четкости в сценариях.

```
## Осторожно.
rm -rf *.zzy   ##  Опции "-rf" с "rm" очень опасны,
               ##+ особенно с символами подстановки.

#+ Продолжение строки.
#  Это первая строка
#+ многострочного комментария,
#+ а это конечная строка.

#* Примечание.

#o Элемент списка.

#> Другая точка зрения.
while [ "$var1" != "end" ]    #> while проверяет "$var1" != "end"
```

- Dotan Barak поддерживает код шаблона индикатора выполнения в сценарии.

Пример 36-17. Индикатор выполнения

```
#!/bin/bash
# progress-bar.sh

# Автор: Dotan Barak (очень небольшие изменения автора ABS Guide).
# Используется в ABS Guide с разрешения (спасибо!).

BAR_WIDTH=50
BAR_CHAR_START="["
BAR_CHAR_END="]"
BAR_CHAR_EMPTY="."
BAR_CHAR_FULL="="
BRACKET_CHARS=2
LIMIT=100

print_progress_bar()
{
        # Подсчет количества полных символов.
        let "full_limit = ((($1 - $BRACKET_CHARS) * $2) / $LIMIT)"

        # Подсчет количества пустых символов.
        let "empty_limit = ($1 - $BRACKET_CHARS) - ${full_limit}"

        # Подготовка панели.
        bar_line="${BAR_CHAR_START}"
        for ((j=0; j<full_limit; j++)); do
                bar_line="${bar_line}${BAR_CHAR_FULL}"
```

```
        done

        for ((j=0; j<empty_limit; j++)); do
                bar_line="${bar_line}${BAR_CHAR_EMPTY}"
        done

        bar_line="${bar_line}${BAR_CHAR_END}"

        printf "%3d%% %s" $2 ${bar_line}
}

# Вот пример кода, который это использует.
MAX_PERCENT=100
for ((i=0; i<=MAX_PERCENT; i++)); do
        #
        usleep 10000
        # ... Или запускает какие-то другие команды ...
        #
        print_progress_bar ${BAR_WIDTH} ${i}
        echo -en "\r"
done

echo ""

exit
```

- Очень умное использование конструкций *проверки* ***if*** блоков комментариев.

```
#!/bin/bash

COMMENT_BLOCK=
#  Попробуйте присвоить переменной выше некоторое значение

#+ для неприятного сюрприза.

if [ $COMMENT_BLOCK ]; then

Блок комментариев --
================================
Строка комментария.
Другая строка комментария.
Еще одна строка комментария.
================================

echo "Это не будет выведено на экран."

Блок комментариев без ошибок! Ура!

fi

echo "Пожалуйста, больше не комментируйте."

exit 0
```

Сравните это с использованием ***here documents*** для *закомментированых блоков кода*.

- Используя *статус выхода переменной* **$?**, сценарий может проверять, содержит ли

параметр только цифры и может ли он рассматривать его как целое число.

```
#!/bin/bash

SUCCESS=0
E_BADINPUT=85

test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
# Целое число, равное 0 или не равное 0.
# 2>/dev/null подавляет сообщение об ошибке.

if [ $? -ne "$SUCCESS" ]
then
  echo "Usage: `basename $0` вводимое_целое_число"
  exit $E_BADINPUT
fi

let "sum = $1 + 25"             # Выдаст ошибку, если $1 не целое число.
echo "Sum = $sum"

#  Любая переменная, а не только параметр командной строки, может быть
#+ проверена таким образом.

exit 0
```

- Для функции диапазон возвращаемых значений `0-255` является серьезным ограничением. Глобальные переменные и другие временные решения часто являются проблематичными. Альтернативный метод, чтобы функция передала значение обратно в основной сценарий, должен иметь функцию записи в `stdout` (обычно ***echo***) возвращаемого значения и присвоения его переменной. На самом деле это вариант *подстановки команды*.

- **Пример 36-18. Трюк с возвращаемым значением**

```
#!/bin/bash
# multiplication.sh

multiply ()                     #  Передаются несколько параметров.
{                               #  Будет приниматься переменное
                                #+ количество аргументов.

  local product=1

  until [ -z "$1" ]             # До использования переданных аргументов.
  do
    let "product *= $1"
    shift
  done

  echo $product                 #  Не выводится echo в stdout,
}                               #+ пока не будет присвоена переменная.

mult1=15383; mult2=25211
val1=`multiply $mult1 $mult2`
# Присваиваем переменной val1 функцию stdout (echo).
```

```
echo "$mult1 X $mult2 = $val1"                    # 387820813

mult1=25; mult2=5; mult3=20
val2=`multiply $mult1 $mult2 $mult3`
echo "$mult1 X $mult2 X $mult3 = $val2"           # 2500

mult1=188; mult2=37; mult3=25; mult4=47
val3=`multiply $mult1 $mult2 $mult3 $mult4`
echo "$mult1 X $mult2 X $mult3 X $mult4 = $val3" # 8173300

exit 0
```

- Этот же способ также работает для буквенно-цифровых строк. Это означает, что функция может «возвращать» не числовые значения.

```
capitalize_ichar ()           #  Инициализация прописными символами
{                             #+ передаваемой строки аргумента(ов).

  string0="$@"                # Принимает несколько аргументов.

  firstchar=${string0:0:1}    # Первый символ.
  string1=${string0:1}        # Остальная строка(и).

  FirstChar=`echo "$firstchar" | tr a-z A-Z`
                              # Прописной первый символ.

  echo "$FirstChar$string1"   # Вывод в stdout.

}

newstring=`capitalize_ichar "каждое предложение должно начинаться с
прописной буквы."`
echo "$newstring"           # Каждое предложение должно начинаться с

                            #+ прописной буквы.
```

- С помощью этого метода функция может " возвращать" даже несколько значений.

Пример 36-19. Еще более трюковое возвращаемое значение

```
#!/bin/bash
# sum-product.sh
# Функция может "возвращать" более одного значения.

sum_and_product ()   # Вычисление суммы и произведения передаваемых
                     #+ аргументов.
{
  echo $(( $1 + $2 )) $(( $1 * $2 ))
# Выводит в stdout каждое вычисленное значение, разделенное пробелом.
}

echo
echo "Введите первое число"
read first
```

```
echo
echo "Введите второе число"
read second
echo

retval=`sum_and_product $first $second`      # Присваивает вывод функции.
sum=`echo "$retval" | awk '{print $1}'`      # Присваивает первое поле.
product=`echo "$retval" | awk '{print $2}'`  # Присваивает второе поле.

echo "$first + $second = $sum"
echo "$first * $second = $product"
echo

exit 0
```

Здесь может быть только одно заявление **echo** в функцию.

Если вы измените предыдущий пример:

```
sum_and_product ()
{
  echo "Это функция sum_and_product." # Это испортит то, что выше!
  echo $(( $1 + $2 )) $(( $1 * $2 ))
}
...
retval=`sum_and_product $first $second`      # Присваивает вывод функции.
# Теперь это не будет работать правильно.
```

- Далее в нашем пакете приемов - это способы для передачи **массива** в **функцию**, а затем «возвращение» массива обратно в основную часть сценария.

  Передача массива включает в себя загрузку элементов массива, разделенных пробелами в переменную *подстановкой команды*. Для возвращения массива, как возвращаемого значения из функции, использует ранее упомянутая стратегия *вывода* массива в функцию, а затем вызов подстановки команды и оператора **(...)** для назначения массива.

Пример 36-20. Передача и возвращение массива

```
#!/bin/bash
# array-function.sh: Передача массива в функцию и ...
#                    "возвращение" массива из функции


Pass_Array ()
{
  local passed_array   # Локальная переменная!
  passed_array=( `echo "$1"` )
  echo "${passed_array[@]}"
  #  Объявляется список всех элементов нового массива
```

```
  #+ и присваивается функции.
}


original_array=( element1 element2 element3 element4 element5 )

echo
echo "original_array = ${original_array[@]}"
#                      Список всех элементов настоящего массива.


# Трюк, который позволяет передавать массив функции.
# *********************************
argument=`echo ${original_array[@]}`
# *********************************
#  Упаковка переменной со всеми разделительными пробелами
#+ между элементами в исходном массиве.
#
# Попытка просто передать сам массив не сработает.


#  Трюк, который позволяет захватывать массив,
#+ как "возвращаемое значение".
# ***************************************
returned_array=( `Pass_Array "$argument"` )
# ***************************************
# Присваивание «выводимого на экран» вывода функции в переменную массива.

echo "returned_array = ${returned_array[@]}"

echo "============================================================="

#  Теперь попробуем опять сначала,
#+ попытка доступа к (списку) массиву из внешней функции.
Pass_Array "$argument"

# Сама функция перечисляет массив, но...
#+ обращение к массиву из внешней функции запрещено.
echo "Переданный массив (внутренняя функция) = ${passed_array[@]}"
# ЗНАЧЕНИЕ NULL, поскольку массив имеет локальную переменную в функции.

echo

###############################################

# А вот еще более наглядный пример:

ret_array ()
{
  for element in {11..20}
  do
    echo "$element "   #  Вывод на экран отдельных элементов,
  done                 #+ которые будут собраны в массив.
}

arr=( $(ret_array) )   #  Сборка в массив.

echo "Захват массива \"arr\" из функции ret_array () ..."
echo "Третий элемент массива \"arr\" это ${arr[2]}."  # 13 (нулевое
                                                      #+  индексирование)
echo -n "Весь массив:"
```

```
echo ${arr[@]}                # 11 12 13 14 15 16 17 18 19 20

echo

exit 0

#  Nathan Coulter указывает на то, что передача массивов с элементами,
#+ содержащими пробелы, срывает этот пример.
```

Более сложный пример передачи массивов в функции, см. Пример A-10.

- Используя ***конструкцию двойных скобок***, можно использовать синтаксис Си-стиля для установки, увеличения/уменьшения переменных и в циклах ***for*** и ***while***. См. Пример 11-13 и Пример 11-18.

- Установка **`path`** и **`umask`** в начале сценария делает его более переносимым - работающим на «другой» машине, пользователь которой, возможно, изменил $PATH и **`umask`**.

```
#!/bin/bash
PATH=/bin:/usr/bin:/usr/local/bin ; export PATH
umask 022   # Файлы, которые создает сценарий, будут иметь права 755.

# Спасибо за замечание Ian D. Allen.
```

- Хорошим методом сценариев является *неоднократная* передача вывода фильтра (по конвейеру) обратно в тот же фильтр, но с другим набором аргументов и/или опций. Особенно подходит это для **`tr`** и **`grep`**.

```
# Из примера "wstrings.sh".

wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
```

- **Пример 36-21. Забава с анаграммами**

```
#!/bin/bash
# agram.sh: Играем в игру с анаграммами.

# Ищем анаграммы...
LETTERSET=etaoinshrdlu
FILTER='.......'       # Минимальное количество букв
#       1234567

anagram "$LETTERSET" | # Поиск всех анаграмм с набором букв...
grep "$FILTER" |       # с, по крайней мере, 7 буквами,
grep '^is' |           # начиная с 'is'
grep -v 's$' |         # нет слов во множественном числе
grep -v 'ed$'          # нет прошедшего времени глаголов
# Можно добавить множество комбинаций условий и фильтров.

#  Используется утилита «анаграмма», которая является
#+  частью авторского 'yawl' - пакета словаря.
```

```
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  http://bash.deta.in/yawl-0.3.2.tar.gz

exit 0                # Окончание кода.


bash$ sh agram.sh
islander
isolate
isolead
isotheral


#  Упражнения:
#  ---------
#  Измените этот сценарий, взяв LETTERSET как параметр командной строки.
#  Параметризуйте фильтры в строках 11-13 (как с $FILTER), что бы они
#+ могли задаваться путем передачи аргументов в функцию.
#  Измените подход к анаграммированию,
#+ см. сценарий agram2.sh.
```

См. также Пример 29-4, Пример 16-25 и Пример A-9.

- Используйте «*анонимные* ***here documents***» для комментариев из блоков кода, для сохранения индивидуальных комментариев из каждой строки с `#`. См. Пример 19-11.

- Запускать сценарий на машине зависящей от команды, которая не может быть установлена - опасно. Используйте **`whatis`**, чтобы избежать этих возможных проблем.

```
CMD=command1                # Первый выбор.
PlanB=command2              # Запасной вариант.

command_test=$(whatis "$CMD" | grep 'nothing appropriate')
#  Если 'command1' нет в системе, 'whatis' возвратит
#+ "command1: nothing appropriate."
#
#  Безопасной альтернативой является:
#     command_test=$(whereis "$CMD" | grep \/)
#  Но тогда смысл следующей проверки должен быть обратным, типа
#+ переменная $command_test имеет содержимое, только если
#+ в системе существует $CMD.
#     (Спасибо, bojster.)


if [[ -z "$command_test" ]]  # Проверка наличия команд.
then
  $CMD опция1 опция2         #  Запуск command1 с опциями.
else                         #  Иначе,
  $PlanB                     #+ запускается command2.
fi
```

- ***Проверка if-grep*** в случае ошибки может не возвратить ожидаемые результаты, когда текст выводится в `stderr`, раньше, чем в `stdout`.

```
if ls -l nonexistent_filename | grep -q 'Нет такого файла или директории'
```

```
  then echo "Файл \"nonexistent_filename\" не существует."
fi
```

- Перенаправление `stderr` в `stdout` исправляет это.

```
if ls -l nonexistent_filename 2>&1 | grep -q 'No such file or directory'
#                              ^^^^
  then echo "Файл \"nonexistent_filename\" не существует."
fi

# Спасибо Chris Martin за разъяснение.
```

- Если вам необходим абсолютный доступ к переменной подоболочки вне ее, вот способ это осуществить.

```
TMPFILE=tmpfile                    #  Создаем временный файл для хранения
                                   #+ переменной.

(   # Внутри subshell ...
inner_variable=Inner
echo $inner_variable
echo $inner_variable >>$TMPFILE  # Добавляем во временный файл.
)

    # Вне subshell ...

echo; echo "-----"; echo
echo $inner_variable               # Null, как и ожидалось.
echo "-----"; echo

# Теперь ...
read inner_variable <$TMPFILE      # Читаем обратно переменную оболочки.
rm -f "$TMPFILE"                   # Избавляемся от временного файла.
echo "$inner_variable"             # Это некрасивый ляп, но это работает.
```

- Команда **`run-parts`** полезна для выполнения набора сценариев команд в определенной последовательности, особенно в сочетании с **`cron`** или **`at`**.

- Что бы сделать несколько изменений в сложном сценарии, используйте пакет *rcs* Revision Control System.

  Среди других преимуществ это автоматическое обновление ID тегов заголовков. Команда **`co`** в *rcs* делает замену параметров некоторых зарезервированных ключевых слов, например, заменяя *`# $Id$`* в сценарии, что-то вроде этого:

```
# $Id: hello-world.sh,v 1.1 2004/10/16 02:43:05 bozo Exp $
```

36.7.2. Виджеты

Было бы неплохо иметь возможность ссылаться на виджеты X-Windows из сценария оболочки. К счастью существует несколько пакетов, которые помогают сделать это, а именно ***Xscript***, ***Xmenu*** и ***widtools***. Первые два из них, кажется, больше не поддерживаются. К счастью все еще возможно получить ***widtools***.

> Пакет ***widtools*** (инструменты для виджета) для установки требует библиотеку *XForms*. Кроме того необходимо разумное редактирование *Makefile* перед тем, как пакет будет встроен в типичную систему Linux. Наконец, три из шести предлагаемых виджетов не работают.

Семейство диалоговых инструментов предлагает способ вызова виджетов «*dialog*» из сценария оболочки. Оригинальная утилита ***dialog*** работает в текстовой консоли, но ее преемники, ***gdialog***, ***Xdialog*** и ***kdialog***, используют наборы виджетов основанных на X-Windows.

Пример 36-22. Виджеты, вызываемые из сценария оболочки

```
#!/bin/bash
# dialog.sh: Использование виджетов 'gdialog'.

# Для запуска этого сценария нужно установить на вашу систему 'gdialog'.
# Или,  можно заменить все примеры «gdialog» на «kdialog»...
# Версия 1.1 (корректирована 04/05/05)

# Этот сценарий был вдохновлен статьей
# "Scripting for X Productivity,"  Marco Fioretti,
# LINUX JOURNAL, Issue 113, September 2003, pp. 86-9.
# Спасибо Вам, добрые люди, от LJ.

# Ошибка ввода в диалоговом окне.
E_INPUT=85
# Размеры окна вводимых виджетов.
HEIGHT=50
WIDTH=60

# Имя выходного файла (создается из названия сценария).
OUTFILE=$0.output

# Этот сценарий можно вывести в текстовый виджет.
gdialog --title "Displaying: $0" --textbox $0 $HEIGHT $WIDTH


# Теперь попробуем сохранить ввод в файл.
echo -n "VARIABLE=" > $OUTFILE
gdialog --title "User Input" --inputbox "Введите пожалуйста переменную:" \
$HEIGHT $WIDTH 2>> $OUTFILE

if [ "$?" -eq 0 ]
# Хорошая привычка — проверять статус выхода.
```

```
then
  echo "\"Диалоговое окно\" выполнено без ошибок."
else
  echo "\"Диалоговое окно\ " выполнено с ошибкой (ами)."
        # Или, кликните "Cancel", вместо кнопки "OK".
  rm $OUTFILE
  exit $E_INPUT
fi

# Теперь мы  извлечем и выведем на экран сохраненную переменную.
.$OUTFILE   # 'Источник' сохраненный файл.
echo "Введенная переменная в \"окно ввода\": "$VARIABLE""


rm $OUTFILE  # Очищаем, удалив временный файл.
             # Некоторым приложениям может потребоваться сохранить этот файл.

exit $?

# Упражнение: Перепишите этот сценарий, используя набор виджетов «zenity».
```

Команда **`xmessage`** является простым способом появления окна сообщения/запроса.

Например:

```
xmessage Грубая ошибка в сценарии! -button exit
```

Последним явлением в соревновании виджетов является ***zenity***. Утилита виджетов всплывающих диалоговых окон *GTK+*, которая очень хорошо работает в сценарии.

```
get_info ()
{
  zenity --entry       #  Выпадает окно запроса ...
                       #+ и выводит запись пользователя в stdout.

                       #  Попробуйте опции --calendar и --scale.
}

answer=$( get_info )   #  Захватываем stdout в переменную $answer.

echo "Запись пользователя: "$answer""
```

Другие варианты сценариев с виджетами: попробуйте ***Tk*** или ***wish*** (производная от *Tcl*), ***PerlTk*** (*Perl* с расширениями *Tk*), ***tksh*** (*ksh* с расширениями *Tk*), ***XForms4Perl*** (*Perl* с расширениями *XForms*), ***Gtk-Perl*** (*Perl* с расширениями *Gtk*), или ***PyQt*** (*Python* с расширениями *Qt*).

36.8. Вопросы безопасности

36.8.1. Инфицированные сценарии оболочки

Это краткое предупреждение о безопасности сценария. Сценарий может содержать червей, трояны или даже вирусы. По этой причине никогда не запускайте сценарий из-под ***root*** (или с установленными правами на сценарий запуска системы `/etc/rc.d`), если только вы не получили его из надежного источника или тщательно проанализировали его на предмет безопасности.

Различные исследователи в Bell Labs и других, включая M. Douglas McIlroy, Tom Duff и Fred Cohen, изучили последствия вирусов сценария оболочки. Они делают вывод, что написать «детский сценарий» слишком легко даже для новичка. [1]

Вот еще одна причина, чтобы изучать сценарии. Вы будете в состоянии видеть и понимать, как сценарии могут защитить вашу систему от заражения больными сценариями.

36.8.2. Сокрытие источника сценария оболочки

В целях безопасности может быть необходимо сделать сценарий не читаемым. Для этого нужна утилита для создания из сценария облегченного исполняемого двоичного файла. ***shc*** -- ***generic shell script compiler***, написанная Francisco Rosales, делает именно это.

К сожалению, в статье от октября 2005 года Linux Journal написано, что двоичный код может быть, некоторых случаях, расшифрован, восстанавливая оригинал сценария. Тем не менее, это хороший способ защиты сохранения сценария от всех, кроме самых опытных хакеров.

36.8.3. Написание безопасных сценариев

Dan Stromberg предлагает следующие основополагающие принципы для написания (относительно) безопасных сценариев.

- Не вставляйте секретные данные в ***переменные окружения***.
- Не передавайте секретные данные аргументам внешних команд (вместо этого передавайте их через ***канал*** или ***перенаправления***).
- Тщательно устанавливайте ваш `$PATH`. Просто не доверяйте наследованию любого пути от вызывающей стороны, если ваш сценарий работает из-под root. В самом деле, всякий раз, когда вы используете переменную среды, унаследованную от вызывающего, подумайте, что может случиться, если вызывающий присвоит какую-то гадость переменной, например, если вызывающий устанавливает $HOME в `/etc`.

Примечания

[1] См. статью Marius van Oers, Unix Shell Scripting Malware, а так же Denning reference в

библиографии.

36.9. Переносимость

Легче портировать оболочку, чем сценарий оболочки.

--Larry Wall

Эта книга конкретно касается сценариев ***Bash*** в системе GNU/Linux. Все же, пользователи ***sh*** и ***ksh*** найдут здесь много полезного .

Как это случается, многие из различных оболочек и языков сценариев стремятся соответствовать стандарту POSIX 1003.2. Вызов Bash с опцией `--posix` или вставка в заголовок сценария `set -o posix` заставляет Bash очень близко соответствовать этому стандарту. Другой альтернативой является использование в сценарии заголовка sha-bang ***#!/bin/sh***, а не ***#!/bin/bash*.** [1] Обратите внимание, что в Linux и некоторых других UNIX `/bin/sh` представляет собой ссылку на `/bin/bash`, и сценарий, вызванный таким образом, отключает расширенную функциональность Bash.

Большинство сценариев Bash будут работать под ***ksh*** и наоборот, так как Chet Ramey оперативно портировал в ***ksh*** возможности последних версий Bash.

На коммерческих UNIX-машинах, сценарии, использующие особенности стандартных команд GNU, могут не работать. В последние несколько лет стало меньше проблем, так как довольно много утилит GNU перенесено даже на «big iron» UNIX.

Bash имеет определенные особенности, которых в традиционном Bourne shell не хватает. Среди них:

- Некоторые опции расширенного вызова
- Команды подстановки в нотации **`$( )`**
- Расширение фигурными скобками
- Некоторые операции с массивами и ассоциативные массивы
- Конструкция расширенной проверки с двойными скобками
- Конструкция арифметической оценки с двойными скобками

- Некоторые операции управления строками
- Процесс подстановки
- Оператор соответствия регулярных выражений
- ***builtins*** специфичные для Bash
- Сопроцессы

См. полный список в F.A.Q. Bash.

36.9.1. Набор тестов

Проиллюстрируем некоторые из несовместимостей между Bash и классической оболочкой Bourne shell. Скачайте и установите "Heirloom Bourne Shell" и запустите следующий сценарий, вначале используя ***Bash***, а затем классический ***sh***.

Пример 36-23. Набор тестов

```
#!/bin/bash
# test-suite.sh
# Небольшой набор тестов совместимости Bash.
# Запустите его на вашей версии Bash, или на других оболочках.

default_option=FAIL         # Тесты ниже не потерпят неудачу при ...

echo
echo -n "Testing "
sleep 1; echo -n ". "
sleep 1; echo -n ". "
sleep 1; echo ". "
echo

# Двойные скобки
String="Двойные скобки поддерживаются?"
echo -n "Тест двойных скобок: "
if [[ "$String" = "Двойные скобки поддерживаются?" ]]
then
  echo "PASS"
else
  echo "FAIL"
fi

# Двойные скобки и соответствие регулярных выражений (regex)
String="Поддерживается соответствие Regex?"
echo -n "Соответствие Regex: "
if [[ "$String" =~ R.....matching* ]]
then
  echo "ПЕРЕДАНО"
else
  echo "НЕ УДАЧНО"
fi

# Массивы
test_arr=$default_option     # НЕУДАЧНО
Array=( Если массивы поддерживаются будет выведено PASS )
test_arr=${Array[5]}
```

```
echo "Array test: $test_arr"

# Подстановка команд
csub_test ()
{
  echo "PASS"
}

test_csub=$default_option    # НЕУДАЧНО
test_csub=$(csub_test)csub_test (){}
echo "Тест подстановки команд: $test_csub"

echo

#  Завершение этого сценария является упражнением для читателя.
#  Добавить к вышеуказанным подобные тесты для двойных скобок,
#+ фигурных скобок, процесса подстановки и т.д.

exit $?
```

Примечания

[1] Или, еще лучше, ***#!/bin/env sh***

36.10. Написание сценариев оболочки под Windows

Даже пользователи, работающие в другой ОС могут запускать UNIX-подобные сценарии, и, следовательно, извлечь выгоду из уроков этой книги. Пакет ***Cygwin*** от Cygnus и утилиты ***MKS*** от Mortice Kern Associates добавляют возможности написания сценариев оболочки для Windows.

Другой альтернативой является UWIN, написанный David Korn из ATT, в Korn Shell.

В 2006 году Microsoft выпустила Windows, PowerShell®, которая содержит ограниченные Bash-подобные возможности сценариев командной строки.

Глава 37. Bash, версии 2, 3 и 4

Содержание

37.1. Bash, версия 2

Текущая версия Bash, установленная на вашем компьютере, наиболее вероятно является версией 2.xx.yy, 3.xx.yy или 4.xx.yy.

```
bash$ echo $BASH_VERSION
3.2.25(1)-release
```

В обновленной версии 2, классического языка сценариев Bash, среди особенностей добавлены массив переменных, расширение строк и параметров и улучшен способ косвенных ссылок на переменные.

Пример 37-1. Расширение строки

```
#!/bin/bash

# Расширение строки.
# Введено в версии 2 Bash.

#  Строки в форме $'xxx'
#+ просто представляют заэкранированные символы.

echo $'Колокольчик прозвонит 3 раза \a \a \a'
     # В некоторых терминалах может прозвонить один раз.
     # Или ...
     # Может не прозвонить вообще, в зависимости от настроек терминала.
echo $'Три формы формата \f \f \f'
echo $'10 переводов строк \n\n\n\n\n\n\n\n\n\n'
echo $'\102\141\163\150'
     #   B   a   s   h
     # Восьмеричный эквивалент символов.

exit
```

Пример 37-2. Косвенные ссылки на переменную - новый способ

```
#!/bin/bash

# Косвенная ссылка на переменную.
# Здесь есть несколько атрибутов ссылок C++.


a=letter_of_alphabet
letter_of_alphabet=z

echo "a = $a"              # Прямая ссылка.

echo "Теперь a = ${!a}"    # Косвенная ссылка.
#  Запись ${!variable} более понятна, чем старая
#+ eval var1=\$$var2

echo

t=table_cell_3
table_cell_3=24
echo "t = ${!t}"                     # t = 24
table_cell_3=387
echo "Значение t изменяется на ${!t}"    # 387
# Нет необходимости в 'eval'.

#  Это удобно для ссылок на элементы массива или таблицы,
```

```
#+ а так же для имитации многомерного массива.
#  Возможность индексации (по аналогии с указателями) была
#+ бы кстати. Вздох.

exit 0

# См. Так же пример ind-ref.sh .
```

Пример 37-3. Приложение простой базы данных, использующей косвенные ссылки на переменные.

```
#!/bin/bash
# resistor-inventory.sh
# Простая база данных / табличное приложение.

# ============================================================== #
# Данные

B1723_value=470                                 # Омы
B1723_powerdissip=.25                           # Ватты
B1723_colorcode="yellow-violet-brown"           # Цвет полос
B1723_loc=173                                   # Где они
B1723_inventory=78                              # Количество

B1724_value=1000
B1724_powerdissip=.25
B1724_colorcode="brown-black-red"
B1724_loc=24N
B1724_inventory=243

B1725_value=10000
B1725_powerdissip=.125
B1725_colorcode="brown-black-orange"
B1725_loc=24N
B1725_inventory=89

# ============================================================== #


echo

PS3='Введите номер из каталога: '

echo

select catalog_number in "B1723" "B1724" "B1725"
do
  Inv=${catalog_number}_inventory
  Val=${catalog_number}_value
  Pdissip=${catalog_number}_powerdissip
  Loc=${catalog_number}_loc
  Ccode=${catalog_number}_colorcode

  echo
  echo "Номер из каталоге $catalog_number:"
  # Теперь получим значение, используя косвенные ссылки.
  echo "Имеется ${!Inv} по [${!Val} ом / ${!Pdissip} ватт]\
  резисторов на складе."    # ^            ^
```

```
  # Как и в Bash 4.2, можно заменить "омы" на \u2126 (с помощью echo -e).
  echo "Находящиеся в корзине # ${!Loc}."
  echo "Их цветной код \"${!Ccode}\"."

  break
done

echo; echo

# Упражнения:
# ---------
# 1) Перепишите этот сценарий для чтения данных из внешнего файла.
# 2) Перепишите этот сценарий используя массивы,
#+   а не косвенные ссылки на переменную.
#    Какой способ является более простым и интуитивно понятным?
#    Какой способ легче записать кодом?


# Примечания:
# -----
#  Сценарии оболочки неуместны нигде, за исключением самых простых приложений
#+ баз данных, и даже тогда они включают в себя обходные пути и кладжи.
#  Намного лучше использовать языки с родной поддержкой структур данных,
#+ таких как C++ или Java (или даже Perl).

exit 0
```

Пример 37-4. Использование массивов и других фокусов для раздачи четырех случайных рук из колоды карт

```
#!/bin/bash
# cards.sh

# Раздача четырех случайных рук из колоды карт.

UNPICKED=0
PICKED=1

DUPE_CARD=99

LOWER_LIMIT=0
UPPER_LIMIT=51
CARDS_IN_SUIT=13
CARDS=52

declare -a Deck
declare -a Suits
declare -a Cards
#  Было бы проще реализовать и более интуитивно с единственным 3-х мерным
#+ массивом.
#  Возможно, будущая версия Bash будет поддерживать многомерные массивы.


initialize_Deck ()
{
i=$LOWER_LIMIT
until [ "$i" -gt $UPPER_LIMIT ]
do
```

```
  Deck[i]=$UNPICKED   # Устанавливаем каждую карту "Deck" как не розданную.
  let "i += 1"
done
echo
}

initialize_Suits ()
{
Suits[0]=C #Кресты
Suits[1]=D #Буби
Suits[2]=H #Черви
Suits[3]=S #Пики
}

initialize_Cards ()
{
Cards=(2 3 4 5 6 7 8 9 10 J Q K A)
# Еще один способ инициализации массива
}

pick_a_card ()
{
card_number=$RANDOM
let "card_number %= $CARDS" # Ограничиваем диапазон 0 - 51, т.е., 52 карты.
if [ "${Deck[card_number]}" -eq $UNPICKED ]
then
  Deck[card_number]=$PICKED
  return $card_number
else
  return $DUPE_CARD
fi
}

parse_card ()
{
number=$1
let "suit_number = number / CARDS_IN_SUIT"
suit=${Suits[suit_number]}
echo -n "$suit-"
let "card_no = number % CARDS_IN_SUIT"
Card=${Cards[card_no]}
printf %-4s $Card
# Вывод карт аккуратными столбцами.
}

seed_random ()  # Источник генерирования случайных чисел.
{               # Что случится, если этого не сделать?
seed=`eval date +%s`
let "seed %= 32766"
RANDOM=$seed
} # Рассмотрите другие виды источников генерирования случайных чисел.

deal_cards ()
{
echo

cards_picked=0
while [ "$cards_picked" -le $UPPER_LIMIT ]
do
  pick_a_card
  t=$?
```

```
  if [ "$t" -ne $DUPE_CARD ]
  then
    parse_card $t

    u=$cards_picked+1
    # Переключаемся обратно на 1 индексации, временно. Почему?
    let "u %= $CARDS_IN_SUIT"
    if [ "$u" -eq 0 ]   # Вложеная проверка условий if/then.
    then
     echo
     echo
    fi                  # Каждая рука отделяется пустой строкой.

    let "cards_picked += 1"
  fi
done

echo

return 0
}


# Структурное программирование:
# Вся логика программы модулирована в функции.

#===============
seed_random
initialize_Deck
initialize_Suits
initialize_Cards
deal_cards
#===============

exit



# Упражнение 1:
# Добавьте комментарии для тщательного документирования этого сценария.

# Упражнение 2:
# Добавление обычный (функция) вывод каждой руки, отсортированный по мастям.
# Если хотите, можете добавить любые няшки и красивки.

# Упражнение 3:
# Упростите и рационализируйте логику сценария.
```

37.2. Bash, версия 3

27 июля 2004 г., Chet Ramey представил 3 версию Bash. В ней исправлен ряд ошибок и добавлены новые функции.

Некоторые из более важных добавленных функций:

- Новый, более обобщенный оператор скобок расширения **{a...z}**.

```
#!/bin/bash

for i in {1..10}
#  Проще и еще проще, чем
#+ for i in $(seq 10)
do
  echo -n "$i "
done

echo

# 1 2 3 4 5 6 7 8 9 10



# Или просто ...

echo {a..z}    #  a b c d e f g h i j k l m n o p q r s t u v w x y z
echo {e..m}    #  e f g h i j k l m
echo {z..a}    #  z y x w v u t s r q p o n m l k j i h g f e d c b a
               #  Срабатывает обратно.
echo {25..30}  #  25 26 27 28 29 30
echo {3..-2}   #  3 2 1 0 -1 -2
echo {X..d}    #  X Y Z [  ] ^ _ ` a b c d
               #  Показывает (некоторые) символы ASCII между Z и a,
               #+ но не полагай1тесь на этот тип поведения, потому что...
echo {]..a}    #  {]..a}
               #  Почему?


# Вы можете играть с префиксами и суффиксами.
echo "Number #"{1..4}, "..."
     # Number #1, Number #2, Number #3, Number #4, ...


# Можно объединять наборы скобок расширения.
echo {1..3}{x..z}" +" "..."
     # 1x + 1y + 1z + 2x + 2y + 2z + 3x + 3y + 3z + ...
     # Создавая алгебраическое выражение.
     # Это может использоваться для поиска перестановок.

# Вы можете вкладывать наборы скобок расширения.
echo {{a..c},{1..3}}
     # a b c 1 2 3
     # "Оператор запятая" соединяет строки вместе.

# ########## ######### ############ ########### ######### ###############
# К сожалению фигурные скобки расширения не поддаются параметризации.
var1=1
var2=5
echo {$var1..$var2}   # {1..5}


#  Но, как поясняет Emiliano G., с помощью «eval» это ограничение
#+ преодолевается.

start=0
end=10
```

```
for index in $(eval echo {$start..$end})
do
  echo -n "$index "   # 0 1 2 3 4 5 6 7 8 9 10
done

echo
```

- Оператор **${!array[@]}**, расширяет все элементы данного массива.

```
#!/bin/bash

Array=(элемент-ноль элемент-один элемент-два элемент-три)

echo ${Array[0]}   # элемент-ноль
                   # первый элемент Array.

echo ${!Array[@]}  # 0 1 2 3
                   # Все элементы Array.

for i in ${!Array[@]}
do
  echo ${Array[i]} # элемент-ноль
                   # элемент-один
                   # элемент-два
                   # элемент-три
                   #
                   # Все элементы в Array.
done
```

- Оператор **=~** совпадения Регулярных Выражений в двойных скобках проверяет выражение. (***Perl*** имеет аналогичный оператор).

```
#!/bin/bash

# Соответствие Регулярных выражений оператором =~ в
variable="This is a fine mess."

echo "$variable"

# Соответствие РВ оператором =~ в [[ двойных скобках ]].
if [[ "$variable" =~ T.........fin*es* ]]
#  ПРИМЕЧАНИЕ: Как и в версии 3.2 Bash, сравниваемые выражения больше не
#+ заключаются в кавычки.
then
  echo "match found"
      # match found
fi
```

- Или, более целесообразно:

```
#!/bin/bash
```

```
input=$1


if [[ "$input" =~ "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" ]]
#                 ^ ПРИМЕЧАНИЕ: Кавычки не обязательны, начиная с версии
#+ 3.2 of Bash.
# NNN-NN-NNNN (где каждая N является цифрой).
then
  echo "Social Security number."
  # Процесс SSN.
else
  echo "He Social Security number!"
  # Или, попросить правильно ввести.
fi
```

Дополнительные примеры использования оператора **=~** см. Пример A-29, Пример 19-14, Пример A-35 и Пример A-24.

- Новая опция **set -o pipefail** полезна для отладки конвейера. Если эта опция установлена, то статусом выхода конвейера является статус выхода последней *не выполненной (или выполненной с ошибкой)* команды (возвращает не нулевое значение), а не фактической конечной команды в конвейере.

  См. Пример 16-43.

Обновление до версии 3 Bash не позволит запускать некоторые сценарии, которые работали под более ранними версиями. *Проверьте критическую наследовательность сценариев, чтобы убедиться, что они будут работать!*

Как это часто случается, пару сценариев в руководстве *Advanced Bash Scripting* нужно исправить (например Пример 9-4).

37.2.1. Bash, версия 3.1

Обновление Bash до версии 3.1 вводит исправление ряда ошибок и несколько незначительных изменений.

- Оператор **+=** теперь разрешен в местах, где в ранее признавался только оператор присваивания **=** .

```
a=1
echo $a        # 1

a+=5           # Не работает с версиями Bash ранее 3.1.
echo $a        # 15

a+=Hello
echo $a        # 15Hello
```

Здесь **=+** функционирует в качестве оператора объединения. Обратите внимание, что его поведение в данном контексте отлично от конструкции **let**.

```
a=1
echo $a        # 1

let a+=5       #  Целочисленная арифметическая операция, а не объединение
               #+ строк.
echo $a        # 6

let a+=Hello   # Ничего не "добавляет" в a.
echo $a        # 6
```

Jeffrey Haemer указывает на то, что этот оператор объединения может быть весьма полезен. В случае добавления директории в $PATH.

```
bash$ echo $PATH
/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin/:/usr/games


bash$ PATH+=:/opt/bin

bash$ echo $PATH
/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin/:/usr/games:/opt/bin
```

37.2.2. Bash, версия 3.2

В этом обновлении довольно много исправлений.

- В *глобальной подстановке параметров*, шаблон больше не якорится в начале строки.
- Опция `--wordexp` отключает процесс подстановки
- Оператору соответствия Регулярных выражений **=~** больше не нужны ***кавычки*** для шаблона в **[[…]]**.

В самом деле, кавычки в этом контексте не целесообразны, так как это может привести к неправильной оценке регулярного выражения. Chet Ramey в Bash FAQ заявляет, что кавычки *явно отключают* оценку регулярного выражения. Смотрите также Ubuntu Bug List и Wikinerds on Bash syntax.

Установка *shopt -s compat31* в сценарий вызывает возврат к оригинальному поведению.

37.3. Bash, версия 4

Chet Ramey анонсировал версию 4 Bash 20 февраля, 2009. Этот релиз содержит ряд важных новых возможностей, а также некоторые важные исправления.

Среди новых свойств:

- Ассоциативные массивы. [1]

> *Ассоциативный* массив можно рассматривать как совокупность двух связанных массивов - единое объединение данных и других *ключей*, указывающих на отдельные элементы *данных* массива.

Пример 37-5. Простая адресная база данных

```
#!/bin/bash4
# fetch_address.sh

declare -A address
#       Опция -A объявляет ассоциативный массив.

address[Charles]="414 W. 10th Ave., Baltimore, MD 21236"
address[John]="202 E. 3rd St., New York, NY 10009"
address[Wilma]="1854 Vermont Ave, Los Angeles, CA 90023"


echo "Адрес Charles  ${address[Charles]}."
# Адресом Charles является 414 W. 10th Ave., Baltimore, MD 21236.
echo "Адрес Wilma  ${address[Wilma]}."
# Адресом Wilma является 1854 Vermont Ave, Los Angeles, CA 90023.
echo "Адрес John  ${address[John]}."
# Адресом John является 202 E. 3rd St., New York, NY 10009.

echo

echo "${!address[*]}"   # Содержимое (индексы) массива ...
# Charles John Wilma
```

Пример 37-6. Несколько более сложная адресная база данных

```
#!/bin/bash4
# fetch_address-2.sh
# Более сложная версия fetch_address.sh.

SUCCESS=0
E_DB=99    # Код ошибки для отсутствующей записи.

declare -A address
#       Опция -A объявляет ассоциативный массив.
```

```
store_address ()
{
  address[$1]="$2"
  return $?
}


fetch_address ()
{
  if [[ -z "${address[$1]}" ]]
  then
    echo "Адрес $1 в базе отсутствует."
    return $E_DB
  fi

  echo "Адрес $1's является ${address[$1]}."
  return $?
}


store_address "Lucas Fayne" "414 W. 13th Ave., Baltimore, MD 21236"
store_address "Arvid Boyce" "202 E. 3rd St., New York, NY 10009"
store_address "Velma Winston" "1854 Vermont Ave, Los Angeles, CA 90023"

#  Упражнение:
#  Перепишите  store_address выше, что бы вызывать для чтения данные из
#+ файла и затем присваивать в массиве имени - поле , а адресу - поле 2.
#  Каждая строка в файле должна иметь формат, как указано выше.
#  С помощью цикла while-read читайте из файла, а с sed или awk
#+ анализируйте поля.

fetch_address "Lucas Fayne"
# Адресом Lucas Fayne является 414 W. 13th Ave., Baltimore, MD 21236.
fetch_address "Velma Winston"
# Адресом Velma Winston является 1854 Vermont Ave, Los Angeles, CA 90023.
fetch_address "Arvid Boyce"
# Адресом Arvid Boyce является 202 E. 3rd St., New York, NY 10009.
fetch_address "Bozo Bozeman"
# Адрес Bozo Bozeman в базе отсутствует.

exit $?   # В этом случае функции возвращается код выхода = 99.
```

См. Пример A-53 использования *ассоциативного массива*.

Элементы индексов массива могут содержать внутри пробелы, или даже ведущие и/или завершающие пробелы. Однако индекс массива элементов *не может состоять только из пробелов*.

```
address[   ]="Blank"   # Ошибка!
```

- Усовершенствование конструкции **case**: завершение *;;&* и *;&* .

Пример 37-7. Проверка символов

```
#!/bin/bash4

test_char ()
{
  case "$1" in
    [[:print:]] )  echo "$1 это печатаемый символ.";;&          # |
    # Завершение ;;& продолжает проверку следующего шаблона.      |
    [[:alnum:]] )  echo "$1 это буквенный/цифровой символ.";;&  # v
    [[:alpha:]] )  echo "$1 это буквенный символ.";;&           # v
    [[:lower:]] )  echo "$1 это символ строчной буквы.";;&
    [[:digit:]] )  echo "$1 это символ цифра.";&               # |
    # Завершение ;& выполняет следующий оператор ...            # |
    %%%@@@@@    )  echo "********************************";;    # v
#   ^^^^^^^^  ... даже дурацкого шаблона.
  esac
}

echo

test_char 3
# 3 это печатаемый символ.
# 3 это буквенный/цифровой символ.
# 3 это цифровой символ.
# ********************************
echo

test_char m
# m это печатаемый символ.
# m это буквенный/цифровой символ.
# m это буквенный символ.
# m это символ строчной буквы.
echo

test_char /
# / это печатаемый символ.

echo

# Завершение ;;& может сохранить сложное условие if/then.
# ;& является несколько менее полезным.
```

- Новый встроенный **`coproc`** позволяет общаться и взаимодействовать двум параллельным процессам. Как написал Chet Ramey в Bash FAQ [2], ver. 4.01:

  > «**`Coproc`**» это новое зарезервированное слово, сокращение от ***coprocess***: асинхронная команда запуска с двумя объединенными конвейерами создающими оболочку. **`Coproc`** могут быть именованы. Дескрипторы ввода и вывода файлов и PID сопроцессов доступны для вызова оболочки переменных с указанными именами **`coproc`**.
  >
  > George Dimitriu объясняет, '... **`coproc`**... является компонентом используемым в Bash процессом замены, который теперь доступен всем.»

Это означает, что он может быть вызван явно в сценарии, а не просто закулисным механизмом используемым Bash.

Сопроцессы используют *дескрипторы файлов*. ***Файловые дескрипторы позволяют процессам и конвейерам взаимодействовать.***

```
#!/bin/bash4
# Взаимодействие сопроцесса с циклом while-read.


coproc { cat mx_data.txt; sleep 2; }
#                         ^^^^^^^
# Попробуй запустить без "sleep 2" и увидишь, что случится.

while read -u ${COPROC[0]} line    #  ${COPROC[0]} это файловый
do                                 #+ дескриптор сопроцесса.
  echo "$line" | sed -e 's/line/NOT-ORIGINAL-TEXT/'
done

kill $COPROC_PID                   #  Сопроцесс больше не нужен,
                                   #+ поэтому его PID убивается.
```

Но будьте осторожны!

```
#!/bin/bash4

echo; echo
a=aaa
b=bbb
c=ccc

coproc echo "one two three"
while read -u ${COPROC[0]} a b c;  #  Обратите внимание, что это цикл
do                                 #+ запускается в subshell.
  echo "Внутри цикла while-read: ";
  echo "a = $a"; echo "b = $b"; echo "c = $c"
  echo "файловый дескриптор coproc: ${COPROC[0]}"
done

# a = one
# b = two
# c = three
# И т.д., прекрасно, но ...

echo "-----------------"
echo "Вне цикла while-read: "
echo "a = $a"  # a =
echo "b = $b"  # b =
echo "c = $c"  # c =
echo "файловый дескриптор coproc: ${COPROC[0]}"
echo
#  coproc все еще запущен, а ...
#+  до сих пор не включено наследование родительским процессом
#+ переменных из дочернего процесса, цикла while-read.
#  Отрывок из сценария "badread.sh".
```

Сопроцесс является *асинхронным*, а это может вызвать проблемы. Он может прекратить общение до завершения другого процесса.

```
#!/bin/bash4

coproc cpname { for i in {0..10}; do echo "index = $i";
done; }
#      ^^^^^^ Это *именованный* сопроцесс.
read -u ${cpname[0]}
echo $REPLY         #  index = 0
echo ${COPROC[0]}   #+ Не выводит ... сопроцесс закончился
                    #+ после первой итерации цикла.



# Однако, George Dimitriu частично это исправил.

coproc cpname { for i in {0..10}; do echo "index = $i"; done;
sleep 1;
echo hi > myo; cat - >> myo; }
#       ^^^^^ это *именованный* сопроцесс.

echo "Я главный"$'\04' >&${cpname[1]}
myfd=${cpname[0]}
echo myfd=$myfd

### while read -u $myfd
### do
###   echo $REPLY;
### done

echo $cpname_PID

#  Запустите с и без закомментированого цикла while-loop,
#+ и очевидно, что каждый процесс, выполняющийся оболочкой и
#+ сопроцесс, ожидает окончания записи собственной записи
#+ конвейером.
```

- Новый встроенный **`mapfile`** позволяет загружать массив содержимого текстового файла без использования цикла или *подстановки команд*.

```
#!/bin/bash4

mapfile Arr1 < $0
# Результат наподобие     Arr1=( $(cat $0) )
echo "${Arr1[@]}"  # Копирует весь сценарий на stdout.

echo "--"; echo

# Но это не то же самое, что  read -a   !!!
read -a Arr2 < $0
echo "${Arr2[@]}"  # Считывает только первую строку сценария в массиве.

exit
```

- Встроенное **`read`** получило незначительное изменение. Опция `timeout -t` теперь принимает дробные значения (десятичные) [3], а опция `-i` позволяет редактировать предустановки буфера. [4] К сожалению эти улучшения все еще в разработке и их (пока) нельзя использовать в сценариях.
- *Подстановка параметров* получила выбор операторов.

```
#!/bin/bash4

var=veryMixedUpVariable
echo ${var}           # veryMixedUpVariable
echo ${var^}          # VeryMixedUpVariable
#         *             Первый символ --> прописной.
echo ${var^^}         # VERYMIXEDUPVARIABLE
#         **            Все символы  --> прописные.
echo ${var,}          # veryMixedUpVariable
#         *             Первый символ --> строчный.
echo ${var,,}         # verymixedupvariable
#         **            Все символы  --> строчные.
```

- Встроенный **`declare`** с опциями `-l` теперь преобразует в строчные, а `-c` в прописные.

```
#!/bin/bash4

declare -l var1           # Будут изменены на строчные
var1=MixedCaseVARIABLE
echo "$var1"              # mixedcasevariable
# Same effect as            echo $var1 | tr A-Z a-z

declare -c var2           # Изменяет только первый символ в прописной.
var2=originally_lowercase
echo "$var2"              # Originally_lowercase
# НЕ такой эффект, как  echo $var2 | tr a-z A-Z
```

- Расширение в фигурных скобках получило больше опций.

  Увеличение/уменьшение, указывается в фигурных скобках в конце .

```
#!/bin/bash4

echo {40..60..2}
# 40 42 44 46 48 50 52 54 56 58 60
# Все четные числа, между 40 и 60.

echo {60..40..2}
# 60 58 56 54 52 50 48 46 44 42 40
# Все четные числа, между 40 и 60, обратный отсчет.
# Эффект уменьшения.
echo {60..40..-2}
# Тот же самый вывод. Знак минус не обязателен.

# А как с буквами и символами?
```

```
echo {X..d}
# X Y Z [  ] ^ _ ` a b c d
# Не выводит \ который экранирует пустое место.
```

Нули, указанные в начале фигурных скобок, предваряют каждый элемент выходных данных тем же числом нулей.

```
bash4$ echo {010..15}
010 011 012 013 014 015


bash4$ echo {000..10}
000 001 002 003 004 005 006 007 008 009 010
```

- Извлечение позиционных параметров ***Substring*** теперь начинается с `$0`, как *нулевого индекса*. (Это исправляет противоречивости при извлечении позиционных параметров).

```
#!/bin/bash
# show-params.bash
# Требуется версия Bash 4+.

# Вызовите этот сценарий хотя бы с одним позиционным параметром.

E_BADPARAMS=99

if [ -z "$1" ]
then
  echo "Usage $0 param1 ..."
  exit $E_BADPARAMS
fi

echo ${@:0}

# bash3 show-params.bash4 one two three
# one two three

# bash4 show-params.bash4 one two three
# show-params.bash4 one two three

# $0               $1  $2  $3
```

- Новый оператор подстановки *рекурсивного* соответствия имен файлов и директорий **`**`**.

```
#!/bin/bash4
# filelist.bash4

shopt -s globstar  #  Необходимо включить globstar, в противном случае **
                   #+ не будет работать.
```

```
                    # Опция о globstar новелла в версии 4 Bash.

echo "Используем *"; echo
for filename in *
do
  echo "$filename"
done   # Только список файлов текущей директории ($PWD).

echo; echo "--------------"; echo

echo "Используем **"
for filename in **
do
  echo "$filename"
done   # Окончательный список фалов дерева, рекурсивно.

exit

Использование *

allmyfiles
filelist.bash4

--------------

Исользование **

allmyfiles
allmyfiles/file.index.txt
allmyfiles/my_music
allmyfiles/my_music/me-singing-60s-folksongs.ogg
allmyfiles/my_music/me-singing-opera.ogg
allmyfiles/my_music/piano-lesson.1.ogg
allmyfiles/my_pictures
allmyfiles/my_pictures/at-beach-with-Jade.png
allmyfiles/my_pictures/picnic-with-Melissa.png
filelist.bash4
```

- Новая внутренняя переменная **$BASHPID**.
- Новая *встроенная* функция обработки ошибок названная **command_not_found_handle**.

```
#!/bin/bash4

command_not_found_handle ()
{ # Принимает неявные параметры.
  echo "Следующая команда является не правильной: \""$1\"""
  echo "Следующие аргументы: \""$2\"" \""$3\"""   # $4, $5 ...
} # $1, $2, и т.д. не переданы функции явно.

bad_command arg1 arg2

# Следующая команда не правильна: "bad_command"
# Следующие аргументы: "arg1" "arg2"
```

Редакционный комментарий

Ассоциативные массивы? Сопроцессы? При изучении и вникании в Bash мы, случайно, желаем узнавать больше? Может быть боитесь (ужас!)? Или возможно завидуете знатокам Korn shell?

Примечание Chet Ramey: Пожалуйста, добавляйте только *необходимые* функции в будущие версии Bash - возможно циклы ***for-each*** и поддержку многомерных массивов. Большинство пользователей Bash не нуждается, не пользуется и скорее всего не оценит очень сложные «особенности», такие как встроенные отладчики, интерфейсы ***Perl*** и ускорители.

37.3.1. Bash, версия 4.1

Версия 4.1 Bash, представленная в в мае 2010 г., являлась главным образом обновлением исправлений .

- Команда **`printf`** теперь принимает опцию `-v` для настройки индексов ***массива***.
- В двойных *квадратных скобках*, операторы сравнения строк **>** и **<**, теперь соответствуют ***локали***. Поскольку локаль может повлиять на порядок сортировки строковых выражений, то имеет побочные эффекты при проверке сравнения выражений в **`[[...]]`** .
- Встроенное **`read`** теперь принимает опцию `-N` (***`read -N chars`***), которая вызывает прекращение действия **`read`** после ***chars*** символов.

  Пример 37-8. Чтение N символов

```
#!/bin/bash
# Требуется версия Bash -ge 4.1 ...

num_chars=61

read -N $num_chars var < $0   # Читает первые 61 символы сценария!
echo "$var"
exit

####### Вывод сценария #######

#!/bin/bash
# Требуется версия Bash -ge 4.1 ...

num_chars=61
```

- ***Here documents*** встроенная в конструкцию подстановки команд **`$(...)`** может завершаться просто **`)`**.

  Пример 37-9. Присваивание переменной с помощью *here document*

```
#!/bin/bash
# here-commsub.sh
# Требуется версия Bash -ge 4.1 ...

multi_line_var=$( cat <<ENDxxx
------------------------------
Это строка 1 переменной
Это строка 2 переменной
Это строка 3 переменной
------------------------------
ENDxxx)

#  Вместо того, как требуется в Bash 4.0:
#+ окончание limit string и завершающая закрывающая
#+ скобка должны находиться на отдельных строках.
# ENDxxx
# )


echo "$multi_line_var"

#  Bash пока еще выдает ошибку.
#  warning: here-document at line 10 delimited
#+ by end-of-file (wanted `ENDxxx')
```

37.3.2. Bash, версия 4.2

Версия 4.2 Bash, вышедшая в феврале 2011 г., помимо исправлений, содержит ряд новых функций и усовершенствований.

- Bash теперь поддерживает экранирование *\u* и *\U Юникода*.

> Юникод является кросс-платформенным стандартом кодирования в числовых значений букв и графических символов. Он позволяет представлять и отображать символы иностранных алфавитов и необычных шрифтов.

```
echo -e '\u2630'   # Горизонтальная тройная панель символа.
# Эквивалентно:
echo -e "\xE2\x98\xB0"
                   # Признаваемое ранними версиями Bash.

echo -e '\u220F'   # PI (Греческие буквы и математические символы)
echo -e '\u0416'   # Заглавные "ZHE" (Буквы кириллицы)
echo -e '\u2708'   # Символ самолета (шрифт Dingbat)
echo -e '\u2622'   # Треугольник Радиация

echo -e "The amplifier circuit requires a 100 \u2126 pull-up resistor."


unicode_var='\u2640'
echo -e $unicode_var      # Женский символ
printf "$unicode_var \n"  # Женский символ и новая строка
```

```
#  А другие разрабатываются ...

#  Мы можем хранить символы Unicode в ассоциативном массиве,
#+ а затем извлекать их по имени.
#  Запустите  gnome-терминал или терминал с крупным, жирным шрифтом
для повышения удобочитаемости.
#+ для повышения удобочитаемости.

declare -A symbol  # Ассоциативный массив.

symbol[script_E]='\u2130'
symbol[script_F]='\u2131'
symbol[script_J]='\u2110'
symbol[script_M]='\u2133'
symbol[Rx]='\u211E'
symbol[TEL]='\u2121'
symbol[FAX]='\u213B'
symbol[care_of]='\u2105'
symbol[account]='\u2100'
symbol[trademark]='\u2122'


echo -ne "${symbol[script_E]}   "
echo -ne "${symbol[script_F]}   "
echo -ne "${symbol[script_J]}   "
echo -ne "${symbol[script_M]}   "
echo -ne "${symbol[Rx]}   "
echo -ne "${symbol[TEL]}   "
echo -ne "${symbol[FAX]}   "
echo -ne "${symbol[care_of]}   "
echo -ne "${symbol[account]}   "
echo -ne "${symbol[trademark]}   "
echo
```

Приведенный выше пример использует конструкцию расширения строки **`$'...'`** .

- Если задана опция оболочки `lastpipe`, последняя команда конвейера в ***subshell*** не запускается.

Пример 37-10. Передача ввода на *read*

```
#!/bin/bash
# lastpipe-option.sh

line=''                    # Значение Null.
echo "\$line = "$line""    # $line =

echo

shopt -s lastpipe          # Ошибка в версии Bash -lt 4.2.
echo "Статус выхода при попытке установить опцию \"lastpipe\" будет $?"
#     1 если версия Bash -lt 4.2, 0 , если иная.
```

```
echo

head -1 $0 | read line    # Передает первую строку сценария в read.
#            ^^^^^^^^^      Не в subshell!!!

echo "\$line = "$line""
# Older Bash releases       $line =
# Bash version 4.2          $line = #!/bin/bash
```

Этот параметр предоставляет возможность «исправления» примеров этих сценариев: Пример 34-3 и Пример 15-8.

- Отрицательные индексы позволяют вести подсчет в обратном направлении от конца массива.

Пример 37-11. Отрицательные индексы массива

```
#!/bin/bash
# neg-array.sh
# Requires Bash, version -ge 4.2.

array=( ноль один два три четыре пять )   # Шести элементный массив.
#         0    1   2    3    4    5
#        -6   -5  -4   -3   -2   -1

# Отрицательные индексы массива теперь разрешены.
echo ${array[-1]}   # пять
echo ${array[-2]}   # четыре
# ...
echo ${array[-6]}   # ноль
# Negative array indices count backward from the last element+1.

# Но, не индексируется последнее начало массива.
echo ${array[-7]}   # array: неправильный индекс массива


# Итак чем же хороша эта новая функция?

echo "Последним элементом массива является "${array[-1]}""
# Что является намного проще, чем:
echo "Последним элементом массива является "${array[${#array[*]}-1]}""
echo

# И ...

index=0
let "neg_element_count = 0 - ${#array[*]}"
# Количество элементов, преобразуемых в отрицательное число.

while [ $index -gt $neg_element_count ]; do
  ((index--)); echo -n "${array[index]} "
done  # Список элементов массива, наоборот.
      # Мы только сымитировали команду "tac" этому массиву.

echo

# См. Так же neg-offset.sh.
```

- Извлечение ***Substring*** с помощью отрицательного параметра длины для указания смещения от конца целевой строки.

Пример 37-12. Отрицательный параметр в конструкции извлечения *string*

```
#!/bin/bash
# Bash, version -ge 4.2
# Отрицательный индекс длины при извлечении substring.
# Важно: Это изменяет интерпретацию этой конструкции!

stringZ=abcABC123ABCabc

echo ${stringZ}                                 # abcABC123ABCabc
#                   Позиции в строке:           0123456789.....
echo ${stringZ:2:3}                             #   cAB
#  Отситываем 2 символа вперед от начала строки, и извлекаем 3 символа.
#  ${string:позиция:длина}

#  Ничего нового, но вот ...

                                                # abcABC123ABCabc
#                   Позиции в строке:           0123....6543210
echo ${stringZ:3:-6}                            #    ABC123
#                  ^
#  3 индекса символов вперед от начала и 6 символов назад от конца,
#+ и извлекаем все между ними.
#  ${string:смещение от начала:смещение от конца}
#  Когда параметр "длина" является отрицательным,
#+ он ***становится параметром смещения от конца***.

#  См. так же neg-array.sh.
```

Примечания

[1] Более конкретно, Bash 4+ имеет *ограниченную* поддержку ассоциативных массивов. Заметим, однако, что ассоциативные массивы в Bash, выполняются быстрее и более эффективно, чем численно-индексированные массивы.

[2] Copyright 1995-2009 by Chester Ramey.

[3] Работает только с конвейерами и некоторыми другими *специальными* файлами.

[4] Но только в сочетании с **`readline`** т.е. из командной строки.

Глава 38. Заключение

Содержание

38.1. Примечания автора

doce ut discas

(Научи, чему ты сам смог научиться.)

Как я решил написать книгу о сценариях? Это странная история. Кажется, несколько лет назад мне нужно было узнать как пишутся сценарии оболочки --а что может быть лучше, чем прочитать хорошую книгу по этому вопросу? Я стал искать, чтобы купить учебник и

справочник, охватывающей все аспекты этого вопроса. Я искал книгу, которая будет объяснять сложные концепции, показывать их изнанку и мучительно подробно, с хорошо прокомментированными примерами, объяснять их. На самом деле, я искал что-то подобное этой самой книге, или даже более, чем она. К сожалению ее не существовало, и если бы я смог, я бы написал ее.

Это напоминает мне рассказ о безумном профессоре. Парень был сумасшедшим, психом. Видя книги, любую книгу – в библиотеке, в книжном магазине, где угодно--он становился одержим идеей, что он мог бы написать ее, он должен написать ее – и написать ее лучше. Он спешил домой и воплощал это, писал книгу с точно таким же названием. Когда, несколько лет спустя, он умер, у него нашлись несколько тысяч книг, вероятно он даже переложил Азимова. Книги не дают никакой пользы тому кто знает, но другим они действительно важны? Этот парень жил своими мечтами, даже был одержим ими, движим ими..., и я не переставая любовался старым глупцом.

38.2. Об авторе

Кто же этот парень?

Автор утверждает, что не имеет ученой степени или специальной квалификации, а имел только желание писать. Эта книга является своего рода форком (ответвлением от его крупной работы, HOW-2 Meet Women: The Shy Man's Guide to Relationships. Он также написал Software-Building HOWTO. В последнее время, он пробует себя в фантастике (тяжелой): Dave Dawson Over Berlin (первое произведение), Dave Dawson Over Berlin (второе произведение) и Dave Dawson Over Berlin (третье произведение). Он также написал несколько Инструкций.

Пользователь Linux с 1995 года (Slackware 2.2, ядро 1.2.1), автор некоторого программного обеспечения, включая утилиту одноразового шифрования ***cruft***, **mcalc** - ипотечный калькулятор, **экспертный** Scrabble® adjudicator, пакет словаря **yawl** и пакет аннаграммных игр **Quacky**. Он начал с компьютерной игры -- программирования FORTRAN IV на CDC 3800 -- и это не последняя часть ностальгии по тем дням.

Живя в не согласии с женой и рыжей полосатой кошкой, он дорожит человеческой слабостью, особенно его собственной.

38.3. Куда обращаться за помощью

Автор не поддерживает и больше не обновляет этот документ. Он не будет отвечать на вопросы об этой книге или общих темах сценариев.

Если вам нужна помощь с освоением, прочитайте соответствующие разделы этого и других

справочников. Сделайте все возможное, чтобы решить проблему используя свои собственные ресурсы и ум. Пожалуйста, не тратьте время автора. Вы не получите ни помощи, ни сочувствия. [1]

Точно так же, любезно воздержитесь от раздражающих обращений к автору, предложениями о трудоустройстве или «бизнес-возможностям. Поймите это правильно - он не требует ни помощи, ни сочувствия, спасибо.

Пожалуйста, обратите внимание, что автор не будет отвечать о сценариях для систем Solaris/Sun/Oracle или Apple. Эти фирмы используют судебные тяжбы, как оружие против сообщества Open Source Community. Пользователи Solaris или Apple, нуждающиеся в помощи сценарии, в виду прямой их озабоченности для обслуживания корпоративных клиентов.

Примечания

[1] Ну если вы очень настаиваете, вы можете попробовать изменить Пример A-44 для удовлетворения ваших потребностей.

38.4. Инструменты, использованные при написании этой книги

38.4.1. Оборудование

Использовались IBM Thinkpad, модель 760XL (P166, 104 meg RAM) с запущенной Red Hat 7.1/7.3.

Обновление: обновленный 770Z Thinkpad (P2-366, 192 meg RAM) с запущенной FC3.

Обновление: обновлен до T61 Thinkpad с запущенной Mandriva 2011.

38.4.2. Программное обеспечение и печать

i. Мощный SGML текстовый редактор **vim** написанный Bram Moolenaar.

ii. **OpenJade**, DSSSL рендеринг-движок для преобразования SGML документы в другие форматы.

iii. **DSSSL stylesheets** написанный Norman Walsh.

iv. *DocBook, The Definitive Guide*, написаная Norman Walsh и Leonard Muellner (O'Reilly, ISBN 1-56592-580-7). Это по-прежнему стандартная ссылка для тех, кто пытается писать документы в формате Docbook SGML.

38.5. Благодарности

Участие Сообщества сделало возможным этот проект. Автор с благодарностью признает, что написание этой книги были бы немыслимо без помощи и обратной связи с этими людьми.

Philippe Martin перевел первую версию (0.1) настоящего документа на DocBook/SGML. Хотя и работает в небольшой французской компании, как разработчик программного обеспечения, он пользуется рабочей документацией GNU/Linux и программным обеспечением, читая литературу и слушая музыку для его успокоения. Вы можете запустить через него что-нибудь для Франции или страны Басков, или отправить ему сообщение на feloy@free.fr.

Philippe Martin так же отметил, что позиционные параметры после `$9` возможно использовать в нотации `{скобок}`. (См. Пример 4-5).

Stéphane Chazelas прислал длинный список исправлений, дополнений и примеров сценариев. Более, чем участник, он, по сути, на некоторое время взял на себя роль редактора этого документа. *Мерси боку!*

Paulo Marcel Coelho Aragao предложил множество исправлений, крупных и мелких, и внес ряд полезных предложений.

Я хотел бы особенно поблагодарить **Patrick Callahan**, **Mike Novak** и **Pal Domokos** за проверку ошибок, указание неясностей и разъяснение их, и изменения в предварительной версии (0.1) настоящего документа. Их оживленные обсуждения вопросов общей документации и сценариев оболочки вдохновил меня, чтобы попытаться сделать этот документ более удобным для чтения.

Я благодарен **Jim Van Zandt** за указания на ошибки и упущения в версии 0.2 настоящего документа. Он также предоставил поучительный пример сценария.

Большое спасибо **Jordi Sanfeliu** за разрешение на использование его прекрасного сценария дерева (Пример A-16) и **Rick Boivie** за его пересмотр.

Так же, благодарю **Michel Charpentier** за разрешение использовать его сценарий факторинга dc (Пример, 16-52).

Слава **Noah Friedman** за разрешение использовать его сценарий строки функции (Пример A-18).

Emmanuel Rouat за предложения исправлений и дополнений подстановки команд, псевдонимов и путей управления. Он также предоставил очень прекрасный образец файла .bashrc (Приложение M).

Heiner Steven за любезное разрешение на использование его сценария базовых преобразований, Пример 16-48. Он также сделал ряд исправлений и много полезных предложений. Особая благодарность.

Rick Boivie внес восхитительный сценарий рекурсии pb.sh (Пример, 36-11), пересмотренный сценарий tree.sh (Пример A-16) и предложил улучшение производительности для сценария monthlypmt.sh (Пример, 16-47).

Florian Wisser просветил меня в некоторых тонкостях проверки строк (см. Пример 7-6) и по другим вопросам.

Oleg Philon направил предложения относительно `cut` и `pidof`.

Michael Zick расширил пример пустого массива, продемонстрировав некоторые удивительные свойства массива. Он также предоставил сценарии isspammer (Пример 16-41 и Пример A-28).

Marc-Jano Knopp прислал исправления и уточнения пакетных файлов DOS.

Hyun Jin Cha нашел несколько опечаток в документе, когда занимался переводом на корейский язык. Спасибо за эти указания.

Andreas Abraham прислал в длинный список опечаток и других исправлений. Особая благодарность!

Остальные, предоставившие сценарии, делавшие полезные предложения и указывавшие на ошибки, были Gabor Kiss, Leopold Toetsch, Peter Tillier, Marcus Berglof, Tony Richardson, Nick Drage (идея сценария!), Rich Bartell, Jess Thrysoee, Adam Lazur, Bram Moolenaar, Baris Cicek, Greg Keraunen, Keith Matthews, Sandro Magi, Albert Reiner, Dim Segebart, Rory Winston, Lee Bigelow, Wayne Pollock, "jipe," "bojster," "nyal," "Hobbit," "Ender," "Little Monster" (Alexis), "Mark," "Patsie," "vladz," Peggy Russell, Emilio Conti, Ian. D. Allen, Hans-Joerg Diers, Arun Giridhar, Dennis Leeuw, Dan Jacobson, Aurelio Marinho Jargas, Edward Scholtz, Jean Helou, Chris Martin, Lee Maschmeyer, Bruno Haible, Wilbert Berendsen, Sebastien Godard, Björn Eriksson, John MacDonald, John Lange, Joshua Tschida, Troy Engel, Manfred Schwarb, Amit Singh, Bill Gradwohl, E. Choroba, David Lombard, Jason Parker, Steve Parker, Bruce W. Clare, William Park, Vernia Damiano, Mihai Maties, Mark Alexander, Jeremy Impson, Ken Fuchs, Jared Martin, Frank Wang, Sylvain Fourmanoit, Matthew Sage, Matthew Walker, Kenny Stauffer, Filip Moritz, Andrzej Stefanski, Daniel Albers, Jeffrey Haemer, Stefano Palmeri, Nils Radtke, Sigurd Solaas, Serghey Rodin, Jeroen Domburg, Alfredo Pironti, Phil Braham, Bruno de Oliveira Schneider, Stefano Falsetto, Chris Morgan, Walter Dnes, Linc Fessenden, Michael Iatrou, Pharis Monalo, Jesse Gough, Fabian Kreutz, Mark Norman, Harald Koenig, Dan Stromberg, Peter Knowles, Francisco Lobo, Mariusz Gniazdowski, Sebastian Arming, Chetankumar Phulpagare, Benno Schulenberg, Tedman Eng, Jochen DeSmet, Juan Nicolas Ruiz, Oliver Beckstein, Achmed Darwish, Dotan Barak, Richard Neill, Albert Siersema, Omair Eshkenazi, Geoff Lee, Graham Ewart, JuanJo Ciarlante, Cliff Bamford, Nathan Coulter, Ramses Rodriguez Martinez, Evgeniy Ivanov, Craig Barnes, George Dimitriu, Kevin LeBlanc, Antonio Macchi, Tomas Pospisek, David Wheeler, Erik Brandsberg, YongYe, Andreas Kühne, Pádraig Brady, Joseph Steinhauser и David Lawyer (автор четырех HOWTOs).

Моя благодарность **Chet Ramey** и **Brian Fox** за написание Bash и встраивание в него элегантные и мощные сценарии дающие возможность соперничать с *ksh*.

Очень особая благодарность трудолюбивым добровольцам **Linux Documentation Project**. Хост LDP размещает репозиторий знаний Linux и эти знания в значительной степени и позволили опубликовать эту книгу.

Благодарность и признательность **IBM**, **Red Hat**, **Google**, **Free Software Foundation** и всем хорошим людям борющимся за то, что бы исходный код программного обеспечения был свободным и открытым.

Запоздалое спасибо мой четвертой учительнице, **Мисс Спенсер**, за эмоциональную поддержку и убеждение во мне того, что, может быть, я еще не совсем потерян.

Благодарю, прежде всего, мою жену, **Anita**, за ее поощрение, вдохновение и эмоциональную поддержку.

38.6. Правовая оговорка

(Это вариант стандартной правовой оговорки LDP .)

За содержание данного документа не несется никакой ответственности. Концепции, примеры и информация используются на свой страх и риск. Документ может содержать ошибки, упущения и неточности, которые могут привести к потери ваших данных или нанести вред вашей системе, поэтому используйте его с соответствующей осторожностью. Автор не несет никакой ответственности за любой ущерб, причиненный случайно или иным способом.

Может произойти случайно, но крайне маловероятно, что вы или ваша система пострадает из-зи последствий, в виду неконтролируемых зависаний. На самом деле, целью этой книги является возможность дать читателям проанализировать сценарии оболочки и определить, будут лт из-за них непредвиденные последствия.

Библиография

Те, кто не понимает UNIX, обречены на его изобретение.

--Henry Spencer

Edited by Peter Denning, *Computers Under Attack: Intruders, Worms, and Viruses*, ACM Press, 1990, 0-201-53067-8.

Этот сборник содержит несколько статей о вирусах в сценариях shell.

*

Ken Burtch, *Linux Shell Scripting with Bash*, 1st edition, Sams Publishing (Pearson), 2004, 0672326426.

Охватывает большую часть материала *ABS Guide*, хотя и в другом стиле.

*

Daniel Goldman, *Definitive Guide to Sed*, 1st edition, 2013.

Эта электронная книга представляет собой отличное введение в ***sed***. Вместо печатного объема, она был специально разработана и отформатирована для просмотра и чтения электронных книг. Хорошо написана, информативна и полезна как в качестве ссылки, так и в качестве учебного пособия. Очень рекомендую.

*

Dale Dougherty and Arnold Robbins, *Sed and Awk*, 2nd edition, O'Reilly and Associates, 1997, 1-156592-225-5.

Развертывание сценариев оболочки на полную мощность требует, по крайней мере, знакомства с ***sed*** и ***awk***. Это классическое руководство. Оно включает в себя прекрасное введение в *Регулярные выражения*. Рекомендую.

*

Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly and Associates, 2002, 0-596-00289-0.

Одно из лучших изданий о *Регулярных выражениях*.

*

Aeleen Frisch, *Essential System Administration*, 3rd edition, O'Reilly and Associates, 2002, 0-596-00343-9.

В этом превосходном руководстве содержится прекрасное введение в язык сценариев командной оболочки с точки зрения системного администратора. Она включает в себя исчерпывающие объяснения запуска и инициализации сценариев в системе UNIX.

*

Stephen Kochan and Patrick Wood, *Unix Shell Programming*, Hayden, 1990, 067248448X.

Все еще стандартное издание, хотя несколько устаревшее, и, стилистически, немного 'деревянное'. [1] На самом деле, эта книга была первым руководством автора *ABS Guide* по сценариям оболочки UNIX, но много лет назад.

*

Neil Matthew and Richard Stones, *Beginning Linux Programming*, Wrox Press, 1996, 1874416680.

Удивительно хорошее, углубленное, освещение различных языков программирования для Linux, где так же довольно сильны главы о сценариях оболочки.

*

Herbert Mayer, *Advanced C Programming on the IBM PC*, Windcrest Books, 1989, 0830693637.

Отличное освещение алгоритмов и общей практики программирования. Настоятельно рекомендуется, но к сожалению больше не издается.

*

David Medinets, *Unix Shell Programming Tools*, McGraw-Hill, 1999, 0070397333.

Довольно хорошее издание о сценариях оболочки, с примерами, с кратким введением в ***Tcl*** и ***Perl***.

*

Cameron Newham and Bill Rosenblatt, *Learning the Bash Shell*, 2nd edition, O'Reilly and Associates, 1998, 1-56592-347-2.

Это - отважное усилие в приличном учебнике для начинающих, но печально несовершенное в охвате написания сценариев и не достаточное в примерах.

*

Anatole Olczak, *Bourne Shell Quick Reference Guide*, ASP, Inc., 1991, 093573922X.

Очень удобное карманное издание, несмотря на отсутствие освещения специфических особенностей Bash.

*

Jerry Peek, Tim O'Reilly, and Mike Loukides, *Unix Power Tools*, 3rd edition, O'Reilly and Associates, Random House, 2002, 0-596-00330-7.

Содержит пару частей очень информативных, углубленных статей о shell-программировании, но не может претендовать на самоучитель. Она воспроизводит большую часть знаний о *Регулярных выражениях* из книги Dougherty и Robbins, выше. Всеобъемлющий охват команд UNIX делает эту книгу достойной места на вашей полке.

*

Clifford Pickover, *Computers, Pattern, Chaos, and Beauty*, St. Martin's Press, 1990, 0-312-04123-3.

Сокровищница идей и рецептов для компьютерного исследования математических странностей.

*

George Polya, *How To Solve It*, Princeton University Press, 1973, 0-691-02356-5.

Классический учебник по методам решения проблем (алгоритмов), с уделением особого внимания их изучению.

*

Chet Ramey and Brian Fox, *The GNU Bash Reference Manua*l, Network Theory Ltd, 2003, 0-9541617-7-7.

> Это руководство является основным изданием по GNU Bash. Авторы данного руководства, Chet Ramey и Brian Fox, являются оригинальными разработчиками GNU Bash. С каждой проданой копии издатель жертвует $1 в Фонд свободного программного обеспечения (Free Software Foundation).
>
> *

Arnold Robbins, *Bash Reference Card*, SSC, 1998, 1-58731-010-5.

> Отличный карманный справочник по Bash(не выходите без него из дома, особенно если вы системный администратор). Стоит $4.95, и, к сожалению, больше не доступен для бесплатного скачивания.
>
> *

Arnold Robbins, *Effective Awk Programming*, Free Software Foundation / O'Reilly and Associates, 2000, 1-882114-26-4.

> Справочник и абсолютный лучший учебник по ***awk***. Бесплатная электронная версия этой книги является частью документации ***awk***, а печатные копии последней версии доступны в *O'Reilly and Associates*.
>
> Эта книга послужила источником вдохновения для автора *ABS Guide*.
>
> *

Bill Rosenblatt, *Learning the Korn Shell*, O'Reilly and Associates, 1993, 1-56592-054-6.

> Хорошо написаная книга содержащая несколько прекрасных указаний по сценариям оболочки в целом.
>
> *

Paul Sheer, *LINUX: Rute User's Tutorial and Exposition*, 1st edition, , 2002, 0-13-033351-4.

> Очень подробное и легко читаемое введение в системной администрирование Linux.
>
> Книга доступна в бумажном варианте или в варианте он-лайн.
>
> *

Ellen Siever and the staff of O'Reilly and Associates, *Linux in a Nutshell*, 2nd edition, O'Reilly and Associates, 1999, 1-56592-585-8.

> Лучший и полнейший справочник по командам Linux. Он даже имеет раздел по Bash.
>
> *

Dave Taylor, *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*, 1st edition, No Starch Press, 2004, 1-59327-012-7.

Содержит очень много из того, что обещает название...

*

The UNIX CD Bookshelf, 3rd edition, O'Reilly and Associates, 2003, 0-596-00392-7.

Собрание из семи книг об UNIX на CD ROM, включающее *UNIX Power Tools*, *Sed and Awk*, и *Learning the Korn Shell*. Полный набор всех справочников и учебников по UNIX, которые вам могут когда-либо понадобиться, стоимость около $130. Купите, даже если нужно залезть в долг и не платить арендную плату.

Новость: Кажется пропало из печати. Хорошо. Но Вы все еще можете купить издания этих книг по отдельности.

*

Книги O'Reilly по ***Perl***. (А так же, *другие* издания O'Reilly.)

* * *

Другие источники

(Перевод предоставляю самим интересующимся)

Fioretti, Marco, "Scripting for X Productivity," *Linux Journal*, Issue 113, September, 2003, pp. 86-9.

Ben Okopnik's well-written *introductory Bash scripting* articles in issues 53, 54, 55, 57, and 59 of the *Linux Gazette*, and his explanation of "The Deep, Dark Secrets of Bash" in issue 56.

Chet Ramey's *Bash - The GNU Shell*, a two-part series published in issues 3 and 4 of the *Linux Journal*, July-August 1994.

Mike G's Bash-Programming-Intro HOWTO.

Richard's Unix Scripting Universe.

Chet Ramey's Bash FAQ.

Greg's WIKI: Bash FAQ.

Example shell scripts at Lucc's Shell Scripts .

Example shell scripts at SHELLdorado .

Example shell scripts at Noah Friedman's script site.

Examples from the *The Bash Scripting Cookbook*, by Albing, Vossen, and Newham.

Example shell scripts at zazzybob.

Steve Parker's Shell Programming Stuff. In fact, all of his shell scripting books are highly recommended. See also Steve's Arcade Games written in a shell script.

An excellent collection of Bash scripting tips, tricks, and resources at the Bash Hackers Wiki.

Giles Orr's Bash-Prompt HOWTO.

The *Pixelbeat* command-line reference.

Very nice **sed**, **awk**, and regular expression tutorials at The UNIX Grymoire.

The GNU sed and gawk manuals. As you recall, gawk is the enhanced GNU version of **awk**.

Many interesting sed scripts at the seder's grab bag.

Tips and tricks at Linux Reviews.

Trent Fisher's groff tutorial.

David Wheeler's Filenames in Shell essay.

"Shelltris" and "shellitaire" at Shell Script Games.

YongYe's wonderfully complex Tetris game script.

Mark Komarinski's Printing-Usage HOWTO.

The Linux USB subsystem (helpful in writing scripts affecting USB peripherals).

There is some nice material on I/O redirection in chapter 10 of the textutils documentation at the University of Alberta site.

Rick Hohensee has written the *osimpa* i386 assembler entirely as Bash scripts.

dgatwood has a very nice shell script games site, featuring a Tetris® clone and solitaire.

Aurelio Marinho Jargas has written a Regular expression wizard. He has also written an informative book on Regular Expressions, in Portuguese.

Ben Tomkins has created the Bash Navigator directory management tool.

William Park has been working on a project to incorporate certain *Awk* and *Python* features into Bash. Among these is a *gdbm* interface. He has released *bashdiff* on Freshmeat.net. He has an article in the November, 2004 issue of the *Linux Gazette* on adding string functions to Bash, with a followup article in the December issue, and yet another in the January, 2005 issue.

Peter Knowles has written an elaborate Bash script that generates a book list on the Sony Librie e-book reader. This useful tool facilitates loading non-DRM user content on the *Librie* (and the newer *PRS-xxx-series* devices).

Tim Waugh's xmlto is an elaborate Bash script for converting Docbook XML documents to other formats.

Philip Patterson's logforbash logging/debugging script.

AuctionGallery, an application for eBay "power sellers" coded in Bash.

Of historical interest are Colin Needham's *original International Movie Database (IMDB) reader polling scripts*, which nicely illustrate the use of awk for string parsing. Unfortunately, the URL link is broken.

Fritz Mehner has written a bash-support plugin for the *vim* text editor. He has also also come up with his own stylesheet for Bash. Compare it with the ABS Guide Unofficial Stylesheet.

Penguin Pete has quite a number of shell scripting tips and hints on his superb site. Highly recommended.

The excellent *Bash Reference Manual*, by Chet Ramey and Brian Fox, distributed as part of the *bash-2-doc* package (available as an rpm). See especially the instructive example scripts in this package.

John Lion's classic, *A Commentary on the Sixth Edition UNIX Operating System*.

The comp.os.unix.shell newsgroup.

The *dd* thread on Linux Questions.

The comp.os.unix.shell FAQ.

Assorted comp.os.unix FAQs.

The *Wikipedia* article covering dc.

The manpages for **bash** and **bash2**, **`date`**, **`expect`**, **`expr`**, **`find`**, **`grep`**, **`gzip`**, **`ln`**, **`patch`**, **`tar`**, **`tr`**, **`bc`**, **`xargs`**. The *texinfo* documentation on **bash**, **`dd`**, **`m4`**, **gawk**, and **sed**.

Примечания

[1] Это, очевидно, трудно переводимый каламбур. Эта книга является довольно приличным введением в основные понятия сценариев оболочки.

Приложение A. Внесенные сценарии

Эти сценарии, хотя и не описываются в тексте этого документа, но иллюстрируют некоторые интересные приемы программирования в shell. Некоторые из них полезны. Проанализируйте и запустите их.

Пример A-1. *mailformat*: Форматирование сообщений e-mail

```
#!/bin/bash
```

```
# mail-format.sh (ver. 1.1): Форматирование сообщений e-mail.

#  Удаляет переводы каретки, табуляции, а также чрезмерно длинные
#+ строки.

# ================================================================
#                 Обычная проверка аргумента(ов) сценария
ARGS=1
E_BADARGS=85
E_NOFILE=86

if [ $# -ne $ARGS ]  # Сценарию передано правильное количество аргументов?
then
  echo "Используйте: `basename $0` filename"
  exit $E_BADARGS
fi

if [ -f "$1" ]       # Проверка наличия файла.
then
    file_name=$1
else
    echo "Файл \"$1\" отсутствует."
    exit $E_NOFILE
fi
# ---------------------------------------------------------------

MAXWIDTH=70          # Общий размер для чрезмерно длинных строк.

# ================================
# Переменная может содержать сценарий sed.
# Это полезный прием.
sedscript='s/^>//
s/^  *>//
s/^  *//
s/              *//'
# ================================

#  Удаление переводов каретки и табуляций в начале строк,
#+ затем создание строки из $MAXWIDTH символов.
sed "$sedscript" $1 | fold -s --width=$MAXWIDTH
                        #  опция -s в 'разбивает', по возможности, разделенные
                        #+ пробелами строки.

#  Этот сценарий был вдохновлен статьей в хорошо известном коммерческом журнале
#+ превозносящем утилиту 164K  ОС Windows с аналогичной функциональностью.
#
#  Хороший набор утилит обработки текста и эффективность
#+ языка сценариев представляет альтернативу раздутым и
#+ неуклюжим исполняемым рабочим программам.

exit $?
```

Пример A-2. *rn*: Бесхитростная утилита для переименования файла

Этот сценарий является модификацией Примера 16-22.

```
#! /bin/bash
# rn.sh

# Очень простая утилита "переименования" файла (основана на "lowercase.sh").
```

```
#
#  Утилита "ren", Vladimir Lanin (lanin@csd2.nyu.edu),
#+ работает гораздо лучше.


ARGS=2
E_BADARGS=85
ONE=1                     #  Получаен единственное/множественное
                          #+ число (см. ниже).

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` старый_шаблон новый_шаблон"
  #  Как и в "rn gif jpg", которая переименовывает все файлы gif
  #+ в рабочей директории в jpg.
  exit $E_BADARGS
fi

number=0                  # Отслеживает, сколько переименовано файлов.


for filename in *$1*      # Проходит все найденные в директории файлы.
do
   if [ -f "$filename" ]  # Если находит соответствие...
   then
     fname=`basename $filename`            # Извлекает путь.
     n=`echo $fname | sed -e "s/$1/$2/"`   # Заменяет новым старое имя файла.
     mv $fname $n                           # Переименовывает.
     let "number += 1"
   fi
done

if [ "$number" -eq "$ONE" ]               # Исправляет грамматические ошибки.
then
 echo "Переименован файл $number."
else
 echo "Переименован файл $number."
fi

exit $?


# Упражнения:
# ---------
# С какими типами файлов этот сценарий не будет работать?
# Как это можно исправить?
```

Пример A-3. *blank-rename*: Исправление имен файлов содержащих пробелы

Даже проще версии предыдущего сценария.

```
#! /bin/bash
# blank-rename.sh
#
# Заменяет подчеркиваниями пробелы во всех файлах в директории.

ONE=1                     #  Получение единственного/множественного
                          #+ числа (см. ниже).
```

```
number=0                  #  Отслеживает, сколько  переименовано файлов.
FOUND=0                   #  Возвращаемое значение при успехе.

for filename in *         #  Проходит все файлы в директории.
do
     echo "$filename" | grep -q " "          #  Проверяет какое имя
     if [ $? -eq $FOUND ]                    #+ содержит пробел(ы).
     then
       fname=$filename                       # Да, это имя должно работать.
       n=`echo $fname | sed -e "s/ /_/g"`    # Замена пробела подчеркиванием.
       mv "$fname" "$n"                      # Переименовывание.
       let "number += 1"
     fi
done

if [ "$number" -eq "$ONE" ]                  # Исправление ошибок.
then
 echo "Переименован файл $number."
else
 echo "Переименован файл $number."
fi

exit 0
```

Пример A-4. *encryptedpw*: Загрузка FTP-сайта с помощью локально зашифрованного пароля

```
#!/bin/bash

#  Измененный пример "ex72.sh" для использования с зашифрованным паролем.

#  Обратите внимание, что отправлять расшифрованный пароль
#+ довольно не безопасно.
#  Используйте для этого что-то типа "ssh".

E_BADARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` имя_файла"
  exit $E_BADARGS
fi

Username=bozo           # Измените на нужное.
pword=/home/bozo/secret/password_encrypted.file
# Файл содержащий зашифрованный пароль.

Filename=`basename $1`  # Извлекает путь имени файла.

Server="XXX"
Directory="YYY"         # Измените имя сервера, выше, и директории.


Password=`cruft <$pword`          # Расшифровывает пароль.
#  Используется авторский пакет шифрования "cruft",
#+ основанный на классическом алгоритме "onetime pad",
#+ Который получает с основного сайта:
```

```
#+    ftp://ibiblio.org/pub/Linux/utils/file
#+      cruft-0.2.tar.gz [16k]


ftp -n $Server <<End-Of-Session
user $Username $Password
binary
bell
cd $Directory
put $Filename
bye
End-Of-Session
# Опция -n в "ftp" отключает автоматический вход.
# Обратите внимание, что звонит "звонок" после передачи каждого файла.

exit 0
```

Пример A-5. *copy-cd*: Копирование CD с данными

```
#!/bin/bash
# copy-cd.sh: Копирование CD с данными

CDROM=/dev/cdrom                            # Устройство CD ROM
OF=/home/bozo/projects/cdimage.iso          # Выходной файл
#       /xxxx/xxxxxxxx/                       Измените, на нужный.
BLOCKSIZE=2048
# SPEED=10                                  # Если не указано, скорость max.
# DEVICE=/dev/cdrom                         # Старая версия.
DEVICE="1,0,0"

echo; echo "Вставьте исходный CD, но *не* монтируйте его."
echo "Нажмите ENTER, когда будете готовы. "
read ready                                  #  Ожидание ввода, $ready не
                                            #+ используется.

echo; echo "Копирование исходного CD в $OF."
echo "Это может занять некоторое время. Пожалуйста, подождите."

dd if=$CDROM of=$OF bs=$BLOCKSIZE           # «Низкоуровневое» копирование.


echo; echo "Извлеките CD."
echo "Вставьте чистый CDR."
echo "Нажмите ENTER, когда будете готовы. "
read ready                                  #  Ожидание ввода, $ready не
                                            #+ используется.

echo "Копируется $OF на CDR."

# cdrecord -v -isosize speed=$SPEED dev=$DEVICE $OF   # Старая версия.
wodim -v -isosize dev=$DEVICE $OF
# Используется пакет Joerg Schilling "cdrecord".
# http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html
# Новые дистрибутивы Linux могут использовать "wodim" вместо "cdrecord" ...


echo; echo "Скопировано $OF на CDR на оборудовании $CDROM."
```

```
echo "Удалить образ файла (y/n)? "  # Возможно большой файл.
read answer

case "$answer" in
[yY]) rm -f $OF
      echo "$OF удален."
      ;;
*)    echo "$OF не удален.";;
esac

echo

# Упражнение:
# Измените объявление "case" на согласие "yes" и "Yes" с ввода.

exit 0
```

Пример A-6. Последовательности Collatz

```
#!/bin/bash
# collatz.sh

#  Печально известная 'градина' или последовательности Collatz.
#  ------------------------------------------
#  1) Получение целого "получаемого" числа из командной строки.
#  2) ЧИСЛО <-- начальное значение
#  3) Вывод ЧИСЛА.
#  4)  Если ЧИСЛО четное, разделить на 2, или
#  5)+ если нечетное, умножить на 3 и прибавить 1.
#  6) ЧИСЛО <-- результат
#  7) Возвращаемся к пункту 3 (для заданного числа повторений).
#
#  Теоретически, каждая такая последовательность,
#+ независимо от того, какое имеет большое начальное значение,
#+ в конечном итоге уменьшается, повторяя циклы '4,2,1 ...',
#+ даже после колебаний в широком диапазоне значений.
#
#  Это пример «повторения»,
#+ операции, которая передает свой вывод обратно себе на вход.
#  Иногда результатом являются «хаотические» ряды.


MAX_ITERATIONS=200
# Для больших исходных чисел (>32000), попробуйте увеличить MAX_ITERATIONS.

h=${1:-$$}                      #  Исходное значение.
                                #  Используем $PID, как исходное значение,
                                #+ если не задано аргументом командной строки.

echo
echo "C($h) -*- $MAX_ITERATIONS повторений"
echo

for ((i=1; i<=MAX_ITERATIONS; i++))
do

# echo -n "$h   "
#           ^^^
```

```
#          табуляция
# printf сделает это получше ...
COLWIDTH=%7d
printf $COLWIDTH $h

  let "remainder = h % 2"
  if [ "$remainder" -eq 0 ]   # Четное?
  then
    let "h /= 2"              # Делим на 2.
  else
    let "h = h*3 + 1"         # Умножаем на 3 и прибавляем 1.
  fi


COLUMNS=10                    # Выводим по 10 значений в строке.
let "line_break = i % $COLUMNS"
if [ "$line_break" -eq 0 ]
then
  echo
fi

done

echo

#  Что бы больше узнать об этой математической функции,
#+ см. _Computers, Pattern, Chaos, and Beauty_, by Pickover, p. 185 ff.,
#+ в Библиографии.

exit 0
```

Пример A-7. *days-between*: Количество дней между двумя датами

```
#!/bin/bash
# days-between.sh:    Число дней между двумя датами.
# Usage: ./days-between.sh [M]M/[D]D/YYYY [M]M/[D]D/YYYY
#
# Примечание: Сценарий изменен с учетом изменений в Bash, v. 2.05b+,
#+            который закрыл лазейки, разрешающие большие отрицательные
#+            целые числа для возвращаемых значений.

ARGS=2                # В командную строку вводятся два параметра.
E_PARAM_ERR=85        # Ошибка параметра.

REFYR=1600            # Исходный год.
CENTURY=100
DIY=365
ADJ_DIY=367           # Поправка на високосный год + величина.
MIY=12
DIM=31
LEAPCYCLE=4

MAXRETVAL=255         #  Максимальное допустимое положительное
                      #+ значение возвращаемое функцией.

diff=                 #  Объявление глобальной переменной для разницы дат.
value=                #  Объявление глобальной переменной для абсолютного
                      #+ значения.
```

```
day=                  #  Объявление глобальных дня, месяца, года.
month=
year=


Param_Error ()        # Не правильные параметры командной строки.
{
  echo "Usage: `basename $0` [M]M/[D]D/YYYY [M]M/[D]D/YYYY"
  echo "       (дата должна быть после 1/3/1600)"
  exit $E_PARAM_ERR
}


Parse_Date ()                 # Проверка параметров командной строки даты.
{
  month=${1%%/**}
  dm=${1%/**}                 # День и месяц.
  day=${dm#*/}
  let "year = `basename $1`"  # Не имя файла, но работает.
}


check_date ()                 # Проверка правильности передаваемой даты.
{
  [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] ||
  [ "$year" -lt "$REFYR" ] && Param_Error
  # Выход сценария при неправильном значении.
  # Используем or-list/and-list.
  #
  # Упражнение: Осуществите более строгую проверку даты.
}


strip_leading_zero () #  Вывод возможных начальных нулей
{                     #+ из дня и/или месяца
  return ${1#0}       #+ т.к. иначе Bash будет понимать их
}                     #+ как восьмеричные значения (POSIX.2, sect 2.9.2.1).


day_index ()          # Формула Gauss:
{                     # Дни от 1 марта, 1600г. до даты заданной параметром.
                      #        ^^^^^^^^^^^^^
  day=$1
  month=$2
  year=$3

  let "month = $month - 2"
  if [ "$month" -le 0 ]
  then
    let "month += 12"
    let "year -= 1"
  fi

  let "year -= $REFYR"
  let "indexyr = $year / $CENTURY"


  let "Days = $DIY*$year + $year/$LEAPCYCLE - $indexyr \
             + $indexyr/$LEAPCYCLE + $ADJ_DIY*$month/$MIY + $day - $DIM"
  #  Для более глубокого понимания этого алгоритма, смотри
  #+   http://weblogs.asp.net/pgreborio/archive/2005/01/06/347968.aspx
```

```
  echo $Days

}


calculate_difference ()           # Разница между двумя выведенными днями.
{
  let "diff = $1 - $2"            # Глобальная переменная.
}


abs ()                            #  Абсолютное значение
{                                 #  Используется глобальная
                                  #+ переменная"value"
  if [ "$1" -lt 0 ]               #  Если отрицательное,
  then                            #+ то
    let "value = 0 - $1"          #+ меняем знак,
  else                            #+ иначе,
    let "value = $1"              #+ продолжаем.
  fi
}



if [ $# -ne "$ARGS" ]             # Требуются два параметра командной строки.
then
  Param_Error
fi

Parse_Date $1
check_date $day $month $year      #  Проверяем правильность даты.

strip_leading_zero $day           #  Удаляем начальные нули
day=$?                            #+ в дне и/или месяце.
strip_leading_zero $month
month=$?

let "date1 = `day_index $day $month $year`"


Parse_Date $2
check_date $day $month $year

strip_leading_zero $day
day=$?
strip_leading_zero $month
month=$?

date2=$(day_index $day $month $year) # Подстановка команд.


calculate_difference $date1 $date2

abs $diff                           # Убеждаемся, что положительное.
diff=$value

echo $diff

exit 0
```

```
#  Упражнение:
#  --------
#  Если указать только один параметр командной строки, то пусть сценарий
#+ использует сегодняшнюю дату в качестве второго параметра.


#  Сравните этот сценарий с
#+ осуществлением формулы Гаусса в программе Си на
#+ http://buschencrew.hypermart.net/software/datedif
```

Пример A-8. Создаем *словарь*

```
#!/bin/bash
# makedict.sh  [создание словаря]

# Изменение сценария /usr/sbin/mkdict (/usr/sbin/cracklib-forman).
# Оригинальный сценарий copyright 1993, Alec Muffett.
#
#  Это модификация сценария, содержащегося в настоящем документе
#+ в соответствии с документом «LICENSE» пакета «Crack»,
#+ являющегося частью оригинального сценария.

#  Этот сценарий обрабатывает текстовые файлы для создания сортированного
#+ списка слов находящихся в файле.
#  Может быть полезен для компиляции словарей
#+ и для других лексикографических целей.


E_BADARGS=85

if [ ! -r "$1" ]                    #  Необходим хотя бы один
then                                #+ правильный файл, как аргумент.
  echo "Usage: $0 файлы-для-обработки"
  exit $E_BADARGS
fi


# SORT="sort"                       #  Нет больше необходимости определять
                                    #+ параметры для сортировки. Изменение
                                    #+ оригинального сценария.

cat $* |                            #  Содержимое указанных файлов в stdout.
        tr A-Z a-z |                #  Конвертирование в строчные буквы.
        tr ' ' '\012' |             #  Новое: Меняем пробелы на перевод строки.
#       tr -cd '\012[a-z][0-9]' |   #  Убираем все не буквенно-цифровое
                                    #+ (оригинальный сценарий)
        tr -c '\012a-z'  '\012' |   #  Вместо удаления не буквенных символов
                                    #+ меняем их на перевод строки.
        sort |                      #  Опция $SORT теперь не нужна.
        uniq |                      #  Удаляем повторения.
        grep -v '^#' |              #  Удаляем строки начинающиеся с #.
        grep -v '^$'                #  Удаляем пустые строки.

exit $?
```

Пример A-9. Преобразование Soundex

```
#!/bin/bash
# soundex.sh: Расчет кода "soundex" для имен

# ======================================================
#        Сценарий Soundex
#         Mendel Cooper
#     thegrendel.abs@gmail.com
#     reldate: 23 January, 2002
#
#     Размещен с Public Domain.
#
# Различные легкие версии этого сценария размещены
#+ Ed Schaefer July, 2002 "Shell Corner" column
#+ в "Unix Review" on-line,
#+ http://www.unixreview.com/documents/uni1026336632258/
# ======================================================


ARGCOUNT=1                      # Необходимо имя в качестве аргумента.
E_WRONGARGS=90

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` имя"
  exit $E_WRONGARGS
fi


assign_value ()                 #  Присваиваем числовые значения
{                               #+ буквам в имени.

  val1=bfpv                     # 'b,f,p,v' = 1
  val2=cgjkqsxz                 # 'c,g,j,k,q,s,x,z' = 2
  val3=dt                       #  и т.д.
  val4=l
  val5=mn
  val6=r

# Очень хорошо здесь использовать 'tr'.
# Посмотрите, что произойдет.

value=$( echo "$1" \
| tr -d wh \
| tr $val1 1 | tr $val2 2 | tr $val3 3 \
| tr $val4 4 | tr $val5 5 | tr $val6 6 \
| tr -s 123456 \
| tr -d aeiouy )

#  Присваиваем буквам числовые значения.
#  Удаляем повторяющиеся числа, за исключением, когда они разделены гласными.
#  Игнорируем гласные, за исключением разделителей, что бы удалить их
#+ последними.
#  Игнорируем 'w' и 'h', даже как разделители, и удаляем их первыми.

}


input_name="$1"
echo
```

```
echo "Имя = $input_name"


# Изменяем все вводимые символы имени на строчные.
# ------------------------------------------------
name=$( echo $input_name | tr A-Z a-z )
# ------------------------------------------------
# Могут быть смешанными только аргументы сценария.


# Префикс кода soundex: Первая буква имени.
# -----------------------------------------


char_pos=0                      # Объявляем позицию символа в имени.
prefix0=${name:$char_pos:1}
prefix=`echo $prefix0 | tr a-z A-Z`
                                # 1-я буква soundex - заглавная.

let "char_pos += 1"             # Перемещаемся на позицию символа 2-й буквы
имени.
name1=${name:$char_pos}


# ++++++++++++++++++++++++++ Исключения ++++++++++++++++++++++++++++++
#  Теперь мы запускаем оба - введенное имя и имя смещенное на один символ
#+ вправо через присвоение значения функции.
#  Если мы получим то же внешнее значение, то это означает, что два первых
#+ символа имени имеют присвоенное одно и то же значение, и что одно следует
#+ отменить.
#  Однако нам также необходимо проверить, является ли первая буква имени
#+ гласной или «w» или «h».

char1=`echo $prefix | tr A-Z a-z`    # Первая буква имени, строчная.

assign_value $name
s1=$value
assign_value $name1
s2=$value
assign_value $char1
s3=$value
s3=9$s3                                 #  Если первая буква имени гласная
                                        #+ или 'w' или 'h',
                                        #+ то ее "значение" будет null (не
                                        #+ присвоено). Поэтому присваиваем ей
                                        #+ 9, иначе значение невозможно будет
                                        #+ проверить.


if [[ "$s1" -ne "$s2" || "$s3" -eq 9 ]]
then
  suffix=$s2
else
  suffix=${s2:$char_pos}
fi
# ++++++++++++++++++++++ Конец исключений ++++++++++++++++++++++++++++


padding=000                     # Используем 3 или более нулей.
```

```
soun=$prefix$suffix$padding    # Место с нулями.

MAXLEN=4                       # Уменьшаем максимум до 4 символов.
soundex=${soun:0:$MAXLEN}

echo "Soundex = $soundex"

echo

#  Код soundex является способом индексации и классификации имен
#+ путем группирования по звучанию.
#  Код soundex присваивает имени сначала букву,
#+ а затем вычисляет три цифры.
#  Подобные созвучные имена будут иметь практически одинаковый код soundex.

#   Примеры:
#   Smith и Smythe оба будут в soundex "S-530".
#   Harrison = H-625
#   Hargison = H-622
#   Harriman = H-655

#  Это на практике работает довольно хорошо, но есть многочисленные аномалии.
#
#
#  U.S. Census и некоторые другие правительственные учреждения используют
#  soundex для генеалогических исследований.
#
#  Что бы больше узнать,
#+ см. "National Archives and Records Administration home page",
#+ http://www.nara.gov/genealogy/soundex/soundex.html



# Упражнение:
# --------
# Упростите часть "Исключения" этого сценария.

exit 0
```

Пример A-10. ***Game of Life***

```
#!/bin/bash
# life.sh: "Life in the Slow Lane"
# Автор: Mendel Cooper
# License: GPL3

# Version 0.2:   Patched by Daniel Albers
#+               to allow non-square grids as input.
# Version 0.2.1: Added 2-second delay between generations.

# ########################################################################## #
# Этот сценарий Bash является версией John Conway "Game of Life".           #
# "Life" это простая реализация клеточного автомата.                         #
# ---------------------------------------------------------------------- #
# Прямоугольная сетка, на которой каждая "ячейка" может быть                #
#                 "живой" или "мертвой".                                     #
#  Обозначаем живую ячейку точкой, а мертвую пустым пространством.           #
#      Выберите произвольно на сетке ячейку с точкой и пустую,               #
```

```
#+       и пусть это будет началом создания: generation 0.              #
#   Определяйте каждую последующую генерацию по следующим правилам      #
# 1) Каждая ячейка имеет 8 соседних, смежных ячеек                       #
#+     слева, справа, сверху, снизу и 4 по диагонали.                   #
#                                                                        #
#                      123                                               #
#                      4*5     * является рассматриваемой ячейкой.       #
#                      678                                               #
#                                                                        #
# 2) Живая ячейка с 2 или 3 живыми соседними ячейками - остается живой. #
SURVIVE=2                                                               #
# 3) Мертвая ячейка с 3 живыми соседними оживает, "рождается".          #
BIRTH=3                                                                 #
# 4) Все остальные случаи являются мертвыми ячейками для               #
#     следующих генераций.                                              #
# ###################################################################### #


startfile=gen0   # Считывать начальную генерацию из файла "gen0" ...
                 # по умолчанию, если при вызове сценария не указан иной файл.
                 #
if [ -n "$1" ]   # Укажите другой файл "generation 0".
then
    startfile="$1"
fi

##############################################
#  Сценарий прерывается, если «startfile» не указан
#+ и
#+ отсутствует файл по умолчанию "gen0".

E_NOSTARTFILE=86

if [ ! -e "$startfile" ]
then
  echo "Начальный файл \""$startfile"\" отсутствует!"
  exit $E_NOSTARTFILE
fi
##############################################


ALIVE1=.
DEAD1=_
                 # Представляем живые и мертвые ячейки в начальном файле.

#  ------------------------------------------------------#
#  Этот сценарий использует сетку 10 x 10 (можно изменить,
#+ но с большой сеткой выполняться будет медленно).
ROWS=10
COLS=10
#  Измените две переменные выше в соответствии с желаемым размером сетки.
#  ------------------------------------------------------#

GENERATIONS=10          #  Количество циклов генераций.
                        #  Увеличьте, если у вас
                        #+ есть много свободного времени.

NONE_ALIVE=85           #  Статус выхода при преждевременном вылете,
                        #+ если нет живой ячейки слева.
DELAY=2                 #  Пауза между генерациями.
TRUE=0
```

```
FALSE=1
ALIVE=0
DEAD=1

avar=                   # Глобальная; содержит текущую генерацию.
generation=0            # Присваивает начало отсчета генераций.

# ================================================================

let "cells = $ROWS * $COLS"   # Количество ячеек.

# Массив содержащий "ячейки."
declare -a initial
declare -a current

display ()
{

alive=0                 # Количество живых ячеек в любой момент времени.
                        # Присваивается ноль.

declare -a arr
arr=( `echo "$1"` )     # Преобразование переданных аргументов в массив.

element_count=${#arr[*]}

local i
local rowcheck

for ((i=0; i<$element_count; i++))
do

  # Вставка символа перевода строки в конце каждой строки.
  let "rowcheck = $i % COLS"
  if [ "$rowcheck" -eq 0 ]
  then
    echo                # Перевод строки.
    echo -n "      "    # Отступ.
  fi

  cell=${arr[i]}

  if [ "$cell" = . ]
  then
    let "alive += 1"
  fi

  echo -n "$cell" | sed -e 's/_/ /g'
  # Вывод из массива, заменяя подчеркивания на пробелы.
done

return

}

IsValid ()                      # Проверка правильности координат ячейки.
{

  if [ -z "$1"  -o -z "$2" ]       # Отсутствуют обязательные аргументы?
  then
    return $FALSE
```

```
  fi

local row
local lower_limit=0                # Запрещаем отрицательные координаты.
local upper_limit
local left
local right

let "upper_limit = $ROWS * $COLS - 1" # Общее число ячеек.


if [ "$1" -lt "$lower_limit" -o "$1" -gt "$upper_limit" ]
then
  return $FALSE                    # Вне границ массива.
fi

row=$2
let "left = $row * $COLS"          # Ограничение слева.
let "right = $left + $COLS - 1"    # Ограничение справа.

if [ "$1" -lt "$left" -o "$1" -gt "$right" ]
then
  return $FALSE                    # Выход за пределы границы строки.
fi

return $TRUE                       # Правильные координаты.

}


IsAlive ()             #  Проверка, является ли ячейка живой.
                       #  Принимает - массив, номер ячейки и состояние
{                      #+ ячейки, в качестве аргументов.
  GetCount "$1" $2     #  Получаем количество соседних живых клеток.
  local nhbd=$?

  if [ "$nhbd" -eq "$BIRTH" ]  # Живые в любом случае.
  then
    return $ALIVE
  fi

  if [ "$3" = "." -a "$nhbd" -eq "$SURVIVE" ]
  then                 # Живые, только если ранее были живыми.
    return $ALIVE
  fi

  return $DEAD         # По умолчанию мертвые.

}


GetCount ()            # Считаем живые соседние ячейки.
                       # Необходимы два аргумента:
                       # $1) общая переменная массива
                       # $2) номер ячейки
{
  local cell_number=$2
  local array
  local top
  local center
  local bottom
```

```
local r
local row
local i
local t_top
local t_cen
local t_bot
local count=0
local ROW_NHBD=3

array=( `echo "$1"` )

let "top = $cell_number - $COLS - 1"    # Присваиваем соседнюю ячейку.
let "center = $cell_number - 1"
let "bottom = $cell_number + $COLS - 1"
let "r = $cell_number / $COLS"

for ((i=0; i<$ROW_NHBD; i++))           # Перемещение слева на право.
do
  let "t_top = $top + $i"
  let "t_cen = $center + $i"
  let "t_bot = $bottom + $i"


  let "row = $r"                        # Вычисляем центр строки.
  IsValid $t_cen $row                   # Правильная позиция ячейки?
  if [ $? -eq "$TRUE" ]
  then
    if [ ${array[$t_cen]} = "$ALIVE1" ] # Ожившая?
    then                                # Если да, то ...
      let "count += 1"                  # Увеличиваем счетчик.
    fi
  fi

  let "row = $r - 1"                    # Вычисляем верхнюю строку.
  IsValid $t_top $row
  if [ $? -eq "$TRUE" ]
  then
    if [ ${array[$t_top]} = "$ALIVE1" ] # Здесь избыточность.
    then                                # Можно ее оптимизировать?
      let "count += 1"
    fi
  fi

  let "row = $r + 1"                    # Вычисляем нижнюю строку.
  IsValid $t_bot $row
  if [ $? -eq "$TRUE" ]
  then
    if [ ${array[$t_bot]} = "$ALIVE1" ]
    then
      let "count += 1"
    fi
  fi

done


if [ ${array[$cell_number]} = "$ALIVE1" ]
then
  let "count -= 1"        #  Убедитесь, что значение самих проверенных
fi                        #+ ячеек не подсчитано.
```

```
  return $count

}

next_gen ()               # Обновляем массив генераций.
{

local array
local i=0

array=( `echo "$1"` )     # Преобразуем переданные аргументы в массив.

while [ "$i" -lt "$cells" ]
do
  IsAlive "$1" $i ${array[$i]}   # Ожившая клетка?
  if [ $? -eq "$ALIVE" ]
  then                           #  Если ожившая, то
    array[$i]=.                  #+ представляем ячейку как точку.
  else
    array[$i]="_"                #  Иначе - подчеркивание
   fi                            #+ (позже будет преобразована в пустое место).
  let "i += 1"
done


#    let "generation += 1"       # Увеличиваем счетчик генераций.
###  Почему строка выше закомментирована?


# Задаем передающуюся переменную, как параметр для функции  «display».
avar=`echo ${array[@]}`   # Преобразуем массив обратно в строковую переменную.
display "$avar"           # Выводим ее на экран.
echo; echo
echo "Генерация $generation  -  $alive живых"

if [ "$alive" -eq 0 ]
then
  echo
  echo "Преждевременный выход: нет больше живых ячеек!"
  exit $NONE_ALIVE        #  Окончание,
fi                        #+ если нет живых ячеек.

}


# ============================================================

# main ()
# {

# Загружаем исходный массив с содержимым начального (стартового) файла.
initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
# Удаляем строки содержащие символ комментария '#'.
          sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# Удаляем символы перевода строки и вставляем пробелы между элементами.

clear          # Очистка экрана.

echo #         Название
```

```
setterm -reverse on
echo "======================="
setterm -reverse off
echo "   Генерация $GENERATIONS"
echo "           закончена"
echo "\"Life in the Slow Lane\""
setterm -reverse on
echo "======================="
setterm -reverse off

sleep $DELAY   # Вывод "splash screen" на 2 секунды.


# -------- Вывод на экран первой генерации. --------
Gen0=`echo ${initial[@]}`
display "$Gen0"           # Только вывод на экран.
echo; echo
echo "Генерация $generation  -  $alive живых"
sleep $DELAY
# ----------------------------------------


let "generation += 1"     # Увеличение счетчика генераций.
echo

# ------- Вывод на экран второй генерации. -------
Cur=`echo ${initial[@]}`
next_gen "$Cur"          # Обновление & вывод на экран.
sleep $DELAY
# ----------------------------------------

let "generation += 1"     # Увеличиваем счетчик генераций.

# ------ Главный цикл для вывода на экран последующих генераций ------
while [ "$generation" -le "$GENERATIONS" ]
do
  Cur="$avar"
  next_gen "$Cur"
  let "generation += 1"
  sleep $DELAY
done
# ============================================================

echo
# }

exit 0   # CEOF:EOF



# Сетка в этом сценарии имеет "проблемы с границами".
# Сверху, снизу и с боков ячейки граничат с пустыми мертвыми ячейками.
# Упражнение: Измените сценарий, чтобы сетка была вокруг,
# +           так что бы 'соприкасались' левая и правая стороны,
# +           а так же верх и низ.
#
# Упражнение: Создайте новый файл "gen0", потомок этого сценария.
#             Используйте сетку 12 x 16, вместо оригинальной 10 x 10.
#             Сделайте соответствующие изменения в сценарии,
#+            что бы запускать с измененным файлом.
#
```

```
# Упражнение: Измените этот сценарий, что бы он мог определять
#+            размер сетки из файла «gen0» и задавать любые
#+            переменные, необходимые для запуска сценария.
#             Это сделало бы ненужным любые замены переменных в
#+            сценарии при изменении сетки.
#
# Упражнение: Оптимизируйте этот сценарий.
#             Он имеет избыточный код.
```

Пример A-11. Файл данных для *Game of Life*

```
# gen0
#
# Это пример начального файла "generation 0" для "life.sh".
# ---------------------------------------------------------------
#  Файл "gen0" является сеткой 10 x 10 использующей точки (.) для живых
#+ ячеек, и подчеркивания (_) для мертвых. Мы не можем использовать пробелы для
#+ мертвых ячеек в этом файле из-за особенностей массивов Bash.
#  Читайте пример: он это объясняет.]
#
# Строки начинающиеся с комментария, '#', сценарий игнорирует.
__.__..___
__..._.____
____.___..
_._______.
____._____
..__...___
____._____
___...____
__.._..___
_..___..__
```

+++

Следующий сценарий Mark Moraes из университета Торонто. См. файл `Moraes-COPYRIGHT` о правах и ограничениях.

Пример A-12. *behead*: Удаление почты и заголовков сообщений новостей

```
#! /bin/sh
#  Удаление заголовков из почты/сообщений новостей т.е.
#+ до первой пустой строки.
#  Автор: Mark Moraes, Университет Торонто

# ==> Такие комментарии добавлены автором этого документа.

if [ $# -eq 0 ]; then
# ==> Если нет аргумента командной строки, то
#+    перенаправляется из файла на stdin.
       sed -e '1,/^$/d' -e '/^[        ]*$/d'
       # --> Удаляются пустые строки и все строки до
       # --> строки начинающейся с отступа.
```

```
else
# ==> Если есть аргумент командной строки, то из указанного файла.
        for i do
                sed -e '1,/^$/d' -e '/^[        ]*$/d' $i
                # --> То же самое, как выше.
        done
fi

exit

# ==> Упражнение: Добавьте проверку ошибок и других опций.
# ==>
# ==> Обратите внимание, что маленький сценарий sed повторяется,
#+    за исключением переданных аргументов.
# ==> Имеет ли смысл вставить ее в функцию? Почему или почему нет?


/*
 * Copyright University of Toronto 1988, 1989.
 * Written by Mark Moraes
 *
 * Permission is granted to anyone to use this software for any purpose on
 * any computer system, and to alter it and redistribute it freely, subject
 * to the following restrictions:
 *
 * 1. The author and the University of Toronto are not responsible
 *    for the consequences of use of this software, no matter how awful,
 *    even if they arise from flaws in it.
 *
 * 2. The origin of this software must not be misrepresented, either by
 *    explicit claim or by omission.  Since few users ever read sources,
 *    credits must appear in the documentation.
 *
 * 3. Altered versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.  Since few users
 *    ever read sources, credits must appear in the documentation.
 *
 * 4. This notice may not be removed or altered.
 */
```

+

Antek Sawicki предоставил следующий сценарий, реализующий параметр подстановки операторов, обсуждающегося в Разделе 10.2.

Пример A-13. *password*: Создание случайного 8-символьного пароля

```
#!/bin/bash
#
#
#  Генератор случайных паролей для Bash 2.x+
#+ Antek Sawicki <tenox@tenox.tc>,
#+ который великодушно дал разрешение использование в ABS Guide.
#
# ==> Комментарии добавленные автором документа ==>


MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
# ==> Пароль содержит буквенно-цифровые символы.
LENGTH="8"
```

```
# ==> Для увеличения пароля 'LENGTH' можно изменить.


while [ "${n:=1}" -le "$LENGTH" ]
# ==> Напомню, что := это оператор "подстановки значения по умолчанию".
# ==> Поэтому, если 'n' не будет объявлена, присваивается 1.
do
        PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
        # ==> Очень заумно.

        # ==> Начнем с вложенного внутрь ...
        # ==> ${#MATRIX} возвращает длину массива MATRIX.

        # ==> $RANDOM%${#MATRIX} возвращает случайное число между 1
        # ==> и [длина MATRIX] - 1.

        # ==> ${MATRIX:$(($RANDOM%${#MATRIX})):1}
        # ==> возвращает расширение MATRIX случайной позиции, длиной 1.
        # ==> См подстановку параметров {var:pos:len} в Главе 9.
        # ==> и сравни примеры.

        # ==> PASS=... просто добавляется этот результат к предыдущей PASS
        #+    (конкатенация).

        # ==> Что бы увидеть более четко, раскомментируйте следующую
        #+    строку
        #                echo "$PASS"
        # ==> Что бы увидеть, что за каждый проход цикла
        # ==> PASS добавляет один символ за раз.

        let n+=1
        # ==> При следующем проходе 'n' увеличивается.
done

echo "$PASS"      # ==> Или, перенаправляется в файл, по желанию.

exit 0
```

+

James R. Van Zandt предоставил этот сценарий, использующий именованные каналы и, с его слов , это "хорошее упражнение для кавычек и экранирования."

Пример A-14. *fifo*: Производство ежедневного резервного копирования с помощью именованных каналов

```
#!/bin/bash
# ==> Сценарий James R. Van Zandt, используется с разрешения.

# ==> Комментарии добавлены автором этого документа.


  HERE=`uname -n`    # ==> имя хоста
  THERE=bilbo
  echo "Запускаем удаленный бэкап на $THERE в `date +%r`"
  # ==> `date +%r` возвращает время в 12-часовом формате, т.е. "08:08:34 PM".
```

```
  # Убедитесь, что /pipe действительно является каналом, а не просто файлом.
  rm -rf /pipe
  mkfifo /pipe       # ==> Создается "именованный канал", назвав "/pipe" ...

  # ==> 'su xyz' запускает команды от имени пользователя "xyz".
  # ==> 'ssh' вызов безопасной оболочки (клиент удаленного входа).
su xyz -c "ssh $THERE \"cat > /home/xyz/backup/${HERE}-daily.tar.gz\" < /pipe"&
  cd /
  tar -czf - bin boot dev etc home info lib man root sbin share usr var > /pipe
  # ==> Используем именованный канал, /pipe, для взаимодействия между
  #+    процессами:
  # ==> 'tar/gzip' пишет в /pipe, а 'ssh' читает из /pipe.

  # ==> Конечным результатом является бэкап основных директорий / вниз.

  # ==>  Каковы преимущества в этой ситуации "именованного канала",
  # ==>+ в отличии от "анонимного канала" с |?
  # ==>  Будет здесь работать анонимный канал?

  # ==>  Необходимо удалять канал перед выходом из сценария?
  # ==>  Как можно это сделать?


  exit 0
```

+

Stéphane Chazelas использует следующий сценарий для демонстрации создания простых чисел без помощи массивов.

Пример A-15. Создание простых чисел с помощью оператора модуль

```
#!/bin/bash
# primes.sh: Создание простых чисел без использования массива.
# Сценарий предоставлен Stephane Chazelas.

#  Здесь *не* используется классический алгоритм "Сито Эратосфена",
#+ а вместо него используется более интуитивный способ проверки каждого числа
#+ кандидата фактором (делением), с помощью оператора остатка от деления «%».


LIMIT=1000                    # Простые, 2 ... 1000.

Primes()
{
 (( n = $1 + 1 ))             # Переход к следующему целому числу.
 shift                        # Следующий параметр в списке.
#  echo "_n=$n i=$i_"

 if (( n == LIMIT ))
 then echo $*
 return
 fi

 for i; do                    # "i" присваивается в "@", начальное значение $n.
#   echo "-n=$n i=$i-"
   (( i * i > n )) && break   # Оптимизация.
   (( n % i )) && continue    # Отсеиваем составные числа, с помощью оператора
                              #+ модуль.
   Primes $n $@               # Рекурсия внутри цикла.
```

```
   return
   done

   Primes $n $@ $n            #  Рекурсия вне цикла.
                              #  Последовательно собираем
                              #+ позиционные параметры.
                              #  "$@" это объединенный список простых чисел.
}

Primes 1

exit $?

# Передайте вывод сценария в «fmt» для более красивого вывода на экран.

#  Раскомментируйте строки 16 и 24, чтобы понять, что происходит.

#  Сравните скорость применения этого алгоритма создания простых чисел
#+ с Ситом Эратосфена (ex68.sh).


#  Упражнение: Перепишите этот сценарий без рекурсий.
```

+

Rick Boivie пересмотрел сценарий *tree*, написаный Jordi Sanfeliu.

Пример A-16. *tree*: Вывод на экран дерева директорий

```
#!/bin/bash
# tree.sh

#  Написан Rick Boivie.
#  Используется с разрешения.
#  Это пересмотренная и упрощенная версия сценария
#+ Jordi Sanfeliu (автор оригинала), и исправленная Ian Kjos.
#  Этот сценарий заменяет более ранние версии использованные
#+ ранее в Advanced Bash Scripting Guide.
#  Copyright (c) 2002, Jordi Sanfeliu, Rick Boivie и Ian Kjos.

# ==> Комментарии добавленные автором этого документа.


search () {
for dir in `echo *`
#  ==> `echo *` Список всех файлов в текущей рабочей директории,
#+ ==> без пробелов.
#  ==> Подобный эффект дает for dir in *
#  ==> но "dir in `echo *`" не будет обрабатывать имена с пробелами.
do
  if [ -d "$dir" ] ; then # ==> Если это директория (-d)...
  zz=0                    # ==> Временная переменная, следит
                          #     за уровнем директории.
  while [ $zz != $1 ]     # Следит за вложенным внутрь цикла.
    do
      echo -n "| "        # ==> Выводит вертикальную черту,
                          # ==> с 2 пробелами & без перевода строки
                          #     для задания отступа.
      zz=`expr $zz + 1`    # ==> zz увеличивается.
```

```
    done

    if [ -L "$dir" ] ; then # ==> Если директория это символическая ссылка...
      echo "+---$dir" `ls -l $dir | sed 's/^.*'$dir' //'`
      # ==> Выводится горизонтальная черта и содержимое директории, но...
      # ==> без части листинга «дата/время».
    else
      echo "+---$dir"             # ==> Выводится горизонтальная черта...
                                  # ==> и выводится имя директории.
      numdirs=`expr $numdirs + 1` # ==> Счетчик директорий увеличивается.
      if cd "$dir" ; then          # ==> Если можно перейти в поддиректорию...
        search `expr $1 + 1`      # с рекурсией ;-)
                                   # ==> Функция вызывающая сама себя.
        cd ..
      fi
    fi
  fi
done
}

if [ $# != 0 ] ; then
  cd $1   # Переходим в указанную директорию.
  #else   # Остаемся в текущей директории
fi

echo "Указанная директория = `pwd`"
numdirs=0

search 0
echo "Всего директорий = $numdirs"

exit 0
```

Версия Patsie сценария *дерева* директории.

Пример A-17. *tree2*: Альтернативный сценарий *дерева* директории

```
#!/bin/bash
# tree2.sh

# Слегка изменен/переделан автором ABS Guide.
# Включен в ABS Guide с разрешения автора сценария (спасибо!).

## Сценарий рекурсивной проверки размера файла/директории написанный Patsie.
##
## Сценарий создает список файлов/директорий и их размер (du -akx)
## и представляет этот список в виде удобочитаемого дерева.
## du -akx' только, если владелец имеет права.
## Чтобы получить наилучший результат желательно запускать из под root*,
## или использовать только для директорий, для которых установлены
## права на чтение. Всего, на что у вас нет прав - в списке не будет.

#* Автор ABS Guide советует соблюдать осторожность запускаясь из-под root!


##########  ЭТО КОНФИГУРАЦИЯ  ##########
```

```
TOP=5                   # Список 5 крупнейших (под)директорий.
MAXRECURS=5             # Максимальная глубина 5 поддиректорий/рекурсий.
E_BL=80                 # Возвращение пустой строки.
E_DIR=81                # Директория не задана.


##########  НИЧЕГО НЕ МЕНЯЙТЕ НИЖЕ ЭТОЙ» СТРОКИ  ##########

PID=$$                          # ID вашего процесса.
SELF=`basename $0`               # Имя вашей программы.
TMP="/tmp/${SELF}.${PID}.tmp"    # Временный результат 'du'.

# Преобразование числа в пунктир из тысячи точек.
function dot { echo "           $*" |
               sed -e :a -e 's/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta' |
               tail -c 12; }

# Usage: tree <рекурсия> <префикс> <мин.размер> <директория>
function tree {
  recurs="$1"           # Глубина вложения
  prefix="$2"           # Что выводим перед именем файла/директории
  minsize="$3"          # Минимальный размер файла/директории
  dirname="$4"          # Проверяемая директория

# Берем ($TOP) крупнейшие поддиректории/подфайлы из файла TMP.
  LIST=`egrep "[[:space:]]${dirname}/[^/]*$" "$TMP" |
        awk '{if($1>'$minsize') print;}' | sort -nr | head -$TOP`
  [ -z "$LIST" ] && return        # Если список пуст, то возвращаемся.

  cnt=0
  num=`echo "$LIST" | wc -l`      # Количество записей в списке.

  ## Основной цикл
  echo "$LIST" | while read size name; do
    ((cnt+=1))                     # Счетчик числа записей.
    bname=`basename "$name"`       # В записи нам необходимо только basename.
    [ -d "$name" ] && bname="$bname/"
                                   # Если это директория, добавляется слэш.
    echo "`dot $size`$prefix +-$bname"
                                   # Выводится результат.
    #  Вызывает себя рекурсивно, если это директория и
    #+ мы не слишком глубоко вошли($MAXRECURS).
    #  Переход рекурсии: $((recurs+1))
    #  Префикс получает пробел, если это последняя запись
    #+ или канал, если еще есть записи.
    #  Минимальный размер файла/директории становится
    #+ десятой долей их родителя: $((size/10)).
    #  Последний аргумент это проверка полного имени директории.
    if [ -d "$name" -a $recurs -lt $MAXRECURS ]; then
      [ $cnt -lt $num ] \
        || (tree $((recurs+1)) "$prefix  " $((size/10)) "$name") \
        && (tree $((recurs+1)) "$prefix |" $((size/10)) "$name")
    fi
  done

  [ $? -eq 0 ] && echo "           $prefix"
  # Каждый раз, когда мы возвращаемся назад добавляется «пустая» строка.
  return $E_BL
  # Возвращается 80, чтобы сказать, что уже добавлена пустая строка.
}
```

```
###                       ###
###  Основная программа   ###
###                       ###

rootdir="$@"
[ -d "$rootdir" ] ||
  { echo "$SELF: Usage: $SELF <directory>" >&2; exit $E_DIR; }
  # Должно вызываться с именем директории.

echo "Создается список, пожалуйста, подождите ..."
     # Вывод сообщения "пожалуйста подождите".
du -akx "$rootdir" 1>"$TMP" 2>/dev/null
     # Создание временного списка всех файлов/директорий и их размеров.
size=`tail -1 "$TMP" | awk '{print $1}'`
     # Какой размер нашей директории root?
echo "`dot $size` $rootdir"
     # Выводит запись директории root.
tree 0 "" 0 "$rootdir"
     # Выводит дерево нашей директории root вниз.

rm "$TMP" 2>/dev/null
     # Очищаем файл TMP.

exit $?
```

Noah Friedman разрешил использовать его сценарий ***string function***. По существу, он воспроизводит некоторые функции обработки строк библиотеки Си.

Пример A-18. *string functions*: Функции обработки строк в стиле Си

```
#!/bin/bash

# string.bash --- bash эмуляция библиотечных подпрограмм string (3)
# Автор: Noah Friedman <friedman@prep.ai.mit.edu>
# ==>     В данном документе используется с его любезного разрешения.
# Создан: 1992-07-01
# Последнее изменение: 1993-09-29
# Public domain

# Преобразование в синтаксис bash v2 осуществил Chet Ramey

# Комментарии:
# Код:

#:docstring strcat:
# Usage: strcat s1 s2
#
# Strcat добавляет значение variable s2 в variable s1.
#
# Пример:
#    a="foo"
#    b="bar"
#    strcat a b
#    echo $a
#    => foobar
#
```

```
#:end docstring:

###;;;autoload   ==> Закомментированная автозагрузка функции.
function strcat ()
{
    local s1_val s2_val

    s1_val=${!1}                # Расширение косвенной ссылки на переменную
    s2_val=${!2}
    eval "$1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' позволяет избежать проблем,
    # ==> если каждая переменная находится внутри одинарных кавычек.
}

#:docstring strncat:
# Usage: strncat s1 s2 $n
#
# Подобна strcat, но strncat добавляет максимально n символов из значения
# variable s2. Если значение variable s2 меньше, чем n символов,
# то копируется меньше символов. Выводит результат на stdout.
#
# Пример:
#    a=foo
#    b=barbaz
#    strncat a b 3
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload
function strncat ()
{
    local s1="$1"
    local s2="$2"
    local -i n="$3"
    local s1_val s2_val

    s1_val=${!s1}              # ==> Расширение косвенной ссылки на переменную
    s2_val=${!s2}

    if [ ${#s2_val} -gt ${n} ]; then
       s2_val=${s2_val:0:$n}   # ==> извлечение substring (содержимого строки)
    fi

    eval "$s1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' позволяет избежать проблем,
    # ==> Если каждая переменная находится внутри одинарных кавычек.
}

#:docstring strcmp:
# Usage: strcmp $s1 $s2
#
# Strcmp сравнивает аргументы и возвращает целое число, меньше,равно,
# или больше, чем ноль, в зависимости от указанного для string s1
# лексикографического знака: меньше чем, равно, или больше чем string s2.
#:end docstring:

###;;;autoload
function strcmp ()
{
```

```
    [ "$1" = "$2" ] && return 0

    [ "${1}" '<' "${2}" ] > /dev/null && return -1

    return 1
}

#:docstring strncmp:
# Usage: strncmp $s1 $s2 $n
#
# Подобна strcmp, но делает сравнение проверкой максимума n
# символов (n меньше чем или равно нулю следующего равенства).
#:end docstring:

###;;;autoload
function strncmp ()
{
    if [ -z "${3}" -o "${3}" -le "0" ]; then
       return 0
    fi

    if [ ${3} -ge ${#1} -a ${3} -ge ${#2} ]; then
       strcmp "$1" "$2"
       return $?
    else
       s1=${1:0:$3}
       s2=${2:0:$3}
       strcmp $s1 $s2
       return $?
    fi
}

#:docstring strlen:
# Usage: strlen s
#
# Strlen возвращает количество символов буквы s в строке.
#:end docstring:

###;;;autoload
function strlen ()
{
    eval echo "\${#${1}}"
    # ==> Возвращает длину значения переменной,
    # ==> имя которой передано в качестве аргумента.
}

#:docstring strspn:
# Usage: strspn $s1 $s2
#
# Strspn возвращает максимальную длину исходного сегмента строки s1,
# который полностью состоит из символов строки s2.
#:end docstring:

###;;;autoload
function strspn ()
{
    # Не объявленный IFS позволяет обрабатывать пробелы, как обычные символы.
    local IFS=
    local result="${1%%[!${2}]*}"

    echo ${#result}
```

```
}

#:docstring strcspn:
# Usage: strcspn $s1 $s2
#
# Strcspn возвращает максимальную длину исходного сегмента строки s1,
# которая состоит исключительно из символов не входящих в строку s2.
#:end docstring:

###;;;autoload
function strcspn ()
{
    # Не объявленный IFS позволяет обрабатывать пробелы как обычные символы.
    local IFS=
    local result="${1%%[${2}]*}"

    echo ${#result}
}

#:docstring strstr:
# Usage: strstr s1 s2
#
# Strstr выводит substring, начиная с первого вхождения строки s2 в
# строке s1, или ничего, если s2 не встречается в строке. Если s2 это
# строка нулевой длинны, то strstr выводит s1.
#:end docstring:

###;;;autoload
function strstr ()
{
    # Если s2 это строка нулевой длины, strstr выводит s1
    [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }

    # strstr ничего не выводит, если s2 не встречается в s1
    case "$1" in
    *$2*) ;;
    *) return 1;;
    esac

    # С помощью кода шаблона соответствия удаляет соответствия и все
    # следующее за ними
    first=${1/$2*/}

    # затем удаляется первая не совпадающая часть строки
    echo "${1##$first}"
}

#:docstring strtok:
# Usage: strtok s1 s2
#
# Strtok считает строки s1, состоящие из последовательностей нулей или других
# текстовых маркеров, разделяя диапазоны одним или более символов из
# строки s2. Первой вызывается строка (с указанной не пустой строкой s1)
# содержащая первый маркер, выводимая на stdout. Эта функция отслеживает
# свою позицию в строке s1 между вызовами, поэтому последующие
# вызовы, сделанные с первым аргументом пустой строки, будут работать сразу
# после этого маркера построчно. Таким образом последующие вызовы будут
# обрабатывать построчно s1 до тех пор, пока не останется маркеров.
# Разделитель строки s2 может отличаться от вызова к вызову. Когда
# заканчиваются маркеры в строке s1, на stdout выводится пустое значение.
#:end docstring:
```

```
###;;;autoload
function strtok ()
{
 :
}

#:docstring strtrunc:
# Usage: strtrunc $n $s1 {$s2} {$...}
#
# Используется многими функциями, типа strncmp для уменьшения количества
# аргументов для сравнения.
# Выводит первые n символов каждой строки s1 s2 ... на stdout.
#:end docstring:

###;;;autoload
function strtrunc ()
{
    n=$1 ; shift
    for z; do
        echo "${z:0:$n}"
    done
}

# предоставленная строка

# string.bash здесь заканчивается


# ========================================================================== #
# ==> Все ниже добавлено автором документа.

# ==> Предлагаю с помощью этого сценария удалить все ниже,
# ==> и "исходник" этого файла в вашем собственном сценарии.

# strcat
string0=one
string1=two
echo
echo "Проверка функции \"strcat\":"
echo "Изначально \"string0\" = $string0"
echo "\"string1\" = $string1"
strcat string0 string1
echo "Новая \"string0\" = $string0"
echo

# strlen
echo
echo "Проверка функции \"strlen\":"
str=123456789
echo "\"str\" = $str"
echo -n "Длина \"str\" = "
strlen str
echo


# Упражнение:
# --------
# Добавьте код для проверки всех других строковых функций выше.
```

```
exit 0
```

Пример комплексного массива, написанного Michael Zick, использующего проверку командой ***md5sum*** зашифрованной информации о директории.

Пример A-19. Информация о директории

```
#! /bin/bash
# directory-info.sh
# Анализ  и листинг информации о директории.

# ПРИМЕЧАНИЕ: Изменены строки 273 и 353 файла "README".

# Автор этого сценария Michael Zick.
# Используется с его разрешения.

# Управление
#  Если переопределяются аргументы командной строки, то они должны быть в
#+ следующем порядке:
#   Arg1: "Дескриптор директории"
#   Arg2: "Исключенные пути"
#   Arg3: "Исключенные директории"
#
# Параметры среды переопределяют значения по умолчанию.
# Команды аргументов переопределяют параметры среды.

# Нахождение содержимого рассматриваемых файловых дескрипторов по умолчанию.
MD5UCFS=${1:-${MD5UCFS:-'/tmpfs/ucfs'}}

# Пути директорий не перечисляются или не вводятся
declare -a \
  EXCLUDE_PATHS=${2:-${EXCLUDE_PATHS:-'(/proc /dev /devfs /tmpfs)'}}

# Директории не перечисляются или не вводятся
declare -a \
  EXCLUDE_DIRS=${3:-${EXCLUDE_DIRS:-'(ucfs lost+found tmp wtmp)'}}

# Файлы не перечисляются или не вводятся
declare -a \
  EXCLUDE_FILES=${3:-${EXCLUDE_FILES:-'(core "Имя с пробелами")'}}


# Here document используется как блок комментариев.
: <<LSfieldsDoc
# # # # # Содержимое информации о директориях файловой системы # # # # #
#
#       ListDirectory "FileGlob" "Field-Array-Name"
# или
#       ListDirectory -of "FileGlob" "Field-Array-Filename"
#       '-of' означает 'output to filename' (вывод в файл)
# # # # #

Форматируем строку дескрипторов основываясь на ls (утилите GNU) версии 4.0.36

Создание форматированной строки (или нескольких):
инод  права      жесткие ссылки владелец группа ...
32736 -rw-------         1       mszick   mszick
```

```
размер   день месяц дата hh:mm:ss год        путь
2756608  Sun   Apr   20  08:53:06 2003 /home/mszick/core

Если по другому, то форматировано:
инод   права жесткая ссылка владелец group ...
266705 crw-rw----    1       root  uucp

старшее младшее день  месяц дата hh:mm:ss год    путь
  4,      68    Sun    Apr   20  09:27:33 2003 /dev/ttyS4
ПРИМЕЧАНИЕ: надоедливая запятая после старшего числа

ПРИМЕЧАНИЕ:'путь' может состоять из нескольких полей:
/home/mszick/core
/proc/982/fd/0 -> /dev/null
/proc/982/fd/1 -> /home/mszick/.xsession-errors
/proc/982/fd/13 -> /tmp/tmpfZVVOCs (deleted)
/proc/982/fd/7 -> /tmp/kde-mszick/ksycoca
/proc/982/fd/8 -> socket:[11586]
/proc/982/fd/9 -> pipe:[11588]

Если этого не достаточно, для сохранения логического анализа,
путь одного или обоих компонентов  может быть относительным:
../Built-Shared -> Built-Static
../linux-2.4.20.tar.bz2 -> ../../../SRCS/linux-2.4.20.tar.bz2

Первый символ из 11 (10?) символов полей:
's' Сокет
'd' Директория
'b' Блочное устройство
'c' Символьное устройство
'l' Символическая ссылка
ПРИМЕЧАНИЕ: Жесткие ссылки не отмечены - проверяется соответствие inode
соответствующим файловым системам. Вся информация о жестких ссылках является
общей, за исключением имен и местоположения имен в системе директорий.
ПРИМЕЧАНИЕ: "Жесткая ссылка" в некоторых системах называется "File Alias".
'-' Неопределенный файл

Следующий - три группы букв: Пользователь, Группа, Остальные
Символ 1: '-' Чтение запрещено; 'r' Чтение разрешено
Символ 2: '-' Запись запрещена; 'w' Запись разрешена
Символ 3, Пользователи и Группа: Совместное исполнение и особенность
'-' Не исполняемый, не специальный
'x' Исполняемый, не специальный
's' Исполняемый, специальный
'S' Не исполняемый, специальный
Символ 3, Остальные: Комбинированный исполняемый и стики-бит (липкий?)
'-' Не исполняемый, не стики
'x' Исполняемый, не стики
't' Исполняемый, стики
'T' Не исполняемый, стики

Далее индикатор доступа
Не проверял ни разу, это может быть одиннадцатый символ или может генерироваться
еще одно поле
' ' Нет альтернативного доступа
'+' Альтернативный доступ
LSfieldsDoc

ListDirectory()
{
        local -a T
```

```
        local -i of=0           # Возвращение переменной по умолчанию
#       OLD_IFS=$IFS            # В BASH по умолчанию ' \t\n'

        case "$#" in
        3)      case "$1" in
                -of)    of=1 ; shift ;;
                 * )    return 1 ;;
                esac ;;
        2)      : ;;            # "continue"
        *)      return 1 ;;
        esac

        # ПРИМЕЧАНИЕ: команда (ls) НЕ заключена в кавычки (")
        T=( $(ls --inode --ignore-backups --almost-all --directory \
        --full-time --color=none --time=status --sort=none \
        --format=long $1) )

        case $of in
        # Назначаем T обратно массиву, имя которого было передано как $2
                0) eval $2=\( \"\$\{T\[@\]\}\" \) ;;
        # Пишем T в файл переданный как $2
                1) echo "${T[@]}" > "$2" ;;
        esac
        return 0
   }

# # # # # Является номер этой строки законным? # # # # # #
#
#       IsNumber "Var"
# # # # # Здесь должен быть лучший способ, вздох ...

IsNumber()
{
        local -i int
        if [ $# -eq 0 ]
        then
                return 1
        else
                (let int=$1)  2>/dev/null
                return $?       # Статус выхода потока let
        fi
}

# # # # # Индексная информация директорий файловой системы  # # # # # #
#
#       IndexList "Field-Array-Name" "Index-Array-Name"
# или
#       IndexList -if Field-Array-Filename Index-Array-Name
#       IndexList -of Field-Array-Name Index-Array-Filename
#       IndexList -if -of Field-Array-Filename Index-Array-Filename
# # # # # #

: <<IndexListDoc
Проход по массиву полей директории произведенный ListDirectory

Подавляются разрывы строк, в противном случае строки определяются сообщением,
создается индекс элемента массива, с которого начинается каждая строка.

Каждая строка получает два индекса, первого элемента каждой строки (inode) и
элемента, содержащего путь к файлу.
Первая запись пары индексов (Строка-номер==0) информационная:
```

```
Index-Array-Name[0] : Количество индексированных "Строк"
Index-Array-Name[1] : "Текущая строка" указанная в Index-Array-Name

Следующая пара индексов (если есть) проводит индексы элементов в
Field-Array-Name:
Index-Array-Name[Line-Number * 2] : Поле элемента "inode".
ПРИМЕЧАНИЕ: Этот диапазон может быть либо +11 или +12 элементов.
Index-Array-Name[(Line-Number * 2) + 1] : Элемент "pathname".
ПРИМЕЧАНИЕ: Этот Диапазон может быть переменным количеством элементов.
Следующая строка пары индексоа для Строка-номер+1.
IndexListDoc

IndexList()
{
        local -a LIST                   # Локальное переданное имя списка
        local -a -i INDEX=( 0 0 )       # Локальный возвращаемый индекс
        local -i Lidx Lcnt
        local -i if=0 of=0              # Имя переменной по умолчанию

        case "$#" in                    # Упрощенный вариант проверки
                0) return 1 ;;
                1) return 1 ;;
                2) : ;;                 # "continue"
                3) case "$1" in
                        -if) if=1 ;;
                        -of) of=1 ;;
                         * ) return 1 ;;
                   esac ; shift ;;
                4) if=1 ; of=1 ; shift ; shift ;;
                *) return 1
        esac

        # Делаем локальную копию списка
        case "$if" in
                0) eval LIST=\( \"\$\{$1\[@\]\}\" \) ;;
                1) LIST=( $(cat $1) ) ;;
        esac

        # Grok (обыскать?) массив
#  Grok -  Мощный инструмент сопоставления с образцом и взаимодействия (прим.
#+ переводчика)

        Lcnt=${#LIST[@]}
        Lidx=0
        until (( Lidx >= Lcnt ))
        do
        if IsNumber ${LIST[$Lidx]}
        then
                local -i inode name
                local ft
                inode=Lidx
                local m=${LIST[$Lidx+2]}        # Поле жестких ссылок
                ft=${LIST[$Lidx+1]:0:1}         # Быстрое состояние
                case $ft in
                b)      ((Lidx+=12)) ;;         # Блочное устройство
                c)      ((Lidx+=12)) ;;         # Символьное устройство
                *)      ((Lidx+=11)) ;;         # Что-то еще
                esac
                name=Lidx
                case $ft in
                -)      ((Lidx+=1)) ;;          # Легкий
```

```
                b)      ((Lidx+=1)) ;;          # Блочное устройство
                c)      ((Lidx+=1)) ;;          # Символьное устройство
                d)      ((Lidx+=1)) ;;          # Другой легкий
                l)      ((Lidx+=3)) ;;          # Еще по КРАЙНЕЙ МЕРЕ два поля
#  А здесь более элегантная обработка каналов,
#+ сокетов, затем  - удаляемых файлов
                *)      until IsNumber ${LIST[$Lidx]} || ((Lidx >= Lcnt))
                        do
                                ((Lidx+=1))
                        done
                        ;;                      # Не нужно
                esac
                INDEX[${#INDEX[*]}]=$inode
                INDEX[${#INDEX[*]}]=$name
                INDEX[0]=${INDEX[0]}+1          # Найдена еще одна "строка"
# echo "Строка: ${INDEX[0]} Тип: $ft Ссылки: $m Иноды: \
# ${LIST[$inode]} Имя: ${LIST[$name]}"

        else
                ((Lidx+=1))
        fi
        done
        case "$of" in
                0) eval $2=\( \"\$\{INDEX\[@\]\}\" \) ;;
                1) echo "${INDEX[@]}" > "$2" ;;
        esac
        return 0                                # Что может пойти не так?
}

# # # # # Определение содержимого файла # # # # # #
#
#       DigestFile Input-Array-Name Digest-Array-Name
# или
#       DigestFile -if Input-FileName Digest-Array-Name
# # # # # #

# Here document используется как блок для комментария.
: <<DigestFilesDoc

Ключом (не каламбур) Unified Content File System (UCFS)
является выделение файлов в системе, основанной на их содержании.
Отличали файлы по имени только в 20-м веке.

Содержимое различается путем вычисления контрольной суммы самого содержимого.
Эта версия использует программу md5sum для создания 128 битной контрольной
суммы представляющей содержимое файла.
Есть шанс, что два файла, имеющие различное содержимое, могут генерировать одну
и ту же контрольную сумму md5sum (или любую контрольную сумму). Если это
проблема, то использование md5sum можно заменить зашифрованными подписями.  Но
до сих пор...

Программа md5sum документирована для вывода трех полей (и их производства), но
после прочтения она выводит два поля (элемента массива). Это вызвано отсутствием
пробелов между второй и третьей областью.
Так что эта функция проверяет вывод md5sum и возвращает:
        [0]     32 символа в шестнадцатеричном формате (имя UCFS)
        [1]     Единственный символ: ' ' текстовый файл, '*' бинарный файл
        [2]     Имя файловой системы (Стиль 20-го века)
        ПРИМЕЧАНИЕ: Это имя может быть символом '-' показывающим чтение STDIN.

DigestFilesDoc
```

```
DigestFile()
{
        local if=0              # Имя переменной, по умолчанию
        local -a T1 T2

        case "$#" in
        3)      case "$1" in
                -if)    if=1 ; shift ;;
                 * )    return 1 ;;
                esac ;;
        2)      : ;;            # "continue"
        *)      return 1 ;;
        esac

        case $if in
        0) eval T1=\( \"\$\{$1\[@\]\}\" \)
           T2=( $(echo ${T1[@]} | md5sum -) )
           ;;
        1) T2=( $(md5sum $1) )
           ;;
        esac

        case ${#T2[@]} in
        0) return 1 ;;
        1) return 1 ;;
        2) case ${T2[1]:0:1} in         # SanScrit-2.0.5
           \*) T2[${#T2[@]}]=${T2[1]:1}
               T2[1]=\*
               ;;
            *) T2[${#T2[@]}]=${T2[1]}
               T2[1]=" "
               ;;
           esac
           ;;
        3) : ;; # Допустим это сработало
        *) return 1 ;;
        esac

        local -i len=${#T2[0]}
        if [ $len -ne 32 ] ; then return 1 ; fi
        eval $2=\( \"\$\{T2\[@\]\}\" \)
}

# # # # # Расположение файла # # # # # #
#
#       LocateFile [-l] FileName Location-Array-Name
# или
#       LocateFile [-l] -of FileName Location-Array-FileName
# # # # #

# Расположением файла является id файловой системы и номер инода

# Here document используется как блок для комментария.
: <<StatFieldsDoc
        Основано на stat, version 2.2
        поля stat -t и stat -lt
        [0]     Имя
        [1]     Общий размер
                Файл – количество байт
                Символическая ссылка – длина строки пути имени
```

```
        [2]     Число находящихся блоков (по 512 байт)
        [3]     Тип файла и права доступа (hex)
        [4]     ID пользователя
        [5]     ID группы
        [6]     Количество оборудования
        [7]     Количество Inode
        [8]     Число жестких ссылок
        [9]     Основной тип оборудования (если inode оборудования)
        [10]    Младший тип оборудования (если inode оборудования)
        [11]    Время последнего доступа
                Может быть отключено в «mount» noatime atime в файлах
                измененных  exec, read, pipe, utime atime mknod (mmap?),
                в директориях, измененных путем добавления/удаления файлов
        [12]    Время последней модификации
                mtime файлов модифицированных write, truncate, utime, mknod
                mtime директорий измененных добавлением/удалением файлов
        [13]    Время последнего изменения
                ctime отражает время изменения информации inode (владелец,
                группа права, количество ссылок)

-*-*- Для:
        Возвращаемый код: 0
        Размер массива: 14
        Содержимое массива
        Элемент 0: /home/mszick
        Элемент 1: 4096
        Элемент 2: 8
        Элемент 3: 41e8
        Элемент 4: 500
        Элемент 5: 500
        Элемент 6: 303
        Элемент 7: 32385
        Элемент 8: 22
        Элемент 9: 0
        Элемент 10: 0
        Элемент 11: 1051221030
        Элемент 12: 1051214068
        Элемент 13: 1051214068

        Для ссылки в виде linkname -> realname
        stat -t  linkname возвращает информацию linkname (ссылку)
        stat -lt linkname возвращает информацию realname

        Поля stat -tf и stat -ltf
        [0]     имя
        [1]     ID-0?           # Возможно в будущем, но структура stat в Linux
        [2]     ID-0?           # не имеет полей LABEL и UUID,
                                # текущая информация должна приходить из
                                # специальных утилит файловой системы
        Они будут передавать:
        [1]     UUID если возможно
        [2]     Значение Label если возможно
        Примечание: 'mount -l' возвращает label и может вернуть UUID

        [3]     Максимальный размер файла
        [4]     Тип файловой системы
        [5]     Общее количество блоков в ФС
        [6]     Свободные блоки
        [7]     Свободные блоки для не-root пользователя(ей)
        [8]     Размер блока ФС
        [9]     Общее количество inode
```

```
        [10]    Свободные inode

-*-*- Для:
        Возвращаемый код: 0
        Размер массива: 11
        Содержимое массива
        Элемент 0: /home/mszick
        Элемент 1: 0
        Элемент 2: 0
        Элемент 3: 255
        Элемент 4: ef53
        Элемент 5: 2581445
        Элемент 6: 2277180
        Элемент 7: 2146050
        Элемент 8: 4096
        Элемент 9: 1311552
        Элемент 10: 1276425

StatFieldsDoc


#       LocateFile [-l] FileName Location-Array-Name
#       LocateFile [-l] -of FileName Location-Array-FileName

LocateFile()
{
        local -a LOC LOC1 LOC2
        local lk="" of=0

        case "$#" in
        0) return 1 ;;
        1) return 1 ;;
        2) : ;;
        *) while (( "$#" > 2 ))
           do
              case "$1" in
               -l) lk=-1 ;;
              -of) of=1 ;;
                *) return 1 ;;
              esac
           shift
           done ;;
        esac

# Далее Sanscrit-2.0.5
      # LOC1=( $(stat -t $lk $1) )
      # LOC2=( $(stat -tf $lk $1) )
      #  Если в системе установлена команда "stat", раскомментируйте
      #+ две строки выше.
        LOC=( ${LOC1[@]:0:1} ${LOC1[@]:3:11}
              ${LOC2[@]:1:2} ${LOC2[@]:4:1} )

        case "$of" in
                0) eval $2=\( \"\$\{LOC\[@\]\}\" \) ;;
                1) echo "${LOC[@]}" > "$2" ;;
        esac
        return 0
# Что получите (если вам повезло, и установлена «stat»)
# -*-*- Расположение дескриптора -*-*-
#       Возвращаемый код: 0
#       Размер массива: 15
```

```
#       Содержимое массива
#       Элемент 0: /home/mszick         Имя 20th Century
#       Элемент 1: 41e8                 Тип и права доступа
#       Элемент 2: 500                  Пользователь
#       Элемент 3: 500                  Группа
#       Элемент 4: 303                  Устройство
#       Элемент 5: 32385                inode
#       Элемент 6: 22                   Количество ссылок
#       Элемент 7: 0                    Главное устройство
#       Элемент 8: 0                    Младшее устройство
#       Элемент 9: 1051224608           Последний доступ
#       Элемент 10: 1051214068          Последняя модификация
#       Элемент 11: 1051214068          Последнее состояние
#       Элемент 12: 0                   UUID (может быть)
#       Элемент 13: 0                   Значение Label (может быть)
#       Элемент 14: ef53                Тип ФС
}


# И затем здесь находился какой-то код проверки

ListArray() # ListArray Name
{
        local -a Ta

        eval Ta=\( \"\$\{$1\[@\]\}\" \)
        echo
        echo "-*-*- List of Array -*-*-"
        echo "Размер массива $1: ${#Ta[*]}"
        echo "Содержимое массива $1:"
        for (( i=0 ; i<${#Ta[*]} ; i++ ))
        do
            echo -e "\tElement $i: ${Ta[$i]}"
        done
        return 0
}

declare -a CUR_DIR
# Для небольших массивов
ListDirectory "${PWD}" CUR_DIR
ListArray CUR_DIR

declare -a DIR_DIG
DigestFile CUR_DIR DIR_DIG
echo "Новое \"имя\" (контрольная сумма) для ${CUR_DIR[9]} это ${DIR_DIG[0]}"

declare -a DIR_ENT
# BIG_DIR # Для очень больших массивов — используем временный файл на ramdisk
# BIG-DIR # ListDirectory -of "${CUR_DIR[11]}/*" "/tmpfs/junk2"
ListDirectory "${CUR_DIR[11]}/*" DIR_ENT

declare -a DIR_IDX
# BIG-DIR # IndexList -if "/tmpfs/junk2" DIR_IDX
IndexList DIR_ENT DIR_IDX

declare -a IDX_DIG
# BIG-DIR # DIR_ENT=( $(cat /tmpfs/junk2) )
# BIG-DIR # DigestFile -if /tmpfs/junk2 IDX_DIG
DigestFile DIR_ENT IDX_DIG
# Маленькие (должны) будут распараллеленны IndexList & DigestFile
# Большие (должны) будут распараллеленны IndexList & DigestFile & назначение
```

```
echo "\"Имя\" (контрольная сумма) для содержимого ${PWD} это ${IDX_DIG[0]}"

declare -a FILE_LOC
LocateFile ${PWD} FILE_LOC
ListArray FILE_LOC

exit 0
```

Stéphane Chazelas продемонстрировал объектно-ориентированное программирование в сценарии Bash.

Mariusz Gniazdowski предоставил хэш библиотеку для использования в сценариях.

Пример A-20. Библиотека хэш функций

```
# Hash:
# Библиотека хэш функции
# Автор: Mariusz Gniazdowski <mariusz.gn-at-gmail.com>
# Date: 2005-04-07

# Функции создающие эмуляцию хэшей в Bash немного менее болезненны.


#    Ограничения:
#  * Поддерживаются только глобальные переменные.
#  * Каждый экземпляр хэш создает одно значение одной глобальной переменной.
#  * Возможны коллизии имен переменных, если вы
#+   определяете переменную как __hash__hashname_key
#  * Ключи должны использовать символы которые могут быть частью имен
#+   переменных Bash (без тире, точек, и т.д.).
#  * Хэш создается как переменная:
#    ... hashname_keyname
#    Так что если кто-то создаст хэши как:
#      myhash_ + mykey = myhash__mykey
#      myhash + _mykey = myhash__mykey ,
#    то может произойти коллизия.
#    (Это не должно стать серьезной проблемой.)


Hash_config_varname_prefix=__hash__


# Эмуляция:  хэш[ключ]=значение
#
# Параметры:
# 1 - хэш
# 2 - ключ
# 3 - значение
function hash_set {
eval "${Hash_config_varname_prefix}${1}_${2}=\"${3}\""
}

# Эмуляция:  значение=хэш[ключ]
#
# Параметры:
# 1 - хэш
# 2 - ключ
# 3 - значение (имя присвоенное глобальной переменной)

function hash_get_into {
```

```
eval "$3=\"\$${Hash_config_varname_prefix}${1}_${2}\""
}

# Эмуляция:  echo хэш[ключ]
#
# Параметры:
# 1 - хэш
# 2 - ключ
# 3 – параметры echo (-n, к примеру)

function hash_echo {
eval "echo $3 \"\$${Hash_config_varname_prefix}${1}_${2}\""
}

# Эмуляция:  хэш[ключ1]=хэш[ключ2]
#
# Параметры:
# 1 - хэш1
# 2 - ключ1
# 3 - хэш2
# 4 - ключ2

function hash_copy {
eval "${Hash_config_varname_prefix}${1}_${2}\
=\"\$${Hash_config_varname_prefix}${3}_${4}\""
}

# Эмуляция:  хэш[ключN-1]=хэш[ключ2]=...хэш[ключ1]
#
# Копирование первого ключа в остальные ключи.
#
# Параметры:
# 1 - хэш1
# 2 - ключ1
# 3 - ключ2
# . . .
# N - ключN

function hash_dup {
  local hashName="$1" keyName="$2"
  shift 2
  until [ ${#} -le 0 ]; do
    eval "${Hash_config_varname_prefix}${hashName}_${1}\
=\"\$${Hash_config_varname_prefix}${hashName}_${keyName}\""
  shift;
  done;
}

# Эмуляция:  unset хэш[ключ]
#
# Параметры:
# 1 - хэш
# 2 - ключ

function hash_unset {
eval "unset ${Hash_config_varname_prefix}${1}_${2}"
}

# Эмуляция типа:  ref=&хэш[ключ]
#
# Ссылка (ref) – это имя переменной, в которой приводится значение.
```

```
#
# Параметры:
# 1 - хэш
# 2 - ключ
# 3 - ref — Имя присвоенное глобальной переменной.

function hash_get_ref_into {
eval "$3=\"${Hash_config_varname_prefix}${1}_${2}\""
}

# Эмуляция типа:  echo &хэш[ключ]
#
# Ссылка — это имя переменной, в которой приводится значение.
#
# Параметры:
# 1 - хэш
# 2 - ключ
# 3 — параметры echo (-n, например)

function hash_echo_ref {
eval "echo $3 \"${Hash_config_varname_prefix}${1}_${2}\""
}

# Эмуляция типа:  $$хэш[ключ](параметр1, параметр2, ...)
#
# Параметры:
# 1 - хэш
# 2 - ключ
# 3,4, ... - параметры функции

function hash_call {
local hash key
hash=$1
key=$2
shift 2
eval "eval \"\$${Hash_config_varname_prefix}${hash}_${key} \\\"\\\$@\\\"\""
}

# Эмуляция типа:  isset(хэш[ключ]) или хэш[ключ]==NULL
#
# Параметры:
# 1 - хэш
# 2 - ключ
# Возвращаются:
# 0 — такой ключ есть
# 1 — такого ключа нет

function hash_is_set {
  eval "if [[ \"\${${Hash_config_varname_prefix}${1}_${2}-a}\" = \"a\" &&
  \"\${${Hash_config_varname_prefix}${1}_${2}-b}\" = \"b\" ]]
    then return 1; else return 0; fi"
}

# Эмуляция типа:
#   foreach($хэш как $ключ => $значение) { fun($ключ,$значение); }
#
# Можно писать различные вариации для этой функции.
# Здесь с помощью функции она производится «в общей форме».
#
# Параметры:
# 1 - хэш
```

```
# 2 – имя функции

function hash_foreach {
  local keyname oldIFS="$IFS"
  IFS=' '
  for i in $(eval "echo \${!${Hash_config_varname_prefix}${1}_*}"); do
    keyname=$(eval "echo \${i##${Hash_config_varname_prefix}${1}_}")
    eval "$2 $keyname \"\$$i\""
  done
IFS="$oldIFS"
}

#  ПРИМЕЧАНИЕ: В строках 103 и 116, амперсанды изменены.
#  Но, это не имеет значения, потому что это так или иначе строки
закомментированны.
```

Вот пример сценария, использующего вышеописанную библиотеку хэш.

Пример A-21. Раскрашивание текста с помощью хэш-функции

```
#!/bin/bash
# hash-example.sh: Раскрашивание текста.
# Автор: Mariusz Gniazdowski <mariusz.gn-at-gmail.com>

.Hash.lib      # Загружает библиотеку функций.

hash_set colors red          "\033[0;31m"
hash_set colors blue         "\033[0;34m"
hash_set colors light_blue   "\033[1;34m"
hash_set colors light_red    "\033[1;31m"
hash_set colors cyan         "\033[0;36m"
hash_set colors light_green  "\033[1;32m"
hash_set colors light_gray   "\033[0;37m"
hash_set colors green        "\033[0;32m"
hash_set colors yellow       "\033[1;33m"
hash_set colors light_purple "\033[1;35m"
hash_set colors purple       "\033[0;35m"
hash_set colors reset_color  "\033[0;00m"


# $1 - ключ
# $2 - значение
try_colors() {
        echo -en "$2"
        echo "Это строка $1."
}
hash_foreach colors try_colors
hash_echo colors reset_color -en

echo -e '\nLet переводит некоторые цвета в желтый цвет.\n'
# В некоторых терминалах тяжело читать желтый текст.
hash_dup colors yellow   red light_green blue green light_gray cyan
hash_foreach colors try_colors
hash_echo colors reset_color -en

echo -e '\nLet удалим желтый и попробуем другие цвета...\n'

for i in red light_green blue green light_gray cyan; do
```

```
        hash_unset colors $i
done
hash_foreach colors try_colors
hash_echo colors reset_color -en

hash_set other txt "Другие примеры ..."
hash_echo other txt
hash_get_into other txt text
echo $text

hash_set other my_fun try_colors
hash_call other my_fun   purple "`hash_echo colors purple`"
hash_echo colors reset_color -en

echo; echo "Возвращаемся в нормальный режим?"; echo

exit $?

#  На некоторых терминалах «светлые» цвета выводятся жирным шрифтом и в
#  результате  выглядят темнее, чем нормальные.
#  Почему так?
```

Пример, иллюстрирующий механику хэширования, но с другой точки зрения.

Пример A-22. Большие возможности хеш-функций

```
#!/bin/bash
# $Id: ha.sh,v 1.2 2005/04/21 23:24:26 oliver Exp $
# Copyright 2005 Oliver Beckstein
# Released under the GNU Public License
# Автор сценария любезно разрешил использование в ABS Guide.
# (Спасибо!)

#-------------------------------------------------------------------
# псевдо-хэш, основанный на косвенных параметрах расширения
# API: доступ через функции:
#
# Создаем хэш:
#
#      newhash Lovers
#
# добавляем записи (обратите внимание на одинарные кавычки для пробелов)
#
#      addhash Lovers Tristan Isolde
#      addhash Lovers 'Romeo Montague' 'Juliet Capulet'
#
# значение ключа доступа
#
#      gethash Lovers Tristan   ---->  Isolde
#
# показываем все ключи
#
#      keyshash Lovers         ----> 'Tristan'  'Romeo Montague'
#
#
# Соглашение: вместо синтаксиса perl' foo{bar} = boing',
# используем
#       '_foo_bar=boing' (два подчеркивания, без пробелов)
```

```
#
# 1) сохраняем ключи в _NAME_keys[]
# 2) сохраняем значения в _NAME_values[] используя тот же целочисленный индекс.
# Целочисленным индексом последней записи является _NAME_ptr
#
# ПРИМЕЧАНИЕ: Это не ошибка или проверка здравомыслия, просто голый скелет.


function _inihash () {
    # Локальная функция
    # вызываемая в начале каждой процедуры
    # определений: _keys _values _ptr
    #
    # Usage: _inihash NAME
    local name=$1
    _keys=_${name}_keys
    _values=_${name}_values
    _ptr=_${name}_ptr
}

function newhash () {
    # Usage: newhash NAME
    # NAME не содержит пробелов или точек.
    # На самом деле: должна быть переменной Bash с правильным именем.
    # Полагаем, что Bash автоматически распознает массивы.
    local name=$1
    local _keys _values _ptr
    _inihash ${name}
    eval ${_ptr}=0
}


function addhash () {
    # Usage: addhash NAME KEY 'VALUE с пробелами'
    # аргументы с пробелами должны быть заключены в одинарные кавычки ''
    local name=$1 k="$2" v="$3"
    local _keys _values _ptr
    _inihash ${name}

    #echo "DEBUG(addhash): ${_ptr}=${!_ptr}''

    eval let ${_ptr}=${_ptr}+1
    eval "$_keys[${!_ptr}]=\"${k}\''''
    eval "$_values[${!_ptr}]=\"${v}\''''
}


function gethash () {
    #  Usage: gethash NAME KEY
    #  Возвращает ERR=0 если запись существует, в противном случае 1
    #  Это не надлежащий хэш --
    #+ Мы просто ищем построчно через ключи.
    local name=$1 key="$2"
    local _keys _values _ptr
    local k v i found h
    _inihash ${name}

    # _ptr holds наибольший индекс в хэш
    found=0

    for i in $(seq 1 ${!_ptr}); do
```

```
        h="\${${_keys}[${i}]}' '  #  Сохраняем в два этапа,
        eval k=${h}               #+ особенно с пробелами в кавычках.
        if [ "${k}" = "${key}" ]; then found=1; break; fi
    done;

    [ ${found} = 0 ] && return 1;
    # else: i это индекс соответствующий ключу
    h="\${${_values}[${i}]}"
    eval echo "${h}"
    return 0;
}

function keyshash () {
    # Usage: keyshash NAME
    # Возвращает список всех ключей определенных для имен хэш.
    local name=$1 key="$2"
    local _keys _values _ptr
    local k i h
    _inihash ${name}

    # _ptr holds наибольший индекс в хэш
    for i in $(seq 1 ${!_ptr}); do
        h="\${${_keys}[${i}]}"   #  Сохраняем в два этапа,
        eval k=${h}              #+ особенно при пробелах в кавычках.
        echo -n "'${k}' "
    done;
}


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

# Теперь давайте проверим.
# (В комментариях в начале сценария.)
newhash Lovers
addhash Lovers Tristan Isolde
addhash Lovers 'Romeo Montague' 'Juliet Capulet'

# Конечный вывод.
echo
gethash Lovers Tristan      # Isolde
echo
keyshash Lovers             # 'Tristan' 'Romeo Montague'
echo; echo


exit 0

# Упражнение:
# --------

# Добавьте в функцию проверку на ошибки.
```

Теперь сценарий, который устанавливает и монтирует эти милые USB брелки твердотельных «жестких дисков.»

Пример A-23. Монтирование USB оборудования хранения данных

```
#!/bin/bash
```

```
# ==> usb.sh
# ==> Сценарий монтирования и установки USB брелка устройств хранения.
# ==> Запускается как root при запуске системы (см. ниже).
# ==>
# ==> Новые дистрибутивы Linux (2004 или позже) автоопределяют
# ==> и устанавливают USB носители, и не нуждаются в этом сценарии.
# ==> Но, он все еще полезен.

#  This code is free software covered by GNU GPL license version 2 or above.
#  Please refer to http://www.gnu.org/ for the full license text.
#
#  Some code lifted from usb-mount by Michael Hamilton's usb-mount (LGPL)
#+ см. http://users.actrix.co.nz/michael/usbmount.html
#
#  ИНСТАЛЯЦИЯ
#  -------
#  Поместите его в /etc/hotplug/usb/diskonkey.
#  Затем загляните в /etc/hotplug/usb.distmap, и скопируйте все записи usb-
#+ носителей в /etc/hotplug/usb.usermap, заменяя "usb-storage" на "diskonkey".
#  В противном случае этот код выполняется только во время вызова/удаления
#+ модуля ядра(по крайней мере в моих тестах).
#
#  TODO (Что еще надо реализовать)
#  ----
#  Обработка более одного устройства одновременно (т.е. /dev/diskonkey1
#+ и /mnt/diskonkey1). Самой большой проблемой здесь является обработка в
#+ devlabel, которую я еще не пробовал.
#
#  АВТОР и ПОДДЕРЖКА
#  ------------------
#  Konstantin Riabitsev, <icon linux duke edu>.
#  Отправляйте сообщения о любой проблеме на мой адрес электронной почты.
#
# ==> Комментарии добавлены автором ABS Guide.



SYMLINKDEV=/dev/diskonkey
MOUNTPOINT=/mnt/diskonkey
DEVLABEL=/sbin/devlabel
DEVLABELCONFIG=/etc/sysconfig/devlabel
IAM=$0

##
# Функции внесены почти дословно из кода монтирования USB.
#

function allAttachedScsiUsb {
  find /proc/scsi/ -path '/proc/scsi/usb-storage*' -type f |
  xargs grep -l 'Attached: Yes'
}

function scsiDevFromScsiUsb {
  echo $1 | awk -F"[-/]" '{ n=$(NF-1);
  print "/dev/sd" substr("abcdefghijklmnopqrstuvwxyz", n+1, 1) }'
}

if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ]; then
    ##
    # перенесено из кода usbcam.
    #
```

```
if [ -f /var/run/console.lock ]; then
    CONSOLEOWNER=`cat /var/run/console.lock`
elif [ -f /var/lock/console.lock ]; then
    CONSOLEOWNER=`cat /var/lock/console.lock`
else
    CONSOLEOWNER=
fi
for procEntry in $(allAttachedScsiUsb); do
    scsiDev=$(scsiDevFromScsiUsb $procEntry)
    #  Какие-то проблемы с носителем usb?
    #  Разделы не находятся в /proc/partitions/, пока к ним не обращались.
    /sbin/fdisk -l $scsiDev >/dev/null
    ##
    #  Большинство устройств имеют информацию разметки, поэтому данные
    #+ будут на /dev/sd?1. Тем не менее, некоторые из них не имеют никакой
    #+ разметки и используют все устройство для хранения данных. Попытаемся
    #+ угадать, есть ли у нас /dev/sd?1, и если нет, то используется
    #+ весь носитель и надеемся на лучшее.
    #
    if grep -q `basename $scsiDev`1 /proc/partitions; then
        part="$scsiDev""1"
    else
        part=$scsiDev
    fi
    ##
    #  Смена владельца раздела пользователя консоли,что бы
    #+ примонтировать ее.
    #
    if [ ! -z "$CONSOLEOWNER" ]; then
        chown $CONSOLEOWNER:disk $part
    fi
    ##
    # Проверка, есть ли у нас UUID определенного devlabel. Если нет,
    # тогда устройство добавляется в список.
    #
    prodid=`$DEVLABEL printid -d $part`
    if ! grep -q $prodid $DEVLABELCONFIG; then
        # Скрестим наши пальцы и будем надеяться, что это сработает
        $DEVLABEL add -d $part -s $SYMLINKDEV 2>/dev/null
    fi
    ##
    # Проверка, существует ли точка монтирования, если нет — создаем ее.
    #
    if [ ! -e $MOUNTPOINT ]; then
        mkdir -p $MOUNTPOINT
    fi
    ##
    # О легком монтировании позаботится /etc/fstab.
    #
    if ! grep -q "^$SYMLINKDEV" /etc/fstab; then
        # Добавляем запись в fstab
        echo -e \
            "$SYMLINKDEV\t\t$MOUNTPOINT\t\tauto\tnoauto,owner,kudzu 0 0" \
            >> /etc/fstab
    fi
done
if [ ! -z "$REMOVER" ]; then
    ##
    # Убеждаемся, что этот сценарий запускается при удалении устройства.
    #
```

```
        mkdir -p `dirname $REMOVER`
        ln -s $IAM $REMOVER
    fi
elif [ "${ACTION}" = "remove" ]; then
    ##
    # Если устройство примонтировано, то правильно его отключает.
    #
    if grep -q "$MOUNTPOINT" /etc/mtab; then
        # отмонтирование
        umount -l $MOUNTPOINT
    fi
    ##
    # Удаление из /etc/fstab, если отмонтировано.
    #
    if grep -q "^$SYMLINKDEV" /etc/fstab; then
        grep -v "^$SYMLINKDEV" /etc/fstab > /etc/.fstab.new
        mv -f /etc/.fstab.new /etc/fstab
    fi
fi

exit 0
```

Преобразование текстового файла в формат HTML.

Пример A-24. Преобразование в HTML

```
#!/bin/bash
# tohtml.sh [v. 0.2.01, reldate: 04/13/12, a teeny bit less buggy]

# Конвертирование текстового файла в формат HTML.
# Автор: Mendel Cooper
# Лицензия: GPL3
# Usage: sh tohtml.sh < текстовый файл > html файл
# Сценарий может быть легко изменен для изменения исходного и целевого имен.

#    Предположения:
# 1) Абзацы в текстовом файле разделены пустыми строками.
# 2) Образы Jpeg (*.jpg) находятся в поддиректории "images".
#    В целевом файле имя образа заключено в квадратные скобки,
#    например, [image01.jpg].
# 3) Выделенные (курсивом) фразы начинаются с Пробел + Подчеркивание или первый
#+   символ в строке является знаком подчеркивания, а конечный с
#+   Подчеркиванием + Пробел или Подчеркиванием + Конец строки.

#  Установки
FNTSIZE=2        # Мелко-средний размер шрифта
IMGDIR="images"  # Директория Image
# Заголовки
HDR01='<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">'
HDR02='<!-- Конвертирование HTML сценарием ***tohtml.sh*** -->'
HDR03='<!-- автор сценария: M. Leo Cooper <thegrendel.abs@gmail.com> -->'
HDR10='<html>'
HDR11='<head>'
HDR11a='</head>'
HDR12a='<title>'
HDR12b='</title>'
HDR121='<META NAME="GENERATOR" CONTENT="tohtml.sh script">'
HDR13='<body bgcolor="#dddddd">'   # Изменяет цвет фона.
```

```
HDR14a='<font size='
HDR14b='>'
# Окончания
FTR10='</body>'
FTR11='</html>'
# Тэги
BOLD="<b>"
CENTER="<center>"
END_CENTER="</center>"
LF="<br>"


write_headers ()
  {
  echo "$HDR01"
  echo
  echo "$HDR02"
  echo "$HDR03"
  echo
  echo
  echo "$HDR10"
  echo "$HDR11"
  echo "$HDR121"
  echo "$HDR11a"
  echo "$HDR13"
  echo
  echo -n "$HDR14a"
  echo -n "$FNTSIZE"
  echo "$HDR14b"
  echo
  echo "$BOLD"        # Все жирным шрифтом (более читабельным).
  }


process_text ()
  {
  while read line     # Считываем одну строку за раз
  do
    {
    if [ ! "$line" ]  # Пустая строка?
    then              # Значит начинается новый абзац.
      echo
      echo "$LF"      # Вставляем два тэга <br>.
      echo "$LF"
      echo
      continue        # Пропускаем проверку подчеркиваний.
    else              # Иначе ...

      if [[ "$line" =~ \[*jpg\] ]]    # Это графическое изображение?
      then                            # Отбрасываем скобки.
        temp=$( echo "$line" | sed -e 's/\[//' -e 's/\]//' )
        line=""$CENTER" <img src="\"$IMGDIR"/$temp\"> "$END_CENTER" "
                                      # Добавляем тэг image.
                                      # и ставим его по центру.
      fi

    fi


    echo "$line" | grep -q _
    if [ "$?" -eq 0 ]    # Если строка содержит подчеркивания ...
```

```
    then
      # ================================================
      # Преобразуем подчеркнутую фразу в курсив.
      temp=$( echo "$line" |
              sed -e 's/ _/ <i>/' -e 's/_/<\/i> /' |
              sed -e 's/^_/<i>/'  -e 's/_/<\/i>/' )
      #  Только подчеркивания пробела в начале,
      #+ или в начале или в конце строки.
      #  Подчеркивания внутри слова не преобразуются!
      line="$temp"
      # Сценарий исполняется медленно. Как можно оптимизировать?
      # ================================================
    fi



#   echo
    echo "$line"
#   echo
#   Дополнительные пустые строки в созданном тексте не желательны!
    } # Окончание цикла while
  done
  }   # Окончание process_text ()


write_footers ()  # Удаляем тэги.
  {
  echo "$FTR10"
  echo "$FTR11"
  }


# main () {
# =========
write_headers
process_text
write_footers
# =========
#         }

exit $?

#  Упражнение:
#  ---------
#  1) Исправьте: Проверить подчеркивания закрытые запятой или точкой.
#  2) Добавьте проверку на наличие закрытых подчеркиваний во фразах,
#+    которые будут выделены курсивом.
```

Вот то, что согреет сердца вебмастеров: сценарий, который сохраняет логи.

Пример A-25. Сохранение веблогов

```
#!/bin/bash
#  archiveweblogs.sh v1.0

#  Troy Engel <tengel@fluid.com>
#  Слегка изменено автором документа.
#  Используется с разрешения.
```

```
#
#  Этот сценарий сохранит обычно обновляемые и удаляемые вебблоги
#+ из установки по умолчанию RedHat/Apache
#  Будут сохраняться сжатые файлы с отметкой даты и времени в имени файла
#+ в заданную директорию.
#
#  Запускается из crontab ночью, когда все выключено:
#  0 2 * * * /opt/sbin/archiveweblogs.sh


PROBLEM=66

# Присвоение директории для резервного копирования.
BKP_DIR=/opt/backups/weblogs

# По умолчанию наполнение Apache/RedHat
LOG_DAYS="4 3 2 1"
LOG_DIR=/var/log/httpd
LOG_FILES="access_log error_log"

# Размещение программ RedHat по умолчанию
LS=/bin/ls
MV=/bin/mv
ID=/usr/bin/id
CUT=/bin/cut
COL=/usr/bin/column
BZ2=/usr/bin/bzip2

# Мы root?
USER=`$ID -u`
if [ "X$USER" != "X0" ]; then
  echo "PANIC: Только root может запускать этот сценарий!"
  exit $PROBLEM
fi

# Директория бэкапа существует/в нее можно писать?
if [ ! -x $BKP_DIR ]; then
  echo "PANIC: $BKP_DIR не существует или в нее нельзя записать!"
  exit $PROBLEM
fi

# Перемещаемся, переименовываем и сжимаем журналы bzip2
for logday in $LOG_DAYS; do
  for logfile in $LOG_FILES; do
    MYFILE="$LOG_DIR/$logfile.$logday"
    if [ -w $MYFILE ]; then
      DTS=`$LS -lgo --time-style=+%Y%m%d $MYFILE | $COL -t | $CUT -d ' ' -f7`
      $MV $MYFILE $BKP_DIR/$logfile.$DTS
      $BZ2 $BKP_DIR/$logfile.$DTS
    else
      # Выводится только сообщение об ошибке.
      if [ -f $MYFILE ]; then
        echo "ERROR: $MYFILE не может быть записан. Сброшен."
      fi
    fi
  done
done

exit 0
```

Как предохранить оболочку от расширения и интерпретации текстовых строк.

Пример A-26. Защита буквенных строк

```
#! /bin/bash
# protect_literal.sh

# set -vx

:<<-'_Protect_Literal_String_Doc'

    Copyright (c) Michael S. Zick, 2003; All Rights Reserved
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None
    Revision: $ID$

    Документация перенаправлена на не производимые в Bash операции.
    При первом прочтении этот блок сценария в Bash будет '/dev/null'.
    (Раскомментируйте команду выше, чтобы увидеть это.)

    Удалите первую строку (Sha-Bang) которая является источником библиотечных
    процедур. Также закомментируйте пример использования кода в двух местах,
    где указано.


    Применение:
        _protect_literal_str 'Ваши ${фантазии} удовлетворяются независимо
        от строки'
        Просто выведите аргумент на стандартный вывод, заключив в жесткие
        кавычки.

        $(_protect_literal_str 'Ваши ${фантазии} удовлетворяются независимо
        от строки') в правой стороне от оператора присваивания.

    Как:
        Правая сторона присваивания сохраняется в жестких кавычках,
        защищая буквальное содержимое во время присваивания.

    Примечания:
        Странные имена (_*) используются для избежания уничтожения выбранных
        пользовательских имен, когда они выступают как источник библиотек.

_Protect_Literal_String_Doc

# Форма функции 'для иллюстрации'

_protect_literal_str() {

# Выбирает неиспользуемый, непечатаемый символ как локальный IFS.
# Не требуется, но показывает, что мы игнорируем это.
    local IFS=$'\x1B'               # Символ \ESC

# Во время присваивания заключите все элементы в жесткие кавычки.
    local tmp=$'\x27'$@$'\x27'
#    local tmp=$'\''$@$'\''         # Еще уродливей.

    local len=${#tmp}               # Только информация
    echo $tmp это длинный $len.     # Вывод И информация
}
```

```
# Версия с укороченным именем.
_pls() {
    local IFS=$'x1B'                # Символ \ESC (не требуется)
    echo $'\x27'$@$'\x27'           # Глобальный параметр в жестких кавычках
}

# :<<-'_Protect_Literal_String_Test'
# # # Удаляем "# " выше для удаления этого кода. # # #

# Посмотрите, как это выглядит при выводе.
echo
echo "- - Первая проверка - -"
_protect_literal_str 'Привет $user'
_protect_literal_str 'Привет "${username}"'
echo

# Что выводится:
# - - Первая проверка - -
# 'Привет $user' величиной 14 символов.
# 'Привет "${username}"' величиной 22 символа.

#  Выглядит, как ожидалось, но почему-то проблемно?
#  Разница скрыта во внутреннем порядке операций в Bash
#  происходящем при присваивании RHS.

# Объявляем массив для проверяемых значений.
declare -a arrayZ

# Назначаем элементы с различными типами кавычек и экранированием.
arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )

# Теперь список массива и посмотрим, что там.
echo "- - Вторая проверка - -"
for (( i=0 ; i<${#arrayZ[*]} ; i++ ))
do
    echo  Элемент $i: ${arrayZ[$i]} длинной: ${#arrayZ[$i]}.
done
echo

# Что выводится:
# - - Вторая проверка - -
# Элемент 0: zero длиной: 4.           # Элемент образец
# Элемент 1: 'Hello ${Me}' длиной: 13. # Наш "$(_pls '...' )"
# Элемент 2: Hello ${You} длиной: 12.  # Кавычки отсутствуют
# Элемент 3: \'Pass: \' длиной: 10 .    # ${pw} ничем не расширено

# Теперь присвойте этот результат.
declare -a array2=( ${arrayZ[@]} )

# И выведите что получилось.
echo "- - Третья проверка - -"
for (( i=0 ; i<${#array2[*]} ; i++ ))
do
    echo  Элемент $i: ${array2[$i]} длиной: ${#array2[$i]}.
done
echo

# Что выводится:
# - - Третья проверка - -
# Элемент 0: zero длиной: 4.           # Элемент образец.
```

```
# Элемент 1: Hello ${Me} длиной: 11.   # Предполагаемый результат.
# Элемент 2: Hello длиной: 5.          # ${You} ничем не расширено.
# Элемент 3: 'Pass: длиной: 6.         # Разделяется пробелом.
# Элемент 4: ' длиной: 1.              # Здесь конечная кавычка.

#  Наш Element 1, заключенный в жесткие кавычки, трактуется буквально.
#  Хотя это и не видно, начальные и конечные пробелы также трактуются
#+ буквально.
#  Теперь, когда содержимое строки установлено, Bash всегда будет выполнять
#+ содержимое жестких кавычек, как требует процесс выполнения.
#  Почему?
#  Рассмотрим нашу конструкцию "$(_pls 'Hello ${Me}')":
#  " ... " -> Требуемое расширение, выделено кавычками.
#  $( ... ) -> Помещаем результат..., выделяем.
#  _pls ' ... ' -> вызывается с буквальными аргументами, выделенными кавычками.
#  Результат возвращает содержимое в жестких кавычках; НО действия выше уже
#  были сделаны, поэтому они становятся частью присвоенного значения.
#  Точно так же, при дальнейшем использовании, строковая переменная ${Me}
#+ является частью содержимого (результата) и выдерживает любые действия
#  (До тех пор, пока строка явно оценивается).

#  Подсказка: Посмотри что получится при замене жестких кавычек ($'\x27')
#+ на мягкие кавычки в действиях выше ($'\x22').
#  Интересно также удаление увеличения любого содержимого в кавычках.

# _Protect_Literal_String_Test
# # # Удалите "#" выше для отключения этого кода. # # #

exit 0
```

Но что делать, если вы хотите расширять и интерпретировать строки в оболочке?

Пример A-27. Снятие защиты буквальности строки

```
#! /bin/bash
# unprotect_literal.sh

# set -vx

:<<-'_UnProtect_Literal_String_Doc'

    Copyright (c) Michael S. Zick, 2003; All Rights Reserved
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None
    Revision: $ID$

    Документация перенаправлена на не производимые в Bash операции.
    При первом прочтении этот блок сценария в Bash будет '/dev/null'.
    (Раскомментируйте команду выше, чтобы увидеть это.)

    Удалите первую строку (Sha-Bang) которая является источником библиотечных
    процедур. Также закомментируйте пример использования кода в двух местах,
    где указано.


    Применение:
        Дополнение к функции "$(_pls 'Literal String')".
        (См. пример protect_literal.sh.)
```

```
        StringVar=$(_upls ProtectedSringVariable)

    Действие:
        Примененяется с права от оператора присваивания;
        производит встраивание подстановки в защищенную строку.

    Примечания:
        Странные имена (_*) используются для избежания уничтожения выбранных
        пользовательских имен, когда они выступают как источник библиотек.


_UnProtect_Literal_String_Doc

_upls() {
    local IFS=$'x1B'                # Символ \ESC (не требуется)
    eval echo $@                    # Подстановка шаблона.
}

# :<<-'_UnProtect_Literal_String_Test'
# # # Удалите "# " выше для удаления этого кода. # # #


_pls() {
    local IFS=$'x1B'                # Символ \ESC (не требуется)
    echo $'\x27'$@$'\x27'           # Шаблоны параметра в жестких кавычках
}

# Объявляем массив для проверки значений.
declare -a arrayZ

# Присваиваем элементам различные типы кавычек и  экранирований.
arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )

# Теперь присвоим результаты.
declare -a array2=( ${arrayZ[@]} )

# Что присваиваем:
# - - Третья проверка - -
# Элемент 0: zero длинной: 4            # Образец.
# Элемент 1: Hello ${Me} длинной: 11    # Предполагаемый результат .
# Элемент 2: Hello длинной: 5           # ${You} ничем не расширяется.
# Элемент 3: 'Pass: длинной: 6          # Разделяющий пробел.
# Элемент 4: ' длиной: 1               # Заключительная кавычка.

# set -vx

#  Объявляем какое-то 'Me' для встраивания подстановки ${Me}.
#  ТОЛЬКО необходимо это сделать ДО оценки защищенной строки.
#  (Именно поэтому она была изначально защищена.)

Me="to the array guy."

# Присваиваем назначенную строковую переменную в результат.
newVar=$(_upls ${array2[1]})

# Смотрим содержимое.
echo $newVar

# Для этого действительно необходима функция?
newerVar=$(eval echo ${array2[1]})
```

```
echo $newerVar

#  Я не знаю, но функция _upls предоставляет нам место для
#+ размещения документации.
#  Это помогает, когда мы забываем что означает конструкция вроде:
#+ $(eval echo ... ).

# Что делать, если Me не присваивается, при оценке защищенной строки?
unset Me
newestVar=$(_upls ${array2[1]})
echo $newestVar

# Только действие, без подсказок, без запусков, без ошибок.

#  Почему?
#  Присвоение содержимому строковой переменной, содержащей последовательности
#+ символов, которые имеют смысл в Bash, является общей проблемой в написании
#+ сценария.
#
#  Эта проблема теперь решена в восьми строках кода
#+ (и четырех страницах описания).

#  Где все это применяется?
#  Динамическое содержимое Web страниц, как массив строк Bash.
#  Содержит присвоение запроса команды Bash 'eval'
#+ для временного сохранения страницы.
#  Не предназначено для замены PHP, просто интересно было это осуществить.
###
#  Нет приложения webserver?
#  Нет проблем, проверьте директорию example источника Bash;
#+ для этого там есть сценарий Bash.

# _UnProtect_Literal_String_Test
# # # Удалите "#" выше для удаления этого кода. # # #

exit 0
```

Этот интересный сценарий помогает выследить спамеров.

Пример A-28. Определение спамеров

```
#!/bin/bash

# $Id: is_spammer.bash,v 1.12.2.11 2004/10/01 21:42:33 mszick Exp $
# Строка выше является информацией RCS.

# Последняя версия этого сценария доступна на http://www.morethan.org.
#
# Идентификация спамеров
# Написан Michael S. Zick
# Используется в ABS Guide с разрешения.



###########################################################
# Документация
# См. Так же "Quickstart" в конце сценария.
###########################################################
```

```
:<<-'__is_spammer_Doc_'

    Copyright (c) Michael S. Zick, 2004
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None -{Its a script; the user is on their own.}-

Не терпится?
    Код приложения: переход к коду "# # # Hunt the Spammer' # # #"
    Пример вывода: ":<<-'_is_spammer_outputs_'"
    Использование: Введите название сценария без аргументов.
                Или перейдите к "Quickstart" в конце сценария.

Обеспечение
    При вводе задается доменное имя или адрес IP(v4):

    Исчерпывающий набор запросов, чтобы найти связанные
    сетевые ресурсы (короткие рекурсии в TLD).

    Проверяет адреса IP(v4), проверяя черный список имен серверов.

    Если будет обнаружен в черном списке IP(v4) адресов,
    текстовое сообщение заносится в черный список.
    (Обычно специфичное сообщение гиперссылки.)

Требования
    Работающее соединение Интернет.
    (Упражнение: Добавьте проверку/или сброс при отсутствии соединения при
    запуске сценария.)
    Bash с массивами (2.05b+).

    Внешняя программа 'dig' --
    программная утилита снабженная набором «связанных» программ.
    В частности, версия, являющаяся частью Bind серии 9.x
    См.: http://www.isc.org

    Все применение 'dig' ограничены функциями обертки,
    которая может быть переписана под свои требования.
    См.: dig_wrappers.bash о деталях.
         ("Дополнительная документация" -- ниже)

Использование
    Сценарию требуется единственный аргумент, которым может быть:
    1) Доменное имя;
    2) IP(v4) адрес;
    3) Файл, с одним именем или адресом в каждой строке.

    Сценарию дополнительно доступен и второй аргумент, которым может быть:
    1) Название сервера Черного списка;
    2) Файл, с одним именем сервера Blacklist в каждой строке.

    Если второй аргумент отсутствует, сценарий использует
    свой список (свободный) серверов Blacklist.

    См. Так же, Quickstart в конце этого сценария (после 'exit').

Возвращаемые коды
    0 - Все OK
    1 — Сбой сценария
    2 — Что то занесено в Blacklist
```

```
Дополнительные переменные среды
    SPAMMER_TRACE
        Если задан файл для записи,
        сценарий будет вести журнал  выполнения потока трассировки.

    SPAMMER_DATA
        Если задан файл для записи, сценарий будет собирать
        обнаруженные данные в виде файла GraphViz
        См.: http://www.research.att.com/sw/tools/graphviz

    SPAMMER_LIMIT
        Ограничение глубины трассировки.

        По умолчанию это 2 уровня.

        Установленный 0 (ноль) подразумевает 'не ограниченно'...
        Внимание: Сценарий может перебирать весь Internet!

        Ограничения 1 или 2 чаще используются при применении
        файла доменных имен и адресов.
        Высший предел может быть полезен при охоте на спамерские банды.


Дополнительная документация
    Загрузите архив сценариев
    объясняющих и иллюстрирующих функции содержащиеся в этом сценарии.
    http://bash.deta.in/mszick_clf.tar.bz2


Примечания для изучающих
    В этом сценарии используется большое количество функций.
    Почти все общие функции имеют свой собственный пример сценария.
    Каждый из примеров сценария имеют комментарии уровня учебника.

Проект развития сценария
    Добавьте поддержку для IP(v6) адресов.
    Адреса IP(v6) признаются, но не обрабатываются.

Расширение проекта
    Добавьте обратный просмотр деталей обнаруженной информации.

    Сообщение о соединениях и злоупотребляющих контактов.

    Измените вывод файла GraphViz для включения в него
    вновь обнаруженной информации.

__is_spammer_Doc_

############################################################




#### Специальные установки IFS, используемые для разбора строк. ####

# Пробел == :Пробел:Табуляция:Новая строка:Перевод каретки:
WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'

# Без пробела == Новая строка:Перевод каретки
NO_WSP=$'\x0A'$'\x0D'
```

```
# Поля десятичного представления IP адреса разделены точкой
ADR_IFS=${NO_WSP}'.'

# Массив для преобразования строки из точек
DOT_IFS='.'${WSP_IFS}

# # # Ожидаемые операции со стеком машины # # #
# Этот набор функций описан в func_stack.bash.
# (См. "Дополнительная документация" выше.)
# # #

# Основной стек отложенных операций.
declare -f -a _pending_
# Основное сохранение для запущенных стеков
declare -i _p_ctrl_
# Основной владелец текущей выполняемой функции
declare -f _pend_current_

# # # Только для отладки — при обычном использовании удаляется # # #
#
# Функция сохранения в _pend_hook_ вызываемая
# непосредственно перед каждым ожиданием функции
# - вычисляется.  Стек очищается, _pend_current_ set.
#
# Это штуковина показана в pend_hook.bash.
declare -f _pend_hook_
# # #

# Ничего не делающая функция
pend_dummy() { : ; }

# Очистка и инициализация стека функции.
pend_init() {
    unset _pending_[@]
    pend_func pend_stop_mark
    _pend_hook_='pend_dummy'  # Только для отладки.
}

# Отказ от верхней функции в стеке.
pend_pop() {
    if [ ${#_pending_[@]} -gt 0 ]
    then
        local -i _top_
        _top_=${#_pending_[@]}-1
        unset _pending_[$_top_]
    fi
}

# pend_func function_name [$(printf '%q\n' arguments)]
pend_func() {
    local IFS=${NO_WSP}
    set -f
    _pending_[${#_pending_[@]}]=$@
    set +f
}

# Функция, которая останавливает выпуск:
pend_stop_mark() {
    _p_ctrl_=0
}
```

```
pend_mark() {
    pend_func pend_stop_mark
}

# Выполнение функций до 'pend_stop_mark' . . .
pend_release() {
    local -i _top_             # Объявляем _top_ целым числом.
    _p_ctrl_=${#_pending_[@]}
    while [ ${_p_ctrl_} -gt 0 ]
    do
       _top_=${#_pending_[@]}-1
       _pend_current_=${_pending_[$_top_]}
       unset _pending_[$_top_]
       $_pend_hook_            # Только для отладки.
       eval $_pend_current_
    done
}

# Опускание функций до 'pend_stop_mark' . . .
pend_drop() {
    local -i _top_
    local _pd_ctrl_=${#_pending_[@]}
    while [ ${_pd_ctrl_} -gt 0 ]
    do
       _top_=$_pd_ctrl_-1
       if [ "${_pending_[$_top_]}" == 'pend_stop_mark' ]
       then
           unset _pending_[$_top_]
           break
       else
           unset _pending_[$_top_]
           _pd_ctrl_=$_top_
       fi
    done
    if [ ${#_pending_[@]} -eq 0 ]
    then
        pend_func pend_stop_mark
    fi
}

#### Редактирование массива ####

# Эта функция описана в edit_exact.bash.
# (См. "Дополнительная документация," выше.)

# edit_exact <имя_исходного_массива> <имя_целевого_массива>
edit_exact() {
    [ $# -eq 2 ] ||
    [ $# -eq 3 ] || return 1
    local -a _ee_Excludes
    local -a _ee_Target
    local _ee_x
    local _ee_t
    local IFS=${NO_WSP}
    set -f
    eval _ee_Excludes=\( \$\{$1\[@\]\} \)
    eval _ee_Target=\( \$\{$2\[@\]\} \)
    local _ee_len=${#_ee_Target[@]}     # Исходный размер.
    local _ee_cnt=${#_ee_Excludes[@]}   # Размер исключенного списка.
    [ ${_ee_len} -ne 0 ] || return 0    # Нулевой размер не редактируется.
    [ ${_ee_cnt} -ne 0 ] || return 0    # Нулевой размер не редактируется.
```

```
    for (( x = 0; x < ${_ee_cnt} ; x++ ))
    do
        _ee_x=${_ee_Excludes[$x]}
        for (( n = 0 ; n < ${_ee_len} ; n++ ))
        do
            _ee_t=${_ee_Target[$n]}
            if [ x"${_ee_t}" == x"${_ee_x}" ]
            then
                unset _ee_Target[$n]     # Отменяем сравнение.
                [ $# -eq 2 ] && break    # Если 2 аргумента, то делаем.
            fi
        done
    done
    eval $2=\( \$\{_ee_Target\[@\]\} \)
    set +f
    return 0
}

# Эта функция описана в edit_by_glob.bash.
# edit_by_glob <excludes_array_name> <target_array_name>
edit_by_glob() {
    [ $# -eq 2 ] ||
    [ $# -eq 3 ] || return 1
    local -a _ebg_Excludes
    local -a _ebg_Target
    local _ebg_x
    local _ebg_t
    local IFS=${NO_WSP}
    set -f
    eval _ebg_Excludes=\( \$\{$1\[@\]\} \)
    eval _ebg_Target=\( \$\{$2\[@\]\} \)
    local _ebg_len=${#_ebg_Target[@]}
    local _ebg_cnt=${#_ebg_Excludes[@]}
    [ ${_ebg_len} -ne 0 ] || return 0
    [ ${_ebg_cnt} -ne 0 ] || return 0
    for (( x = 0; x < ${_ebg_cnt} ; x++ ))
    do
        _ebg_x=${_ebg_Excludes[$x]}
        for (( n = 0 ; n < ${_ebg_len} ; n++ ))
        do
            [ $# -eq 3 ] && _ebg_x=${_ebg_x}'*'  #  Редактируем префикс
            if [ ${_ebg_Target[$n]:=} ]          #+ если записан & присвоен.
            then
                _ebg_t=${_ebg_Target[$n]/#${_ebg_x}/}
                [ ${#_ebg_t} -eq 0 ] && unset _ebg_Target[$n]
            fi
        done
    done
    eval $2=\( \$\{_ebg_Target\[@\]\} \)
    set +f
    return 0
}

# Эта функция описана unique_lines.bash.
# unique_lines <in_name> <out_name>
unique_lines() {
    [ $# -eq 2 ] || return 1
    local -a _ul_in
    local -a _ul_out
    local -i _ul_cnt
    local -i _ul_pos
```

```
    local _ul_tmp
    local IFS=${NO_WSP}
    set -f
    eval _ul_in=\( \$\{$1\[@\]\} \)
    _ul_cnt=${#_ul_in[@]}
    for (( _ul_pos = 0 ; _ul_pos < ${_ul_cnt} ; _ul_pos++ ))
    do
        if [ ${_ul_in[${_ul_pos}]:=} ]      # Если присвоен & не записан
        then
            _ul_tmp=${_ul_in[${_ul_pos}]}
            _ul_out[${#_ul_out[@]}]=${_ul_tmp}
            for (( zap = _ul_pos ; zap < ${_ul_cnt} ; zap++ ))
            do
                [ ${_ul_in[${zap}]:=} ] &&
                [ 'x'${_ul_in[${zap}]} == 'x'${_ul_tmp} ] &&
                    unset _ul_in[${zap}]
            done
        fi
    done
    eval $2=\( \$\{_ul_out\[@\]\} \)
    set +f
    return 0
}

# Эта функция описана в char_convert.bash.
# to_lower <строка>
to_lower() {
    [ $# -eq 1 ] || return 1
    local _tl_out
    _tl_out=${1//A/a}
    _tl_out=${_tl_out//B/b}
    _tl_out=${_tl_out//C/c}
    _tl_out=${_tl_out//D/d}
    _tl_out=${_tl_out//E/e}
    _tl_out=${_tl_out//F/f}
    _tl_out=${_tl_out//G/g}
    _tl_out=${_tl_out//H/h}
    _tl_out=${_tl_out//I/i}
    _tl_out=${_tl_out//J/j}
    _tl_out=${_tl_out//K/k}
    _tl_out=${_tl_out//L/l}
    _tl_out=${_tl_out//M/m}
    _tl_out=${_tl_out//N/n}
    _tl_out=${_tl_out//O/o}
    _tl_out=${_tl_out//P/p}
    _tl_out=${_tl_out//Q/q}
    _tl_out=${_tl_out//R/r}
    _tl_out=${_tl_out//S/s}
    _tl_out=${_tl_out//T/t}
    _tl_out=${_tl_out//U/u}
    _tl_out=${_tl_out//V/v}
    _tl_out=${_tl_out//W/w}
    _tl_out=${_tl_out//X/x}
    _tl_out=${_tl_out//Y/y}
    _tl_out=${_tl_out//Z/z}
    echo ${_tl_out}
    return 0
}

#### Приложение вспомогательных функций ####
```

```
# Не каждая функция использует точки в качестве разделителей (например APNIC).
# Эта функция описана в to_dot.bash
# to_dot <строка>
to_dot() {
    [ $# -eq 1 ] || return 1
    echo ${1//[#|@|%]/.}
    return 0
}

# Эта функция описана в is_number.bash.
# is_number <input>
is_number() {
    [ "$#" -eq 1 ]    || return 1  # пустая?
    [ x"$1" == 'x0' ] && return 0  # ноль?
    local -i tst
    let tst=$1 2>/dev/null         # значит это числа!
    return $?
}

# Эта функция описывается в is_address.bash.
# is_address <ввод>
is_address() {
    [ $# -eq 1 ] || return 1    # Пусто ==> ложно
    local -a _ia_input
    local IFS=${ADR_IFS}
    _ia_input=( $1 )
    if  [ ${#_ia_input[@]} -eq 4 ]  &&
        is_number ${_ia_input[0]}   &&
        is_number ${_ia_input[1]}   &&
        is_number ${_ia_input[2]}   &&
        is_number ${_ia_input[3]}   &&
        [ ${_ia_input[0]} -lt 256 ] &&
        [ ${_ia_input[1]} -lt 256 ] &&
        [ ${_ia_input[2]} -lt 256 ] &&
        [ ${_ia_input[3]} -lt 256 ]
    then
        return 0
    else
        return 1
    fi
}

#  Эта функция описывается в split_ip.bash.
#  split_ip <IP_address>
#+ <array_name_norm> [<array_name_rev>]
split_ip() {
    [ $# -eq 3 ] ||              #  Или три
    [ $# -eq 2 ] || return 1     #+ или два аргумента
    local -a _si_input
    local IFS=${ADR_IFS}
    _si_input=( $1 )
    IFS=${WSP_IFS}
    eval $2=\(\ \$\{_si_input\[@\]\}\ \)
    if [ $# -eq 3 ]
    then
        # Вставляем запрос в массив.
        local -a _dns_ip
        _dns_ip[0]=${_si_input[3]}
        _dns_ip[1]=${_si_input[2]}
        _dns_ip[2]=${_si_input[1]}
        _dns_ip[3]=${_si_input[0]}
```

```
        eval $3=\(\ \$\{_dns_ip\[@\]\}\ \)
    fi
    return 0
}

# Эта функция описывается в dot_array.bash.
# dot_array <array_name>
dot_array() {
    [ $# -eq 1 ] || return 1     # Требуется единственный аргумент.
    local -a _da_input
    eval _da_input=\(\ \$\{$1\[@\]\}\ \)
    local IFS=${DOT_IFS}
    local _da_output=${_da_input[@]}
    IFS=${WSP_IFS}
    echo ${_da_output}
    return 0
}

# Эта функция описана в file_to_array.bash
# file_to_array <file_name> <line_array_name>
file_to_array() {
    [ $# -eq 2 ] || return 1  # Требуется два аргумента.
    local IFS=${NO_WSP}
    local -a _fta_tmp_
    _fta_tmp_=( $(cat $1) )
    eval $2=\( \$\{_fta_tmp_\[@\]\} \)
    return 0
}

#  Вывод массива колонками в виде многополевых строк.
#  col_print <array_name> <min_space> <tab_stop [tab_stops]>
col_print() {
    [ $# -gt 2 ] || return 0
    local -a _cp_inp
    local -a _cp_spc
    local -a _cp_line
    local _cp_min
    local _cp_mcnt
    local _cp_pos
    local _cp_cnt
    local _cp_tab
    local -i _cp
    local -i _cpf
    local _cp_fld
    # ВНИМАНИЕ: СЛЕДУЮЩАЯ СТРОКА НЕ ПУСТАЯ — ЭТО ПРОБЕЛ ЗАКЛЮЧЕННЫЙ В КАВЫЧКИ.
    local _cp_max=' '
    set -f
    local IFS=${NO_WSP}
    eval _cp_inp=\(\ \$\{$1\[@\]\}\ \)
    [ ${#_cp_inp[@]} -gt 0 ] || return 0 # Пусто это легко.
    _cp_mcnt=$2
    _cp_min=${_cp_max:1:${_cp_mcnt}}
    shift
    shift
    _cp_cnt=$#
    for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
    do
        _cp_spc[${#_cp_spc[@]}]="${_cp_max:2:$1}" #"
        shift
    done
    _cp_cnt=${#_cp_inp[@]}
```

```
    for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
    do
        _cp_pos=1
        IFS=${NO_WSP}$'\x20'
        _cp_line=( ${_cp_inp[${_cp}]} )
        IFS=${NO_WSP}
        for (( _cpf = 0 ; _cpf < ${#_cp_line[@]} ; _cpf++ ))
        do
            _cp_tab=${_cp_spc[${_cpf}]:${_cp_pos}}
            if [ ${#_cp_tab} -lt ${_cp_mcnt} ]
            then
                _cp_tab="${_cp_min}"
            fi
            echo -n "${_cp_tab}"
            (( _cp_pos = ${_cp_pos} + ${#_cp_tab} ))
            _cp_fld="${_cp_line[${_cpf}]}"
            echo -n ${_cp_fld}
            (( _cp_pos = ${_cp_pos} + ${#_cp_fld} ))
        done
        echo
    done
    set +f
    return 0
}

# # # # Поток данных 'Hunt the Spammer' # # # #

# Код возврата приложения
declare -i _hs_RC

# Ввод оригинала, из которого удалены IP-адреса
# После чего проверяется доменное имя.
declare -a uc_name

# Ввод оригинала IP адресов перемещенных сюда
# После чего проверяется IP адрес
declare -a uc_address

# Имена, против которых запущено расширение адресов
# Готово для подробного поиска имен
declare -a chk_name

# Адреса, против которых запущено расширение имен
# Готово для подробного поиска адресов
declare -a chk_address

#  Рекурсия — начальная глубина имени.
#  expand_input_address поддерживает это листинг
#+ для предотвращения повторного поиска адресов во время
#+ рекурсии доменного имени.
declare -a been_there_addr
been_there_addr=( '127.0.0.1' ) # Обычный localhost

# Имена, которые проверены(или заданы)
declare -a known_name

# Адреса, котрые проверены (или заданы)
declare -a known_address

#  Список нуля или более серверов Blacklist для проверки.
#  Каждый 'известный адрес' будет проверен для каждого сервера,
```

```
#+ с отрицательными ответами и подавлением неудачных проверок.
declare -a list_server

# Косвенный предел — установлен в ноль == без предела
indirect=${SPAMMER_LIMIT:=2}

# # # # Информация выходных данных 'Hunt the Spammer' # # # #

# Каждое доменное имя может иметь несколько IP адресов.
# Каждый IP адрес может иметь несколько доменных имен.
# Поэтому отслеживается уникальная пара адрес-имя.
declare -a known_pair
declare -a reverse_pair

#  В дополнение к переменным потока данных; known_address
#+ known_name и list_server, следует вывод в
#+ файл внешнего графического интерфейса.

# Управляющая связь, родитель -> поля SOA.
declare -a auth_chain

# Ссылочная связь, имя родителя -> имя потомка
declare -a ref_chain

# Связь DNS — доменное имя -> адрес
declare -a name_address

# Пары имя и служба — доменное имя -> служба
declare -a name_srvc

# Пары имя и ресурсы — доменное имя -> запись ресурса
declare -a name_resource

# Пары родитель и потомок — имя родителя -> имя потомка
# Это  МОЖЕТ НЕ быть таким же, как в ref_chain!
declare -a parent_child

# Пары адрес и попавшие в Blacklist - адрес->сервер
declare -a address_hits

# Дамп файла данных интерфейса
declare -f _dot_dump
_dot_dump=pend_dummy   # Изначально no-op

#  Дамп данных это включение присваивания переменной окружения SPAMMER_DATA
#+ имени записываемого файла.
declare _dot_file

# Вспомогательная функция для функции dump-to-dot-file
# dump_to_dot <array_name> <prefix>
dump_to_dot() {
    local -a _dda_tmp
    local -i _dda_cnt
    local _dda_form='    '${2}'%04u %s\n'
    local IFS=${NO_WSP}
    eval _dda_tmp=\(\ \$\{$1\[@\]\}\ \)
    _dda_cnt=${#_dda_tmp[@]}
    if [ ${_dda_cnt} -gt 0 ]
    then
        for (( _dda = 0 ; _dda < _dda_cnt ; _dda++ ))
        do
```

```
            printf "${_dda_form}" \
                   "${_dda}" "${_dda_tmp[${_dda}]}" >>${_dot_file}
        done
    fi
}

# Которая так же присваивает _dot_dump в эту функцию ...
dump_dot() {
    local -i _dd_cnt
    echo '# Устаревшие данные: '$(date -R) >${_dot_file}
    echo '# ABS Guide: is_spammer.bash; v2, 2004-msz' >>${_dot_file}
    echo >>${_dot_file}
    echo 'digraph G {' >>${_dot_file}

    if [ ${#known_name[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Известные узлы доменных имен' >>${_dot_file}
        _dd_cnt=${#known_name[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    N%04u [label="%s"] ;\n' \
                   "${_dd}" "${known_name[${_dd}]}" >>${_dot_file}
        done
    fi

    if [ ${#known_address[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Известные узлы адресов' >>${_dot_file}
        _dd_cnt=${#known_address[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    A%04u [label="%s"] ;\n' \
                   "${_dd}" "${known_address[${_dd}]}" >>${_dot_file}
        done
    fi

    echo                                    >>${_dot_file}
    echo '/*'                               >>${_dot_file}
    echo ' * Известные отношения :: пользователя преобразуется'  >>${_dot_file}
    echo ' * Нужна  ручная или программная графическая форма.'  >>${_dot_file}
    echo ' *'                               >>${_dot_file}

    if [ ${#auth_chain[@]} -gt 0 ]
    then
      echo >>${_dot_file}
      echo '# Значение поддерживаемой ссылки & исходное поле.' >>${_dot_file}
        dump_to_dot auth_chain AC
    fi

    if [ ${#ref_chain[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Имя поддерживаемой ссылки и исходного поля.' >>${_dot_file}
        dump_to_dot ref_chain RC
    fi

    if [ ${#name_address[@]} -gt 0 ]
    then
        echo >>${_dot_file}
```

```
        echo '# Known name->address edges' >>${_dot_file}
        dump_to_dot name_address NA
    fi

    if [ ${#name_srvc[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->service edges' >>${_dot_file}
        dump_to_dot name_srvc NS
    fi

    if [ ${#name_resource[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->resource edges' >>${_dot_file}
        dump_to_dot name_resource NR
    fi

    if [ ${#parent_child[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known parent->child edges' >>${_dot_file}
        dump_to_dot parent_child PC
    fi

    if [ ${#list_server[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Известные узлы Blacklist' >>${_dot_file}
        _dd_cnt=${#list_server[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    LS%04u [label="%s"] ;\n' \
                   "${_dd}" "${list_server[${_dd}]}" >>${_dot_file}
        done
    fi

    unique_lines address_hits address_hits
    if [ ${#address_hits[@]} -gt 0 ]
    then
      echo >>${_dot_file}
      echo '# Known address->Blacklist_hit edges' >>${_dot_file}
      echo '# ВНИМАНИЕ:  предупреждения могут вызвать ложные значения.' >>$
{_dot_file}
       dump_to_dot address_hits AH
    fi
    echo          >>${_dot_file}
    echo ' *'     >>${_dot_file}
    echo ' * Это масса отношений. Удачное представление.' >>${_dot_file}
    echo ' */'    >>${_dot_file}
    echo '}'      >>${_dot_file}
    return 0
}

# # # # Поток выполнения 'Hunt the Spammer' # # # #

#  Выполнение трассировки включается присвоением переменной
#+ среды SPAMMER_TRACE имени файла для записи.
declare -a _trace_log
declare _log_file
```

```
# Функция для заполнения журнала трассировки.
trace_logger() {
    _trace_log[${#_trace_log[@]}]=${_pend_current_}
}

# Дамп журнала трассировки в файле функции переменной.
declare -f _log_dump
_log_dump=pend_dummy   # Изначально no-op.

# Дамп журнала трассировки в файле.
dump_log() {
    local -i _dl_cnt
    _dl_cnt=${#_trace_log[@]}
    for (( _dl = 0 ; _dl < _dl_cnt ; _dl++ ))
    do
        echo ${_trace_log[${_dl}]} >> ${_log_file}
    done
    _dl_cnt=${#_pending_[@]}
    if [ ${_dl_cnt} -gt 0 ]
    then
        _dl_cnt=${_dl_cnt}-1
        echo '# # # Стек операций не пустой # # #' >> ${_log_file}
        for (( _dl = ${_dl_cnt} ; _dl >= 0 ; _dl-- ))
        do
            echo ${_pending_[${_dl}]} >> ${_log_file}
        done
    fi
}

# # # Служебная программа оберток «dig» # # #
#
#  Эти обертки приводятся из примеров, показанных в dig_wrappers.bash.
#
#  Основное отличие заключается в возвращении ими своих результатов
#+ в массив в виде списка.
#
#  Детали см. dig_wrappers.bash.
#+ С помощью этого сценария разрабатываются любые изменения.
#
# # #

# Короткая форма ответа: анализ ответа 'dig'.

# Поиск вперед :: Имя -> Адрес
# short_fwd <domain_name> <array_name>
short_fwd() {
    local -a _sf_reply
    local -i _sf_rc
    local -i _sf_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'sfwd: '${1}
  _sf_reply=( $(dig +short ${1} -c in -t a 2>/dev/null) )
  _sf_rc=$?
  if [ ${_sf_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='## Ошибка поиска '${_sf_rc}' на '${1}' ##'
# [ ${_sf_rc} -ne 9 ] && pend_drop
        return ${_sf_rc}
    else
        # Некоторые версии 'dig' возвращают предупреждения на stdout.
```

```
        _sf_cnt=${#_sf_reply[@]}
        for (( _sf = 0 ; _sf < ${_sf_cnt} ; _sf++ ))
        do
            [ 'x'${_sf_reply[${_sf}]:0:2} == 'x;;' ] &&
                unset _sf_reply[${_sf}]
        done
        eval $2=\( \$\{_sf_reply\[@\]\} \)
    fi
    return 0
}

# Обратный поиск :: Адрес -> Имя
# short_rev <ip_address> <array_name>
short_rev() {
    local -a _sr_reply
    local -i _sr_rc
    local -i _sr_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'srev: '${1}
  _sr_reply=( $(dig +short -x ${1} 2>/dev/null) )
  _sr_rc=$?
  if [ ${_sr_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='## Ошибка поиска '${_sr_rc}' на '${1}' ##'
# [ ${_sr_rc} -ne 9 ] && pend_drop
        return ${_sr_rc}
    else
        # Некоторые версии 'dig' возвращают предупреждения на stdout.
        _sr_cnt=${#_sr_reply[@]}
        for (( _sr = 0 ; _sr < ${_sr_cnt} ; _sr++ ))
        do
            [ 'x'${_sr_reply[${_sr}]:0:2} == 'x;;' ] &&
                unset _sr_reply[${_sr}]
        done
        eval $2=\( \$\{_sr_reply\[@\]\} \)
    fi
    return 0
}

# Специальный формат поиска используется для запросов серверов черного списка.
# short_text <ip_address> <array_name>
short_text() {
    local -a _st_reply
    local -i _st_rc
    local -i _st_cnt
    IFS=${NO_WSP}
# echo 'stxt: '${1}
  _st_reply=( $(dig +short ${1} -c in -t txt 2>/dev/null) )
  _st_rc=$?
  if [ ${_st_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='##Ошибка поиска текста '${_st_rc}' на
'${1}'##'
# [ ${_st_rc} -ne 9 ] && pend_drop
        return ${_st_rc}
    else
        # Некоторые версии 'dig' возвращают предупреждения на stdout.
        _st_cnt=${#_st_reply[@]}
        for (( _st = 0 ; _st < ${#_st_cnt} ; _st++ ))
        do
```

```
            [ 'x'${_st_reply[${_st}]:0:2} == 'x;;' ] &&
                unset _st_reply[${_st}]
        done
        eval $2=\( \$\{_st_reply\[@\]\} \)
    fi
    return 0
}

# Длинные формы, анализ самих версий

# RFC 2782   Служба поисков
# dig +noall +nofail +answer _ldap._tcp.openldap.org -t srv
# _<service>._<protocol>.<domain_name>
# _ldap._tcp.openldap.org. 3600   IN     SRV    0 0 389 ldap.openldap.org.
# domain TTL Class SRV Priority Weight Port Target

# Поиск вперед :: Имя -> передача зоны poor man
# long_fwd <domain_name> <array_name>
long_fwd() {
    local -a _lf_reply
    local -i _lf_rc
    local -i _lf_cnt
    IFS=${NO_WSP}
echo -n ':'
# echo 'lfwd: '${1}
  _lf_reply=( $(
     dig +noall +nofail +answer +authority +additional \
         ${1} -t soa ${1} -t mx ${1} -t any 2>/dev/null) )
  _lf_rc=$?
  if [ ${_lf_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='# Ошибка поиска зоны '${_lf_rc}' на '${1}' #'
# [ ${_lf_rc} -ne 9 ] && pend_drop
        return ${_lf_rc}
    else
        # Некоторые версии 'dig' возвращают предупреждения на stdout.
        _lf_cnt=${#_lf_reply[@]}
        for (( _lf = 0 ; _lf < ${_lf_cnt} ; _lf++ ))
        do
            [ 'x'${_lf_reply[${_lf}]:0:2} == 'x;;' ] &&
                unset _lf_reply[${_lf}]
        done
        eval $2=\( \$\{_lf_reply\[@\]\} \)
    fi
    return 0
}
#  Обратный поиск доменного имени соответствующий адресу IPv6
#      4321:0:1:2:3:4:567:89ab
#  будет (частями, т.е.: Hexdigit) задом на перед:
#  b.a.9.8.7.6.5.0.4.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.0.0.0.0.1.2.3.4.IP6.ARPA.

# Обратный поиск :: Адрес -> передача связи poor man
# long_rev <rev_ip_address> <array_name>
long_rev() {
    local -a _lr_reply
    local -i _lr_rc
    local -i _lr_cnt
    local _lr_dns
    _lr_dns=${1}'.in-addr.arpa.'
    IFS=${NO_WSP}
echo -n ':'
```

```
# echo 'lrev: '${1}
  _lr_reply=( $(
       dig +noall +nofail +answer +authority +additional \
           ${_lr_dns} -t soa ${_lr_dns} -t any 2>/dev/null) )
  _lr_rc=$?
  if [ ${_lr_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='# Deleg lkp error '${_lr_rc}' on '${1}' #'
# [ ${_lr_rc} -ne 9 ] && pend_drop
       return ${_lr_rc}
    else
       # Некоторые версии 'dig' возвращают предупреждения на stdout.
       _lr_cnt=${#_lr_reply[@]}
       for (( _lr = 0 ; _lr < ${_lr_cnt} ; _lr++ ))
       do
           [ 'x'${_lr_reply[${_lr}]:0:2} == 'x;;' ] &&
               unset _lr_reply[${_lr}]
       done
       eval $2=\( \$\{_lr_reply\[@\]\} \)
    fi
    return 0
}

# # # Применение конкретных функций # # #

# Портит возможное имя; подавляет root и TLD (домены верхнего уровня).
# name_fixup <string>
name_fixup(){
    local -a _nf_tmp
    local -i _nf_end
    local _nf_str
    local IFS
    _nf_str=$(to_lower ${1})
    _nf_str=$(to_dot ${_nf_str})
    _nf_end=${#_nf_str}-1
    [ ${_nf_str:${_nf_end}} != '.' ] &&
        _nf_str=${_nf_str}'.'
    IFS=${ADR_IFS}
    _nf_tmp=( ${_nf_str} )
    IFS=${WSP_IFS}
    _nf_end=${#_nf_tmp[@]}
    case ${_nf_end} in
    0) # Без точек, только точки.
        echo
        return 1
    ;;
    1) # Только TLD.
        echo
        return 1
    ;;
    2) # Возможно нормально.
       echo ${_nf_str}
       return 0
       # Нужно просмотреть таблицу?
       if [ ${#_nf_tmp[1]} -eq 2 ]
       then # Коды стран TLD.
           echo
           return 1
       else
           echo ${_nf_str}
           return 0
```

```
      fi
    ;;
    esac
    echo ${_nf_str}
    return 0
}

# Нашупывает и портит оригинальный ввод(ы).
split_input() {
    [ ${#uc_name[@]} -gt 0 ] || return 0
    local -i _si_cnt
    local -i _si_len
    local _si_str
    unique_lines uc_name uc_name
    _si_cnt=${#uc_name[@]}
    for (( _si = 0 ; _si < _si_cnt ; _si++ ))
    do
        _si_str=${uc_name[$_si]}
        if is_address ${_si_str}
        then
            uc_address[${#uc_address[@]}]=${_si_str}
            unset uc_name[$_si]
        else
            if ! uc_name[$_si]=$(name_fixup ${_si_str})
            then
                unset ucname[$_si]
            fi
        fi
    done
  uc_name=( ${uc_name[@]} )
  _si_cnt=${#uc_name[@]}
  _trace_log[${#_trace_log[@]}]='#Input '${_si_cnt}' unchkd name input(s).#'
  _si_cnt=${#uc_address[@]}
  _trace_log[${#_trace_log[@]}]='#Input '${_si_cnt}' unchkd addr input(s).#'
    return 0
}

# # # Обнаружение функций--рекурсивно блокированные внешние данные  # # #
#  В каждом начале 'если список пуст; возвращается 0 ' это требование. #

# Ограничитель рекурсии
# limit_chk() <next_level>
limit_chk() {
    local -i _lc_lmt
    # Проверка прямых ограничений.
    if [ ${indirect} -eq 0 ] || [ $# -eq 0 ]
    then
        # Выбор 'делать всегда'
        echo 1                # Любое значение будет сделано.
        return 0              # OK для продолжения.
    else
        # Ограничение действует.
        if [ ${indirect} -lt ${1} ]
        then
            echo ${1}         # Без разницы.
            return 1          # Здесь прекращается.
        else
            _lc_lmt=${1}+1    # Увеличиваем заданный предел.
            echo ${_lc_lmt}   # Выводим его на экран.
            return 0          # OK для продолжения.
        fi
```

```
    fi
}

# Для каждого имени в uc_name:
#     Перемещаем имя в chk_name.
#     Добавляем адреса в uc_address.
#     Виснет expand_input_address.
#     Повторяется до тех пор, пока не обнаружится новое.
# expand_input_name <indirection_limit>
expand_input_name() {
    [ ${#uc_name[@]} -gt 0 ] || return 0
    local -a _ein_addr
    local -a _ein_new
    local -i _ucn_cnt
    local -i _ein_cnt
    local _ein_tst
    _ucn_cnt=${#uc_name[@]}

    if  ! _ein_cnt=$(limit_chk ${1})
    then
        return 0
    fi

     (( _ein = 0 ; _ein < _ucn_cnt ; _ein++ ))

        if short_fwd ${uc_name[${_ein}]} _ein_new
        then
          for (( _ein_cnt = 0 ; _ein_cnt < ${#_ein_new[@]}; _ein_cnt++ ))
          do
              _ein_tst=${_ein_new[${_ein_cnt}]}
              if is_address ${_ein_tst}
              then
                  _ein_addr[${#_ein_addr[@]}]=${_ein_tst}
              fi
          done
        fi
    done
    unique_lines _ein_addr _ein_addr     # Отменяет повторения.
    edit_exact chk_address _ein_addr     # Отменяет зависшие детали.
    edit_exact known_address _ein_addr   # Подробности отмены.
 if [ ${#_ein_addr[@]} -gt 0 ]           # Что-нибудь новое?
 then
   uc_address=( ${uc_address[@]} ${_ein_addr[@]} )
   pend_func expand_input_address ${1}
   _trace_log[${#_trace_log[@]}]='#Добавление '${#_ein_addr[@]}' введенного
непроверенного адреса inp.#'
 fi
    edit_exact chk_name uc_name          # Отменяет зависшие детали.
    edit_exact known_name uc_name        # Подробности отмены.
    if [ ${#uc_name[@]} -gt 0 ]
    then
        chk_name=( ${chk_name[@]} ${uc_name[@]}  )
        pend_func detail_each_name ${1}
    fi
    unset uc_name[@]
    return 0
}

# Для каждого адреса в uc_address:
#     Перемещаем адрес в chk_address.
#     Добавляем имя в uc_name.
```

```
#     Зависание expand_input_name.
#     Повторяется до тех пор, пока не обнаружится новое
# expand_input_address <indirection_limit>
expand_input_address() {
    [ ${#uc_address[@]} -gt 0 ] || return 0
    local -a _eia_addr
    local -a _eia_name
    local -a _eia_new
    local -i _uca_cnt
    local -i _eia_cnt
    local _eia_tst
    unique_lines uc_address _eia_addr
    unset uc_address[@]
    edit_exact been_there_addr _eia_addr
    _uca_cnt=${#_eia_addr[@]}
    [ ${_uca_cnt} -gt 0 ] &&
        been_there_addr=( ${been_there_addr[@]} ${_eia_addr[@]} )

    for (( _eia = 0 ; _eia < _uca_cnt ; _eia++ ))
     do
       if short_rev ${_eia_addr[${_eia}]} _eia_new
       then
         for (( _eia_cnt = 0 ; _eia_cnt < ${#_eia_new[@]} ; _eia_cnt++ ))
         do
           _eia_tst=${_eia_new[${_eia_cnt}]}
           if _eia_tst=$(name_fixup ${_eia_tst})
           then
             _eia_name[${#_eia_name[@]}]=${_eia_tst}
       fi
     done
           fi
    done
    unique_lines _eia_name _eia_name     # Отменяет повторения.
    edit_exact chk_name _eia_name        # Отменяет зависшие детали.
    edit_exact known_name _eia_name      # Подробности отмены.
 if [ ${#_eia_name[@]} -gt 0 ]           # Что-то новое?
 then
   uc_name=( ${uc_name[@]} ${_eia_name[@]} )
   pend_func expand_input_name ${1}
   _trace_log[${#_trace_log[@]}]='#Добавление '${#_eia_name[@]}' ввода
непроверенного имени.#'
    fi
    edit_exact chk_address _eia_addr     # Отменяет зависшие детали.
    edit_exact known_address _eia_addr   # Подробности отмены.
    if [ ${#_eia_addr[@]} -gt 0 ]        # Что-то новое?
    then
        chk_address=( ${chk_address[@]} ${_eia_addr[@]} )
        pend_func detail_each_address ${1}
    fi
    return 0
}

# Самоанализ ответа зоны.
# Ввод является списком chk_name.
# detail_each_name <indirection_limit>
detail_each_name() {
    [ ${#chk_name[@]} -gt 0 ] || return 0
    local -a _den_chk       # Имена для проверки
    local -a _den_name      # Имена найденные здесь
    local -a _den_address   # Адреса найденные здесь
    local -a _den_pair      # Пары найденные здесь
```

```
    local -a _den_rev       # Обратные пары найденные здесь
    local -a _den_tmp       # Анализируемая строка
    local -a _den_auth      # Анализируемое соединение SOA
    local -a _den_new       # Ответ зоны
    local -a _den_pc        # Родитель-потомок, быстрое большое получение
    local -a _den_ref       # Делает ссылку связей
    local -a _den_nr        # Имя-Источник может быть большое
    local -a _den_na        # Имя-Адрес
    local -a _den_ns        # Имя-Служба
    local -a _den_achn      # Управляющая связь
    local -i _den_cnt       # Детали подсчета имен
    local -i _den_lmt       # Косвенное ограничение
    local _den_who          # Обработанные имена
    local _den_rec          # Запись типа обработки
    local _den_cont         # Домен соединения
    local _den_str          # Исправление имени строки
    local _den_str2         # Обратное исправление
    local IFS=${WSP_IFS}

    # Локальное, уникальное копирование имен для проверки
    unique_lines chk_name _den_chk
    unset chk_name[@]       # Производится глобально.

    # Несколько любых имен, уже известных
    edit_exact known_name _den_chk
    _den_cnt=${#_den_chk[@]}

    # Если что-нибудь осталось, добавляет в known_name.
    [ ${_den_cnt} -gt 0 ] &&
        known_name=( ${known_name[@]} ${_den_chk[@]} )

    # Для списка (прежде всего) неизвестных имен ...
    for (( _den = 0 ; _den < _den_cnt ; _den++ ))
    do
        _den_who=${_den_chk[${_den}]}
        if long_fwd ${_den_who} _den_new
        then
            unique_lines _den_new _den_new
            if [ ${#_den_new[@]} -eq 0 ]
            then
                _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
            fi

            # Анализ каждой строки ответов.
            for (( _line = 0 ; _line < ${#_den_new[@]} ; _line++ ))
            do
                IFS=${NO_WSP}$'\x09'$'\x20'
                _den_tmp=( ${_den_new[${_line}]} )
                IFS=${WSP_IFS}
              # Если используется запись а не сообщение об ошибке ...
              if [ ${#_den_tmp[@]} -gt 4 ] && [ 'x'${_den_tmp[0]} != 'x;;' ]
              then
                    _den_rec=${_den_tmp[3]}
                    _den_nr[${#_den_nr[@]}]=${_den_who}' '${_den_rec}
                    # Начало с RFC1033 (+++)
                    case ${_den_rec} in

#<name> [<ttl>]  [<class>] SOA <origin> <person>
                    SOA) # Начало управления
    if _den_str=$(name_fixup ${_den_tmp[0]})
    then
```

```
      _den_name[${#_den_name[@]}]=${_den_str}
      _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_str}' SOA'
      # Оригинальный SOA — запись доменного имени основной зоны
      if _den_str2=$(name_fixup ${_den_tmp[4]})
      then
        _den_name[${#_den_name[@]}]=${_den_str2}
        _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_str2}' SOA.O'
      fi
      # Ответная сторона адреса электронной почты (возможно фиктивная).
      # Возможность игнорировать first.last@domain.name.
      set -f
      if _den_str2=$(name_fixup ${_den_tmp[5]})
      then
        IFS=${ADR_IFS}
        _den_auth=( ${_den_str2} )
        IFS=${WSP_IFS}
        if [ ${#_den_auth[@]} -gt 2 ]
        then
          _den_cont=${_den_auth[1]}
          for (( _auth = 2 ; _auth < ${#_den_auth[@]} ; _auth++ ))
          do
            _den_cont=${_den_cont}'.'${_den_auth[${_auth}]}
          done
          _den_name[${#_den_name[@]}]=${_den_cont}'.'
          _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_cont}'. SOA.C'
                                  fi
        fi
        set +f
                        fi
                    ;;


      A) # Запись адреса IP(v4)
      if _den_str=$(name_fixup ${_den_tmp[0]})
      then
        _den_name[${#_den_name[@]}]=${_den_str}
        _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' '${_den_str}
        _den_na[${#_den_na[@]}]=${_den_str}' '${_den_tmp[4]}
        _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' A'
      else
        _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' unknown.domain'
        _den_na[${#_den_na[@]}]='unknown.domain '${_den_tmp[4]}
        _den_ref[${#_den_ref[@]}]=${_den_who}' unknown.domain A'
      fi
      _den_address[${#_den_address[@]}]=${_den_tmp[4]}
      _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_tmp[4]}
             ;;

             NS) # Запись имени сервера
             # Обслуживаемый домен (может отличаться от текущего)
               if _den_str=$(name_fixup ${_den_tmp[0]})
                 then
                   _den_name[${#_den_name[@]}]=${_den_str}
                   _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' NS'

             # Провайдер обслуживаемого домена
             if _den_str2=$(name_fixup ${_den_tmp[4]})
             then
               _den_name[${#_den_name[@]}]=${_den_str2}
               _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str2}' NSH'
               _den_ns[${#_den_ns[@]}]=${_den_str2}' NS'
```

```
  _den_pc[${#_den_pc[@]}]=${_den_str}' '${_den_str2}
 fi
  fi
       ;;

MX) # Запись почтового сервера
    # Обслуживаемый домен (без символов групповой подстановки)
if _den_str=$(name_fixup ${_den_tmp[0]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' MX'
fi
# Доменное имя провайдера сервиса
if _den_str=$(name_fixup ${_den_tmp[5]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' MXH'
  _den_ns[${#_den_ns[@]}]=${_den_str}' MX'
  _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
fi
       ;;

PTR) # Запись обратного адреса
     # Особое имя
if _den_str=$(name_fixup ${_den_tmp[0]})
then
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' PTR'
  # Host name (not a CNAME)
  if _den_str2=$(name_fixup ${_den_tmp[4]})
  then
    _den_rev[${#_den_rev[@]}]=${_den_str}' '${_den_str2}
    _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str2}' PTRH'
    _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
  fi
fi
       ;;

AAAA) # Запись адреса IP(v6)
if _den_str=$(name_fixup ${_den_tmp[0]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' '${_den_str}
  _den_na[${#_den_na[@]}]=${_den_str}' '${_den_tmp[4]}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' AAAA'
  else
    _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' unknown.domain'
    _den_na[${#_den_na[@]}]='unknown.domain '${_den_tmp[4]}
    _den_ref[${#_den_ref[@]}]=${_den_who}' unknown.domain'
  fi
  # Не производится для адресов IPv6
  _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_tmp[4]}
       ;;

CNAME) # Запись алиаса имени
       # Никнейм
if _den_str=$(name_fixup ${_den_tmp[0]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' CNAME'
  _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
fi
```

```
                    # Имя хоста
              if _den_str=$(name_fixup ${_den_tmp[4]})
              then
                _den_name[${#_den_name[@]}]=${_den_str}
                _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' CHOST'
                _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
              fi
                    ;;
#             TXT)
#             ;;
                    esac
                fi
            done
        else # Поиск ошибки == 'A' запись 'неизвестный адрес'
            _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
        fi
    done

    # Точка управления ростом массива.
    unique_lines _den_achn _den_achn      # Работает лучше всего, все время.
    edit_exact auth_chain _den_achn       # Работает лучше всего, временами.
    if [ ${#_den_achn[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        auth_chain=( ${auth_chain[@]} ${_den_achn[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_ref _den_ref      # Работает лучше всего, все время.
    edit_exact ref_chain _den_ref       # Работает лучше всего, временами.
    if [ ${#_den_ref[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        ref_chain=( ${ref_chain[@]} ${_den_ref[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_na _den_na
    edit_exact name_address _den_na
    if [ ${#_den_na[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_address=( ${name_address[@]} ${_den_na[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_ns _den_ns
    edit_exact name_srvc _den_ns
    if [ ${#_den_ns[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_srvc=( ${name_srvc[@]} ${_den_ns[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_nr _den_nr
    edit_exact name_resource _den_nr
    if [ ${#_den_nr[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_resource=( ${name_resource[@]} ${_den_nr[@]} )
```

```
        IFS=${WSP_IFS}
    fi

    unique_lines _den_pc _den_pc
    edit_exact parent_child _den_pc
    if [ ${#_den_pc[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        parent_child=( ${parent_child[@]} ${_den_pc[@]} )
        IFS=${WSP_IFS}
    fi

    # Обновление списка known_pair (Адреса и имена).
    unique_lines _den_pair _den_pair
    edit_exact known_pair _den_pair
    if [ ${#_den_pair[@]} -gt 0 ]  # Что-то новое?
    then
        IFS=${NO_WSP}
        known_pair=( ${known_pair[@]} ${_den_pair[@]} )
        IFS=${WSP_IFS}
    fi

    # Обновление списка обратных пар.
    unique_lines _den_rev _den_rev
    edit_exact reverse_pair _den_rev
    if [ ${#_den_rev[@]} -gt 0 ]   # Что-то новое?
    then
        IFS=${NO_WSP}
        reverse_pair=( ${reverse_pair[@]} ${_den_rev[@]} )
        IFS=${WSP_IFS}
    fi

    # Проверка косвенного ограничения — сбрасывается, если достигнуто.
    if ! _den_lmt=$(limit_chk ${1})
    then
        return 0
    fi

# Исполняемый движок - LIFO. Важен порядок выполнения операций.
# Мы определили новые адреса?
unique_lines _den_address _den_address    # Сброс повторений.
edit_exact known_address _den_address     # Сброс всех обработанных.
edit_exact un_address _den_address        # Сброс всех ожидающих.
if [ ${#_den_address[@]} -gt 0 ]          # Что-то новое?
then
  uc_address=( ${uc_address[@]} ${_den_address[@]} )
  pend_func expand_input_address ${_den_lmt}
  _trace_log[${#_trace_log[@]}]='# Добавление '${#_den_address[@]}' не
проверенных адресов. #'
    fi

# Найдены новые имена?
unique_lines _den_name _den_name          # Сброс повторений.
edit_exact known_name _den_name           # Сброс всех обработанных.
edit_exact uc_name _den_name              # Сброс всех ожидающих.
if [ ${#_den_name[@]} -gt 0 ]             # Что-то новое?
then
  uc_name=( ${uc_name[@]} ${_den_name[@]} )
  pend_func expand_input_name ${_den_lmt}
  _trace_log[${#_trace_log[@]}]='#Добавленные '${#_den_name[@]}' не проверенные
имена#'
```

```
    fi
    return 0
}

# Самоанализ передачи ответа
# Ввод является списком chk_address.
# detail_each_address <indirection_limit>
detail_each_address() {
    [ ${#chk_address[@]} -gt 0 ] || return 0
    unique_lines chk_address chk_address
    edit_exact known_address chk_address
    if [ ${#chk_address[@]} -gt 0 ]
    then
        known_address=( ${known_address[@]} ${chk_address[@]} )
        unset chk_address[@]
    fi
    return 0
}

# # # Применение конкретного вывода функций # # #

# Вывод на экран читабельных известных пар.
report_pairs() {
    echo
    echo 'Известные пары сети.'
    col_print known_pair 2 5 30

    if [ ${#auth_chain[@]} -gt 0 ]
    then
        echo
        echo 'Известная цепь управления.'
        col_print auth_chain 2 5 30 55
    fi

    if [ ${#reverse_pair[@]} -gt 0 ]
    then
        echo
        echo 'Известные обратные пары.'
        col_print reverse_pair 2 5 55
    fi
    return 0
}

# Проверка адресов С черным списком серверов.
# Хорошее место для захвата GraphViz: адрес->состояние(сервер(сообщение))
# check_lists <ip_address>
check_lists() {
    [ $# -eq 1 ] || return 1
    local -a _cl_fwd_addr
    local -a _cl_rev_addr
    local -a _cl_reply
    local -i _cl_rc
    local -i _ls_cnt
    local _cl_dns_addr
    local _cl_lkup

    split_ip ${1} _cl_fwd_addr _cl_rev_addr
    _cl_dns_addr=$(dot_array _cl_rev_addr)'.'
    _ls_cnt=${#list_server[@]}
    echo '    Проверьте адрес '${1}
    for (( _cl = 0 ; _cl < _ls_cnt ; _cl++ ))
```

```
    do
      _cl_lkup=${_cl_dns_addr}${list_server[${_cl}]}
      if short_text ${_cl_lkup} _cl_reply
      then
        if [ ${#_cl_reply[@]} -gt 0 ]
        then
          echo '        Записи из '${list_server[${_cl}]}
          address_hits[${#address_hits[@]}]=${1}' '${list_server[${_cl}]}
          _hs_RC=2
          for (( _clr = 0 ; _clr < ${#_cl_reply[@]} ; _clr++ ))
          do
            echo '            '${_cl_reply[${_clr}]}
          done
        fi
      fi
    done
    return 0
}

# # # Обыкновенное применение склеивания # # #

# Кто это сделал?
credits() {
   echo
   echo 'Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz'
}

# Как этим пользоваться?
# (См. Так же, "Quickstart" в конце сценария.)
usage() {
    cat <<-'_usage_statement_'
    Сценарию is_spammer.bash требуется один или два аргумента.

    arg 1) Может быть одним из:
        a) Доменное имя
        b) Адрес IPv4
        c) Имя файла с смешанными именами
           и адресами, на каждой строке.

    arg 2) Может быть одним из:
        a) Доменное имя сервера Blacklist
        b) Имя файла с именами серверов Blacklist
           на каждой строке.
        c) Если не указано, по умолчанию список (свободный)
           ис пользуемых серверов Blacklist.
        d) Если задан пустой, читабельный файл,
           то поиск сервера Blacklist отключается

    Все выводы сценария пишутся на stdout.

    Возвращаемые коды: 0 -> Все OK, 1 -> Ошибка сценария,
                  2 -> Что-то находится в Blacklist.

    Требуется сторонняя программа 'dig' из набора 'bind-9'
    программ DNS.  См.: http://www.isc.org

    Ограничение глубины поиска доменного имени, по умолчанию 2 уровня.
    Установите переменную окружения SPAMMER_LIMIT для изменения.
    SPAMMER_LIMIT=0 означает 'неограниченно'

    Ограничение может быть установлено в командной строке.
```

```
    Если arg#1 -целое число, ограничение устанавливается в данное значение
    и потом  применяются правила аргумента выше.

    Установка переменной окружения 'SPAMMER_DATA' для файла
    вызовет сценарий, для записи графического файла GraphViz.


_usage_statement_
}

# Список серверов Blacklist по умолчанию:
# См.: http://www.spews.org/lists.html

declare -a default_servers
# См.: http://www.spamhaus.org (Постоянный, хорошо сопровождаемый)
default_servers[0]='sbl-xbl.spamhaus.org'
# См.: http://ordb.org (Открытые почтовые переключения)
default_servers[1]='relays.ordb.org'
# См.: http://www.spamcop.net/ (Сообщайте о спаммерах сюда)
default_servers[2]='bl.spamcop.net'
# См.: http://www.spews.org (Система 'раннего обнаружения')
default_servers[3]='l2.spews.dnsbl.sorbs.net'
# См.: http://www.dnsbl.us.sorbs.net/using.shtml
default_servers[4]='dnsbl.sorbs.net'
# См.: http://dsbl.org/usage (Различные списки почтовых переключений)
default_servers[5]='list.dsbl.org'
default_servers[6]='multihop.dsbl.org'
default_servers[7]='unconfirmed.dsbl.org'

# Пользователь вводит аргумент #1
setup_input() {
    if [ -e ${1} ] && [ -r ${1} ]  # Имя читабельного файла
    then
        file_to_array ${1} uc_name
        echo 'Используем файл >'${1}'< как ввод.'
    else
        if is_address ${1}          # IP адрес?
        then
            uc_address=( ${1} )
            echo 'Начинаем с адреса >'${1}'<'
        else                        # Должно быть имя.
            uc_name=( ${1} )
            echo 'Начинаем с доменного имени >'${1}'<'
        fi
    fi
    return 0
}

# Пользователь вводит аргумент #2
setup_servers() {
    if [ -e ${1} ] && [ -r ${1} ]  # Имя читабельного файла
    then
        file_to_array ${1} list_server
        echo 'Используемый файл >'${1}'< как список сервера blacklist.'
    else
        list_server=( ${1} )
        echo 'Используемый сервер blacklist >'${1}'<'
    fi
    return 0
}
```

```
# Пользовательская переменная окружения SPAMMER_TRACE
live_log_die() {
    if [ ${SPAMMER_TRACE:=} ]    # Нужен журнал трассировки?
    then
        if [ ! -e ${SPAMMER_TRACE} ]
        then
            if ! touch ${SPAMMER_TRACE} 2>/dev/null
            then
                pend_func echo $(printf '%q\n' \
                'Невозможно создать логфайл >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            _pend_hook_=trace_logger
            _log_dump=dump_log
        else
            if [ ! -w ${SPAMMER_TRACE} ]
            then
                pend_func echo $(printf '%q\n' \
                'Невозможно записать >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            echo '' > ${_log_file}
            _pend_hook_=trace_logger
            _log_dump=dump_log
        fi
    fi
    return 0
}

# Пользовательская переменная окружения SPAMMER_DATA
data_capture() {
    if [ ${SPAMMER_DATA:=} ]    # Нужен дамп данных?
    then
        if [ ! -e ${SPAMMER_DATA} ]
        then
            if ! touch ${SPAMMER_DATA} 2>/dev/null
            then
                pend_func echo $(printf '%q]n' \
                'Не удается создать выходной файл данных >'${SPAMMER_DATA}'<')
                pend_release
                exit 1
            fi
            _dot_file=${SPAMMER_DATA}
            _dot_dump=dump_dot
        else
            if [ ! -w ${SPAMMER_DATA} ]
            then
                pend_func echo $(printf '%q\n' \
                'Не удается записать выходной файл данных >'${SPAMMER_DATA}'<')
                pend_release
                exit 1
            fi
            _dot_file=${SPAMMER_DATA}
            _dot_dump=dump_dot
        fi
    fi
    return 0
```

```
}

# Ищем указанные пользователем аргументы.
do_user_args() {
    if [ $# -gt 0 ] && is_number $1
    then
        indirect=$1
        shift
    fi

    case $# in                      # Рассматриваем правильно?
        1)
            if ! setup_input $1     # Необходима проверка ошибок.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            list_server=( ${default_servers[@]} )
            _list_cnt=${#list_server[@]}
            echo 'Using default blacklist server list.'
            echo 'Ограничение глубины поиска: '${indirect}
            ;;
        2)
            if ! setup_input $1     # Необходима проверка ошибок.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            if ! setup_servers $2   # Необходима проверка ошибок.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            echo 'Ограничение глубины поиска: '${indirect}
            ;;
        *)
            pend_func usage
            pend_release
            $_log_dump
            exit 1
            ;;
    esac
    return 0
}

# Инструмент общей отладки.
# list_array <array_name>
list_array() {
    [ $# -eq 1 ] || return 1  # Требуется один аргумент.

    local -a _la_lines
    set -f
    local IFS=${NO_WSP}
    eval _la_lines=\(\ \$\{$1\[@\]\}\ \)
    echo
    echo "Подсчет элементов "${#_la_lines[@]}" array "${1}
    local _ln_cnt=${#_la_lines[@]}
```

```
    for (( _i = 0; _i < ${_ln_cnt}; _i++ ))
    do
        echo 'Элемент '$_i' >'${_la_lines[$_i]}'<'
    done
    set +f
    return 0
}

# # # Код программы 'Hunt the Spammer' # # #
pend_init                               # Готовый стек движка.
pend_func credits                       # Последнее выведенное на экран.

# # # Предоставляется пользователю # # #
live_log_die                            # Настройка журнала отладки трассировки.
data_capture                            # Настройка файла сбора данных.
echo
do_user_args $@

# # # Не вышли еще - есть надежда   # # #
# Открытие группы - выполнение движком  LIFO -
# выкладывание в обратном порядке исполнения.
_hs_RC=0                                # Возвращаемый код Hunt the Spammer
return code
pend_mark
    pend_func report_pairs              # Сообщение пар имя-адрес.

    # Обе detail_* являются взаимно обратными.
    # Поэтому они выкладывают функции expand_* как затребованные.
    # Обе (последними ???) рекурсивно выходят.
    pend_func detail_each_address       # Получает все ресурсы адресов.
    pend_func detail_each_name          # Получает все ресурсы имен.

    #  Обе expand_* взаимно обратны,
    #+ выкладывают добавленные функции detail_* как затребованные.
    pend_func expand_input_address 1    # Расширяет ввод имени по адресу.
    pend_func expand_input_name 1       # Расширяет ввод адреса по имени.

    # Начинаем с присвоения уникальных имен и адресов.
    pend_func unique_lines uc_address uc_address
    pend_func unique_lines uc_name uc_name

    # Разделяем смешанный ввод имен и адресов.
    pend_func split_input
pend_release

# # # Сообщения пар — Уникальный список обнаруженных IP адресов
echo
_ip_cnt=${#known_address[@]}
if [ ${#list_server[@]} -eq 0 ]
then
    echo 'Черный список серверов пуст, не проверен.'
else
    if [ ${_ip_cnt} -eq 0 ]
    then
        echo 'Список известных адресов пуст, не проверен.'
    else
        _ip_cnt=${_ip_cnt}-1   # Начинаем сверху.
        echo 'Проверка серверов черного списка.'
        for (( _ip = _ip_cnt ; _ip >= 0 ; _ip-- ))
        do
          pend_func check_lists $( printf '%q\n' ${known_address[$_ip]} )
```

```
        done
    fi
fi
pend_release
$_dot_dump                  # Графический дамп файла
$_log_dump                  # Трассировка выполнения
echo


##############################
# Пример вывода сценария #
##############################
:<<-'_is_spammer_outputs_'

./is_spammer.bash 0 web4.alojamentos7.com

Начинаем с доменного имени >web4.alojamentos7.com<
Используется черный список серверов по умолчанию.
Ограничение глубины поиска: 0
.:...:::::..:::::.::::::::.::.::.::::::::::::
Известные сетевые пары.
    66.98.208.97            web4.alojamentos7.com.
    66.98.208.97            ns1.alojamentos7.com.
    69.56.202.147           ns2.alojamentos.ws.
    66.98.208.97            alojamentos7.com.
    66.98.208.97            web.alojamentos7.com.
    69.56.202.146           ns1.alojamentos.ws.
    69.56.202.146           alojamentos.ws.
    66.235.180.113          ns1.alojamentos.org.
    66.235.181.192          ns2.alojamentos.org.
    66.235.180.113          alojamentos.org.
    66.235.180.113          web6.alojamentos.org.
    216.234.234.30          ns1.theplanet.com.
    12.96.160.115           ns2.theplanet.com.
    216.185.111.52          mail1.theplanet.com.
    69.56.141.4             spooling.theplanet.com.
    216.185.111.40          theplanet.com.
    216.185.111.40          www.theplanet.com.
    216.185.111.52          mail.theplanet.com.

Проверка серверов черного списка.
  Проверка адреса 66.98.208.97
      Запись из dnsbl.sorbs.net
  "Принятый спам См.: http://www.dnsbl.sorbs.net/lookup.shtml?66.98.208.97"
    Проверяется адрес 69.56.202.147
    Проверяется адрес 69.56.202.146
    Проверяется адрес 66.235.180.113
    Проверяется адрес 66.235.181.192
    Проверяется адрес 216.185.111.40
    Проверяется адрес 216.234.234.30
    Проверяется адрес 12.96.160.115
    Проверяется адрес 216.185.111.52
    Проверяется адрес 69.56.141.4

Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz

_is_spammer_outputs_

exit ${_hs_RC}

################################################################################
```

```
#  Сценарий игнорирует все отсюда вниз из-за команды «exit», чуть выше. #
################################################################################



Quickstart
==========

 Условия

  Версия Bash 2.05b или 3.00 (bash --version)
  Версия Bash поддерживающая массивы. Поддерживаемые
  массивы являются по умолчанию встроенными в конфигурацию Bash.

  'dig,' версия 9.x.x (dig $HOSTNAME, см. первую строку вывода)
  Версия dig поддерживающая +короткие опции.
  См.: dig_wrappers.bash .


 Дополнительные условия

  Локальные «именованные» программы кэширования DNS. На любой вкус.
  Дважды: dig $HOSTNAME
  Проверка нижней части выходных данных для: SERVER: 127.0.0.1#53
  Это означает, что одно запущено.


 Дополнительная графическая поддержка

  'date,' обычная вещь *nix. (date -R)

  Программа dot преобразовывает описанный графический файл в
  диаграмму. (dot -V)
  Часть из набора программ Graph-Viz.
  См.: [http://www.research.att.com/sw/tools/graphviz||GraphViz]

  'dotty,' визуальный редактор для графически описанных файлов.
  То же часть из набора программ Graph-Viz.


 Quick Start

В той же директории, что и is_spammer.bash script;
Выполняем: ./is_spammer.bash

 Детали использования
1. Выбор сервера черного списка.

  (a) Использование по умолчанию, встроенный список: Ничего не делаем.

  (b) Использование своего собственного списка:

    i. Создаем файл с одним доменным именем сервера черного списка
       на каждой строке.

    ii. Предоставляем этот файл в качестве последнего аргумента сценарию.

  (c) Используя единственный сервер черного списка: Последний аргумент
      сценария.

  (d) Отключение черного списка:
```

```
    i. Создаем пустой файл (касается spammer.nul)
       Выбираете файл.

    ii. Предоставляем этот пустой файл, как
        последний аргумент сценария.

2. Ограничение глубины поиска.

  (a) По умолчанию, значение 2 : Ничего не делаем.

  (b) Установить свое ограничение:
      Ограничение 0 означает: без ограничений.

    i. export SPAMMER_LIMIT=1
       или любое нужное ограничение.

    ii. ИЛИ предоставляем требуемое ограничение, как первый
       аргумент сценария.

3. Дополнительный журнал выполнения трассировки.

  (a) Использовать установку по умолчанию без вывода лога: Ничего не делаем.

  (b) Записывать лог выполнения трассировки:
      export SPAMMER_TRACE=spammer.log
      или в любой другой файл.

4. Дополнительный графически описанный файл.

  (a) По умолчанию: Ничего не делаем.

  (b) Записать графический файл Graph-Viz:
      export SPAMMER_DATA=spammer.dot
      или в любой другой файл.

5. Откуда начинать поиск.

  (a) Начать с одного доменного имени:

    i. Без ограничения поиска командной строкой: Первый
       аргумент сценария.

    ii. С ограничением поиска командной строкой: Второй
        аргумент сценария.

  (b) Начать с одного IP адреса:

    i. Без ограничения поиска командной строкой: Первый
       аргумент сценария.

    ii. С ограничением поиска командной строкой: Второй
        аргумент сценария.

  (c) Начать с (смешанно) нескольких имен и/или адресов:
      Создаем файл с одним именем или адресом на строке.
      Выбираем имя файла.

    i. Без ограничения поиска командной строкой: Файл как
       первый аргумент сценария.
```

```
    ii. С ограничением поиска командной строкой: Файл как
        второй аргумент сценария.

6. Что делать с выводом на экран.

  (a) Просмотр вывода на экране: Ничего не делаем.

  (b) Сохранить вывод в файл: Перенаправить stdout в файл.

  (c) Отменить вывод на экран: Перенаправить stdout в /dev/null.

7. Отмена решения.
   нажмите RETURN
   ждите (не обязательно, увидите точки и запятые).

8. Необязательная проверка возвращаемого кода.

  (a) Возвращаемый код 0: Все OK

  (b) Возвращаемый код 1: Ошибка сценария

  (c) Возвращаемый код 2: Что-то из черного списка.

9. Где взять график (диаграмму)?

Сценарий непосредственно не производит график (диаграмму). Он  производит только
описание графического файла. Вы можете обработать выводимый файл графического
дескриптора программой «dot».

Пока вы не измените этот дескриптор файла, для описания того, что вы хотите
увидеть, все, что вы получите - это кучу имен и адресов узлов.

Все обнаруживаемые сценарием связи, находятся в блоке комментариев файла
графического дескриптора, каждый с описательным заголовком.

Необходимое редактирование, для того, чтобы провести линию между парой узлов из
информации в файле дескриптора, может быть сделано с помощью текстового
редактора.

Данные этих строк в файле дескриптора:

# Известные узлы доменного имени

N0000 [label="guardproof.info."] ;

N0002 [label="third.guardproof.info."] ;



# Известный адрес узла

A0000 [label="61.141.32.197"] ;

/*

# Известное имя->вложенный адрес

NA0000 third.guardproof.info. 61.141.32.197
```

```
# Известный родитель->вложенный потомок

PC0000 guardproof.info. third.guardproof.info.

 */
```

Включите следующие строки, заменив узлы идентификаторов в отношениях:

```
# Известные узлы доменного имени

N0000 [label="guardproof.info."] ;

N0002 [label="third.guardproof.info."] ;


# Известные узлы адреса

A0000 [label="61.141.32.197"] ;


# PC0000 guardproof.info. third.guardproof.info.

N0000->N0002 ;


# NA0000 third.guardproof.info. 61.141.32.197

N0002->A0000 ;


/*

# Известное имя->вложенный адрес

NA0000 third.guardproof.info. 61.141.32.197


# Известный родитель->вложенный потомок

PC0000 guardproof.info. third.guardproof.info.

 */
```

Реализуйте это программой «dot», и вы получите ваш первый сетевой график.

Помимо обычных графических вложений, дескриптор файла включает аналогичный формат пар данных, которые описывают сервисы, записи зон (подграфики?), черный список адресов и другие вещи, которые можно было бы включить в график. Эта дополнительная информация может быть отображена как узел другой формы, цвета, размера и т.д.

Файл-дескриптор можно также читать и редактировать сценарием Bash (конечно же). Вы найдете основную часть необходимых функций в "is_spammer.bash" script.

```
# Завершение Quickstart.
```

```
Примечание
========== ====

Michael Zick указывает, что есть интерактивный веб-сайт «makeviz.bash» на
rediris.es. Не могу дать полный URL-адрес, так как это не публично доступный
узел.
```

Другой антиспамный сценарий.

Пример A-29. Spammer Hunt

```
#!/bin/bash
# whx.sh: поиск спамеров с помощью "whois"
# Автор: Walter Dnes
# Небольшие изменения (первой секции) автором ABS Guide.
# Используется в ABS Guide с разрешения.

# Нужна версия Bash 3.x или выше (присутствует оператор =~).
# Сценарий прокомментирован автором и автором ABS Guide.



E_BADARGS=85        # Отсутствие аргумента командной строки.
E_NOHOST=86         # Узел не найден.
E_TIMEOUT=87        # Время поиска узла истекло.
E_UNDEF=88          # Другие ошибки.

HOSTWAIT=10         # Задаем 10 секунд для ответа узла на запрос.
                    # Ожидание может быть увеличено.
OUTFILE=whois.txt   # Выходной файл.
PORT=4321


if [ -z "$1" ]      # Проверка аргументов командной строки (требуемых).
then
  echo "Usage: $0 доменное имя или IP адрес"
  exit $E_BADARGS
fi


if [[ "$1" =~ [a-zA-Z][a-zA-Z]$ ]]  #  Оканчивается двумя буквенными символами?
then                                #  То это доменное имя &&
                                    #+ и производится поиск узла.
  IPADDR=$(host -W $HOSTWAIT $1 | awk '{print $4}')
                                      #  Поиск узла для
                                      #+ получения IP адреса.
                                      #  Извлечение последнего поля.
else
  IPADDR="$1"                         #  Аргументом командной строки был IP
                                      #+ адрес.
fi

echo; echo "IP адресом является: "$IPADDR""; echo

if [ -e "$OUTFILE" ]
then
  rm -f "$OUTFILE"
  echo "Устаревший выходной файл \"$OUTFILE\" удален."; echo
```

```
fi


#  Проверка исправности.
#  (Этот раздел требует больше усилий)
#  ==============================
if [ -z "$IPADDR" ]
# Нет ответа.
then
  echo "Узел не найден!"
  exit $E_NOHOST
fi

if [[ "$IPADDR" =~ ^[;;] ]]
#  ;; Время соединения истекло; нет доступа к серверам.
then
  echo "Время поиска узла истекло!"
  exit $E_TIMEOUT
fi

if [[ "$IPADDR" =~ [(NXDOMAIN)]$ ]]
#  Узел xxxxxxxxx.xxx не найден: 3(NXDOMAIN)
then
  echo "Узел не найден!"
  exit $E_NOHOST
fi

if [[ "$IPADDR" =~ [(SERVFAIL)]$ ]]
#  Узел xxxxxxxxx.xxx не найден: 2(SERVFAIL)
then
  echo "Узел не найден!"
  exit $E_NOHOST
fi


# ======================== Основное тело сценария ========================

AFRINICquery() {
#  Определяет запрашивающую функцию AFRINIC. Выводит уведомление на экран,
#+ а затем делает фактический запрос, перенаправляя вывод в $OUTFILE.

  echo "Поиск $IPADDR на whois.afrinic.net"
  whois -h whois.afrinic.net "$IPADDR" > $OUTFILE

#  Проверка наличия ссылки на rwhois.
#  Предупреждает о не работающем сервере rwhois.infosat.net
#+ и попытается запросить rwhois.
  if grep -e "^remarks: .*rwhois\.[^ ]\+" "$OUTFILE"
  then
    echo " " >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "Предупреждение: rwhois.infosat.net не работает \
      с 2005/02/02" >> $OUTFILE
    echo "         когда был написан этот сценарий." >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "***" >> $OUTFILE
    echo " " >> $OUTFILE
    RWHOIS=`grep "^remarks: .*rwhois\.[^ ]\+" "$OUTFILE" | tail -n 1 |\
    sed "s/\(^.*\)\(rwhois\..*\)\(:4.*\)/\2/"`
    whois -h ${RWHOIS}:${PORT} "$IPADDR" >> $OUTFILE
```

```
  fi
}

APNICquery() {
  echo "Поиск $IPADDR на whois.apnic.net"
  whois -h whois.apnic.net "$IPADDR" > $OUTFILE

#  Почти каждая страна имеет свой собственный регистратор Интернета.
#  Я не беспокою их консультациями, потому что региональный реестр, как
#+ правило, предоставляет достаточную информацию.
#  Есть несколько исключений, когда региональный реестр соотносится с прямыми
#+ данными национального реестра.
#  Это APNIC в Японии и Южной Корее, и LACNIC в Бразилии.
#  Следующее положение проверяет $OUTFILE (whois.txt) на наличие в поле страны
#+ «KR» (Южная Корея) или «JP» (Япония).
#  Если кто-либо из них будет найден, то повторно запускается запрос
#+ соответствующему национальному реестру.

  if grep -E "^country:[ ]+KR$" "$OUTFILE"
  then
    echo "Поиск $IPADDR на whois.krnic.net"
    whois -h whois.krnic.net "$IPADDR" >> $OUTFILE
  elif grep -E "^country:[ ]+JP$" "$OUTFILE"
  then
    echo "Поиск $IPADDR на whois.nic.ad.jp"
    whois -h whois.nic.ad.jp "$IPADDR"/e >> $OUTFILE
  fi
}

ARINquery() {
  echo "Поиск $IPADDR на whois.arin.net"
  whois -h whois.arin.net "$IPADDR" > $OUTFILE

#  Несколько крупных интернет-провайдеров, перечисленных в ARIN, имеют свой
#+ собственный внутренний whois-сервис, называемый «rwhois».
#  Провайдерами из реестра ARIN перечислен большой блок IP-адресов.
#  Чтобы получить IP-адреса второго уровня ISPs или других крупных клиентов,
#+ надо сослаться на порт 4321сервера rwhois.
#  Сначала я начал с кучи операторов «if» проверяя более крупных провайдеров.
#  Этот подход является громоздким и всегда находится другой сервер rwhois, о
#+ котором я не знал.
#  Более элегантным подходом является проверка ссылки $OUTFILE whois-сервера,
#+ анализ этого имени сервера в разделе комментариев и повторный запуск запроса
#+ на соответствующий сервер rwhois.
#  Анализ кажется немного страшным, длинной строкой внутри обратных кавычек.
#  Но сделайте только один раз, и он будет работать по мере добавления новых
#+ серверов.
#@   Комментарий автора ABS Guide : это все не так страшно и это, по сути,
#@+  поучительное использование регулярных выражений.

  if grep -E "^Comment: .*rwhois.[^ ]+" "$OUTFILE"
  then
    RWHOIS=`grep -e "^Comment:.*rwhois\.[^ ]\+" "$OUTFILE" | tail -n 1 |\
    sed "s/^\(.*\)\(rwhois\.[^ ]\+\)\(.*$\)/\2/"`
    echo "Поиск $IPADDR на ${RWHOIS}"
    whois -h ${RWHOIS}:${PORT} "$IPADDR" >> $OUTFILE
  fi
}

LACNICquery() {
  echo "Поиск $IPADDR на whois.lacnic.net"
```

```
  whois -h whois.lacnic.net "$IPADDR" > $OUTFILE

#  Следующий оператор if проверяет $OUTFILE (whois.txt) на наличие 'BR'
#+ (Бразилия) в поле страны.
#  Если он найден, запрос повторно запускается для whois.registro.br.

  if grep -E "^country:[ ]+BR$" "$OUTFILE"
  then
    echo "Поиск $IPADDR на whois.registro.br"
    whois -h whois.registro.br "$IPADDR" >> $OUTFILE
  fi
}

RIPEquery() {
  echo "Поиск $IPADDR на whois.ripe.net"
  whois -h whois.ripe.net "$IPADDR" > $OUTFILE
}

#  Инициализируем несколько переменных.
#  * slash8 это наиболее значимый октет
#  * slash16 состоит из двух наиболее значимых октетов
#  * octet2 это второй наиболее значимый октет


slash8=`echo $IPADDR | cut -d. -f 1`
  if [ -z "$slash8" ]  # Другая проверка.
  then
    echo "Неопределенная ошибка!"
    exit $E_UNDEF
  fi
slash16=`echo $IPADDR | cut -d. -f 1-2`
#                             ^ Период указанный как разделитель 'cut'.
  if [ -z "$slash16" ]
  then
    echo "Неопределенная ошибка!"
    exit $E_UNDEF
  fi
octet2=`echo $slash16 | cut -d. -f 2`
  if [ -z "$octet2" ]
  then
    echo "Неопределенная ошибка!"
    exit $E_UNDEF
  fi


#  Проверка различных мелочей и окончания зарезервированного пространства.
#  Нет смысла в запросах этих адресов.

if [ $slash8 == 0 ]; then
  echo $IPADDR это пространство '"Данной сети"'\; Не запрашивается
elif [ $slash8 == 10 ]; then
  echo $IPADDR это пространство RFC1918\; Не запрашивается
elif [ $slash8 == 14 ]; then
  echo $IPADDR это пространство '"Public Data Network"'\; Не запрашивается
elif [ $slash8 == 127 ]; then
  echo $IPADDR это пространство loopback\; Не запрашивается
elif [ $slash16 == 169.254 ]; then
  echo $IPADDR это пространство локальной ссылки\; Не запрашивается
elif [ $slash8 == 172 ] && [ $octet2 -ge 16 ] && [ $octet2 -le 31 ];then
  echo $IPADDR это пространство RFC1918\; Не запрашивается
```

```
elif [ $slash16 == 192.168 ]; then
  echo $IPADDR это пространство RFC1918\; Не запрашивается
elif [ $slash8 -ge 224 ]; then
  echo $IPADDR это многоадресное или зарезервированное пространство\; Не
запрашивается
elif [ $slash8 -ge 200 ] && [ $slash8 -le 201 ]; then LACNICquery "$IPADDR"
elif [ $slash8 -ge 202 ] && [ $slash8 -le 203 ]; then APNICquery "$IPADDR"
elif [ $slash8 -ge 210 ] && [ $slash8 -le 211 ]; then APNICquery "$IPADDR"
elif [ $slash8 -ge 218 ] && [ $slash8 -le 223 ]; then APNICquery "$IPADDR"

#  Если бы мы получили этот запрос ARIN, то ничего бы он нам не дал.
#  Если ссылка находится в $ OUTFILE  APNIC, AFRINIC, LACNIC или RIPE,
#+ запрашивается соответствующий сервер whois.

else
  ARINquery "$IPADDR"
  if grep "whois.afrinic.net" "$OUTFILE"; then
    AFRINICquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+RIPE$" "$OUTFILE"; then
    RIPEquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+APNIC$" "$OUTFILE"; then
    APNICquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+LACNIC$" "$OUTFILE"; then
    LACNICquery "$IPADDR"
  fi
fi

#@  ----------------------------------------------------------------
#   Попробуйте так же:
#   wget http://logi.cc/nw/whois.php3?ACTION=doQuery&DOMAIN=$IPADDR
#@  ----------------------------------------------------------------

#  Теперь запросы закончены.
#  Вывод копии окончательного результата на экран.

cat $OUTFILE
# Или "less $OUTFILE" ...


exit 0

#@  Комментарии автора ABS Guide:
#@  Ничего особенного здесь нет, но очень полезный инструмент для охоты на
#@+ спамеров.
#@  Конечно, этот сценарий нуждается в улучшении и  все еще немного глючит
#@+ (упражнение для читателя), но все же, это хороший кусок кода Walter Dnes.
#@  Спасибо!
```

Интерфейс к ***wget*** "Маленького монстра"

Пример A-30. Делаем *wget* легким в использовании

```
#!/bin/bash
# wgetter2.bash

# Автор: Маленький монстр [monster@monstruum.co.uk]
# ==> Используется в ABS Guide с разрешения автора сценария.
# ==> Сценарий до сих пор нуждается в отладке (упражнение для читателей).
```

```
# ==> Он так же может дополнительно внести некоторые изменения в комментариях.


#  Это wgetter2 --
#+ сценарий Bash делающий Wget немного более дружественным, и сохраняющий
#+ набранный текст.


# ============================================================================

# Коды ошибок для не нормального выхода.
E_USAGE=67        # Используется сообщение, а потом выход.
E_NO_OPTS=68      # Не введены аргументы командной строки.
E_NO_URLS=69      # Нет URL переданных сценарию.
E_NO_SAVEFILE=70  # Нет сохраненных файлов переданных сценарию.
E_USER_EXIT=71    # Выход по решению пользователя.


#  Мы будем использовать основные команды wget.
#  Можно изменить, если потребуется.
#  NB: если используется proxy, установите http_proxy = yourproxy in .wgetrc.
#  Иначе удалите --proxy=on, ниже.
# ============================================================================
CommandA="wget -nc -c -t 5 --progress=bar --random-wait --proxy=on -r"
# ============================================================================


# ----------------------------------------------------------------------------
# Устанавливаем переменные и объясняем их.

pattern=" -A .jpg,.JPG,.jpeg,.JPEG,.gif,.GIF,.htm,.html,.shtml,.php"
                    # опции wget позволяющие получать только определенные виды
                    #+ файлов.
                    # Закомментируйте не нужное
today=`date +%F`    # Применяется для имени файла.
home=$HOME          # Присваиваем HOME внутренней переменной.
                    # Если используете другой путь, измените.
depthDefault=3      # Присваиваем чувствительность рекурсии по умолчанию.
Depth=$depthDefault # В противном случае обратная связь с пользователем будет
                    #+ не правильной.
RefA=""             # Устанавливаем пустую ссылку на страницу.
Flag=""             #  По умолчанию не сохраняем ничего,
                    #+ или то, что потом посчитаем нужным.
lister=""           # Используется для передачи списка url прямо в wget.
Woptions=""         # Используется для передачи wget других своих опций.
inFile=""           # Используется для запуска функции.
newFile=""          # Используется для запуска функции.
savePath="$home/w-save"
Config="$home/.wgetter2rc"
                    #  Здесь могут быть сохранены некоторые параметры,
                    #+ окончательно измененные в сценарии.
Cookie_List="$home/.cookielist"
                    # Итак, мы знаем, где хранятся cookie ...
cFlag=""            # Часть процедуры выбора файла cookie.

# Определяем доступные параметры. Если необходимо, то буквы легко изменить.
# Это необязательные параметры; просто подождите, чтобы предложили.

save=s   # Сохранить команду вместо ее выполнения.
cook=c   # Изменить файл cookie для этой сессии.
help=h   # Справка.
```

```
list=l   # Передать wget опции -i и списка URL.
runn=r   # Запустить сохраненные команды в качестве опции аргумента.
inpu=i   # Запустить сохраненные команды интерактивно.
wopt=w   # Ввод параметров, передавая непосредственно в wget.
# ---------------------------------------------------------------------


if [ -z "$1" ]; then   # Убедитесь, что у нас есть что-то для ввода в wget.
   echo "Нужно ввести, по крайней мере, URL или опцию!"
   echo "Используйте -$help."
   exit $E_NO_OPTS
fi


# +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# добавлено добавлено добавлено добавлено добавлено добавлено добавлено

if [ ! -e "$Config" ]; then   # Смотрим, существует ли файл конфигурации.
   echo "Создание конфигурационного файла, $Config"
   echo "# Это файл конфигурации для wgetter2" > "$Config"
   echo "# Ваши личные параметры будут сохранены в этом файле" >> "$Config"
else
   source $Config             #  Импорт переменных, установленных
                              #+ нами вне сценария.
fi

if [ ! -e "$Cookie_List" ]; then
   # Настройка списка файлов cookie, если нет ни одного.
   echo "Собираем cookie ..."
   find -name cookies.txt >> $Cookie_List # Создаем список файлов cookie.
fi #  Изолируйте у себя 'if',
   #+ в случае, если поиск прерывается.

if [ -z "$cFlag" ]; then # Если мы еще не сделали этого...
   echo                  # Создаем пространство, после приглашения
                         #+ командной строки.
   echo "Похоже, вы еще не настроили источник cookie."
   n=0                   #  Убедитесь, что счетчик не содержит
                         #+ случайных значений.
   while read; do
      Cookies[$n]=$REPLY # Помещаем найденные файлы cookie в массив.
      echo "$n) ${Cookies[$n]}"  # Создаем меню.
      n=$(( n + 1 ))     # Увеличиваем счетчик.
   done < $Cookie_List   # Читаем поток.
   echo "Для использования введите номер файла cookie."
   echo "Если не хотите использовать cookies, просто нажмите RETURN."
   echo
   echo "Я не буду спрашивать снова. Редактируем $Config"
   echo "Если решили изменить последние данные"
   echo "или использовать опцию -${cook}  для изменения сессии."
   read
   if [ ! -z $REPLY ]; then   # Пользователь не нажал return.
      Cookie=" --load-cookies ${Cookies[$REPLY]}"
      # Задаем переменную также, как и в конфигурационном файле.

      echo "Cookie=\" --load-cookies ${Cookies[$REPLY]}\"" >> $Config
   fi
   echo "cFlag=1" >> $Config  # Теперь знаем, это чтобы не запрашивать снова.
fi

# окончание секции добавлено окончание секции добавлено
```

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


# Другие переменные.
# Могут или не могут быть предметом изменения.
# Немного похоже на мелкий шрифт.
CookiesON=$Cookie
# echo "файл cookie это $CookiesON"     # Для отладки.
# echo "домашний это ${home}"           # Для отладки.


wopts()
{
echo "Введите опции для передачи в wget."
echo "Предполагается вы знаете, что делаете."
echo
echo "Можно так же здесь передать аргументы опции."
# То есть все помещенное сюда передается в wget.

read Wopts
# Читение опций передаваемых в wget.

Woptions=" $Wopts"
#         ^  Зачем начальный пробел?
# Назначение другой переменной.
# Просто для удовольствия, или ...

echo "опции ${Wopts} переданные в wget"
# В основном для отладки.
# Мило.

return
}


save_func()
{
echo "Настройки будут сохранены."
if [ ! -d $savePath ]; then  #  Проверяем существование директории.
   mkdir $savePath           #  Создаем директорию, если она не существовала,
                             #+ для сохранения.
fi

Flag=S
# Последняя часть кода предлагает выбрать действие.
# В основном надо установить флаг.

return
}


usage() # Рассказывает, как все это работает.
{
    echo "Добро пожаловать в wgetter. Это оболочка к wget."
    echo "Она будет всегда запускать wget с такими параметрами:"
    echo "$CommandA"
    echo "и шаблоном для сравнения: $pattern \
(который можно изменить в верху этого сценария)."
    echo "Будет всегда запрашивать глубину рекурсии, \
и, если хотите, использовать ссылки на страницу."
    echo "Wgetter доступны следующие опции:"
```

```
    echo ""
    echo "-$help : Вывод справки."
    echo "-$save : Сохранение команды в файл $savePath/wget-($today) \
вместо его запуска."
    echo "-$runn : Запуск сохраненных команд wget вместо запуска новых -"
    echo "Для этой опции введите имя файла как аргумент."
    echo "-$inpu : Интерактивный запуск сохраненных команд wget --"
    echo "Сценарий будет запрашивать у вас имя файла."
    echo "-$cook : Смена файла cookies для данной сессии."
    echo "-$list : Указывает wget использовать URL из списка, а не \
из командной строки."
    echo "-$wopt : Передает любые другие опции прямо в wget."
    echo ""
    echo "См. Справочные страницы wget о дополнительных опциях, \
которые можно передать в wget."
    echo ""

    exit $E_USAGE  # Здесь все. Все остальное не обрабатывается.
}


list_func() #  Задание опции пользователя с помощью опции -i в wget,
            #+ и список URL.
{
while [ 1 ]; do
   echo "Введите имя файла содержащего URL (нажмите q если
передумали)."
   read urlfile
   if [ ! -e "$urlfile" ] && [ "$urlfile" != q ]; then
       # Ищет файл, или опцию quit (выход).
       echo "Этот файл не существует!"
   elif [ "$urlfile" = q ]; then # Проверка опции quit.
       echo "Список url не будет использоваться."
       return
   else
      echo "используем $urlfile."
      echo "Если вы задали URL в командной строке, я сначала буду \
использовать ее."
                              # Стандартное сообщение wget пользователю.
      lister=" -i $urlfile" # Это то, что мы хотим передать в wget.
      return
   fi
done
}


cookie_func() #  Задание опции пользователя для использования
              #+ другого файла cookie.
{
while [ 1 ]; do
   echo "Меняем файл cookies. Нажмите return если не хотите менять"
   read Cookies
   # NB: это не те Cookie, что ранее.
   # На конце 's'.
   if [ -z "$Cookies" ]; then
      return
   elif [ ! -e "$Cookies" ]; then
      echo "Файл не существует. Попробуйте еще раз."
   else
       CookiesON=" --load-cookies $Cookies"   # файл есть -- используем!
       return
```

```
   fi
done
}


run_func()
{
if [ -z "$OPTARG" ]; then
# Проверка, чтобы увидеть: используется параметр командной строки или запрос.
   if [ ! -d "$savePath" ]; then      # Если директория не существует ...
      echo "$savePath не существует."
      echo "Пожалуйста укажите путь и файл сохраненных команд wget:"
      read newFile
         until [ -f "$newFile" ]; do  # Продолжаем до получения.
            echo "Извините, этот файл не существует. Попробуйте снова."
            # Попробуйте, действительно трудно получить что-то.
            read newFile
         done


# ---------------------------------------------------------------------------
#       if [ -z ( grep wget ${newfile} ) ]; then
        # Предположим,  нужный файл не получен.
#       echo "Извините, этот файл не содержит команд wget.  Сброс."
#       exit
#       fi
#
# Это фиктивный код.
# Он фактически не работает.
# Если хотите — исправьте его!
# ---------------------------------------------------------------------------


      filePath="${newFile}"
   else
   echo "Путь сохранения $savePath"
     echo "Введите имя файла который хотите использовать."
     echo "У вас есть выбор:"
     ls $savePath                                    # Задаем выбор.
     read inFile
       until [ -f "$savePath/$inFile" ]; do         #  Продолжаем до получения
                                                    #+ чего-нибудь.
          if [ ! -f "${savePath}/${inFile}" ]; then # Если файл не существует.
             echo "Этот файл не существует. Выберите из:"
             ls $savePath                           # Если допущена ошибка.
             read inFile
          fi
         done
      filePath="${savePath}/${inFile}"  # Создаем одну переменную ...
   fi
else filePath="${savePath}/${OPTARG}"   # Которая может быть любой ...
fi

if [ ! -f "$filePath" ]; then           # Если получено через фиктивный файл.
   echo "Вы не указали подходящий файл."
   echo "Запустите этот сценарий с первой опцией -${save}."
   echo "Сброс."
   exit $E_NO_SAVEFILE
fi
echo "Используйте: $filePath"
while read; do
```

```
    eval $REPLY
    echo "Завершено: $REPLY"
done < $filePath  # Поток указанного файла для использования в цикле 'while'.

exit
}



# Выуживаем любые опции, используемые сценарием.
# Это показано в "Learning The Bash Shell" (O'Reilly).
while getopts ":$save$cook$help$list$runn:$inpu$wopt" opt
do
  case $opt in
     $save) save_func;;   #  Сохранение некоторых сессий wgetter на будущее.
     $cook) cookie_func;; #  Смена файла cookie.
     $help) usage;;       #  Вызов справки.
     $list) list_func;;   #  Разрешить wget использовать список URL.
     $runn) run_func;;    #  Полезно, если wgetter вызывается,
                          #+ например, сценарием cron.
     $inpu) run_func;;    #  Когда не знаете, как называются ваши файлы.
     $wopt) wopts;;       #  Передача опций напрямую в wget.
        \?) echo "Не правильная опция."
            echo "Используйте -${wopt} для передачи опций прямо в wget,"
            echo "или -${help} для справки";;      # Что-нибудь дополнительно.
  esac
done
shift $((OPTIND - 1))     # Делаем волшебные вещи с $#.


if [ -z "$1" ] && [ -z "$lister" ]; then
                          #  Мы должны ввести, по крайней мере,
                          #+ один URL в командной строке,
                          #+ если не используется список.
   echo "Нет заданного URL! Вы должны ввести их на той же строке, что и
wgetter2.»
   echo "См.,  wgetter2 http://somesite http://anothersite."
   echo "Используйте опцию $help для информации."
   exit $E_NO_URLS        # Выход с соответствующим кодом ошибки.
fi

URLS=" $@"
# Список URL может быть изменен, если мы установим опцию in в цикле.

while [ 1 ]; do
   # Запрос наиболее часто используемых опций.
   # (Главным образом не измененных после версии 1 wgetter)
   if [ -z $curDepth ]; then
      Current=""
   else Current=" Текущее значение $curDepth"
   fi
       echo "Как глубоко зайдем? \
(Целое число: По умолчанию это $depthDefault.$Current)"
       read Depth   # Рекурсия — Как глубоко заходить?
       inputB=""    # Сброс пробела при каждом проходе цикла.
       echo "Введите имя ссылки на страницу ( по умолчанию — none)."
       read inputB  # Нужно для некоторых сайтов.

       echo "Хотите вывести журнал на терминал"
       echo "(y/n, по умолчанию yes)?"
       read noHide  # В противном случае Wget просто занесет его в файл.
```

```
      case $noHide in
         y|Y ) hide="";;
         n|N ) hide=" -b";;
           * ) hide="";;
      esac

      if [ -z ${Depth} ]; then
      #  Пользователь принял умолчание или текущую глубину,
      #+ в любом случае Depth пуста.
         if [ -z ${curDepth} ]; then
         #  Смотрим установленную глубину предыдущей итерации.
            Depth="$depthDefault"
            #  Устанавливаем глубину рекурсии по умолчанию,
            #+ если не надо ничего применять.
         else Depth="$curDepth" #  Иначе, остается предыдущая установка.
         fi
      fi
   Recurse=" -l $Depth"          # Устанавливаем желаемую глубину.
   curDepth=$Depth               # Запоминаем настройки.

      if [ ! -z $inputB ]; then
         RefA=" --referer=$inputB"  # Опция использующейся ссылки на страницу.
      fi

   WGETTER="${CommandA}${pattern}${hide}${RefA}${Recurse}\
${CookiesON}${lister}${Woptions}${URLS}"
   #  Только совместная целая строка ...
   #  NB: без пробелов.
   #  Она состоит из отдельных элементов, поэтому, если какие-либо из них
   #+ пустые, мы не получим дополнительных пробелов.

   if [ -z "${CookiesON}" ] && [ "$cFlag" = "1" ] ; then
       echo "Внимание — не возможно найти файл cookie"
       #  Может быть изменено,
       #+ в случае, если пользователь решил не использовать cookie.
   fi

   if [ "$Flag" = "S" ]; then
      echo "$WGETTER" >> $savePath/wget-${today}
      #  Создаем уникальное имя файла на сегодня, или добавляем в него,
      #+ если существует.
      echo "$inputB" >> $savePath/список сайтов-${today}
      #  Создаем список, очень легко вернуться назад,
      #+ поскольку вся команда немного запутанна.
      echo "Команда сохранена в файле $savePath/wget-${today}"
           # Сообщение пользователю.
      echo "Ссылка URL на страницу сохранена в файле file$ \
savePath/site-list-${today}"
           # Сообщение пользователю.
      Saver=" с сохранением опции"
   else
       echo "*****************"
       echo "*****Получение*****"
       echo "*****************"
       echo ""
       echo "$WGETTER"
       echo ""
       echo "*****************"
       eval "$WGETTER"
   fi
```

```
      echo ""
      echo "Запускается over$Saver."
      echo "Если желаете выйти, нажмите q."
      echo "в ином случае введите URL:"
      # Теперь снова. Сообщает об устанавливаемом варианте сохранения.

      read
      case $REPLY in
      # Нужно изменить положение «ловушки».
         q|Q ) exit $E_USER_EXIT;;  # Упражнение для читателя?
           * ) URLS=" $REPLY";;
      esac

      echo ""
done


exit 0
```

Пример A-31. Сценарий *podcasting*

```
#!/bin/bash

#  bashpodder.sh:
#  By Linc 10/1/2004
#  Ищите последнюю версию сценария на
#+ http://linc.homeunix.org:8080/scripts/bashpodder
#  Last revision 12/14/2004 - Many Contributors!
#  If you use this and have made improvements or have comments
#+ drop me an email at linc dot fessenden at gmail dot com
#  I'd appreciate it!

# ==>  Дополнительные комментарии ABS Guide.

# ==>  Автор этого сценария любезно предоставил право на включение в ABS Guide.

# ==> ################################################################
#
# ==> Что такое "podcasting"?

# ==> Это широковещательное "radio shows" через Internet.
# ==> Может воспроизводить музыкальные файлы на iPods и других  проигрывателях.

# ==> Сценарий создает эту возможность.
# ==> См. документацию на сайте автора сценария, выше.

# ==> ################################################################


# Создаем сценарий дружественный crontab:
cd $(dirname $0)
# ==> Переходим в директорию, где размещен этот сценарий.

# datadir это директория, куда вы будете сохранять подкасты:
datadir=$(date +%Y-%m-%d)
# ==> Мы создали директорию с именем на основе даты: YYYY-MM-DD
```

```
# Проверяем создание datadir если необходимо:
if test ! -d $datadir
        then
        mkdir $datadir
fi

# Удаляем все временные файлы:
rm -f temp.log

#  Считываем файл bp.conf и wget некоторые url не имеющиеся
#+ в файле podcast.log file:
while read podcast
  do # ==> Переход к основному действию.
  file=$(wget -q $podcast -O - | tr '\r' '\n' | tr \' \" | \
sed -n 's/.*url="\([^"]*\)".*/\1/p')
  for url in $file
                do
                echo $url >> temp.log
                if ! grep "$url" podcast.log > /dev/null
                        then
                        wget -q -P $datadir "$url"
                fi
                done
    done < bp.conf

# Перемещаем динамически созданный файл журнала в постоянный лог-файл:
cat podcast.log >> temp.log
sort temp.log | uniq > podcast.log
rm temp.log
# Создаем плэй-лист m3u:
ls $datadir | grep -v m3u > $datadir/podcast.m3u


exit 0
```

Пример A-32. Ночной бэкап firewire HD

```
#!/bin/bash
# nightly-backup.sh
# http://www.richardneill.org/source.php#nightly-backup-rsync
# Copyright (c) 2005 Richard Neill <backup@richardneill.org>.
# Лицензировано Free Software под GNU GPL.
# ==> Включено в ABS Guide с разрешения автора сценария.
# ==> (Спасибо!)

#  Создает резервную копию хост-компьютера на локально подключенный firewire
#+ HDD с помощью rsync и ssh.
#  (Сценарий будет работать с подключенным USB-устройством (см. строки 40-43).
#  Он обновляет резервные копии.
#  Запускается cron-ом каждую ночь в 5 часов утра.
#  Осуществляет резервное копирование только домашней директории.
#  Если следует сохранять права (кроме пользовательских), то запустите
#+ процесс rsync как root (а вновь устанавливаемые -o).
#  Сохраняем ежедневно в течение 7 дней, затем каждую неделю в течение 4
#+ недель, затем каждый месяц в течении 3-х месяцев.

#  См: http://www.mikerubel.org/computers/rsync_snapshots/
#  Сохраняем как: $HOME/bin/nightly-backup_firewire-hdd.sh
```

```
#  Известные баги:
#  ----------
#  i)  В идеале мы хотим исключить ~/.tmp и кэш браузера.

#  ii) Если пользователь находится за компьютером в 5 утра,
#+     а файлы в это время изменяются rsync,
#+     то срабатывает BACKUP_JUSTINCASE.
#      В некоторой степени это особенность, но
#+     она вызывает утечку дискового пространства.


##### НАЧАЛО КОНФИГУРАЦИОННОЙ СЕКЦИИ ########################################
LOCAL_USER=rjn                # Пользователь, чья домашняя директория
                              #+ архивируется.
MOUNT_POINT=/backup           # Точка монтирования диска.
                              # НЕ ОКАНЧИВАЕТСЯ слэшем!
                              # Должна быть уникальной (например
                              #+ используйте ссылки udev)
# MOUNT_POINT=/media/disk     # Для USB-оборудования.
SOURCE_DIR=/home/$LOCAL_USER  # НЕ ОКАНЧИВАЕТСЯ слэшем - это важно для rsync.
BACKUP_DEST_DIR=$MOUNT_POINT/backup/`hostname -s`.${LOCAL_USER}.nightly_backup
DRY_RUN=false                 # Если true, вызывается rsync с -n,
                              #+производя холодный запуск.
                              # Закомментируйте или установите false, для
                              #+ нормального использования.
VERBOSE=false                 # Если true, производится подробный rsync.
                              # Закомментируйте или установите false.
COMPRESS=false                # Если true, сжимается.
                              # Хорошо в интернете, плохо на LAN.
                              # Закомментируйте или установите false.

### Коды выхода ###
E_VARS_NOT_SET=64
E_COMMANDLINE=65
E_MOUNT_FAIL=70
E_NOSOURCEDIR=71
E_UNMOUNTED=72
E_BACKUP=73
##### ОКОНЧАНИЕ КОНФИГУРАЦИОННОЙ СЕКЦИИ ######################################


# Проверка установки всех важных переменных:
if [ -z "$LOCAL_USER" ] ||
   [ -z "$SOURCE_DIR" ] ||
   [ -z "$MOUNT_POINT" ]  ||
   [ -z "$BACKUP_DEST_DIR" ]
then
   echo 'Одна из переменных не установлена! Редактируйте файл: $0. BACKUP
FAILED.'
   exit $E_VARS_NOT_SET
fi

if [ "$#" != 0 ]  # Если параметры командной строки ...
then              # Here document(действие).
  cat <<-ENDOFTEXT
    Автоматический ночной бэкап запускаемый cron.
    Читаемый источник: $0
    Директория бэкап $BACKUP_DEST_DIR.
    Она будет создаваться по мере необходимости; инициализация уже не нужна.

    ВНИМАНИЕ: Содержимое $BACKUP_DEST_DIR будет изменяться.
```

```
    Директории именованные 'backup.\$i' будут в итоге УДАЛЕНЫ.
    Бэкап производится каждый день в течении 7 дней (1-8),
    каждую неделю в течении 4 недель (9-12),
    каждый месяц в течении 3 месяцев (13-15).

    Вы можете это добавить в ваш crontab, с помощью 'crontab -e'
    #  Бэкап файлов: $SOURCE_DIR в $BACKUP_DEST_DIR
    #+ каждую ночь в 3:15 утра
         15 03 * * * /home/$LOCAL_USER/bin/nightly-backup_firewire-hdd.sh

    Не забывайте проверять производство резервных копий, особенно если вы не
читаете почтовые сообщения cron!'
        ENDOFTEXT
   exit $E_COMMANDLINE
fi


# Анализируем опции.
# ==================

if [ "$DRY_RUN" == "true" ]; then
  DRY_RUN="-n"
  echo "ВНИМАНИЕ:"
  echo "ЭТО 'ХОЛОДНЫЙ ЗАПУСК'!"
  echo "Фактически никакие данные не будут переданы!"
else
  DRY_RUN=""
fi

if [ "$VERBOSE" == "true" ]; then
  VERBOSE="-v"
else
  VERBOSE=""
fi

if [ "$COMPRESS" == "true" ]; then
  COMPRESS="-z"
else
  COMPRESS=""
fi


#  Дополнительные резервные копии сохраняются каждую неделю
#+ (фактически 8 дней) и каждый месяц.
DAY_OF_MONTH=`date +%d`            # День месяца (01..31).
if [ $DAY_OF_MONTH = 01 ]; then    # Первый месяц.
  MONTHSTART=true
elif [ $DAY_OF_MONTH = 08 \
    -o $DAY_OF_MONTH = 16 \
    -o $DAY_OF_MONTH = 24 ]; then
    # Дни 8,16,24  (используем 8, а не 7 для обработки 31-дневных месяцев)
      WEEKSTART=true
fi


#  Проверяем, что HDD примонтирован.
#  По крайней мере, проверьте, что *что-то* примонтировано!
#  Мы можем использовать какое-то уникальное устройства, вместо предположенных
#+ scsi-id при наличии соответствующего правила udev в
#+ /etc/udev/rules.d/10-rules.local
#+ и внеся соответствующую запись в /etc/fstab.
```

```
#  Например: такое правило udev :
# BUS="scsi", KERNEL="sd*", SYSFS{vendor}="WDC WD16",
# SYSFS{model}="00JB-00GVA0     ", NAME="%k", SYMLINK="lacie_1394d%n"

if mount | grep $MOUNT_POINT >/dev/null; then
  echo "Точка монтирования $MOUNT_POINT действительно примонтирована. OK"
else
  echo -n "Попытка примонтировать $MOUNT_POINT..."
           # Если не примонтирована, пробуем примонтировать.
  sudo mount $MOUNT_POINT 2>/dev/null

  if mount | grep $MOUNT_POINT >/dev/null; then
    UNMOUNT_LATER=TRUE
    echo "OK"
    #  Примечание: Если мы преждевременно выходим с ошибкой, то
    #+ убедитесь, что отмонтирована.
  else
    echo "НЕУДАЧА"
    echo -e "Ничего не примонтировано к $MOUNT_POINT. БЭКАП НЕУДАЧЕН!"
    exit $E_MOUNT_FAIL
  fi
fi


# Проверка, что исходная директория существует и читабельна.
if [ ! -r  $SOURCE_DIR ] ; then
  echo "$SOURCE_DIR не существует, или не читается. BACKUP FAILED."
  exit $E_NOSOURCEDIR
fi


# Убедитесь, что структура директории бэкапа правильна.
# Если нет - создаем.
# Создаем поддиректории.
# Обратите внимание, что backup.0 создается rsync по мере необходимости.

for ((i=1;i<=15;i++)); do
  if [ ! -d $BACKUP_DEST_DIR/backup.$i ]; then
    if /bin/mkdir -p $BACKUP_DEST_DIR/backup.$i ; then
    #  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  Нет [ ] скобок проверки.
    #+                                                   Почему?
      echo "Внимание: директория $BACKUP_DEST_DIR/backup.$i отсутствует,"
      echo "или не инициализирована. (Пере-)Создайте ее."
    else
      echo "ОШИБКА: директория $BACKUP_DEST_DIR/backup.$i"
      echo "отсутствует или не была создана."
    if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
        # Прежде, чем выйти, отмонтируйтесь от точки монтирования.
        cd
        sudo umount $MOUNT_POINT &&
        echo "$MOUNT_POINT снова отмонтирована. Отказ."
    fi
      exit $E_UNMOUNTED
  fi
fi
done


#  Устанавливаем права доступа 700 для обеспечения
#+ безопасности прав на другой многопользовательской системе.
if ! /bin/chmod 700 $BACKUP_DEST_DIR ; then
```

```
  echo "ОШИБКА: Не установлены права доступа $BACKUP_DEST_DIR в 700."

  if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
  # Прежде, чем выйти, отмонтируйте от точки монтирования.
     cd ; sudo umount $MOUNT_POINT \
     && echo "$MOUNT_POINT снова отмонтирована. Отказ."
  fi

  exit $E_UNMOUNTED
fi

# Создаем ссылку: current -> backup.1 если нужно.
# Неудача здесь не критична.
cd $BACKUP_DEST_DIR
if [ ! -h current ] ; then
  if ! /bin/ln -s backup.1 current ; then
    echo "ВНИМАНИЕ: не создана ссылка current -> backup.1"
  fi
fi


# Теперь делаем rsync.
echo "Теперь производим бэкап rsync..."
echo "Исходная директория: $SOURCE_DIR"
echo -e "Директория назначения бэкапа: $BACKUP_DEST_DIR\n"


/usr/bin/rsync $DRY_RUN $VERBOSE -a -S --delete --modify-window=60 \
--link-dest=../backup.1 $SOURCE_DIR $BACKUP_DEST_DIR/backup.0/

#  Если rsync ошибся, то только предупреждение, а не выход,
#+ так как это могут быть незначительные проблемы.
#  Например, если один файл не читаемый, rsync ошибется.
#  Это не должно препятствовать обновлению.
#  Не используйте, например, `date +%a`, поскольку директории просто
#+ полны ссылками и съедают *так много* пространства.

if [ $? != 0 ]; then
  BACKUP_JUSTINCASE=backup.`date +%F_%T`.justincase
  echo "ВНИМАНИЕ: процесс rsync  полностью не совершен."
  echo "Возможно где-то ошибка."
  echo "Сохранение дополнительной копии: $BACKUP_JUSTINCASE"
  echo "ВНИМАНИЕ: Если это происходит регулярно, будет задействовано много"
  echo "пространства, даже несмотря на то, что это просто жесткие ссылки!"
fi

# Сохраняем readme в родительской директории бэкап.
# Сохранено в последней поддиректории.
echo "Бэкап $SOURCE_DIR на `hostname` последней запущенной \
`date`" > $BACKUP_DEST_DIR/README.txt
echo "Бэкап $SOURCE_DIR на `hostname` созданный \
`date`" > $BACKUP_DEST_DIR/backup.0/README.txt

# Если мы не находимся в холодном запуске, дополняем бэкапы.
[ -z "$DRY_RUN" ] &&

  #  Проверка заполненности диска бэкапа.
  #  Предупреждение, если занято если 90%. Если 98% или более, вероятен провал,
  #+ поэтому отказ.
  #  (Примечание: df дает вывод более, чем одной строки.)
```

```
 #  Проверяем это перед rsync, возможно, есть шанс.
 DISK_FULL_PERCENT=`/bin/df $BACKUP_DEST_DIR |
 tr "\n" ' ' | awk '{print $12}' | grep -oE [0-9]+ `
 echo "Проверка дискового пространства раздела бэкап \
 $MOUNT_POINT $DISK_FULL_PERCENT% full."
 if [ $DISK_FULL_PERCENT -gt 90 ]; then
   echo "Внимание: Диск заполнен более чем на 90%."
 fi
 if [ $DISK_FULL_PERCENT -gt 98 ]; then
   echo "Ошибка: Диск заполнен! Откат."
     if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
       # Прежде, чем выйти, отмонтируйте от точки монтирования.
       cd; sudo umount $MOUNT_POINT &&
       echo "$MOUNT_POINT снова отмонтирован. Откат."
     fi
   exit $E_UNMOUNTED
 fi


# Создание дополнительного бэкапа.
# Если копирование неудачно, откат.
if [ -n "$BACKUP_JUSTINCASE" ]; then
  if ! /bin/cp -al $BACKUP_DEST_DIR/backup.0 \
     $BACKUP_DEST_DIR/$BACKUP_JUSTINCASE
  then
    echo "ОШИБКА: Создание дополнительной копии неудачно \
    $BACKUP_DEST_DIR/$BACKUP_JUSTINCASE"
    if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
      # Прежде, чем выйти, отмонтируйте от точки монтирования.
      cd ;sudo umount $MOUNT_POINT &&
      echo "$MOUNT_POINT снова отмонтирована. Откат."
    fi
    exit $E_UNMOUNTED
  fi
fi


# Начало месяца, перезапись старее 8.
if [ "$MONTHSTART" == "true" ]; then
  echo -e "\nStart of month. \
  Удаление старых бэкапов: $BACKUP_DEST_DIR/backup.15"  &&
  /bin/rm -rf  $BACKUP_DEST_DIR/backup.15  &&
  echo "Перезапись ежемесячных,еженедельных бэкапов: \
  $BACKUP_DEST_DIR/backup.[8-14] -> $BACKUP_DEST_DIR/backup.[9-15]"  &&
    /bin/mv $BACKUP_DEST_DIR/backup.14 $BACKUP_DEST_DIR/backup.15  &&
    /bin/mv $BACKUP_DEST_DIR/backup.13 $BACKUP_DEST_DIR/backup.14  &&
    /bin/mv $BACKUP_DEST_DIR/backup.12 $BACKUP_DEST_DIR/backup.13  &&
    /bin/mv $BACKUP_DEST_DIR/backup.11 $BACKUP_DEST_DIR/backup.12  &&
    /bin/mv $BACKUP_DEST_DIR/backup.10 $BACKUP_DEST_DIR/backup.11  &&
    /bin/mv $BACKUP_DEST_DIR/backup.9 $BACKUP_DEST_DIR/backup.10  &&
    /bin/mv $BACKUP_DEST_DIR/backup.8 $BACKUP_DEST_DIR/backup.9

# Начало недели, перезапись старее 4.
elif [ "$WEEKSTART" == "true" ]; then
  echo -e "\nНачало недели. \
  Удаление старых еженедельных бэкапов: $BACKUP_DEST_DIR/backup.12"  &&
  /bin/rm -rf  $BACKUP_DEST_DIR/backup.12  &&

  echo "Перезапись еженедельных бэкапов: \
  $BACKUP_DEST_DIR/backup.[8-11] -> $BACKUP_DEST_DIR/backup.[9-12]"  &&
    /bin/mv $BACKUP_DEST_DIR/backup.11 $BACKUP_DEST_DIR/backup.12  &&
```

```
    /bin/mv $BACKUP_DEST_DIR/backup.10 $BACKUP_DEST_DIR/backup.11  &&
    /bin/mv $BACKUP_DEST_DIR/backup.9 $BACKUP_DEST_DIR/backup.10  &&
    /bin/mv $BACKUP_DEST_DIR/backup.8 $BACKUP_DEST_DIR/backup.9

 else
  echo -e "\nУдаление старых ежедневных бэкапов: $BACKUP_DEST_DIR/backup.8"  &&
    /bin/rm -rf  $BACKUP_DEST_DIR/backup.8

 fi  &&

 # Ежедневно, перезапись новыми 8.
 echo "Перезапись ежедневных бэкапов: \
 $BACKUP_DEST_DIR/backup.[1-7] -> $BACKUP_DEST_DIR/backup.[2-8]"  &&
    /bin/mv $BACKUP_DEST_DIR/backup.7 $BACKUP_DEST_DIR/backup.8  &&
    /bin/mv $BACKUP_DEST_DIR/backup.6 $BACKUP_DEST_DIR/backup.7  &&
    /bin/mv $BACKUP_DEST_DIR/backup.5 $BACKUP_DEST_DIR/backup.6  &&
    /bin/mv $BACKUP_DEST_DIR/backup.4 $BACKUP_DEST_DIR/backup.5  &&
    /bin/mv $BACKUP_DEST_DIR/backup.3 $BACKUP_DEST_DIR/backup.4  &&
    /bin/mv $BACKUP_DEST_DIR/backup.2 $BACKUP_DEST_DIR/backup.3  &&
    /bin/mv $BACKUP_DEST_DIR/backup.1 $BACKUP_DEST_DIR/backup.2  &&
    /bin/mv $BACKUP_DEST_DIR/backup.0 $BACKUP_DEST_DIR/backup.1  &&

 SUCCESS=true


if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
  #  Отмонтирование от точки монтирования, если вначале она не была
  #+ примонтирована.
  cd ; sudo umount $MOUNT_POINT && echo "$MOUNT_POINT Снова отмонтирована."
fi


if [ "$SUCCESS" == "true" ]; then
  echo 'SUCCESS!'
  exit 0
fi

# Выход, если резервное копирование работает.
echo 'БЭКАП НЕУДАЛСЯ! Холодный запуск? Заполнен диск?) '
exit $E_BACKUP
```

Пример A-33. Расширение команды *cd*

```
###############################################################################
#
#       cdll
#       by Phil Braham
#
#       ############################################
#       Latest version of this script available from
#       http://freshmeat.net/projects/cd/
#       ############################################
#
#       .cd_new
#
#       Расширение Unix команды cd
```

```
#
#       Существует неограниченный стек записей и стек записи событий.
#       Стек записей сохраняет последние директории cd_maxhistory,
#       которые использовались. Записи событий могут быть назначены
#       часто используемым директориям.
#
#       Записи событий назначаются заранее, установкой переменной
#       среды CDSn или с помощью команды -u или – U.
#
#       Ниже приводится файл .profile:
#
#               . cdll              #  Настройка команды cd
#       alias cd='cd_new'           #  Замена команды cd
#               cd -U               #  Загрузка предварительно назначенных
#                                   #+ записей стека и записей событий
#               cd -D               #  Установка режима non-default
#               alias @="cd_new @"  #  Позволяет использовать @ для получения
#                                   #+ истории
#
#       Для справки:
#
#               cd -h или
#               cd -H
#
#
###############################################################################
#
#       Version 1.2.1
#
#       Автор Phil Braham - Realtime Software Pty Ltd
#       (realtime@mpx.com.au)
#       Please send any suggestions or enhancements to the author (also at
#       phil@braham.net)
#
###############################################################################

cd_hm ()
{
        ${PRINTF} "%s" "cd [dir] [0-9] [@[s|h] [-g [<dir>]] [-d] \
[-D] [-r<n>] [dir|0-9] [-R<n>] [<dir>|0-9]
   [-s<n>] [-S<n>] [-u] [-U] [-f] [-F] [-h] [-H] [-v]
    <dir>       Переход в директорию
    0-n         Переход в предыдущую директорию (0 предыдущая, 1 следующая
                перед ней, и т.д.)
                n это вершина максимума записи историй (умолчание 50)
    @           Список записей истории и записей событий
    @h          Список записей истории
    @s          Список записей событий
    -g [<dir>]  Переход к буквальному имени (обходя специальные имена)
                Дает доступ к директориям с именами '0','1','-h' и т.д.
    -d          Изменение действия по умолчанию - подробно. (См. примечание)
    -D          Изменение действия по умолчанию - просто. (См. примечание)
    -s<n>       Переход к записи события <n>*
    -S<n>       Переход к записи события <n> и изменение ее текущей dir*

    -r<n> [<dir>] Переход в директорию <dir>, а затем вложение в нее
                  записи событий <n>*
    -R<n> [<dir>] Переход в директорию <dir> и ввод событий <n>* в
                  текущую директорию

    -a<n>       Альтернативно предлагаемая директория. См. примечание ниже.
```

```
    -f [<file>] Содержимое файла <file>.
    -u [<file>] Обновление записи в <file>.
                Если имя файла не указано, то по умолчанию используется файл
                вида (${CDPath}${2:-"$CDFile"}) с -F и -U
    -v          Вывод номера версии
    -h          Справка
    -H          Подробная справка

    *Запись событий (0 - 9) ведется до тех пор, пока не заменяется другими
     записями или не обновляется командой -u

    Альтернативно предлагаемые директории:
    Если директория не найдена, то cd будет предлагать другие. Т.е. директории
    начинающиеся с той же буквы, и если таковые будут найдены - они
    перечисляются по префиксам  -a<n>, где <n> это номер
    Это позволяет перейти в директорию, введя в командной строке cd -a<n>

    Директории -r<n> или -R<n> могут обозначаться числом.
    Например:
        $ cd -r3 4  Перейти к записи истории 4 и вложить ее в
                    запись события 3
        $ cd -R3 4  Вложить в текущую директорию записи события 3 и
                    перейти к записи истории 4
        $ cd -s3    Перейти к записи события 3

    Обратите внимание, что команды R,r,S и s могут использоваться без числа
    и т.е. Будут соответствовать 0:
        $ cd -s     Перейти к записи события 0
        $ cd -S     Перейти к записи события 0 и
                    записать события 0 в текущую директорию
        $ cd -r 1   Перейти к записи истории 1 и вложить в нее
                    записи события 0
        $ cd -r     Перейти к записи истории 0 и вложить в нее
                    записи события 0

        if ${TEST} "$CD_MODE" = "PREV"
        then
                ${PRINTF} "$cd_mnset"
        else
                ${PRINTF} "$cd_mset"
        fi
}
"
cd_Hm ()
{
        cd_hm
        ${PRINTF} "%s" "
        Предыдущая директория (0-$cd_maxhistory), сохраненная
        переменными среды CD[0] - CD[$cd_maxhistory]
        Подобна директориям событий S0 - $cd_maxspecial в
        переменной среды CDS[0] - CDS[$cd_maxspecial]
        и может быть доступна из командной строки

        По умолчанию путем к имени файла для команд -f и -u  является $CDPath
        По умолчанию именем файла для команд -f и -u является $CDFile

        Устанавливаются следующие переменные среды:
            CDL_PROMPTLEN  - Устанавливает требуемый размер строки приглашения.
                Строка приглашения находится справа от символов текущей
                директории.
                Если не установлена, то приглашение остается слева.
```

```
            CDL_PROMPT_PRE - Устанавливает в строке префикс приглашения.
                По умолчанию:
                    не-root:  \"\\[\\e[01;34m\\]\"  (синий цвет).
                    root:      \"\\[\\e[01;31m\\]\"  (красный цвет).
            CDL_PROMPT_POST    - Устанавливает в строке суффикс строки
                                 приглашения.
                По умолчанию:
                    non-root:  \"\\[\\e[00m\\]$\"
                                (сбрасывает цвет и выводит $).
                    root:      \"\\[\\e[00m\\]#\"
                                (сбрасывает цвет и выводит #).
            CDPath – Установка пути по умолчанию для опций -f и -u.
                     Умолчание – домашняя директория
            CDFile – Установка по умолчанию имени файла для опций -f и -u.
                     Умолчание cdfile

"
    cd_version

}

cd_version ()
{
 printf "Версия: ${VERSION_MAJOR}.${VERSION_MINOR} Дата: ${VERSION_DATE}\n"
}

#
# Обрезается справа.
#
# параметры:
#   p1 - строка
#   p2 – размер обрезания
#
# возвращает строку в tcd
#
cd_right_trunc ()
{
    local tlen=${2}
    local plen=${#1}
    local str="${1}"
    local diff
    local filler="<--"
    if ${TEST} ${plen} -le ${tlen}
    then
        tcd="${str}"
    else
        let diff=${plen}-${tlen}
        elen=3
        if ${TEST} ${diff} -le 2
        then
            let elen=${diff}
        fi
        tlen=-${tlen}
        let tlen=${tlen}+${elen}
        tcd=${filler:0:elen}${str:tlen}
    fi
}

#
# Три версии производства истории:
#    cd_dohistory  - Пакеты истории и событий шаг за шагом
```

```
#    cd_dohistoryH – Показать только историю
#    cd_dohistoryS – Показать только события
#
cd_dohistory ()
{
    cd_getrc
        ${PRINTF} "История:\n"
    local -i count=${cd_histcount}
    while ${TEST} ${count} -ge 0
    do
        cd_right_trunc "${CD[count]}" ${cd_lchar}
            ${PRINTF} "%2d %-${cd_lchar}.${cd_lchar}s " ${count} "${tcd}"

        cd_right_trunc "${CDS[count]}" ${cd_rchar}
            ${PRINTF} "S%d %-${cd_rchar}.${cd_rchar}s\n" ${count} "${tcd}"
        count=${count}-1
    done
}

cd_dohistoryH ()
{
    cd_getrc
        ${PRINTF} "История:\n"
        local -i count=${cd_maxhistory}
        while ${TEST} ${count} -ge 0
        do
          ${PRINTF} "${count} %-${cd_flchar}.${cd_flchar}s\n" ${CD[$count]}
          count=${count}-1
        done
}

cd_dohistoryS ()
{
    cd_getrc
        ${PRINTF} "События:\n"
        local -i count=${cd_maxspecial}
        while ${TEST} ${count} -ge 0
        do
          ${PRINTF} "S${count} %-${cd_flchar}.${cd_flchar}s\n" ${CDS[$count]}
          count=${count}-1
        done
}

cd_getrc ()
{
    cd_flchar=$(stty -a | awk -F \;
    '/rows/ { print $2 $3 }' | awk -F \  '{ print $4 }')
    if ${TEST} ${cd_flchar} -ne 0
    then
        cd_lchar=${cd_flchar}/2-5
        cd_rchar=${cd_flchar}/2-5
            cd_flchar=${cd_flchar}-5
    else
            cd_flchar=${FLCHAR:=75}
            # cd_flchar использует for для историй @s & @h
            cd_lchar=${LCHAR:=35}
            cd_rchar=${RCHAR:=35}
    fi
}

cd_doselection ()
```

```
{
        local -i nm=0
        cd_doflag="TRUE"
        if ${TEST} "${CD_MODE}" = "PREV"
        then
                if ${TEST} -z "$cd_npwd"
                then
                        cd_npwd=0
                fi
        fi
        tm=$(echo "${cd_npwd}" | cut -b 1)
    if ${TEST} "${tm}" = "-"
    then
        pm=$(echo "${cd_npwd}" | cut -b 2)
        nm=$(echo "${cd_npwd}" | cut -d $pm -f2)
        case "${pm}" in
             a) cd_npwd=${cd_sugg[$nm]} ;;
             s) cd_npwd="${CDS[$nm]}" ;;
             S) cd_npwd="${CDS[$nm]}" ; CDS[$nm]=`pwd` ;;
             r) cd_npwd="$2" ; cd_specDir=$nm ; cd_doselection "$1" "$2";;
             R) cd_npwd="$2" ; CDS[$nm]=`pwd` ; cd_doselection "$1" "$2";;
        esac
    fi

    if ${TEST} "${cd_npwd}" != "." -a "${cd_npwd}" \
!= ".." -a "${cd_npwd}" -le ${cd_maxhistory} >>/dev/null 2>&1
    then
      cd_npwd=${CD[$cd_npwd]}
     else
       case "$cd_npwd" in
                @)  cd_dohistory ; cd_doflag="FALSE" ;;
               @h) cd_dohistoryH ; cd_doflag="FALSE" ;;
               @s) cd_dohistoryS ; cd_doflag="FALSE" ;;
               -h) cd_hm ; cd_doflag="FALSE" ;;
               -H) cd_Hm ; cd_doflag="FALSE" ;;
               -f) cd_fsave "SHOW" $2 ; cd_doflag="FALSE" ;;
               -u) cd_upload "SHOW" $2 ; cd_doflag="FALSE" ;;
               -F) cd_fsave "NOSHOW" $2 ; cd_doflag="FALSE" ;;
               -U) cd_upload "NOSHOW" $2 ; cd_doflag="FALSE" ;;
               -g) cd_npwd="$2" ;;
               -d) cd_chdefm 1; cd_doflag="FALSE" ;;
               -D) cd_chdefm 0; cd_doflag="FALSE" ;;
               -r) cd_npwd="$2" ; cd_specDir=0 ; cd_doselection "$1" "$2";;
               -R) cd_npwd="$2" ; CDS[0]=`pwd` ; cd_doselection "$1" "$2";;
               -s) cd_npwd="${CDS[0]}" ;;
               -S) cd_npwd="${CDS[0]}"  ; CDS[0]=`pwd` ;;
               -v) cd_version ; cd_doflag="FALSE";;
       esac
    fi
}

cd_chdefm ()
{
        if ${TEST} "${CD_MODE}" = "PREV"
        then
                CD_MODE=""
                if ${TEST} $1 -eq 1
                then
                        ${PRINTF} "${cd_mset}"
                fi
        else
```

```
                CD_MODE="PREV"
                if ${TEST} $1 -eq 1
                then
                        ${PRINTF} "${cd_mnset}"
                fi
        fi
}

cd_fsave ()
{
        local sfile=${CDPath}${2:-"$CDFile"}
        if ${TEST} "$1" = "SHOW"
        then
                ${PRINTF} "Сохранено в %s\n" $sfile
        fi
        ${RM} -f ${sfile}
        local -i count=0
        while ${TEST} ${count} -le ${cd_maxhistory}
        do
                echo "CD[$count]=\"${CD[$count]}\"" >> ${sfile}
                count=${count}+1
        done
        count=0
        while ${TEST} ${count} -le ${cd_maxspecial}
        do
                echo "CDS[$count]=\"${CDS[$count]}\"" >> ${sfile}
                count=${count}+1
        done
}

cd_upload ()
{
        local sfile=${CDPath}${2:-"$CDFile"}
        if ${TEST} "${1}" = "SHOW"
        then
                ${PRINTF} "Загрузка из %s\n" ${sfile}
        fi
        . ${sfile}
}

cd_new ()
{
    local -i count
    local -i choose=0

        cd_npwd="${1}"
        cd_specDir=-1
        cd_doselection "${1}" "${2}"

        if ${TEST} ${cd_doflag} = "TRUE"
        then
                if ${TEST} "${CD[0]}" != "`pwd`"
                then
                        count=$cd_maxhistory
                        while ${TEST} $count -gt 0
                        do
                                CD[$count]=${CD[$count-1]}
                                count=${count}-1
                        done
                        CD[0]=`pwd`
                fi
```

```
                command cd "${cd_npwd}" 2>/dev/null
        if ${TEST} $? -eq 1
        then
            ${PRINTF} "Неизвестная директория: %s\n" "${cd_npwd}"
            local -i ftflag=0
            for i in "${cd_npwd}"*
            do
                if ${TEST} -d "${i}"
                then
                    if ${TEST} ${ftflag} -eq 0
                    then
                        ${PRINTF} "Предлагается:\n"
                        ftflag=1
                    fi
                    ${PRINTF} "\t-a${choose} %s\n" "$i"
                                        cd_sugg[$choose]="${i}"
                    choose=${choose}+1
                fi
            done
        fi
        fi

        if ${TEST} ${cd_specDir} -ne -1
        then
                CDS[${cd_specDir}]=`pwd`
        fi

        if ${TEST} ! -z "${CDL_PROMPTLEN}"
        then
        cd_right_trunc "${PWD}" ${CDL_PROMPTLEN}
            cd_rp=${CDL_PROMPT_PRE}${tcd}${CDL_PROMPT_POST}
                export PS1="$(echo -ne ${cd_rp})"
        fi
}
###############################################################################
#                                                                             #
#                        Инициализация                                        #
#                                                                             #
###############################################################################
#
VERSION_MAJOR="1"
VERSION_MINOR="2.1"
VERSION_DATE="24-MAY-2003"
#
alias cd=cd_new
#
# Устанавливаем команды
RM=/bin/rm
TEST=test
PRINTF=printf              # Используем встроенный printf

###############################################################################
#                                                                             #
# Измените, чтобы изменить умолчание до и после строки приглашения.           #
# Вступит в силу при установке CDL_PROMPTLEN.                                 #
#                                                                             #
###############################################################################
if ${TEST} ${EUID} -eq 0
then
#   CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="$HOSTNAME@"}
    CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="\\[\\e[01;31m\\]"}  # Root - красный
```

```
    CDL_PROMPT_POST=${CDL_PROMPT_POST:="\\[\\e[00m\\]#"}
else
    CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="\\[\\e[01;34m\\]"}  # Юзеры - синий
    CDL_PROMPT_POST=${CDL_PROMPT_POST:="\\[\\e[00m\\]$"}
fi
##############################################################################
#
# cd_maxhistory определяет максимальное количество записей истории.
typeset -i cd_maxhistory=50

##############################################################################
#
# cd_maxspecial определяет максимальное количество записи событий.
typeset -i cd_maxspecial=9
#
#
##############################################################################
#
#  cd_histcount определяет число записей, выводимых на экран,
#+ истории команд.
typeset -i cd_histcount=9
#
##############################################################################
export CDPath=${HOME}/
#  Измените,  если используется другие                                      #
#+ путь и имя файла                                                        #
export CDFile=${CDFILE:=cdfile}           # для команд -u и -f             #
#
##############################################################################
                                                                           #
typeset -i cd_lchar cd_rchar cd_flchar
                        #  Количество символов для cd_flchar               #
cd_flchar=${FLCHAR:=75} #+ используемых для записей @s и @h                #

typeset -ax CD CDS
#
cd_mset="\n\tУстановлен режим по умолчанию - вводится cd без параметров \
имеющих действие по умолчанию\n\Используйте cd -d или -D для перехода cd \
в предыдущую директорию без параметров\n"
cd_mnset="\n\tУстановлен режим не по умолчанию - ввод cd без \
параметров будет равнозначен вводу cd 0\n\Используйте cd -d или \
-D для изменения действия по молчанию cd\n"

# ============================================================== #



: <<DOCUMENTATION

Автор Phil Braham. Realtime Software Pty Ltd.
Released under GNU license. Free to use. Please pass any modifications
or comments to the author Phil Braham:

realtime@mpx.com.au
======================================================================

cdll это замена для cd, включающая в себя аналогичные функциональные
возможности команд bash pushd и popd, но независима от них.

Эта версия cdll была протестирована на Linux с помощью Bash. Будет работать
большинстве версий Linux, но вероятно не будет работать в других оболочках без
```

```
изменения.

Введение
============

cdll позволяет легко перемещаться между директориями. При переходе в новую
директорию текущая директория автоматически помещается в стек. По умолчанию
хранятся 50 записей, но это можно изменять. Директории событий, сохраняемые для
быстрого доступа - по умолчанию 10, но можно изменять. Самые последние записи
событий в стеке могут быть быстро просмотрены.

Стек директорий и записей событий можно сохранять и загружать из файла. Это
позволяет создавать для них введенное имя, сохраняемое перед выходом или
изменять его при переходе к другому проекту.

Кроме того cdll предоставляет гибкость приглашения команд, позволяющую,
например, выделять имя директории цветом, которое усекается слева, если будет
слишком длинным.


Установка cdll
===============

Скопируйте cdll в вашу локальную домашнюю директорию или в основную директорию,
например в /usr/bin (потребуется доступ с правами root).

Скопируйте файл cdfile в вашу домашнюю директорию. Потребуются права доступа
для чтения и записи. Этот файл по умолчанию, содержит директорию стека и записи
событий.

Для замены команды cd необходимо добавить команды в сценарий входа в систему.
Сценарием входа является одна из:

    /etc/profile
    ~/.bash_profile
    ~/.bash_login
    ~/.profile
    ~/.bashrc
    /etc/bash.bashrc.local

Установите Ваш логин, рекомендуется в ~/.bashrc, глобальные (и root) установки
добавляются командами в /etc/bash.bashrc.local

Установка логина, добавляется командой:
    . <dir>/cdll
Например, если cdll находится в вашей локальной домашней директории:
    . ~/cdll
Если в /usr/bin то:
    . /usr/bin/cdll

Если вы хотите использовать ее вместо встроенной команды cd, то добавьте:
    alias cd='cd_new'
Мы также рекомендуем следующие команды:
    alias @='cd_new @'
    cd -U
    cd -D

Если вы хотите использовать быстрое приглашение cdll, то добавьте следующее:
    CDL_PROMPTLEN=nn
Где nn это число описываемое ниже. Назначение 99 будет вполне подходящим
числом.
```

Таким образом, сценарий выглядит примерно так:

```
    ##########################################################################
    # Настройка CD
    ##########################################################################
    CDL_PROMPTLEN=21        # Допускает строку длиной до 21 символа
    . /usr/bin/cdll         # Инициализация cdll
    alias cd='cd_new'       # Замена встроенной команды cd
    alias @='cd_new @'      # Допускает @ в командной строке для вывода истории
    cd -U                   # Загрузка директорий
    cd -D                   # Задает действие по умолчанию для не-posix
    ##########################################################################
```

Смысл этих команд станет ясным позже.

Есть несколько предостережений. Если другая программа изменяет директорию без вызова cdll, то директория не попадет в стек и, если используется объект строки, она не будет обновлена. Две программы, которые могут это сделать - pushd и popd. Чтобы обновить строку и стек, просто введите:

```
    cd .
```

Обратите внимание, что если предыдущая запись стека является текущей директорией, то стек не обновляется.

Использование
=====

```
cd [dir] [0-9] [@[s|h] [-g <dir>] [-d] [-D] [-r<n>]
   [dir|0-9] [-R<n>] [<dir>|0-9] [-s<n>] [-S<n>]
   [-u] [-U] [-f] [-F] [-h] [-H] [-v]

    <dir>       Переход в директорию
    0-n         Переход в предыдущую директорию (0 это предыдущая,
                1 это следующая перед ней, и т.д.)
                n максимальное число историй (по умолчанию 50)
    @           Список записей истории и событий (Обычно доступных как $ @)
    @h          Список записей истории
    @s          Список записей событий
    -g [<dir>]  Переход к буквальному имени (обходя имена событий)
                Это позволит доступ к директориям названным '0','1','-h' и
                т.д.
    -d          Изменение действия по умолчанию - verbose. (См. примечание)
    -D          Изменение действия по умолчанию - silent. (См. примечание)
    -s<n>       Перейти к записи события <n>
    -S<n>       Перейти к записи события <n>
                и заменить ее текущей директорией
    -r<n> [<dir>] Перейти в директорию <dir>
                  и поместить ее в запись события <n>
    -R<n> [<dir>] Перейти в директорию <dir>
                  и поместить текущую директорию в запись события <n>
    -a<n>       Альтернативно предложенная директория. См. примечание ниже.
    -f [<file>] Записать файл в <file>.
    -u [<file>] Обновить записи из <file>.
                Если имя файла не указано, то по умолчанию файл (~/cdfile)
                используется silent версиями -F и -U
    -v          Вывод на экран номера версии
    -h          Справка
    -H          Подробная справка
```

```
Примеры
========

В этих примерах предполагается, что установлен режим по умолчанию (то есть, cd
без параметров будет переходить в последнюю директорию стека), что aliase были
созданы для cd и @ как описано выше и что строка приглашения cd активна и длина
строки составляет 21 символ.

    /home/phil$ @
    # Список записей @
    History:
    # Вывод команды @
    .....
    # Пропускаем эти записи для краткости
    1 /home/phil/ummdev               S1 /home/phil/perl
    # Выводятся две самые последние записи истории
    0 /home/phil/perl/eg              S0 /home/phil/umm/ummdev
    # и две записи событий

    /home/phil$ cd /home/phil/utils/Cdll
    # Теперь изменяем директории
    /home/phil/utils/Cdll$ @
    # Строка отображает директорию.
    History:
    # Новая история
    .....
    1 /home/phil/perl/eg              S1 /home/phil/perl
    # Запись истории 0 перемещена в 1
    0 /home/phil                      S0 /home/phil/umm/ummdev
    # и в самую последнюю запись

Переход к записи истории:

    /home/phil/utils/Cdll$ cd 1
    # Переход к записи истории 1.
    /home/phil/perl/eg$
    # Текущая директория теперь та, что была 1

Переход к записи события:

    /home/phil/perl/eg$ cd -s1
    # Переход к записи события 1
    /home/phil/umm/ummdev$
    # Текущая директория S1

Переход в указанную директорию, например 1:

    /home/phil$ cd -g 1
    # -g игнорирует значение события 1
    /home/phil/1$

Помещение текущей директории в список событий как S1:
    cd -r1 .        #  ИЛИ
    cd -R1 .        #  Тот же эффект, как если директория является
                    #+ . (текущей директорией)

Переход в директорию и добавление ее, как события
  Директория для -r<n> или -R<n> может быть числом.
  Например:
        $ cd -r3 4  Переход к записи истории 4 и помещение ее в запись
```

```
                    события 3
        $ cd -R3 4  Помещение текущей директории в запись события 3 и переход
                    к записи истории 4
        $ cd -s3    Переход к записи события 3

    Обратите внимание, что команды R,r,S и s могут использоваться без чисел и
    обращаться в 0:
        $ cd -s     Переход к записи события 0
        $ cd -S     Переход к записи события 0 и создание записи события 0
                    текущей директории
        $ cd -r 1   Переход к записи истории 1 и помещение ее в запись
                    события 0
        $ cd -r     Переход к записи истории 0 и помещение ее в запись
                    события 0


    Альтернативно предлагаемые директории:

    Если директория не найдена, то CD предложит другую
    возможность. Эти директории начинаются с каких-то букв
    и, если таковые будут найдены, они перечисляются префиксом с -a<n>
    где <n> это число. Это позволяет перейти в директорию, введя
    cd -a<n> в командной строке

        Используйте cd -d или -D для изменения действия cd по умолчанию.
        cd -H покажет текущее действие

        Записи истории (0-n) сохраняются в переменных окружения
        CD[0] - CD[n]
        Аналогичным образом директории событий S0 - 9 находятся в
        переменных окружения CDS[0] - CDS[9]
        и могут быть доступны из командной строки, например:

            ls -l ${CDS[3]}
            cat ${CD[8]}/file.txt

        По умолчанию путем для команд -f и -u является ~
        По умолчанию именем для команд -f и -u является cdfile


Конфигурация
=============

    Могут быть установлены следующие переменные окружения:

        CDL_PROMPTLEN  - Задает требуемую длину строки.
            Строка запроса устанавливается справа от символов текущей
            директории. Если не задана, то строка остается неизменной.
            Обратите внимание, что количество символов, до которых
            сокращается директория, не является общим количеством символов
            в строке.

            CDL_PROMPT_PRE - Установка строки в префикс запроса.
                Умолчание:
                    не-root:  "\\[\\e[01;34m\\]"  (установлено синим).
                    root:      "\\[\\e[01;31m\\]"  (установлено красным).

            CDL_PROMPT_POST    - Установка строки в суффикс запроса.
                По умолчанию:
                    non-root:  "\\[\\e[00m\\]$"
                               (меняет цвет и выводит на экран $).
```

```
                    root:      "\\[\\e[00m\\]#"
                               (меняет цвет и выводит на экран #).

        Примечание:
            CDL_PROMPT_PRE & _POST только t

        CDPath – Устанавливает путь по умолчанию для опций -f и -u.
                 По умолчанию – домашняя директория.
        CDFile – Устанавливает файл по умолчанию для опций -f и -u.
                 По умолчанию cdfile


    Существуют три переменные, определенные в файле cdll, которые
    управляют количеством сохраненных или отображаемых записей.
    Они находятся в разделе «Initialisation here» в конце файла.

        cd_maxhistory       - Число сохраненных записей истории.
                              По умолчанию 50.
        cd_maxspecial       - Число допустимых записей событий.
                              По умолчанию 9.
        cd_histcount        - Число выводимых на экран записей истории и
                              событий. По умолчанию 9.

    Обратите внимание, что cd_maxspecial может быть >= cd_histcount для
    избежания вывода записей событий, которые не были установлены.


Version: 1.2.1 Date: 24-MAY-2003

DOCUMENTATION
```

Пример A-34. Сценарий настройки звуковой карты

```
#!/bin/bash
# soundcard-on.sh

#  Автор: Mkarcher
#  http://www.thinkwiki.org/wiki  ...
#  /Script_for_configuring_the_CS4239_sound_chip_in_PnP_mode
#  Автор ABS Guide сделал небольшие изменения и добавил комментарии.
#  Couldn't contact script author to ask for permission to use, but ...
#+ the script was released under the FDL,
#+ so its use here should be both legal and ethical.

#  Сценарий-звука-через-pnp для Thinkpad 600E
#+ и других компьютеров с CS4239/CS4610
#+ которые не работают без драйверов PCI
#+ и не распознаются кодом PnP snd-cs4236.
#  Так же для других Thinkpad 770-серии, таких как 770x.
#  Запускается root, конечно.
#
#  Это старые и очень устаревшие портативные компьютеры,
#+ а этот сценарий является весьма поучительным,
#+ он показывает, как создавать и взламывать файлы устройств.
```

```
#  Поиск звуковой pnp карты:

for dev in /sys/bus/pnp/devices/*
do
  grep CSC0100 $dev/id > /dev/null && WSSDEV=$dev
  grep CSC0110 $dev/id > /dev/null && CTLDEV=$dev
done
# Для 770x:
# WSSDEV = /sys/bus/pnp/devices/00:07
# CTLDEV = /sys/bus/pnp/devices/00:06
# Это символические ссылки в /sys/devices/pnp0/ ...


#  Активация устройств:
#  ThinkPad грузится с отключенными устройствами, «быстрая загрузка» отключена
#+ (в BIOS).

echo activate > $WSSDEV/resources
echo activate > $CTLDEV/resources


# Анализ параметров ресурсов.

{ read # Discard "state = active" (см. ниже).
  read bla port1
  read bla port2
  read bla port3
  read bla irq
  read bla dma1
  read bla dma2
 # "bla" в первом поле имеет метки: "io," "state," и т.п..
 # Они отбрасываются.

 #  Взлом: с PnPBIOS: порты: port1: WSS, port2:
 #+ OPL, port3: sb (не обязателен)
 #         с ACPI-PnP:порты: port1: OPL, port2: sb, port3: WSS
 #  (ACPI bios думает, что здесь ошибка, код-PnP-карты в snd-cs4236.c
 #+  использует порт PnPBIOS)
 #  Определяем порядок портов, использующих фиксированный порт OPL, как ссылку.
  if [ ${port2%%-*} = 0x388 ]
 #            ^^^^  Вырезаем все, что следует после дефиса в адресе порта.
 #                  Поэтому, если port1 это 0x530-0x537, то
 #+                 остается 0x530 — начало адреса порта.
 then
   # PnPBIOS: обычный порядок
   port=${port1%%-*}
   oplport=${port2%%-*}
 else
   # ACPI: смешанный порядок
   port=${port3%%-*}
   oplport=${port1%%-*}
 fi
 } < $WSSDEV/resources
# Посмотрим, что здесь происходит:
# ----------------------------
#   cat /sys/devices/pnp0/00:07/resources
#
#   state = active
#   io 0x530-0x537
#   io 0x388-0x38b
#   io 0x220-0x233
```

```
#   irq 5
#   dma 1
#   dma 0
#   ^^^   метка "bla" в первом поле (сброшена).


{ read # Сбрасываем первую строку, как выше.
  read bla port1

  cport=${port1%%-*}
  #            ^^^^
  # Нужен только _начальный_ адрес порта.
} < $CTLDEV/resources


# Загружаем модуль:

modprobe --ignore-install snd-cs4236 port=$port cport=$cport\
fm_port=$oplport irq=$irq dma1=$dma1 dma2=$dma2 isapnp=0 index=0
# См. Справочную страницу modprobe.

exit $?
```

Пример A-35. Положение отдельных абзацев в текстовом файле

```
#!/bin/bash
# find-splitpara.sh
#  Поиск отдельных абзацев в текстовом файле
#+ и тэгов числа строк.


ARGCOUNT=1       # Нужен один аргумент.
OFF=0            # Флаг состояния.
ON=1
E_WRONGARGS=85

file="$1"        # Целевой файл.
lineno=1         # Номер строки. Начинается с 1.
Flag=$OFF        # Флаг пустой строки.

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` FILENAME"
  exit $E_WRONGARGS
fi

file_read ()     # Сканирование файла для шаблона, а затем вывод строки.
{
while read line
do

  if [[ "$line" =~ ^[a-z] && $Flag -eq $ON ]]
     then  # Строка начинается со строчного символа, следующая строка пустая.
     echo -n "$lineno::   "
     echo "$line"
  fi
```

```
  if [[ "$line" =~ ^$ ]]
     then       #  Если строка пустая,
     Flag=$ON   #+ устанавливается флаг.
  else
     Flag=$OFF
  fi

  ((lineno++))

done
} < $file  # Файл перенаправляется на stdin функции.

file_read


exit $?


# ---------------------------------------------------------------
Это строка один примера абзаца, bla, bla, bla.
Это строка два, а строка три находится на следующей строке, но

пустая строка разделяет две части абзаца.
# ---------------------------------------------------------------

Запускаем этот сценарий с файлом, содержащим вышеуказанные абзацы:

4::   Пустая строка, разделяющая две части абзаца.


Будет дополнительный вывод всех остальных отдельных абзацев  целевого файла.
```

Пример A-36. Вставка сортировки

```
#!/bin/bash
# insertion-sort.bash: Реализация вставки сортировки в Bash
#                      Максимально используем возможности массива Bash:
#+                     разделение (строки), объединение, и т.д.
# URL: http://www.lugmen.org.ar/~jjo/jjotip/insertion-sort.bash.d
#+          /insertion-sort.bash.sh
#
# Автор: JuanJo Ciarlante <jjo@irrigacion.gov.ar>
# Слегка изменено автором ABS Guide.
# License: GPLv2
# Используется в ABS Guide с разрешения автора (спасибо!).
#
# Проверка:   ./insertion-sort.bash -t
# Или:          bash insertion-sort.bash -t
# Это работать *не будет*:
#              sh insertion-sort.bash -t
#  Почему? Подсказка: какие Bash-специфичные функции отключены по умолчанию
#+ при запуске сценария 'sh script.sh'?
#
: ${DEBUG:=0}  # Отладка, переопределение:  DEBUG=1 ./scriptname ...
# Подстановка параметра -- установите DEBUG в 0 если нет предыдущей установки.

# Глобальный массив: "list"
typeset -a list
# Загружаем числа, разделенные пробелами, из stdin.
if [ "$1" = "-t" ]; then
```

```
DEBUG=1
        read -a list <<( od -Ad -w24 -t u2 /dev/urandom ) # Случайный список.
#                    ^^  процесс подстановки
else
        read -a list
fi
numelem=${#list[*]}

#  Выводит отмеченный элемент list, индекс которого $1,
#+ окруженный двумя символами - передается как $2.
#  Вся строка начинается с $3.
showlist()
  {
  echo "$3"${list[@]:0:$1} ${2:0:1}${list[$1]}${2:1:1} ${list[@]:$1+1};
  }

# Цикл _pivot_ -- второго элемента от конца list.
for(( i=1; i<numelem; i++ )) do
        ((DEBUG))&&showlist i "[]" " "
        # Из текущего _pivot_, возвращаемся к первому элементу.
        for(( j=i; j; j-- )) do
                # Поиск 1-го элемента. Меньшего, чем текущий "pivot" ...
                [[ "${list[j-1]}" -le "${list[i]}" ]] && break
        done
        (( i==j )) && continue ## Нет содержимого нужного для этого элемента.
        # ... Помещаем list[i] (pivot) слева от list[j]:
        list=(${list[@]:0:j} ${list[i]} ${list[j]}\
        #         {0,j-1}        {i}       {j}
              ${list[@]:j+1:i-(j+1)} ${list[@]:i+1})
        #         {j+1,i-1}              {i+1,последний}
        ((DEBUG))&&showlist j "<>" "*"
done


echo
echo  "------"
echo $'Итог:\n'${list[@]}

exit $?
```

Пример A-37. Общее отклонение

```
#!/bin/bash
# sd.sh: Общее отклонение

#  Общее отклонение подразумевает последовательно установленные данные.
#  Показывает, насколько отдельные элементы данных отклоняются от среднего
#+ арифметического значения, то есть их 'разброс' (или кластер).
#  Это, по существу, среднее отклонение данных от среднего значения.

# ============================================================ #
#    Подсчет общего отклонения:
#
# 1  Находим среднее арифметическое (обычное) всех элементов.
# 2  Вычитаем каждый элемент данных из среднего арифметического,
#    и эту разницу возводим в квадрат.
```

```
# 3  Добавляем все отдельные квадраты разниц в #2.
# 4  Делим сумму в #3 на число элементов данных.
#    Это известно как «отклонение».
# 5  Корень квадратный из #4 даст общее отклонение.
# ========================================================== #

count=0         # Количество пунктов данных; глобальная переменная.
SC=9            # Масштаб, который используется bc. 9 десятичных знаков.
E_DATAFILE=90   # Ошибка файла данных.

# ----------------- Настройки файла данных ---------------------
if [ ! -z "$1" ]  # Файл задан как аргумент командной строки?
then
  datafile="$1" #  текстовый файл ASCII,
else            #+ один (числовой) элемент данных в строке!
  datafile=sample.dat
fi              #  См. Пример файла данных ниже.

if [ ! -e "$datafile" ]
then
  echo "\""$datafile"\" не существует!"
  exit $E_DATAFILE
fi
# ------------------------------------------------------


arith_mean ()
{
  local rt=0         # Запущено всего.
  local am=0         # Среднее арифметическое.
  local ct=0         # Количество элементов данных.

  while read value   # Считывается за раз один элемент данных.
  do
    rt=$(echo "scale=$SC; $rt + $value" | bc)
    (( ct++ ))
  done

  am=$(echo "scale=$SC; $rt / $ct" | bc)

  echo $am; return $ct   # Эта функция " возвращает" ДВА значения!
  #  Внимание: Этот трюк не будет работать, если $ct > 255!
  #  Для обработки большего количества элементов данных
  #+ просто закомментируйте "return $ct" выше.
} <"$datafile"   # Из файла данных.

sd ()
{
  mean1=$1  # Среднее арифметическое (переданное в функцию).
  n=$2      # Количество элементов данных.
  sum2=0    # Сумма квадратов разностей ("отклонение").
  avg2=0    # Среднее $sum2.
  sdev=0    # Общее отклонение.

  while read value   # Считывается одна строка за раз.
  do
    diff=$(echo "scale=$SC; $mean1 - $value" | bc)
    # Разница между средним арифметическим и элементом данных.
    dif2=$(echo "scale=$SC; $diff * $diff" | bc) # Возведение в квадрат.
    sum2=$(echo "scale=$SC; $sum2 + $dif2" | bc) # Сумма квадратов.
  done
```

```
    avg2=$(echo "scale=$SC; $sum2 / $n" | bc)  # среднее суммы квадратов.
    sdev=$(echo "scale=$SC; sqrt($avg2)" | bc) # Квадратный корень =
    echo $sdev                                 # общему отклонению.

} <"$datafile"   # Из файла данных.


# ======================================================== #
mean=$(arith_mean); count=$?   # Два возвращаемых из функции!
std_dev=$(sd $mean $count)

echo
echo "Число элементов данных в \""$datafile"\" = $count"
echo "Среднее арифметическое (усредненное) = $mean"
echo "Общее отклонение = $std_dev"
echo
# ======================================================== #

exit

#  Этот сценарий может быть упрощен, но не за счет уменьшения точности.

# ++++++++++++++++++++++++++++++++++++++ #
# Пример файла данных (sample1.dat):

# 18.35
# 19.0
# 18.88
# 18.91
# 18.64


# $ sh sd.sh sample1.dat

# Количество элементов данных в "sample1.dat" = 5
# Среднее арифметическое (среднее) = 18.756000000
# Общее отклонение = .235338054
# ++++++++++++++++++++++++++++++++++++++ #
```

Пример A-38. Файловый генератор *pad* для условно-бесплатных продуктов

```
#!/bin/bash
# pad.sh

###########################################################
#               PAD (xml) генератор файлов
#+ Написан Mendel Cooper <thegrendel.abs@gmail.com>.
#+ Released to the Public Domain.
#
#  Создает дескриптор файла "PAD" для условно-бесплатных
#+ пакетов, согласно спецификациям ASP.
#  http://www.asp-shareware.org/pad
###########################################################
```

```
#  Принимает (не обязательно) сохраненное имя файла, как аргумент командной
#+ строки.
if [ -n "$1" ]
then
  savefile=$1
else
  savefile=save_file.xml               # По умолчанию имя save_file.
fi


# ===== Заголовки файла PAD =====
HDR1="<?xml version=\"1.0\" encoding=\"Windows-1252\" ?>"
HDR2="<XML_DIZ_INFO>"
HDR3="<MASTER_PAD_VERSION_INFO>"
HDR4="\t<MASTER_PAD_VERSION>1.15</MASTER_PAD_VERSION>"
HDR5="\t<MASTER_PAD_INFO>Portable Application Description, или для краткости
PAD, это набор данных, который используется авторами условно бесплатных
продуктов, для распространения информации для тех, кто заинтересован в этих
программных продуктах.
Подробнее на http://www.asp-shareware.org/pad</MASTER_PAD_INFO>"
HDR6="</MASTER_PAD_VERSION_INFO>"
# ============================


fill_in ()
{
  if [ -z "$2" ]
  then
    echo -n "$1? "      # Получены данные введенные пользователем.
  else
    echo -n "$1 $2? "   # Запросить дополнительно?
  fi

  read var              # Можно вставить для заполнения поля.
                        # Это показывает, насколько «read» может быть гибким.

  if [ -z "$var" ]
  then
    echo -e "\t\t<$1 />" >>$savefile    # Отступ с 2 табуляциями.
    return
  else
    echo -e "\t\t<$1>$var</$1>" >>$savefile
    return ${#var}      # Возвращается длина введенной строки.
  fi
}

check_field_length ()  # Проверка длины полей описания программы.
{
  # $1 = максимальный размер поля
  # $2 = фактический размер поля
  if [ "$2" -gt "$1" ]
  then
    echo "Внимание: Максимальное количество символов поля $1 превышена!"
  fi
}

clear                  # Очистка экрана.
echo "Генератор файлов PAD"
echo "--- ---- -------"
echo
```

```
# Записываем заголовки файла в файл.
echo $HDR1 >$savefile
echo $HDR2 >>$savefile
echo $HDR3 >>$savefile
echo -e $HDR4 >>$savefile
echo -e $HDR5 >>$savefile
echo $HDR6 >>$savefile


# Company_Info
echo "ИНФОРМАЦИЯ О КОМПАНИИ"
CO_HDR="Company_Info"
echo "<$CO_HDR>" >>$savefile

fill_in Company_Name
fill_in Address_1
fill_in Address_2
fill_in City_Town
fill_in State_Province
fill_in Zip_Postal_Code
fill_in Country

# Если применимо:
# fill_in ASP_Member "[Y/N]"
# fill_in ASP_Member_Number
# fill_in ESC_Member "[Y/N]"

fill_in Company_WebSite_URL

clear   # Очистка экрана между разделами.

   # Contact_Info
echo "КОНТАКТЫ"
CONTACT_HDR="Contact_Info"
echo "<$CONTACT_HDR>" >>$savefile
fill_in Author_First_Name
fill_in Author_Last_Name
fill_in Author_Email
fill_in Contact_First_Name
fill_in Contact_Last_Name
fill_in Contact_Email
echo -e "\t</$CONTACT_HDR>" >>$savefile
   # END Contact_Info

clear

   # Support_Info
echo "ПОДДЕРЖКА"
SUPPORT_HDR="Support_Info"
echo "<$SUPPORT_HDR>" >>$savefile
fill_in Sales_Email
fill_in Support_Email
fill_in General_Email
fill_in Sales_Phone
fill_in Support_Phone
fill_in General_Phone
fill_in Fax_Phone
echo -e "\t</$SUPPORT_HDR>" >>$savefile
   # END Support_Info

echo "</$CO_HDR>" >>$savefile
```

```
# END Company_Info

clear

# Program_Info
echo "ИНФОРМАЦИЯ О ПРОГРАММЕ"
PROGRAM_HDR="Program_Info"
echo "<$PROGRAM_HDR>" >>$savefile
fill_in Program_Name
fill_in Program_Version
fill_in Program_Release_Month
fill_in Program_Release_Day
fill_in Program_Release_Year
fill_in Program_Cost_Dollars
fill_in Program_Cost_Other
fill_in Program_Type "[Shareware/Freeware/GPL]"
fill_in Program_Release_Status "[Beta, Major Upgrade, etc.]"
fill_in Program_Install_Support
fill_in Program_OS_Support "[Win9x/Win2k/Linux/etc.]"
fill_in Program_Language "[English/Spanish/etc.]"

echo; echo

  # File_Info
echo "ИНФОРМАЦИЯ О ПРОГРАММЕ"
FILEINFO_HDR="File_Info"
echo "<$FILEINFO_HDR>" >>$savefile
fill_in Filename_Versioned
fill_in Filename_Previous
fill_in Filename_Generic
fill_in Filename_Long
fill_in File_Size_Bytes
fill_in File_Size_K
fill_in File_Size_MB
echo -e "\t</$FILEINFO_HDR>" >>$savefile
  # END File_Info

clear

  # Expire_Info
echo "ИНФОРМАЦИЯ ОБ ОКОНЧАНИИ ДЕЙСТВИЯ"
EXPIRE_HDR="Expire_Info"
echo "<$EXPIRE_HDR>" >>$savefile
fill_in Has_Expire_Info "Y/N"
fill_in Expire_Count
fill_in Expire_Based_On
fill_in Expire_Other_Info
fill_in Expire_Month
fill_in Expire_Day
fill_in Expire_Year
echo -e "\t</$EXPIRE_HDR>" >>$savefile
  # END Expire_Info

clear

  # More Program_Info
echo "ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ О ПРОГРАММЕ"
fill_in Program_Change_Info
fill_in Program_Specific_Category
fill_in Program_Categories
fill_in Includes_JAVA_VM "[Y/N]"
```

```
fill_in Includes_VB_Runtime "[Y/N]"
fill_in Includes_DirectX "[Y/N]"
  # END More Program_Info

echo "</$PROGRAM_HDR>" >>$savefile
# END Program_Info

clear

# Program_Description
echo "ОПИСАНИЕ ПРОГРАММЫ"
PROGDESC_HDR="Program_Descriptions"
echo "<$PROGDESC_HDR>" >>$savefile

LANG="English"
echo "<$LANG>" >>$savefile

fill_in Keywords "[запятая + разделительный пробел]"
echo
echo "45, 80, 250, 450, 2000 слов описания программы"
echo "(можно вырезать и вставлять в это поле)"
#  Было бы весьма целесообразно составить текстовым редактором
#+ следующие поля "Char_Desc", а
#+ затем вырезать-и-вставить текст в поля ответов.
echo
echo "              |---------------45 символов---------------|"
fill_in Char_Desc_45
check_field_length 45 "$?"
echo
fill_in Char_Desc_80
check_field_length 80 "$?"

fill_in Char_Desc_250
check_field_length 250 "$?"

fill_in Char_Desc_450
fill_in Char_Desc_2000

echo "</$LANG>" >>$savefile
echo "</$PROGDESC_HDR>" >>$savefile
# END Program Description

clear
echo "Сделано."; echo; echo
echo "Сохранено в файл:  \""$savefile"\""

exit 0
```

Пример A-39. Редактор *man page*

```
#!/bin/bash
# maned.sh
# Простой редактор man page

# Version: 0.1 (Альфа, возможны ошибки)
# Автор: Mendel Cooper <thegrendel.abs@gmail.com>
# Reldate: 16 June 2008
# License: GPL3
```

```
savefile=      # Глобальная переменная, используется несколькими функциями.
E_NOINPUT=90   # Отсутствие пользовательского ввода (ошибка). Может быть или
               #+ не быть критической.

# =========== Теги разметки ============ #
TopHeader=".TH"
NameHeader=".SH NAME"
SyntaxHeader=".SH SYNTAX"
SynopsisHeader=".SH SYNOPSIS"
InstallationHeader=".SH INSTALLATION"
DescHeader=".SH DESCRIPTION"
OptHeader=".SH OPTIONS"
FilesHeader=".SH FILES"
EnvHeader=".SH ENVIRONMENT"
AuthHeader=".SH AUTHOR"
BugsHeader=".SH BUGS"
SeeAlsoHeader=".SH SEE ALSO"
BOLD=".B"
# Если необходимо — добавьте тэги.
# См. документацию groff по значениям разметки.
# ================================== #

start ()
{
clear                  # Очистка экрана.
echo "ManEd"
echo "-----"
echo
echo "Простой создатель man page"
echo "Автор: Mendel Cooper"
echo "License: GPL3"
echo; echo; echo
}

progname ()
{
  echo -n "Название программы? "
  read name

  echo -n "Раздел Manpage? [Нажмите RETURN для умолчания (\"1\") ]  "
  read section
  if [ -z "$section" ]
  then
    section=1   # Увеличьте количество страниц раздела.
  fi

  if [ -n "$name" ]
  then
    savefile=""$name"."$section""       #  Суффикс имени файла = раздел.
    echo -n "$1 " >>$savefile
    name1=$(echo "$name" | tr a-z A-Z)  #  Изменение на строчные,
    echo -n "$name1" >>$savefile
  else
    echo "Ошибка! Отсутствует ввод."    # Ввод обязателен.
    exit $E_NOINPUT                     # Критично!
    #  Упражнение: Прерывите сценарий, если введенное имя ошибочно.
    #              Перепишите этот раздел, чтобы использовалось имя файла по
    #+             умолчанию, если не введены входные данные.
  fi
```

```
  echo -n "  \"$section\""">>$savefile   # Добавление, всегда добавляем.

  echo -n "Версия? "
  read ver
  echo -n " \"Версия $ver \""">>$savefile
  echo >>$savefile

  echo -n "Краткое описание [0 - 5 слов]? "
  read sdesc
  echo "$NameHeader">>$savefile
  echo ""$BOLD" "$name""">>$savefile
  echo "\- "$sdesc""">>$savefile

}

fill_in ()
{ # Эта функция в большей или меньшей степени скопирована из сценария "pad.sh".
  echo -n "$2? "       #  Получение ввода пользователя.
  read var             #  Можно вставить (только одну строку!) при
                       #+ заполнения поля.

  if [ -n "$var" ]
  then
    echo "$1 " >>$savefile
    echo -n "$var" >>$savefile
  else                 # Запись поля в файл не добавлено.
    return $E_NOINPUT  # Здесь это не критично.
  fi

  echo >>$savefile

}


end ()
{
clear
echo -n "Вы хотели бы просмотреть сохраненные страницы (y/n)? "
read ans
if [ "$ans" = "n" -o "$ans" = "N" ]; then exit; fi
exec less "$savefile"  #  Выход сценария и передача управления  'less' ...
                       #+ ... Которая форматирует исходник man page для
                       #+ просмотра.
}


# ---------------------------------------- #
start
progname "$TopHeader"
fill_in "$SynopsisHeader" "Синопсис"
fill_in "$DescHeader" "Подробное описание"
# Можно вставлять в текст *одной строкой*.
fill_in "$OptHeader" "Опции"
fill_in "$FilesHeader" "Файлы"
fill_in "$AuthHeader" "Автор"
fill_in "$BugsHeader" "Баги"
fill_in "$SeeAlsoHeader" "См. Так же"
# fill_in "$OtherHeader" ... при необходимости.
end    # ... выход не требуется.
# ---------------------------------------- #
```

```
#  Обратите внимание, что созданной странице обычно требуется тонкая ручная
#+ настройка с помощью текстового редактора.
#  Тем не менее, намного облегчается написание исходника man с нуля или даже
#+ редактирования пустого шаблона man page.
#  Основным недостатком сценария является то, что он позволяет вставлять в поля
#+ ввода только единственную текстовую строку.
#  Это могут быть длинные, составленные вместе строки, которые groff
#+ автоматически складывает и расстанавливает переносы.
#  Однако если вам необходимы несколько абзацев (разделенных переводами
#+ каретки), они должны быть вставлены путем ручного редактирования созданного
#+ сценария  man page.
#  Упражнение (трудное): Исправьте это!

#  Этот сценарий не столь подробен, как полнофункциональный пакет «manedit»
#+ http://freshmeat.net/projects/manedit/
#+ но все равно он облегчает работу.
```

Пример A-40. Лепестки розы

```
#!/bin/bash -i
# petals.sh

##############################################################################
# Лепестки розы                                                              #
#                                                                            #
# Version 0.1 Создан Serghey Rodin                                           #
# Version 0.2 Изменен автором ABS Guide Author                               #
#                                                                            #
# License: GPL3                                                              #
# Используется в ABS Guide с разрешения.                                     #
# ########################################################################## #

hits=0      # Правильные ответы.
WIN=6       # Освоил игру.
ALMOST=5    # Не хватает мастерства.
EXIT=exit   # Рано закончили?

RANDOM=$$   # Случайные числа генерируются из PID сценария.


# Кости (ASCII графика для костей)
bone1[1]="|         |"
bone1[2]="|       o |"
bone1[3]="|       o |"
bone1[4]="| o     o |"
bone1[5]="| o     o |"
bone1[6]="| o     o |"
bone2[1]="|    o    |"
bone2[2]="|         |"
bone2[3]="|    o    |"
bone2[4]="|         |"
bone2[5]="|    o    |"
bone2[6]="| o     o |"
bone3[1]="|         |"
bone3[2]="| o       |"
bone3[3]="| o       |"
bone3[4]="| o     o |"
```

```
bone3[5]="| o     o |"
bone3[6]="| o     o |"
bone="+---------+"



# Функции

instructions () {

  clear
  echo -n "Вам нужны инструкции? (y/n) "; read ans
  if [ "$ans" = "y" -o "$ans" = "Y" ]; then
    clear
    echo -e '\E[34;47m'  # Синий шрифт.

#  "cat document"
    cat <<INSTRUCTIONSZZZ
Игра называется Лепестки розы,
и это не просто название.
Бросаются пять костей, а вы должны угадать 'результат' каждого броска.
Это может быть ноль или другое число.
После вашего ответа, вам будет показан результат этого броска, но ...
это вся информация, которую вы получите.

Шесть последовательных правильных ответов присоединят вас к Братству Розы.
INSTRUCTIONSZZZ

    echo -e "\033[0m"    # Выключение синего цвета.
    else clear
  fi

}


fortune ()
{
  RANGE=7
  FLOOR=0
  number=0
  while [ "$number" -le $FLOOR ]
  do
    number=$RANDOM
    let "number %= $RANGE"   # 1 - 6.
  done

  return $number
}



throw () { # Подсчет каждой отдельной неудачи.
  fortune; B1=$?
  fortune; B2=$?
  fortune; B3=$?
  fortune; B4=$?
  fortune; B5=$?

  calc () { # функция в функции!
    case "$1" in
       3   ) rose=2;;
```

```
      5   ) rose=4;;
      *   ) rose=0;;
    esac    # Упрощенный алгоритм.
            # Действительно, не добраться до сути вопроса.
    return $rose
  }

  answer=0
  calc "$B1"; answer=$(expr $answer + $(echo $?))
  calc "$B2"; answer=$(expr $answer + $(echo $?))
  calc "$B3"; answer=$(expr $answer + $(echo $?))
  calc "$B4"; answer=$(expr $answer + $(echo $?))
  calc "$B5"; answer=$(expr $answer + $(echo $?))
}



game ()
{ # Создание графического отображения броска кости.
  throw
    echo -e "\033[1m"    # Жирный.
  echo -e "\n"
  echo -e "$bone\t$bone\t$bone\t$bone\t$bone"
  echo -e \
 "${bone1[$B1]}\t${bone1[$B2]}\t${bone1[$B3]}\t${bone1[$B4]}\t${bone1[$B5]}"
  echo -e \
 "${bone2[$B1]}\t${bone2[$B2]}\t${bone2[$B3]}\t${bone2[$B4]}\t${bone2[$B5]}"
  echo -e \
 "${bone3[$B1]}\t${bone3[$B2]}\t${bone3[$B3]}\t${bone3[$B4]}\t${bone3[$B5]}"
  echo -e "$bone\t$bone\t$bone\t$bone\t$bone"
  echo -e "\n\n\t\t"
    echo -e "\033[0m"    # Выключение жирного.
  echo -n "Сколько лепестков у розы? "
}



# ============================================================== #

Инструкции

while [ "$petal" != "$EXIT" ]    # Основной цикл.
do
  game
  read petal
  echo "$petal" | grep [0-9] >/dev/null  # Фильтр для цифр ответа.
                                         # В противном случае просто бросаем
                                         #+ кости снова.
  if [ "$?" -eq 0 ]   # Цикл If #1.
  then
    if [ "$petal" == "$answer" ]; then    # Цикл If #2.
        echo -e "\Правильно. У розы $petal лепестков.\n"
        (( hits++ ))

        if [ "$hits" -eq "$WIN" ]; then   # Цикл If #3.
          echo -e '\E[31;47m'  # Красный шрифт.
          echo -e "\033[1m"    # Жирный.
          echo "Вы разгадали тайну Лепестков розы!"
          echo "Добро пожаловать в Братство розы!!!"
          echo "(При этом Вы клянетесь хранить эту тайну.)"; echo
          echo -e "\033[0m"    # Отключаем красный & жирный.
```

```
          break                  # Выход!
        else echo "Пока у вас $hits правильных ответов."; echo

        if [ "$hits" -eq "$ALMOST" ]; then
          echo "Последний ответ приведет вас к сути тайны!"; echo
        fi

      fi                                  # Закрываем цикл if #3.

    else
      echo -e "\Не верно. У розы $answer лепестков.\n"
      hits=0   # Сброс количества правильных ответов.
    fi                                    # Закрываем цикл if #2.

    echo -n "Нажмите ENTER для следующего броска, или введите \"exit\" для
окончания. "
    read
    if [ "$REPLY" = "$EXIT" ]; then exit
    fi

  fi                  # Закрываем цикл if #1.

  clear
done                  # Конец основного (while) цикла.

###

exit $?

# Источники:
# ---------
# 1) http://en.wikipedia.org/wiki/Petals_Around_the_Rose
# 2) http://www.borrett.id.au/computing/petals-bg.htm
```

Пример A-41. Quacky: игра слов Perquackey

```
#!/bin/bash
# qky.sh

######################################################################
# QUACKEY: Облегченная версия Perquackey [TM].                       #
#                                                                    #
# Автор: Mendel Cooper  <thegrendel.abs@gmail.com>                   #
# version 0.1.02      03 May, 2008                                   #
# License: GPL3                                                      #
######################################################################

WLIST=/usr/share/dict/word.lst
#                     ^^^^^^^^  Словарь находится здесь.
#  Словарь в ASCII, в каждой строке одно слово, формат UNIX.
#  Предлагаемый словарь - это пакет словаря автора "yawl".
#  http://bash.deta.in/yawl-0.3.2.tar.gz
#    или
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz

NONCONS=0     # Слово не из букв.
CONS=1        # Слово из букв.
SUCCESS=0
NG=1
FAILURE=''
```

```
NULL=0          # Обнуление значения буквы (если найдена).
MINWLEN=3       # Минимальный размер слова.
MAXCAT=5        # Максимальное число слов в данной категории.
PENALTY=200     # Штраф за неприемлемые слова.
total=
E_DUP=70        # Ошибка при повторении слова.

TIMEOUT=10      # Время для ввода слова.

NVLET=10        # 10 не правильных букв.
VULET=13        # 13 правильных букв (еще не реализовано!).

declare -a Words
declare -a Status
declare -a Score=( 0 0 0 0 0 0 0 0 0 0 0 )


letters=( a n s r t m l k p r b c i d s i d z e w u e t f
e y e r e f e g t g h h i t r s c i t i d i j a t a o l a
m n a n o v n w o s e l n o s p a q e e r a b r s a o d s
t g t i t l u e u v n e o x y m r k )
#  Таблица распределения букв нагло заимствована из игры «Wordy»,
#+ ca. 1992, написанной одним хорошим парнем по имени Mendel Cooper.

declare -a LS

numelements=${#letters[@]}
randseed="$1"

instructions ()
{
  clear
  echo "Добро пожаловать в QUACKEY, игру анаграмм."; echo
  echo -n "Нужны инструкции? (y/n) "; read ans

   if [ "$ans" = "y" -o "$ans" = "Y" ]; then
     clear
     echo -e '\E[31;47m'  # Основной цвет красный. '\E[34;47m' для синего.
     cat <<INSTRUCTION1

QUACKEY это вариант Perquackey [TM].
Правила те же, но подсчет упрощен и допускается повторение ранее
сыгранных слов. 'Уязвимая' игра еще не реализована, но тогда игра
будет полноценной.
В начале игры, игроку задаются 10 букв. Цель - отгадать правильные
словарные слова длиной не менее 3 букв из этого набора букв.
Размер слов каждой категории
-- 3-буквенные, 4-буквенные, 5-буквенные, ... --
завершается пятым введенным словом, и никакие дальнейшие слова в данной
категории принимаются.
Штраф за слишком короткие (двух буквенные), повторяющиеся, не состоящие
из букв, и неправильные (нет в словаре) слова -200. Такое же наказание
применяется к попыткам ввести слово в категорию заполненную выше.

ИНСТРУКЦИЯ1

  echo -n "Нажмите ENTER для вывода следующей страницы инструкции. "; read az1

     cat <<ИНСТРУКЦИЯ2

Очки начисляемые в классической Perquackey:
```

```
Первое 3-буквенное слово    60 очков, плюс   10 за каждое добавленное слово.
Первое 4-буквенное слово   120 очков, плюс   20 за каждое добавленное слово.
Первое 5-буквенное слово   200 очков, плюс   50 за каждое добавленное слово.
Первое 6-буквенное слово   300 очков, плюс  100 за каждое добавленное слово.
Первое 7-буквенное слово   500 очков, плюс  150 за каждое добавленное слово.
Первое 8-буквенное слово   750 очков, плюс  250 за каждое добавленное слово.
Первое 9-буквенное слово  1000 очков, плюс  500 за каждое добавленное слово.
Первое 10-буквенное слово 2000 очков, плюс 2000 за каждое добавленное слово.

Завершающая категория бонусов:
3-буквенные слова   100
4-буквенные слова   200
5-буквенные слова   400
6-буквенные слова   800
7-буквенные слова  2000
8-буквенные слова 10000
Это упрощение абсурдной балльной системы бонусов Perquackey.

ИНСТРУКЦИЯ2

  echo -n "Нажмите ENTER для вывода конечной страницы инструкции. "; read az1

     cat <<ИНСТРУКЦИЯ3


Нажатие ENTER при вводе слова завершает игру.

Время ввода каждого слова ограничено 10 секундами.
*** По истечении времени игра прекращается. ***
Кроме этого время игры не ограниченно.

-------------------------------------------------
Статистика игры автоматически сохраняется в файл.
-------------------------------------------------

Для конкурсной ('повторной') игры, предыдущий набор букв может быть повторен в
сценарии параметром командной строки
Например, указывая набор букв "qky 7633"
c a d i f r h u s k ...

ИНСТРУКЦИЯ3

  echo; echo -n "Нажмите ENTER для запуска игры. "; read az1

       echo -e "\033[0m"    # Отключаем красный цвет.
     else clear
  fi

  clear

}



seed_random ()
{                         #  Источник генератора случайных чисел
  if [ -n "$randseed" ]   #  В соревновательном режиме можно задавать
  then                    #+ для игры случайное число.
#   RANDOM="$randseed"
    echo "Источнику RANDOM присваивается "$randseed""
  else
```

```
    randseed="$$"         # Или он получает случайное число из ID процесса.
    echo "Источник RANDOM не указан, задается из ID процесса сценария ($$)."
  fi

  RANDOM="$randseed"

  echo
}


get_letset ()
{
  element=0
  echo -n "Набор букв:"

  for lset in $(seq $NVLET)
  do  # Выбор случайных букв для внесения в набор букв.
    LS[element]="${letters[$((RANDOM%numelements))]}"
    ((element++))
  done

  echo
  echo "${LS[@]}"

}


add_word ()
{
  wrd="$1"
  local idx=0

  Status[0]=""
  Status[3]=""
  Status[4]=""

  while [ "${Words[idx]}" != '' ]
  do
    if [ "${Words[idx]}" = "$wrd" ]
    then
      Status[3]="ШТРАФ-за-повторение-слова"
      let "Score[0]= 0 - $PENALTY"
      let "Score[1]-=$PENALTY"
      return $E_DUP
    fi

    ((idx++))
  done

  Words[idx]="$wrd"
  get_score

}

get_score()
{
  local wlen=0
  local score=0
  local bonus=0
  local first_word=0
  local add_word=0
```

```
  local numwords=0

  wlen=${#wrd}
  numwords=${Score[wlen]}
  Score[2]=0
  Status[4]=""   # Присвоение "бонуса" 0.

  case "$wlen" in
    3) first_word=60
       add_word=10;;
    4) first_word=120
       add_word=20;;
    5) first_word=200
       add_word=50;;
    6) first_word=300
       add_word=100;;
    7) first_word=500
       add_word=150;;
    8) first_word=750
       add_word=250;;
    9) first_word=1000
       add_word=500;;
   10) first_word=2000
       add_word=2000;;   # Эта категория отличается от первоначальной!
      esac

  ((Score[wlen]++))
  if [ ${Score[wlen]} -eq $MAXCAT ]
  then   # Категория завершает упрощенный подсчет бонусов!
    case $wlen in
      3 ) bonus=100;;
      4 ) bonus=200;;
      5 ) bonus=400;;
      6 ) bonus=800;;
      7 ) bonus=2000;;
      8 ) bonus=10000;;
    esac  # Не нужно беспокоиться о 9-й и 10-й.
    Status[4]="Категория-Завершается-$wlen-***БОНУСАМИ***"
    Score[2]=$bonus
  else
    Status[4]=""   # Обнуление.
  fi


    let "score =  $first_word +   $add_word * $numwords"
    if [ "$numwords" -eq 0 ]
    then
      Score[0]=$score
    else
      Score[0]=$add_word
    fi   #  Все это, чтобы отделить очки за последнее слово
         #+ от общего счета.
  let "Score[1] += ${Score[0]}"
  let "Score[1] += ${Score[2]}"

}

get_word ()
{
  local wrd=''
  read -t $TIMEOUT wrd   # Значение времени чтения
```

```
  echo $wrd
}

is_constructable ()
{ # Это наиболее сложная и трудно описываемая функция.
  local -a local_LS=( "${LS[@]}" )  # Локальная копия набора букв.
  local is_found=0
  local idx=0
  local pos
  local strlen
  local local_word=( "$1" )
  strlen=${#local_word}

  while [ "$idx" -lt "$strlen" ]
  do
    is_found=$(expr index "${local_LS[*]}" "${local_word:idx:1}")
    if [ "$is_found" -eq "$NONCONS" ] # Не соответствует конструкции!
    then
      echo "$FAILURE"; return
    else
      ((pos = ($is_found - 1) / 2))   # Восстановление пробелов между буквами!
      local_LS[pos]=$NULL             # Обнуление использованных букв.
      ((idx++))                       # Увеличение индекса.
    fi
  done

  echo "$SUCCESS"
  return
}

is_valid ()
{ # Удивительно просто проверяет, если слово в словаре ...
  fgrep -qw "$1" "$WLIST"   # ... любезным 'grep' ...
  echo $?
}

check_word ()
{
  if [ -z "$1" ]
  then
    return
  fi

  Status[1]=""
  Status[2]=""
  Status[3]=""
  Status[4]=""

  iscons=$(is_constructable "$1")
  if [ "$iscons" ]
  then
    Status[1]="constructable"
    v=$(is_valid "$1")
    if [ "$v" -eq "$SUCCESS" ]
    then
      Status[2]="valid"
      strlen=${#1}

      if [ ${Score[strlen]} -eq "$MAXCAT" ]   # Полная категория!
      then
        Status[3]="ШТРАФ-за переполнение-категории-$strlen"
```

```
        return $NG
      fi

      case "$strlen" in
        1 | 2 )
        Status[3]="ШТРАФ-за-двух-буквенное-слово"
        return $NG;;
        * )
        Status[3]=""
        return $SUCCESS;;
      esac
    else
      Status[3]="Не-правильно-ШТРАФ"
      return $NG
    fi
  else
    Status[3]="Не-составлено-ШТРАФ"
      return $NG
  fi

  ### FIXME: Поток строк блока кода выше.

}


display_words ()
{
  local idx=0
  local wlen0

  clear
  echo "Набор букв:   ${LS[@]}"
  echo "Трех:    Четырех:    Пяти:     Шести:    Семи:    Восьми:"
  echo "-------------------------------------------------------------"



  while [ "${Words[idx]}" != '' ]
  do
   wlen0=${#Words[idx]}
   case "$wlen0" in
     3) ;;
     4) echo -n "           " ;;
     5) echo -n "                     " ;;
     6) echo -n "                               " ;;
     7) echo -n "                                         " ;;
     8) echo -n "                                                   " ;;
   esac
   echo "${Words[idx]}"
   ((idx++))
  done

  ### FIXME: Слово на экране довольно сырое.
}


play ()
{
  word="Начало игры"   # Образец слова, для начала ...

  while [ "$word" ]   #  Если игрок нажал на возврат (пустое слово),
```

```
  do                    #+ то игра заканчивается.
    echo "$word: "${Status[@]}""
    echo -n "Последние очки: [${Score[0]}]   ВСЕГО очков: [${Score[1]}]:
Следующее слово: "
    total=${Score[1]}
    word=$(get_word)
    check_word "$word"

    if [ "$?" -eq "$SUCCESS" ]
    then
      add_word "$word"
    else
      let "Score[0]= 0 - $PENALTY"
      let "Score[1]-=$PENALTY"
    fi

  display_words
  done   # Выход из игры.

  ### FIXME: Функция play() вызывает много других функций.
  ### Это "код спагетти" !!!
}

end_of_game ()
{ # Сохранение и вывод статистики.

  ########################Автосохранение############################
  savefile=qky.save.$$
  #                 ^^ PID сценария
  echo `date` >> $savefile
  echo "Набор букв # $randseed  (случайный источник) ">> $savefile
  echo -n "Набор букв: " >> $savefile
  echo "${LS[@]}" >> $savefile
  echo "---------" >> $savefile
  echo "Слово составлено:" >> $savefile
  echo "${Words[@]}" >> $savefile
  echo >> $savefile
  echo "Очки: $total" >> $savefile

  echo "Статистика этого раунда сохранена в \""$savefile"\""
  ############################################################

  echo "Очки этого раунда: $total"
  echo "Слов:  ${Words[@]}"
}

# ---------#
инструкции
seed_random
get_letset
play
end_of_game
# ---------#

exit $?

# СДЕЛАТЬ:
#
# 1) Очистить код!
# 2) Приукрасить функцию display_words()  (возможно виджетами?).
# 3) Улучшить тайм-аут... возможно изменить запись untimed,
```

```
#+   но с ограничением времени для общего раунда.
# 4) Будет красиво, если время будет отсчитываться на экране.
# 5) Реализация «уязвимого» режима игры для совместимости с
#+   классической версией игры.
# 6) Улучшить возможность сохранения в файл (и, возможно, сделать ее
#+   не обязательной).
# 7) Исправить баги!!!

# Больше информации на:
# http://bash.deta.in/qky.README.html
```

Пример A-42. Nim

```
#!/bin/bash
# nim.sh: Игра Nim

# Автор: Mendel Cooper
# Reldate: 15 July 2008
# License: GPL3

ROWS=5     # Пять рядов колышков (или спичек).
WON=91     # Код выхода при победе/поражении.
LOST=92    # Возможно, полезно, при работе в пакетном режиме.
QUIT=99
peg_msg=   # Колышек/Колышки?
Rows=( 0 5 4 3 2 1 )   # Массив данных проведения игры.
# ${Rows[0]} содержит общее количество колышков, обновляется после каждого
# прохода.
# Другие элементы массива хранящие количество колышков соответствующего
# ряда.

instructions ()
{
  clear
  tput bold
  echo "Добро пожаловать в игру Nim."; echo
  echo -n "Нужна подсказка? (y/n) "; read ans

   if [ "$ans" = "y" -o "$ans" = "Y" ]; then
     clear
     echo -e '\E[33;41m'  # желтый передний план, красный фон; жирный.
     cat <<INSTRUCTIONS

Nim это игра, имеющая корни в далеком прошлом.
Этот, особый, вариант начинается пятью рядами колышков.

1:    | | | | |
2:     | | | |
3:      | | |
4:       | |
5:        |

Номер слева обозначает ряд.

Человек-игрок ходит первым и чередуется с ботом.
Игра состоит в удалении, по меньшей мере, одного колышка из одного ряда.
Цель - удалить все колышки из ряда.
Например, в ряде 2, выше, игрок может удалить 1, 2, 3, или 4 колышек.
```

```
Игрок, который удаляет последний колышек проигрывает.

Стратегия состоит в попытке  убрать колышек(и),
оставляя неудачника с последним колышком.

Для выхода из игры, при Вашем ходе, нажмите ENTER.
INSTRUCTIONS

echo; echo -n "Нажмите ENTER для запуска игры. "; read azx

      echo -e "\033[0m"    # Восстановление отображения на экране.
      else tput sgr0; clear
  fi

clear

}

tally_up ()
{
  let "Rows[0] = ${Rows[1]} + ${Rows[2]} + ${Rows[3]} + ${Rows[4]} + \
  ${Rows[5]}"    # Суммирование оставшихся колышков.
}

display ()
{
  index=1   # Начинаем с верхнего ряда.
  echo

  while [ "$index" -le "$ROWS" ]
  do
    p=${Rows[index]}
    echo -n "$index:   "          # Вывод номера строки.

  # ------------------------------------------------
  # Два параллельных внутренних цикла.

      indent=$index
      while [ "$indent" -gt 0 ]
      do
        echo -n " "               # Ряды в шахматном порядке.
        ((indent--))              # Пробелы между колышками.
      done

    while [ "$p" -gt 0 ]
    do
      echo -n "| "
      ((p--))
    done
  # ------------------------------------------------

  echo
  ((index++))
  done

  tally_up

  rp=${Rows[0]}

  if [ "$rp" -eq 1 ]
  then
```

```
    peg_msg=peg
    final_msg="Игра окончена."
  else              # Игра еще не окончена ...
    peg_msg=pegs
    final_msg=""    # ... Поэтому «финальное сообщение» пустое.
  fi

  echo "Осталось      $rp $peg_msg."
  echo "      "$final_msg""


  echo
}

player_move ()
{

  echo "Ваш ход:"

  echo -n "Какой ряд? "
  while read idx
  do                    # Проверка правильности и т.д..

    if [ -z "$idx" ]    # Нажав возврат - выходим.
    then
        echo "Преждевременный выход."; echo
        tput sgr0       # Восстанавливаем изображение на экране.
        exit $QUIT
    fi

    if [ "$idx" -gt "$ROWS" -o "$idx" -lt 1 ]   # Проверка границ.
    then
      echo "Не правильный номер ряда!"
      echo -n "Какой ряд? "
    else
      break
    fi
    # СДЕЛАТЬ:
    #  Добавить проверку не числового ввода.
    #  Кроме того, сценарий падает при вводе большого количества повторов
    #+ вне диапазона.
    #  Исправьте это.

  done

  echo -n "Сколько  удалить? "
  while read num
  do                    # Проверка правильности.

  if [ -z "$num" ]
  then
    echo "Преждевременный выход."; echo
    tput sgr0           # Восстанавливаем изображение на экране.
    exit $QUIT
  fi

    if [ "$num" -gt ${Rows[idx]} -o "$num" -lt 1 ]
    then
      echo "Не возможно удалить $num!"
      echo -n "Сколько удалить? "
    else
```

```
      break
    fi
  done
  # СДЕЛАТЬ:
  #  Добавить проверку не числового ввода.
  #  Кроме того сценарий падает при вводе большого количества повторов
  #+ вне диапазона.
  #  Исправьте это.

  let "Rows[idx] -= $num"

  display
  tally_up

  if [ ${Rows[0]} -eq 1 ]
  then
   echo "      Победил человек!"
   echo "      Поздравления!"
   tput sgr0   # Восстановление изображения на экране.
   echo
   exit $WON
  fi

  if [ ${Rows[0]} -eq 0 ]
  then          # Получите поражение, а не победу . . .
    echo "      Дурачок!"
    echo "      Вы должны удалить последний колышек!"
    echo "      Победил бот!"
    tput sgr0   # Восстановление изображения на экране.
    echo
    exit $LOST
  fi
}


bot_move ()
{

  row_b=0
  while [[ $row_b -eq 0 || ${Rows[row_b]} -eq 0 ]]
  do
    row_b=$RANDOM          # Выбираем случайный ряд.
    let "row_b %= $ROWS"
  done


  num_b=0
  r0=${Rows[row_b]}

  if [ "$r0" -eq 1 ]
  then
    num_b=1
  else
    let "num_b = $r0 - 1"
        #  Оставляет только один колышек в ряду.
  fi    #  Не очень хорошая стратегия,
        #+ но, вероятно, намного лучше, чем совершенно случайно.

  let "Rows[row_b] -= $num_b"
  echo -n "Bot:  "
  echo "Удаляем из ряда $row_b ... "
```

```
  if [ "$num_b" -eq 1 ]
  then
    peg_msg=peg
  else
    peg_msg=pegs
  fi

  echo "      $num_b $peg_msg."

  display
  tally_up

  if [ ${Rows[0]} -eq 1 ]
  then
   echo "      Победил бот!"
   tput sgr0   # Восстановление изображения на экране.
   exit $WON
  fi

}


# ================================================= #
инструкции       # Если человеку нужны ...
tput bold        # Жирные символы для более удобного просмотра.
display          # Показывает игровое поле.

while [ true ]   # Основной цикл.
do               # Чередование ходов бота и человека.
  player_move
  bot_move
done
# ================================================= #

# Упражнения:
# --------
# Улучшите стратегию бота.
# Существует, по сути, стратегия Nim, которая может принести победу.
# См. раздел Wikipedia о Nim:  http://en.wikipedia.org/wiki/Nim
# Перекодируйте бот для использования этой стратегии (довольно трудно).
```

Пример A-43. Секундомер в командной строке

```
#!/bin/sh
# sw.sh
# Секундомер в командной строке

# Автор: Pádraig Brady
#    http://www.pixelbeat.org/scripts/sw
#    (Небольшие изменения автора ABS Guide.)
#    Используется в ABS Guide с разрешения автора сценария.
# Примечания:
#    Этот сценарий запускает несколько процессов, дополняющих цикл
#    обработки оболочки, поэтому предполагается, что они занимают
#    незначительное количество времени по сравнению со временем отклика людей
#    (~.1C) (или прерываний клавиатуры (~.05s))
```

```
#    Для разделения - знак'?' должен быть введен дважды, если перед ним
#    (в той же строке) введены символы (ошибочные).
#    Пока не вызывается '?', сигнал, на сильно
#    загруженных системах, может слегка отставать.


# Изменения:
#    V1.0, 23 Aug 2005, Initial release
#    V1.1, 26 Jul 2007, Allow both splits and laps from single invocation.
#                       Only start timer after a key is pressed.
#                       Indicate lap number
#                       Cache programs at startup so there is less error
#                       due to startup delays.
#    V1.2, 01 Aug 2007, Work around `date` commands that don't have
#                       nanoseconds.
#                       Use stty to change interrupt keys to space for
#                       laps etc.
#                       Ignore other input as it causes problems.
#    V1.3, 01 Aug 2007, Testing release.
#    V1.4, 02 Aug 2007, Various tweaks to get working under ubuntu
#                       and Mac OS X.
#    V1.5, 27 Jun 2008, set LANG=C as got vague bug report about it.

export LANG=C

ulimit -c 0   # Без дампов ядра из SIGQUIT.
trap '' TSTP  # На всякий случай игнорируем Ctrl-Z.
save_tty=`stty -g` && trap "stty $save_tty" EXIT  #  Восстанавливает tty при
                                                  #+ выходе.
stty quit ' ' # Вместо Ctrl-\ используются пробелы.
stty eof  '?' # Вместо Ctrl-D для разделения используется ?.
stty -echo    # Не выводить ввод на экран.

cache_progs() {
    stty > /dev/null
    date > /dev/null
    grep . < /dev/null
    (echo "import time" | python) 2> /dev/null
    bc < /dev/null
    sed '' < /dev/null
    printf '1' > /dev/null
    /usr/bin/time false 2> /dev/null
    cat < /dev/null
}
cache_progs   # Минимизация задержки запуска.

date +%s.%N | grep -qF 'N' && use_python=1 # Если `date` без наносекунд.
now() {
    if [ "$use_python" ]; then
        echo "import time; print time.time()" 2>/dev/null | python
    else
        printf "%.2f" `date +%s.%N`
    fi
}

fmt_seconds() {
    seconds=$1
    mins=`echo $seconds/60 | bc`
    if [ "$mins" != "0" ]; then
        seconds=`echo "$seconds - ($mins*60)" | bc`
```

```
        echo "$mins:$seconds"
    else
        echo "$seconds"
    fi
}

total() {
    end=`now`
    total=`echo "$end - $start" | bc`
    fmt_seconds $total
}

stop() {
    [ "$lapped" ] && lap "$laptime" "display"
    total
    exit
}

lap() {
    laptime=`echo "$1" | sed -n 's/.*real[^0-9.]*\(.*\)/\1/p'`
    [ ! "$laptime" -o "$laptime" = "0.00" ] && return
    # Слишком частые сигналы.
    laptotal=`echo $laptime+0$laptotal | bc`
    if [ "$2" = "display" ]; then
        lapcount=`echo 0$lapcount+1 | bc`
        laptime=`fmt_seconds $laptotal`
        echo $laptime "($lapcount)"
        lapped="true"
        laptotal="0"
    fi
}

echo -n "Пространство | с ? для разделения | Ctrl-C для остановки |
Пространство для запуска">&2

while true; do
    trap true INT QUIT  # Установка сигнала обработки.
    laptime=`/usr/bin/time -p 2>&1 cat >/dev/null`
    ret=$?
    trap '' INT QUIT    # Игнорировать сигналы сценария.
    if [ $ret -eq 1 -o $ret -eq 2 -o $ret -eq 130 ]; then # SIGINT = стоп
        [ ! "$start" ] && { echo >&2; exit; }
        stop
    elif [ $ret -eq 3 -o $ret -eq 131 ]; then            # SIGQUIT = этап
        if [ ! "$start" ]; then
            start=`сейчас` || exit 1
            echo >&2
            continue
        fi
        lap "$laptime" "display"
    else                 # eof = разделитель
        [ ! "$start" ] && continue
        total
        lap "$laptime"  # Обновление.
    fi
done

exit $?
```

Пример A-44. Универсальность решения домашних заданий сценариев оболочки

```
#!/bin/bash
#  homework.sh: Универсальность решения домашних заданий.
#  Автор: M. Leo Cooper
#  Если вы замените имя автора на ваше собственное имя, то это будет плагиат,
#+ возможно меньший грех, чем обман при решении домашнего задания!
#  License: Public Domain

#  Этот сценарий можете отдать вашему преподавателю, для выполнения ВСЕХ
#+ домашних заданий сценариев оболочки.
#  Он содержит мало комментариев, но вы, изучающий, легко исправите это.
#  Автор сценария снимает с себя всякую ответственность!

DLA=1
P1=2
P2=4
P3=7
PP1=0
PP2=8
MAXL=9
E_LZY=99

declare -a L
L[0]="3 4 0 17 29 8 13 18 19 17 20 2 19 14 17 28"
L[1]="8 29 12 14 18 19 29 4 12 15 7 0 19 8 2 0 11 11 24 29 17 4 6 17 4 19"
L[2]="29 19 7 0 19 29 8 29 7 0 21 4 29 13 4 6 11 4 2 19 4 3"
L[3]="19 14 29 2 14 12 15 11 4 19 4 29 19 7 8 18 29"
L[4]="18 2 7 14 14 11 22 14 17 10 29 0 18 18 8 6 13 12 4 13 19 26"
L[5]="15 11 4 0 18 4 29 0 2 2 4 15 19 29 12 24 29 7 20 12 1 11 4 29"
L[6]="4 23 2 20 18 4 29 14 5 29 4 6 17 4 6 8 14 20 18 29"
L[7]="11 0 25 8 13 4 18 18 27"
L[8]="0 13 3 29 6 17 0 3 4 29 12 4 29 0 2 2 14 17 3 8 13 6 11 24 26"
L[9]="19 7 0 13 10 29 24 14 20 26"

declare -a \
alph=( A B C D E F G H I J K L M N O P Q R S T U V W X Y Z . , : ' ' )


pt_lt ()
{
  echo -n "${alph[$1]}"
  echo -n -e "\a"
  sleep $DLA
}

b_r ()
{
 echo -e '\E[31;48m\033[1m'
}

cr ()
{
 echo -e "\a"
 sleep $DLA
}

restore ()
{
```

```
  echo -e '\033[0m'            # Отключаем жирный.
  tput sgr0                    # Включаем обычный.
}


p_l ()
{
  for ltr in $1
  do
    pt_lt "$ltr"
  done
}

# ----------------------
b_r

for i in $(seq 0 $MAXL)
do
  p_l "${L[i]}"
  if [[ "$i" -eq "$P1" || "$i" -eq "$P2" || "$i" -eq "$P3" ]]
  then
    cr
  elif [[ "$i" -eq "$PP1" || "$i" -eq "$PP2" ]]
  then
    cr; cr
  fi
done

restore
# ----------------------

echo

exit $E_LZY

#  Типичный пример запутанного сценария, который трудно
#  понять и сложно поддерживать.
#  В карьере администратора вы будете довольно часто
#+ работать с такими гадостями
```

Пример A-45. Рыцарский турнир

```
#!/bin/bash
# ktour.sh

# Автор: mendel cooper
# reldate: 12 Jan 2009
# license: public domain
# (Not much sense GPLing something that's pretty much in the common
#+ domain anyhow.)

###############################################################################
#             Рыцарский турнир, обычные проблемы.                             #
#             ===================================                             #
#  Рыцарь ходит на следующий квадрат шахматной доски,                         #
#  но ему нельзя возвращаться на любой, уже посещенный, квадрат.              #
#                                                                             #
#  А правда, почему Сэру Рыцарю нельзя отступать?                             #
#  Может быть он имеет привычку развлекаться в предрассветные                 #
#+ утренние часы                                                              #
```

```
#  Возможно он оставляет крошки от пиццы в постели, пустые пивные #
#+ бутылки на полу и засоряет сантехнику ....                      #
#                                                                  #
#  ---------------------------------------------------------------  #
#                                                                  #
#  Использование: ktour.sh [начальная клетка] [глупость]           #
#                                                                  #
#  Обратите внимание, что номер стартовой клетки может находиться #
#+ в диапазоне 0 - 63 ... или                                      #
#  клетка обозначается в обычной шахматной нотации,                #
#  как a1, f5, h3, и т.д..                                         #
#                                                                  #
#  Если стартовая клетка не указана,                               #
#+ то стартует со случайной клетки доски.                          #
#                                                                  #
# "глупость" второй параметр в бестолковой стратегии.              #
#                                                                  #
#  Примеры:                                                        #
#  ktour.sh 23          начать с клетки #23 (h3)                   #
#  ktour.sh g6 stupid   начать с клетки #46,                       #
#                 используется стратегия "stupid" (не-Warnsdorff). #
####################################################################

DEBUG=      #  Вывод  отладочной информации в stdout.
SUCCESS=0
FAIL=99
BADMOVE=-999
FAILURE=1
LINELEN=21  # Количество ходов выводимое в строке.
# ------------------------------------- #
# Параметры массива доски
ROWS=8   # доска 8 x 8.
COLS=8
let "SQUARES = $ROWS * $COLS"
let "MAX = $SQUARES - 1"
MIN=0
# 64 клетки доски, индексируются от 0 до 63.

VISITED=1
UNVISITED=-1
UNVSYM="##"
# ------------------------------------- #
# Глобальные переменные.
startpos=    # Стартовая позиция (клетки #0 - 63).
currpos=     # Текущая позиция.
movenum=     # Номер хода.
CRITPOS=37   # Временная стартовая позиция f5!

declare -i board
# Используется одномерный массив для имитации двумерного.
# Это может осложнить жизнь и привести к уродству; См. ниже.
declare -i moves  # Смещения от текущей позиции рыцаря.


initialize_board ()
{
  local idx

  for idx in {0..63}
  do
    board[$idx]=$UNVISITED
```

```
  done
}



print_board ()
{
  local idx

  echo "    _________________________________________"
  for row in {7..0}               #  Обратный порядок рядов ...
  do                              #+ поэтому выводится как на
                                  #+ шахматной доске.
    let "rownum = $row + 1"       #  Исходная нумерация рядов с 1.
    echo -n "$rownum  |"          #  Отмечает края границ доски и
    for column in {0..7}          #+ "математическая запись."
    do
      let "idx = $ROWS*$row + $column"
      if [ ${board[idx]} -eq $UNVISITED ]
      then
        echo -n "$UNVSYM   "      ##
      else                        # Отмечает квадрат с номером хода.
        printf "%02d " "${board[idx]}"; echo -n "  "
      fi
    done
    echo -e -n "\b\b\b|"  # \b Это пробел.
    echo                  # -e позволяет выводить экранированные символы.
  done

  echo "    ------------------------------------"
  echo "     a    b    c    d    e    f    g    h"
}



failure()
{
  echo
  print_board
  echo
  echo    "   O!!! Нет клеток для ходов!"
  echo -n "   Рыцарский турнир окончен"
  echo    " на $(to_algebraic $currpos) [square #$currpos]"
  echo    "   после $movenum ходов!"
  echo
  exit $FAIL
}



xlat_coords ()   #  Перевод координат x/y в позиции доски
{                #+ (элемент массива доски #).
  #  Ввод пользователем стартового положения на доске, как координаты X/Y.
  #  Эта функция не использовалась в первоначальном релизе ktour.sh.
  #  Можно использовать обновленную версию совместимую со стандартной
  #+ реализацией Рыцарского Турнира на Си, Python и т.д.
  if [ -z "$1" -o -z "$2" ]
  then
    return $FAIL
  fi
```

```
  local xc=$1
  local yc=$2

  let "board_index = $xc * $ROWS + yc"

  if [ $board_index -lt $MIN -o $board_index -gt $MAX ]
  then
    return $FAIL    # Выход за границы доски!
  else
    return $board_index
  fi
}



to_algebraic ()   #  Перевод позиции на доске (# элемента массива доски)
{                 #+ в математическую нотацию используемую шахматистами.
  if [ -z "$1" ]
  then
    return $FAIL
  fi

  local element_no=$1   # Числовая позиция на доске.
  local col_arr=( a b c d e f g h )
  local row_arr=( 1 2 3 4 5 6 7 8 )

  let "row_no = $element_no / $ROWS"
  let "col_no = $element_no % $ROWS"
  t1=${col_arr[col_no]}; t2=${row_arr[row_no]}
  local apos=$t1$t2   # Конкатенация (объединение).
  echo $apos
}



from_algebraic ()   #  Перевод математической шахматной нотации
{                   #+ в числовую позицию на доске (# элемента массива доски).
                    #  Или определение числового ввода и возвращение его
                    #+ не измененным.
  if [ -z "$1" ]
  then
    return $FAIL
  fi   # Если нет аргумента командной строки, то, по умолчанию, стартует со
       #+ случайной позиции.

  local ix
  local ix_count=0
  local b_index     # Индекс доски [0-63]
  local alpos="$1"

  arow=${alpos:0:1} # позиция = 0, величина = 1
  acol=${alpos:1:1}

  if [[ $arow =~ [[:digit:]] ]]   #  Введены числа?
  then       #  POSIX char class
    if [[ $acol =~ [[:alpha:]] ]] # Число следует за буквой? Не правильно!
      then return $FAIL
    else if [ $alpos -gt $MAX ]   # За границами доски?
      then return $FAIL
    else return $alpos            #  Возвращает не измененное число(а) ...
      fi                          #+ если в диапазоне.
```

```
    fi
  fi

  if [[ $acol -eq $MIN || $acol -gt $ROWS ]]
  then        # 1 - 8 за границами диапазона?
    return $FAIL
  fi

  for ix in a b c d e f g h
  do  # Преобразование колонки букв в колонку цифр.
   if [ "$arow" = "$ix" ]
   then
     break
   fi
  ((ix_count++))    # Увеличиваем индекс отсчета.
  done

  ((acol--))        # Уменьшение, преобразующее индексируемый массив в ноль.
  let "b_index = $ix_count + $acol * $ROWS"

  if [ $b_index -gt $MAX ]   # За границами доски?
  then
    return $FAIL
  fi

  return $b_index

}


generate_moves ()   #  Подсчет всех правильных ходов рыцаря,
{                   #+ относительно текущей позиции ($1),
                    #+ и сохранение в массиве ${moves}.
  local kt_hop=1    #  Одна клетка  :: короткий ход рыцаря.
  local kt_skip=2   #  Две клетки :: длинный ход рыцаря.
  local valmov=0    #  Правильные ходы.
  local row_pos; let "row_pos = $1 % $COLS"


  let "move1 = -$kt_skip + $ROWS"           # 2 влево, 1 вверх
    if [[ `expr $row_pos - $kt_skip` -lt $MIN ]]   #  Ужасно!
    then                                           #  Не выходите за границы
                                                   #+ доски.
      move1=$BADMOVE                               #  Даже временно.
    else
      ((valmov++))
    fi
  let "move2 = -$kt_hop + $kt_skip * $ROWS" # 1 влево, 2 вверх
    if [[ `expr $row_pos - $kt_hop` -lt $MIN ]]    # Кошмар продолжается ...
    then
      move2=$BADMOVE
    else
      ((valmov++))
    fi
  let "move3 =  $kt_hop + $kt_skip * $ROWS" # 1 вправо, 2 вверх
    if [[ `expr $row_pos + $kt_hop` -ge $COLS ]]
    then
      move3=$BADMOVE
    else
      ((valmov++))
```

```
    fi
  let "move4 =  $kt_skip + $ROWS"           # 2 вправо, 1 вверх
    if [[ `expr $row_pos + $kt_skip` -ge $COLS ]]
    then
      move4=$BADMOVE
    else
      ((valmov++))
    fi
  let "move5 =  $kt_skip - $ROWS"           # 2 вправо, 1 вниз
    if [[ `expr $row_pos + $kt_skip` -ge $COLS ]]
    then
      move5=$BADMOVE
    else
      ((valmov++))
    fi
  let "move6 =  $kt_hop - $kt_skip * $ROWS" # 1 вправо, 2 вниз
    if [[ `expr $row_pos + $kt_hop` -ge $COLS ]]
    then
      move6=$BADMOVE
    else
      ((valmov++))
    fi
  let "move7 = -$kt_hop - $kt_skip * $ROWS" # 1 влево, 2 вниз
    if [[ `expr $row_pos - $kt_hop` -lt $MIN ]]
    then
      move7=$BADMOVE
    else
      ((valmov++))
    fi
  let "move8 = -$kt_skip - $ROWS"           # 2 влево, 1 вниз
    if [[ `expr $row_pos - $kt_skip` -lt $MIN ]]
    then
      move8=$BADMOVE
    else
      ((valmov++))
    fi   # Должен быть лучший способ, что бы сделать это.

  local m=( $valmov $move1 $move2 $move3 $move4 $move5 $move6 $move7 $move8 )
  # ${moves[0]} = число правильных ходов.
  # ${moves[1]} ... ${moves[8]} = возможные ходы.
  echo "${m[*]}"    # Элементы массива захватаченной переменной stdout.

}

is_on_board ()  # Это фактическое положение  на доске?
{
  if [[ "$1" -lt "$MIN" || "$1" -gt "$MAX" ]]
  then
    return $FAILURE
  else
    return $SUCCESS
  fi
}


do_move ()      # Перемещение рыцаря!
{
  local valid_moves=0
  local aapos
  currposl="$1"
```

```
  lmin=$ROWS
  iex=0
  squarel=
  mpm=
  mov=
  declare -a p_moves

  ########################## РЕШЕНИЕ-ХОД ##############################
  if [ $startpos -ne $CRITPOS ]
  then   # CRITPOS = square #37
    decide_move
  else                    # Нужен особый патч для startpos=37 !!!
    decide_move_patched   # Почему именно этот ход, а не другой ???
  fi
  ######################################################################

  (( ++movenum ))         # Увеличивает счетчик ходов.
  let "square = $currposl + ${moves[iex]}"

  ##################    ОТЛАДКА    ###############
  if [ "$DEBUG" ]
    then debug   # Вывод отладочной информации.
  fi
  ################################################

  if [[ "$square" -gt $MAX || "$square" -lt $MIN ||
        ${board[square]} -ne $UNVISITED ]]
  then
    (( --movenum ))             #  Уменьшает счетчик ходов,
    echo "RAN OUT OF SQUARES!!!" #+ если предыдущий ход был не правильным.
    return $FAIL
  fi

  board[square]=$movenum
  currpos=$square       # Обновление текущей позиции.
  ((valid_moves++));    # ходы[0]=$valid_moves
  aapos=$(to_algebraic $square)
  echo -n "$aapos "
  test $(( $Moves % $LINELEN )) -eq 0 && echo
  # Вывод ходов LINELEN=21 построчно. Правильный тур выводит 3 строки.
  return $valid_moves   # Выбор клетки для хода!
}


do_move_stupid()   #  Глупый алгоритм,
{                  #+ любезность автора сценария, *не* Warnsdorff.
  local valid_moves=0
  local movloc
  local squareloc
  local aapos
  local cposloc="$1"

  for movloc in {1..8}
  do       # Переход к первой найденной свободной клетке.
    let "squareloc = $cposloc + ${moves[movloc]}"
    is_on_board $squareloc
    if [ $? -eq $SUCCESS ] && [ ${board[squareloc]} -eq $UNVISITED ]
    then   # Добавляет условия к проверке if, выше, для улучшения алгоритма.
      (( ++movenum ))
      board[squareloc]=$movenum
      currpos=$squareloc     # Обновляет текущую позицию.
```

```
      ((valid_moves++));     # ходы[0]=$valid_moves
      aapos=$(to_algebraic $squareloc)
      echo -n "$aapos "
      test $(( $Moves % $LINELEN )) -eq 0 && echo   # Вывод 21 хода/строки.
      return $valid_moves    # Выбор клетки для хода!
    fi
  done

  return $FAIL
  #  Если во всех 8 проходах не найдено свободных клеток,
  #+ то Рыцарский турнир завершается поражением.

  #  Глупый алгоритм обычно терпит неудачу примерно через 30 - 40 ходов,
  #+ но выполняется _гораздо_ быстрее, чем функция Warnsdorff do_move().
}


decide_move ()          #  Какой будет сделан ход?
{                       #  О, неправильная startpos=37 !!!
  for mov in {1..8}
  do
    let "squarel = $currposl + ${moves[mov]}"
    is_on_board $squarel
    if [[ $? -eq $SUCCESS && ${board[squarel]} -eq $UNVISITED ]]
    then   #  Ищем свободную клетку с возможно меньшими будущими ходами.
           #  Это алгоритм Warnsdorff.
           #  Что случится, если рыцарь пойдет по направлению к внешнему
           #+ краю доски, а затем по спирали внутрь.
           #  Учитываем два, или более, вероятных хода с возможно меньшими
           #+ будущими ходами, данная реализация выбирает _первый_ из
           #+ этих шагов.
           #  Это означает, что не обязательно существует уникальное решение
           #+ для любой заданной стартовой позиции.

      possible_moves $squarel
      mpm=$?
      p_moves[mov]=$mpm

      if [ $mpm -lt $lmin ]  # Если меньше, чем предыдущий минимум ...
      then   #   ^^
        lmin=$mpm            # Обновляем минимум.
        iex=$mov             # Сохраняем индекс.
      fi

    fi
  done
}


decide_move_patched ()          #  Решаем какой ход сделать,
{  #       ^^^^^^^              #+ но только, если startpos=37 !!!
  for mov in {1..8}
  do
    let "squarel = $currposl + ${moves[mov]}"
    is_on_board $squarel
    if [[ $? -eq $SUCCESS && ${board[squarel]} -eq $UNVISITED ]]
    then
      possible_moves $squarel
      mpm=$?
      p_moves[mov]=$mpm
```

```
      if [ $mpm -le $lmin ]  # Если меньше чем или равно предыдущему минимуму!
      then #     ^^
        lmin=$mpm
        iex=$mov
      fi

    fi
  done                       # Это можно улучшить
}



possible_moves ()            #  Подсчет числа возможных ходов,
{                            #+ задаваемых текущей позицией.

  if [ -z "$1" ]
  then
    return $FAIL
  fi

  local curr_pos=$1
  local valid_movl=0
  local icx=0
  local movl
  local sq
  declare -a movesloc

  movesloc=( $(generate_moves $curr_pos) )

  for movl in {1..8}
  do
    let "sq = $curr_pos + ${movesloc[movl]}"
    is_on_board $sq
    if [ $? -eq $SUCCESS ] && [ ${board[sq]} -eq $UNVISITED ]
    then
      ((valid_movl++));
    fi
  done

  return $valid_movl         # Выбираем клетку для хода!
}


strategy ()
{
  echo

  if [ -n "$STUPID" ]
  then
    for Moves in {1..63}
    do
      cposl=$1
      moves=( $(generate_moves $currpos) )
      do_move_stupid "$currpos"
      if [ $? -eq $FAIL ]
      then
        failure
      fi
      done
  fi
```

```
  #  Здесь не нужно "else",
  #+ потому, что Глупая стратегия всегда приводит к поражению и выходу!
  for Moves in {1..63}
  do
    cposl=$1
    moves=( $(generate_moves $currpos) )
    do_move "$currpos"
    if [ $? -eq $FAIL ]
    then
      failure
    fi

  done
        #  Два цикла do, выше, можно объединить в один, но это
  echo  #+ замедлит все исполнение.

  print_board
  echo
  echo "Рыцарский турнир окончен на $(to_algebraic $currpos) [square
#$currpos]."
  return $SUCCESS
}

debug ()
{       # Включаем настройку DEBUG=1 в начале сценария.
  local n

  echo "================================="
  echo "  Ход номер  $movenum:"
  echo " *** возможных ходов = $mpm ***"
# echo "### клетка = $square ###"
  echo "lmin = $lmin"
  echo "${moves[@]}"

  for n in {1..8}
  do
    echo -n "($n):${p_moves[n]} "
  done

  echo
  echo "iex = $iex :: moves[iex] = ${moves[iex]}"
  echo "square = $square"
  echo "================================="
  echo
} # Дает довольно полное представление о состоянии после каждого хода.



# ================================================================ #
# int main () {
from_algebraic "$1"
startpos=$?
if [ "$startpos" -eq "$FAIL" ]          # Нормально, даже если нет $1.
then   #         ^^^^^^^^^^^^             Нормально, даже если введено -lt 0.
  echo "Нет указания на стартовую клетку (или не правильный ввод)."
  let "startpos = $RANDOM % $SQUARES"   # Допустимый диапазон 0 - 63.
fi


if [ "$2" = "stupid" ]
then
```

```
  STUPID=1
  echo -n "     ### Глупая стратегия ###"
else
  STUPID=''
  echo -n "  *** Алгоритм Warnsdorff ***"
fi


initialize_board

movenum=0
board[startpos]=$movenum   # Отмечает каждую клетку доски с номером хода.
currpos=$startpos
algpos=$(to_algebraic $startpos)

echo; echo "Начали с $algpos [square #$startpos] ..."; echo
echo -n "Ходы:"

strategy "$currpos"

echo

exit 0   # return 0;

# }      # Окончание псевдо-функции main().
# ================================================================ #


# Упражнения:
# ---------
#
# 1) Расширьте этот пример для доски 10 x 10 или более.
# 2) Улучшите 'глупую' стратегию, изменив
#    функцию do_move_stupid.
#    Подсказка: Избегацте ходов в угловые клетки в начале игры
#               (в противоположность алгоритму Warnsdorff!).
# 3) Этот сценарий можно значительное улучшить и оптимизировать,
#    особенно в плохо написанной функции generate_moves ()
#    и в патче DECIDE-MOVE функции do_move().
#    Необходимо понять, почему стандартный алгоритм не выполняется для
#+   startpos=37 ... а _не_ для_любой_другой, в том числе симметричной
#+   startpos=26.
#    Возможно, при расчете возможных ходов, считать обратные ходы в исходный
#+   квадрат. Если так, то это можно относительно легко исправить.
```

Пример A-46. Волшебные квадраты

```
#!/bin/bash
# msquare.sh
# Генератор волшебных квадратов (квадратов только нечетного порядка!)

# Автор: mendel cooper
# reldate: 19 Jan. 2009
# License: Public Domain
# A C-program by the very talented Kwon Young Shin inspired this script.
#     http://user.chollian.net/~brainstm/MagicSquare.htm

# Объяснение: "Волшебный квадрат" это двумерный массив
```

```
#             целых чисел в котором все ряды, столбцы
#             и *большие* диагонали равны одной и той же сумме чисел в них.
#             Этот "квадрат," массив, имеет какое-то одинаковое число рядов и
#             стобцов. Это число является "порядком."
# Пример магического квадрата 3 порядка:
#   8  1  6
#   3  5  7
#   4  9  2
# Все ряды, столбцы, и две диагонали имеют в сумме 15.


# Глобальные
EVEN=2
MAXSIZE=31   # 31 ряд x 31 колонку.
E_usage=90   # Ошибка вызова.
dimension=
declare -i square

usage_message ()
{
  echo "Usage: Порядок $0"
  echo "   ... где \"порядок\" (размер квадрата) нечетное целое число"
  echo "       в диапазоне 3 - 31."
  #  На самом деле работает с квадратами до порядка 159,
  #+ но большие квадраты не будут нормально выводиться в окне терминала.
  #  Попробуйте увеличить MAXSIZE, выше.
  exit $E_usage
}


calculate ()       # Здесь производится основная работа.
{
  local row col index dimadj j k cell_val=1
  dimension=$1

  let "dimadj = $dimension * 3"; let "dimadj /= 2"   # x 1.5, затем обрезается.

  for ((j=0; j < dimension; j++))
  do
    for ((k=0; k < dimension; k++))
    do  # Расчет индексов, а затем преобразование в индекс 1-мерного массива.
        # Bash не поддерживает многомерные массивы. Жаль.
      let "col = $k - $j + $dimadj"; let "col %= $dimension"
      let "row = $j * 2 - $k + $dimension"; let "row %= $dimension"
      let "index = $row*($dimension) + $col"
      square[$index]=cell_val; ((cell_val++))
    done
  done
}     # Простая математика, не требующая виртуализации.


print_square ()              # Вывод квадрата, один ряд за раз.
{
  local row col idx d1
  let "d1 = $dimension - 1"   # Настройка нулевого индекса массива.

  for row in $(seq 0 $d1)
  do

    for col in $(seq 0 $d1)
    do
```

```
      let "idx = $row * $dimension + $col"
      printf "%3d " "${square[idx]}"; echo -n "  "
    done   # Выводится порядок 13, аккуратные 80-колонок в окне терминала.

    echo   # Перевод строки после каждого ряда.
  done
}


###################################################
if [[ -z "$1" ]] || [[ "$1" -gt $MAXSIZE ]]
then
  usage_message
fi

let "test_even = $1 % $EVEN"
if [ $test_even -eq 0 ]
then           # Не удается обработать даже порядок квадратов.
  usage_message
fi

calculate $1
print_square   # echo "${square[@]}"   # ОТЛАДКА

exit $?
###################################################


# Упражнения:
# ---------
# 1) Добавьте функцию для подсчета суммы каждого ряда, столбца,
#    и *длинной* диагонали. Суммирование должно быть математическим.
#    Это "волшебная постоянная" конкретного порядка данного квадрата.
# 2) Автоматическое вычисление функцией print_square, сколько места нужно
#    выделить между элементами квадрата для оптимизации изображения.
#    Может потребоваться параметризация строки «printf».
# 3) Добавьте соответствующие функции для создания магических квадратов
#    *даже* с числом строк/столбцов. Это не обычно(!).
#    См. URL Kwon Young Shin, выше, для справки.
```

Пример A-47. «Пятнашки»

```
#!/bin/bash
# fifteen.sh

# Классические "Пятнашки"
# Автор: Antonio Macchi
# Слегка отредактировано и прокомментировано автором ABS Guide.
# В ABS Guide используется с разрешения. (C!)

#  Изобретение «пятнашек» приписываются
#+ Sam Loyd или Noyes Palmer Chapman.
#  Головоломка была очень популярна в прошлом, 19, веке.

#  Объект: Переставьте числа так, что бы они стояли по порядку,
#+ от 1 до 15:   _________________
#                |  1   2   3   4 |
#                |  5   6   7   8 |
```

```
#                |  9  10  11  12 |
#                | 13  14  15     |
#                 ----------------


######################
# Константы          #
  SQUARES=16         #
  FAIL=70            #
  E_PREMATURE_EXIT=80 #
######################


##########
# Данные #
##########

Puzzle=( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 " " )


############
# Функции #
############

function swap
{
  local tmp

  tmp=${Puzzle[$1]}
  Puzzle[$1]=${Puzzle[$2]}
  Puzzle[$2]=$tmp
}


function Jumble
{ # Перемешивание перед началом игры.
  local i pos1 pos2

  for i in {1..100}
  do
    pos1=$(( $RANDOM % $SQUARES))
    pos2=$(( $RANDOM % $SQUARES ))
    swap $pos1 $pos2
  done
}


function PrintPuzzle
{
  local i1 i2 puzpos
  puzpos=0

  clear
  echo "Введите  quit  для выхода."; echo   # Лучше чем Ctl-C.

  echo ",----.----.----.----."   # Верхняя граница.
  for i1 in {1..4}
  do
    for i2 in {1..4}
    do
      printf "| %2s " "${Puzzle[$puzpos]}"
```

```
      (( puzpos++ ))
    done
    echo "|"                      # Правая граница.
    test $i1 = 4 || echo "+----+----+----+----+"
  done
  echo "'----'----'----'----'"   # Нижняя граница.
}


function GetNum
{ # Проверка правильности ввода.
  local puznum garbage

  while true
  do
          echo "Ходы: $moves" # Подсчет правильных ходов.
    read -p "Число для хода: " puznum garbage
      if [ "$puznum" = "quit" ]; then echo; exit $E_PREMATURE_EXIT; fi
    test -z "$puznum" -o -n "${puznum//[0-9]/}" && continue
    test $puznum -gt 0 -a $puznum -lt $SQUARES && break
  done
  return $puznum
}


function GetPosFromNum
{ # $1 = puzzle-number
  local puzpos

  for puzpos in {0..15}
  do
    test "${Puzzle[$puzpos]}" = "$1" && break
  done
  return $puzpos
}


function Move
{ # $1=Puzzle-pos
  test $1 -gt 3 && test "${Puzzle[$(( $1 - 4 ))]}" = " "\
       && swap $1 $(( $1 - 4 )) && return 0
  test $(( $1%4 )) -ne 3 && test "${Puzzle[$(( $1 + 1 ))]}" = " "\
       && swap $1 $(( $1 + 1 )) && return 0
  test $1 -lt 12 && test "${Puzzle[$(( $1 + 4 ))]}" = " "\
       && swap $1 $(( $1 + 4 )) && return 0
  test $(( $1%4 )) -ne 0 && test "${Puzzle[$(( $1 - 1 ))]}" = " " &&\
       swap $1 $(( $1 - 1 )) && return 0
  return 1
}


function Solved
{
  local pos

  for pos in {0..14}
  do
    test "${Puzzle[$pos]}" = $(( $pos + 1 )) || return $FAIL
    # Проверка, является ли число в каждом квадрате = номеру квадрата.
  done
  return 0   # Успешное решение.
```

```
}


################### MAIN () ######################{
moves=0
Jumble

while true   # Цикл не прерывается до окончания решения головоломки.
do
  echo; echo
  PrintPuzzle
  echo
  while true
  do
    GetNum
    puznum=$?
    GetPosFromNum $puznum
    puzpos=$?
    ((moves++))
    Move $puzpos && break
  done
  Solved && break
done

echo;echo
PrintPuzzle
echo; echo "БРАВО!"; echo

exit 0
#########################################################}

#  Упражнение:
#  --------
#  Перепишите сценарий для вывода на экран, вместо чисел 1 – 15,
#+ букв от A до O.
```

Пример A-48. Ханойские пагоды, *графическая версия*

```
#! /bin/bash
# Ханойские пагоды
# Оригинальный сценарий (hanoi.bash) copyright (C) 2000 Amit Singh.
# Все права защищены.
# http://hanoi.kernelthread.com

#  hanoi2.bash
#  Version 2.00: modded for ASCII-graphic display.
#  Version 2.01: fixed no command-line param bug.
#  Используемый код предоставлен Antonio Macchi,
#+ и сильно отредактирован автором ABS Guide.
#  Этот вариант подпадает под настоящие авторские права, см. выше.
#  В ABS Guide используется с разрешения Amit Singh (спасибо!).


###   Переменные && проверка правильности   ###

E_NOPARAM=86
E_BADPARAM=87            # Неправильное количество дисков переданных сценарию.
E_NOEXIT=88
```

```
DISKS=${1:-$E_NOPARAM}   # Должно быть указано количество дисков.
Moves=0

MWIDTH=7
MARGIN=2
#  Произвольные «волшебные» константы ; хорошо работают для относительно
 #+ малого количества дисков.
# BASEWIDTH=51   # Оригинальный код.
let "basewidth = $MWIDTH * $DISKS + $MARGIN"       # «База» под стержнями.
#  «Алгоритм» вероятно был улучшен.

###   Выводимые на экран переменные   ###
let "disks1 = $DISKS - 1"
let "spaces1 = $DISKS"
let "spaces2 = 2 * $DISKS"

let "lastmove_t = $DISKS - 1"                      # Последний ход?


declare -a Rod1 Rod2 Rod3

###   ########################   ###


function repeat  {  # $1=символ $2=число повторений
  local n           # Повторный вывод символа на экран.

  for (( n=0; n<$2; n++ )); do
    echo -n "$1"
  done
}

function FromRod  {
  local rod summit weight sequence

  while true; do
    rod=$1
    test ${rod/[^123]/} || continue

    sequence=$(echo $(seq 0 $disks1 | tac))
    for summit in $sequence; do
      eval weight=\${Rod${rod}[$summit]}
      test $weight -ne 0 &&
           { echo "$rod $summit $weight"; return; }
    done
  done
}


function ToRod  { # $1=предыдущее (FromRod) значение
  local rod firstfree weight sequence

  while true; do
    rod=$2
    test ${rod/[^123]} || continue

    sequence=$(echo $(seq 0 $disks1 | tac))
    for firstfree in $sequence; do
      eval weight=\${Rod${rod}[$firstfree]}
      test $weight -gt 0 && { (( firstfree++ )); break; }
```

```
    done
    test $weight -gt $1 -o $firstfree = 0 &&
         { echo "$rod $firstfree"; return; }
  done
}


function PrintRods  {
  local disk rod empty fill sp sequence


  repeat " " $spaces1
  echo -n "|"
  repeat " " $spaces2
  echo -n "|"
  repeat " " $spaces2
  echo "|"

  sequence=$(echo $(seq 0 $disks1 | tac))
  for disk in $sequence; do
    for rod in {1..3}; do
      eval empty=$(( $DISKS - (Rod${rod}[$disk] / 2) ))
      eval fill=\${Rod${rod}[$disk]}
      repeat " " $empty
      test $fill -gt 0 && repeat "*" $fill || echo -n "|"
      repeat " " $empty
    done
    echo
  done
  repeat "=" $basewidth   # Отрисовка "базы" под стержни.
  echo
}


display ()
{
  echo
  PrintRods

  # Получает номер стержня, сумму и значение.
  first=( `FromRod $1` )
  eval Rod${first[0]}[${first[1]}]=0

  # Получает номер стержня и первую свободную позицию
  second=( `ToRod ${first[2]} $2` )
  eval Rod${second[0]}[${second[1]}]=${first[2]}


echo; echo; echo
if [ "${Rod3[lastmove_t]}" = 1 ]
then   # Последний ход? Если да, то выводится финальная позиция.
    echo "+  Финальная позиция: $Moves ходов"; echo
    PrintRods
  fi
}


# Отсюда вниз, все как в оригинальном сценарии (hanoi.bash).

dohanoi() {   # Рекурсивная функция.
    case $1 in
```

```
    0)
        ;;
    *)
        dohanoi "$(($1-1))" $2 $4 $3
        if [ "$Moves" -ne 0 ]
        then
          echo "+  Позиция после хода $Moves"
        fi
        ((Moves++))
        echo -n "   Следующий ход будет:  "
        echo $2 "-->" $3
          display $2 $3
        dohanoi "$(($1-1))" $4 $3 $2
        ;;
    esac
}


setup_arrays ()
{
  local dim n elem

  let "dim1 = $1 - 1"
  elem=$dim1

  for n in $(seq 0 $dim1)
  do
   let "Rod1[$elem] = 2 * $n + 1"
   Rod2[$n]=0
   Rod3[$n]=0
   ((elem--))
  done
}


###   Главная   ###

setup_arrays $DISKS
echo; echo "+  Start Position"

case $# in
    1) case $(($1>0)) in     # Должен иметься в наличии хотя бы один диск.
       1)
           disks=$1
           dohanoi $1 1 3 2
#          Total moves = 2^n - 1, где n = количество дисков.
           echo
           exit 0;
           ;;
       *)
           echo "$0: Неправильное значение количества дисков";
           exit $E_BADPARAM;
           ;;
       esac
    ;;
    *)
       clear
       echo "Используйте: $0 N"
       echo "       Где \"N\" это число дисков."
       exit $E_NOPARAM;
       ;;
```

```
esac

exit $E_NOEXIT   # Не следует здесь выходить.

#  Примечание:
#  Перенаправьте вывод сценария в файл, в противном случае он выведется на
#+ экран.
```

Пример A-49. Ханойские пагоды, *другая графическая версия*

```
#! /bin/bash
# Ханойские пагоды
# Оригинальный сценарий (hanoi.bash) copyright (C) 2000 Amit Singh.
# Все права защищены.
# http://hanoi.kernelthread.com

#  hanoi2.bash
#  Version 2: modded for ASCII-graphic display.
#  Uses code contributed by Antonio Macchi,
#+ with heavy editing by ABS Guide author.
#  This variant also falls under the original copyright, see above.
#  Используется в ABS Guide с разрешения Amit Singh (спасибо!).


#   Переменные   #
E_NOPARAM=86
E_BADPARAM=87   # Не правильное число дисков передаваемых сценарию.
E_NOEXIT=88
DELAY=2         # Интервал, в секундах, между ходами. Измените, если нужно.
DISKS=$1
Moves=0

MWIDTH=7
MARGIN=2
#  Произвольные «волшебные» константы ; хорошо работают для относительно малого
#+ количества дисков.
# BASEWIDTH=51   # Оригинальный код.
let "basewidth = $MWIDTH * $DISKS + $MARGIN" # "База" для стержней.
# "Алгоритм" выше вероятно улучшен.

# Выводимые на экран переменные.
let "disks1 = $DISKS - 1"
let "spaces1 = $DISKS"
let "spaces2 = 2 * $DISKS"

let "lastmove_t = $DISKS - 1"                # Последний ход?


declare -a Rod1 Rod2 Rod3

##################


function repeat  {  # $1=символ $2=число повторений
  local n           # Повторный вывод символа.

  for (( n=0; n<$2; n++ )); do
    echo -n "$1"
```

```
  done
}

function FromRod  {
  local rod summit weight sequence

  while true; do
    rod=$1
    test ${rod/[^123]/} || continue

    sequence=$(echo $(seq 0 $disks1 | tac))
    for summit in $sequence; do
      eval weight=\${Rod${rod}[$summit]}
      test $weight -ne 0 &&
           { echo "$rod $summit $weight"; return; }
    done
  done
}


function ToRod  { # $1=предыдущее (FromRod) значение
  local rod firstfree weight sequence

  while true; do
    rod=$2
    test ${rod/[^123]} || continue

    sequence=$(echo $(seq 0 $disks1 | tac))
    for firstfree in $sequence; do
      eval weight=\${Rod${rod}[$firstfree]}
      test $weight -gt 0 && { (( firstfree++ )); break; }
    done
    test $weight -gt $1 -o $firstfree = 0 &&
         { echo "$rod $firstfree"; return; }
  done
}


function PrintRods  {
  local disk rod empty fill sp sequence

  tput cup 5 0

  repeat " " $spaces1
  echo -n "|"
  repeat " " $spaces2
  echo -n "|"
  repeat " " $spaces2
  echo "|"

  sequence=$(echo $(seq 0 $disks1 | tac))
  for disk in $sequence; do
    for rod in {1..3}; do
      eval empty=$(( $DISKS - (Rod${rod}[$disk] / 2) ))
      eval fill=\${Rod${rod}[$disk]}
      repeat " " $empty
      test $fill -gt 0 && repeat "*" $fill || echo -n "|"
      repeat " " $empty
    done
    echo
  done
```

```
  repeat "=" $basewidth   # Отрисовка "базы" стержней.
  echo
}


display ()
{
  echo
  PrintRods

  # Получает номер стержня, сумму и значение
  first=( `FromRod $1` )
  eval Rod${first[0]}[${first[1]}]=0

  # Получает номер стержня и первую свободную позицию
  second=( `ToRod ${first[2]} $2` )
  eval Rod${second[0]}[${second[1]}]=${first[2]}


  if [ "${Rod3[lastmove_t]}" = 1 ]
  then   # Последний ход? Если да, то показывается финальная позиция.
    tput cup 0 0
    echo; echo "+  Финальная позиция: $Moves ходов"
    PrintRods
  fi

  sleep $DELAY
}

# Отсюда вниз, оригинальный сценарий (hanoi.bash).

dohanoi() {   # Рекурсивная функция.
    case $1 in
    0)
        ;;
    *)
        dohanoi "$(($1-1))" $2 $4 $3
        if [ "$Moves" -ne 0 ]
        then
          tput cup 0 0
          echo; echo "+  Позиция после хода $Moves"
        fi
        ((Moves++))
        echo -n "   Следующий ход будет:  "
        echo $2 "-->" $3
        display $2 $3
        dohanoi "$(($1-1))" $4 $3 $2
        ;;
    esac
}

setup_arrays ()
{
  local dim n elem

  let "dim1 = $1 - 1"
  elem=$dim1

  for n in $(seq 0 $dim1)
  do
   let "Rod1[$elem] = 2 * $n + 1"
```

```
   Rod2[$n]=0
   Rod3[$n]=0
   ((elem--))
  done
}


###   Основное   ###

trap "tput cnorm" 0
tput civis
clear

setup_arrays $DISKS

tput cup 0 0
echo; echo "+  Начальная позиция"

case $# in
    1) case $(($1>0)) in     # Должен иметься хотя бы один диск.
       1)
           disks=$1
           dohanoi $1 1 3 2
#          Total moves = 2^n - 1, где n = # диска.
           echo
           exit 0;
           ;;
       *)
           echo "$0: Неправильное значение для номера диска";
           exit $E_BADPARAM;
           ;;
       esac
    ;;
    *)
       echo "usage: $0 N"
       echo "       Где \"N\" это количество дисков."
       exit $E_NOPARAM;
       ;;
esac

exit $E_NOEXIT   # Здесь не выходят.

#  Упражнение:
#  --------
#  Есть небольшая ошибка в сценарии, которая вызывает сброс вывода
#+ предпоследнего хода.
#+ Исправьте это.
```

Пример A-50.Альтернативная версия сценария *getopt-simple.sh*

```
#!/bin/bash
# UseGetOpt.sh

# Автор: Peggy Russell <prusselltechgroup@gmail.com>

UseGetOpt () {
  declare inputOptions
  declare -r E_OPTERR=85
```

```
  declare -r ScriptName=${0##*/}
  declare -r ShortOpts="adf:hlt"
  declare -r LongOpts="aoption,debug,file:,help,log,test"

DoSomething () {
    echo "Имя функции '${FUNCNAME}'"
    #  Напомним, что $FUNCNAME является внутренней переменной,
    #+ содержащей имя функции.
  }

  inputOptions=$(getopt -o "${ShortOpts}" --long \
              "${LongOpts}" --name "${ScriptName}" -- "${@}")

  if [[ ($? -ne 0) || ($# -eq 0) ]]; then
    echo "Используйте: ${ScriptName} [-dhlt] {OPTION...}"
    exit $E_OPTERR
  fi

  eval set -- "${inputOptions}"

  # Только в образовательных целях. Можно удалить.
  #-----------------------------------------------
  echo "++ Проверка: Количество аргументов: [$#]"
  echo '++ Проверка: Проходов цикла "$@"'
  for a in "$@"; do
    echo "  ++ [$a]"
  done
  #-----------------------------------------------

  while true; do
    case "${1}" in
      --aoption | -a)  # Найденные аргументы.
        echo "Опция [$1]"
        ;;

      --debug | -d)    # Включение информационных сообщений.
        echo "Опция [$1] Отладка включена"
        ;;

      --file | -f)     #  Проверка аргумента на обязательность.
        case "$2" in   #+ Вторая колонка является необязательным аргументом.
          "")          #  Не существует.
              echo "Опция [$1] Используется умолчание"
              shift
              ;;

          *) # Got it
             echo "Опция [$1] Используется ввод [$2]"
             shift
             ;;

        esac
        DoSomething
        ;;

      --log | -l) # Включение журналирования.
        echo "Опция [$1] Журналирование включено"
        ;;

      --test | -t) # Включение проверки.
        echo "Опция [$1] Проверка включена"
```

```
        ;;

      --help | -h)
        echo "Опция [$1] Вывод справки"
        break
        ;;

Not there      --)   # Сделано! $# номер аргумента для "--", $@ это "--"
        echo "Опция [$1] Двойное тире"
        break
        ;;

       *)
        echo "Главная внутренняя ошибка!"
        exit 8
        ;;

    esac
    echo "Количество аргументов: [$#]"
    shift
  done

  shift
  # Только в образовательных целях. Можно удалить
  #-----------------------------------------------------------------------
  echo "++ Проверка: Количество аргументов после \"--\" это [$#] Они: [$@]"
  echo '++ Проверка: Проходов цикла "$@"'
  for a in "$@"; do
    echo "  ++ [$a]"
  done
  #-----------------------------------------------------------------------

}

################################### О С Н О В Н О Е ########################
#  Если удалить "функцию UseGetOpt () {" и передачу "}",
#+ можно не раскомментировать строку "exit 0" ниже, а вызывать этот сценарий
#+ с различными опциями из командной строки.
#-----------------------------------------------------------------------
# exit 0

echo "Проверка 1"
UseGetOpt -f myfile one "two three" four

echo;echo "Проверка 2"
UseGetOpt -h

echo;echo "Проверка 3 – Немного опций"
UseGetOpt -adltf myfile  anotherfile

echo;echo "Проверка 4 – Много опций"
UseGetOpt --aoption --debug --log --test --file myfile anotherfile

exit
```

Пример A-51.Версия примера *UseGetOpt.sh*, использующего приложение *Табличных расширений*

```
#!/bin/bash

#  UseGetOpt-2.sh
#  Измененный сценарий для иллюстрации табличных расширений
#+ опций командной строки.
#  См. приложение "Введение в табличные расширения".

#  Доступные опции: -a -d -f -l -t -h
#+                   --aoption, --debug --file --log --test -- help --

#  Автор оригинала сценария: Peggy Russell <prusselltechgroup@gmail.com>


# UseGetOpt () {
  declare inputOptions
  declare -r E_OPTERR=85
  declare -r ScriptName=${0##*/}
  declare -r ShortOpts="adf:hlt"
  declare -r LongOpts="aoption,debug,file:,help,log,test"

DoSomething () {
    echo "Имя функции '${FUNCNAME}'"
  }

  inputOptions=$(getopt -o "${ShortOpts}" --long \
              "${LongOpts}" --name "${ScriptName}" -- "${@}")

  if [[ ($? -ne 0) || ($# -eq 0) ]]; then
    echo "Используйте: ${ScriptName} [-dhlt] {OPTION...}"
    exit $E_OPTERR
  fi

  eval set -- "${inputOptions}"


  while true; do
    case "${1}" in
      --aoption | -a)  # Найденный аргумент.
        echo "Option [$1]"
        ;;

      --debug | -d)    # Включение информационных сообщений.
        echo "Опция [$1] Отладка включена"
        ;;

      --file | -f)     #  Проверка необязательного аргумента.
        case "$2" in   #+ Вторая колонка является необязательным аргументом.
          "")          #  Не существует.
              echo "Опция [$1] Используется умолчание"
              shift
              ;;

          *) # Передано
             echo "Опция [$1] Используется ввод [$2]"
             shift
             ;;

        esac
        DoSomething
        ;;
```

```
      --log | -l) # Включение журналирования.
        echo "Опция [$1] Журналирование включено"
        ;;

      --test | -t) # Включение проверки.
        echo "Опция [$1] Проверка включена"
        ;;

      --help | -h)
        echo "Опция [$1] Вывод справки"
        break
        ;;

      --)   # Сделано! $# это номер аргумента "--", $@ это "--"
        echo "Опция [$1] Двойное тире"
        break
        ;;

       *)
        echo "Главная внутренняя ошибка!"
        exit 8
        ;;

    esac
    echo "Количество аргументов: [$#]"
    shift
  done

  shift

#  }

exit
```

Пример A-52. Перебор всех возможных цветов фона

```
#!/bin/bash

# show-all-colors.sh
# Вывод всех 256 цветов фона, с помощью управляющих последовательностей ANSI.
# Автор: Chetankumar Phulpagare
# В ABS Guide используется с разрешения.

T1=8
T2=6
T3=36
offset=0

for num1 in {0..7}
do {
   for num2 in {0,1}
       do {
          shownum=`echo "$offset + $T1 * ${num2} + $num1" | bc`
          echo -en "\E[0;48;5;${shownum}m color ${shownum} \E[0m"
          }
       done
   echo
   }
```

```
done

offset=16
for num1 in {0..5}
do {
   for num2 in {0..5}
       do {
          for num3 in {0..5}
              do {
                 shownum=`echo "$offset + $T2 * ${num3} \
                 + $num2 + $T3 * ${num1}" | bc`
                 echo -en "\E[0;48;5;${shownum}m color ${shownum} \E[0m"
                 }
               done
          echo
          }
       done
}
done

offset=232
for num1 in {0..23}
do {
   shownum=`expr $offset + $num1`
   echo -en "\E[0;48;5;${shownum}m ${shownum}\E[0m"
}
done

echo
```

Пример A-53. Упражнения в Азбуке Морзе

```
#!/bin/bash
# sam.sh, v. .01a
# Еще один Morse (код сценария упражнения)
# С большими извинениями перед  Sam (F.B.) Morse.
# Автор: Mendel Cooper
# License: GPL3
# Reldate: 05/25/11

# Сценарий упражнения в коде Морзе.
# Преобразует аргументы в звуковые точки и тире.
# Примечания: вводятся ТОЛЬКО СТРОЧНЫЕ буквы.



# Извлечение файлов wav из tarball:
# http://bash.deta.in/abs-guide-latest.tar.bz2
DOT='soundfiles/dot.wav'
DASH='soundfiles/dash.wav'
# Может быть поместить звуковые файлы в /usr/local/sounds?

LETTERSPACE=300000  # Микросекунды.
WORDSPACE=980000
# Ясно и медленно, для начинающих. Может быть 5 знаков в минуту?

EXIT_MSG="Вы знаете азбуку Морзе!"
E_NOARGS=75         # Отсутствие аргументов командной строки?
```

```
declare -A morse    # Ассоциативный массив!
# ===================================== #
morse[a]="точка; тире"
morse[b]="тире; точка; точка; точка"
morse[c]="тире; точка; тире; точка"
morse[d]="тире; точка; точка"
morse[e]="точка"
morse[f]="точка; точка; тире; точка"
morse[g]="тире; тире; точка"
morse[h]="точка; точка; точка; точка"
morse[i]="точка; точка;"
morse[j]="точка; тире; тире; тире"
morse[k]="тире; точка; тире"
morse[l]="точка; тире; точка; точка"
morse[m]="тире; тире"
morse[n]="тире; точка"
morse[o]="тире; тире; тире"
morse[p]="точка; тире; тире; точка"
morse[q]="тире; тире; точка; тире"
morse[r]="точка; тире; точка"
morse[s]="точка; точка; точка"
morse[t]="тире"
morse[u]="точка; точка; тире"
morse[v]="точка; точка; точка; тире"
morse[w]="точка; тире; тире"
morse[x]="тире; точка; точка; тире"
morse[y]="тире; точка; тире; тире"
morse[z]="тире; тире; точка; точка"
morse[0]="тире; тире; тире; тире; тире"
morse[1]="точка; тире; тире; тире; тире"
morse[2]="точка; точка; тире; тире; тире"
morse[3]="точка; точка; точка; тире; тире"
morse[4]="точка; точка; точка; точка; тире"
morse[5]="точка; точка; точка; точка; точка"
morse[6]="тире; точка; точка; точка; точка"
morse[7]="тире; тире; точка; точка; точка"
morse[8]="тире; тире; тире; точка; точка"
morse[9]="тире; тире; тире; тире; точка"
# Следующее должно быть заэкранировано или в кавычках.
morse[?]="точка; точка; тире; тире; точка; точка"
morse[.]="точка; тире; точка; тире; точка; тире"
morse[,]="тире; тире; точка; точка; тире; тире"
morse[/]="тире; точка; точка; тире; точка"
morse[\@]="точка; тире; тире; точка; тире; точка"
# ===================================== #

play_letter ()
{
  eval ${morse[$1]}   # Проигрываются точки и тире из звуковых файлов.
  # Почему здесь необходим 'eval'?
  usleep $LETTERSPACE # Пауза между буквами.
}

extract_letters ()
{                     # Фрагмент строки поэлементно, по буквам.
  local pos=0         # Начинается с левого конца строки.
  local len=1         # По одной букве.
  strlen=${#1}
```

```
  while [ $pos -lt $strlen ]
  do
    letter=${1:pos:len}
    #      ^^^^^^^^^^^^    См. Главу 10.1.
    play_letter $letter
    echo -n "*"       #    Отмечается только что проигранная буква.
    ((pos++))
  done
}

######### Проигрываемые звуки ############
dot()  { aplay "$DOT" 2&>/dev/null;  }
dash() { aplay "$DASH" 2&>/dev/null; }
##########################################

no_args ()
{
    declare -a usage
    usage=( $0 word1 word2 ... )

    echo "Usage:"; echo
    echo ${usage[*]}
    for index in 0 1 2 3
    do
      extract_letters ${usage[index]}
      usleep $WORDSPACE
      echo -n " "     # Выводимый пробел между словами.
    done
#   echo "Usage: $0 word1 word2 ... "
    echo; echo
}


# int main()
# {

clear                # Очищаем экран.
echo "            SAM"
echo "Еще один тренажер кода Морзе"
echo "    Автор: Mendel Cooper"
echo; echo;

if [ -z "$1" ]
then
  no_args
  echo; echo; echo "$EXIT_MSG"; echo
  exit $E_NOARGS
fi

echo; echo "$*"       # Ввод текста, который должен прозвучать.

until [ -z "$1" ]
do
  extract_letters $1
  shift           # К следующему слову.
  usleep $WORDSPACE
  echo -n " "     # Ввод пробелов между словами.
done

echo; echo; echo "$EXIT_MSG"; echo
```

```
exit 0
# }

#  Упражнения:
#  ---------
#  1) Сценарий, в качестве аргументов, должен принимать слова в нижнем или
#+    верхнем регистре. Подсказка: Используйте 'tr'...
#  2) Сценарий должен принимать ввод из текстового файла.
```

Пример A-54. Кодирование/декодирование Base64

```
#!/bin/bash
# base64.sh: Реализация кодирования и декодирования Base64 с помощью Bash.
#
# Copyright (c) 2011 vladz <vladz@devzero.fr>
# В ABSG используется с разрешения (спасибо!).
#
#  Кодирование или декодирование оригинального Base64 (а так же Base64url)
#+ из STDIN в STDOUT.
#
#    Используем:
#
#    Кодирование
#    $ ./base64.sh < binary-file > binary-file.base64
#    Декодирование
#    $ ./base64.sh -d < binary-file.base64 > binary-file
#
# Ссылка:
#
#    [1]  RFC4648 - "The Base16, Base32, and Base64 Data Encodings"
#         http://tools.ietf.org/html/rfc4648#section-5


# Массив base64_charset[] содержит всю кодировку символов base64,
# и дополнительно символ "=" ...
base64_charset=( {A..Z} {a..z} {0..9} + / = )
                # Прекрасная иллюстрация скобок расширения.

#  Раскомментируйте строки ###, ниже, для использования кодирования base64url
#+ вместо оригинального base64.
### base64_charset=( {A..Z} {a..z} {0..9} - _ = )

#  Выводимый размер текста при кодировании
#+ (64 символа, как при выводе openssl).
text_width=64

function display_base64_char {
#  Преобразование 6-битного числа (от 0 до 63) в его соответствующее значение
#+ Base64, затем вывод результата с указанным размером написанного.
  printf "${base64_charset[$1]}"; (( width++ ))
  (( width % text_width == 0 )) && printf "\n"
}

function encode_base64 {
#  Кодирование трех 8-разрядных шестнадцатеричных кодов в четыре 6-битовых
#+ числа.
  #    Нам нужны два локальных массива переменных целых чисел:
```

```
  #    c8[]: для хранения кодов 8-битных символов для кодирования
  #    c6[]: для хранения закодированных 6-битных значений
  declare -a -i c8 c6

  #  Конвертация шестнадцатеричных в десятичные.
  c8=( $(printf "ibase=16; ${1:0:2}\n${1:2:2}\n${1:4:2}\n" | bc) )

  #  Давайте поиграем с побитовыми операторами
  #+ (преобразование 3x8-битного в 4x6-битное).
  (( c6[0] = c8[0] >> 2 ))
  (( c6[1] = ((c8[0] &  3) << 4) | (c8[1] >> 4) ))

  # The следующие операции зависят от числа элементов c8.
  case ${#c8[*]} in
    3) (( c6[2] = ((c8[1] & 15) << 2) | (c8[2] >> 6) ))
       (( c6[3] = c8[2] & 63 )) ;;
    2) (( c6[2] = (c8[1] & 15) << 2 ))
       (( c6[3] = 64 )) ;;
    1) (( c6[2] = c6[3] = 64 )) ;;
  esac

  for char in ${c6[@]}; do
    display_base64_char ${char}
  done
}

function decode_base64 {
# Декодирование четырех символов base64 в три шестнадцатеричных символа ASCII.
  #  c8[]: для хранения кодов 8-битных символов
  #  c6[]: для хранения значений соответствующих 6-битным  Base64
  declare -a -i c8 c6

  # Ищем десятичное значение, соответствующее текущему символу base64.
  for current_char in ${1:0:1} ${1:1:1} ${1:2:1} ${1:3:1}; do
     [ "${current_char}" = "=" ] && break

     position=0
     while [ "${current_char}" != "${base64_charset[${position}]}" ]; do
        (( position++ ))
     done

     c6=( ${c6[*]} ${position} )
  done

  #  Давайте поиграем с побитовыми операторами
  #+ (преобразование 4x8-битное в 3x6-битное).
  (( c8[0] = (c6[0] << 2) | (c6[1] >> 4) ))

  # Следующие операции зависят от числа элементов c6.
  case ${#c6[*]} in
    3) (( c8[1] = ( (c6[1] & 15) << 4) | (c6[2] >> 2) ))
       (( c8[2] = (c6[2] & 3) << 6 )); unset c8[2] ;;
    4) (( c8[1] = ( (c6[1] & 15) << 4) | (c6[2] >> 2) ))
       (( c8[2] = ( (c6[2] &  3) << 6) |  c6[3] )) ;;
  esac

  for char in ${c8[*]}; do
     printf "\x$(printf "%x" ${char})"
  done
}
```

```
# main ()

if [ "$1" = "-d" ]; then   # декодирование

  # Переформатирование STDIN в псевдо 4x6-битные группы.
  content=$(cat - | tr -d "\n" | sed -r "s/(.{4})/\1 /g")

  for chars in ${content}; do decode_base64 ${chars}; done

else
  # Делаем hexdump stdin и переформатируем в 3-байтовые группы.
  content=$(cat - | xxd -ps -u | sed -r "s/(\w{6})/\1 /g" |
            tr -d "\n")

  for chars in ${content}; do encode_base64 ${chars}; done

  echo

fi
```

Пример A-55. Вставка текста в файл с помощью *sed*

```
#!/bin/bash
#  Добавляет строку в заданную строку в
#+ файлы, с именами заканчивающимися «sample»,
#+ в текущей рабочей директории
#  000000000000000000000000000000000000
#  Этот сценарий перезаписывает файлы!
#  Будьте осторожны, запуская его в директории,
#+ где есть важные файлы!!!
#  000000000000000000000000000000000000

#  Создаем пару файлов для работы ...
#  01sample
#  02sample
#  ... и т.д.
#  Эти файлы не должны быть пустыми, иначе сценарий не будет работать.

lineno=1            # Добавить в строку 1 (Добавлено).
filespec="*sample"  # Шаблон имени для работы.

string=$(whoami)    # Установка пользовательского имени, как
#+ вставляемой строки.
                    # Может быть любая другая строка.

for file in $filespec # Указываем, какие файлы нужно изменить.
do #        ^^^^^^^^^
 sed -i ""$lineno"i "$string"" $file
#    ^^ опция -i option редактирует файлы на месте.
#                  ^ Вставка (i) команды.
 echo ""$file" изменен!"
done

echo "Внимание: возможны испорченные файлы!"
```

```
exit 0

# Упражнение:
# Добавьте в этом сценарии проверку ошибок.
# Это ужасно необходимо.
```

Пример A-56. Шифр Gronsfeld

```
#!/bin/bash
# gronsfeld.bash

# License: GPL3
# Reldate 06/23/11

#  Это реализация шифра Гронсфилда.
#  По сути, урезанный вариант многоалфавитного
#+ Vigenère Tableau, только 10 алфавитов.
#  Классический Gronsfeld имеет ключ из числовых последовательностей,
#+ а здесь, для простоты использования, заменен на буквенную строку.
#  Предположительно этот шифр был изобретен графом Гронсфилдом
#+ в XVII веке. Одно время он считался не взламываемым.
#  Обратите внимание, что это, по современным меркам, ###НЕ### безопасный шифр.

#  Глобальные переменные  #
Enc_suffix="29379"   #  Вывод зашифрованного текста с указанным 5-значным
#+ суффиксом.
                     #  Он работает, как флаг для расшифровки, f когда
                     #+ используется для создания паролей добавляет
                     #+ секретностb.
Default_key="gronsfeldk"
                     #  Сценарий использует ее, если не указан ключ ниже
                     #  ("Keychain").
                     #  Измените два значения частоты, выше, для
                     #+ дополнительной безопасности.

GROUPLEN=5           #  Вывод групп из 5 букв, по традиции.
alpha1=( abcdefghijklmnopqrstuvwxyz )
alpha2=( {A..Z} )    #  Вывод всех заглавных, по традиции.
                     #  С помощью alpha2=( {a..z} ) создается пароль.
wraplen=26           #  Обертка, возвращающая при окончании алфавита.
dflag=               #  Флаг дешифровки (устанавливается если
                     #+ присутствует $Enc_suffix).
E_NOARGS=76          #  Ошибочный аргумент командной строки?
DEBUG=77             #  Флаг отладки.
declare -a offsets   #  Массив содержащий числовой сдвиг значения для
                     #+ шифрования и расшифровки.


########Keychain#########
key=  ### Помещаем ключ сюда!!!
      # 10 символов!
#########################

# Действие
: ()
{ #  Шифрование или расшифровка зависит от установки $dflag.
  #  Почему ": ()" как имя функции? Просто чтобы доказать, что это возможно.
```

```
  local idx keydx mlen off1 shft
  local plaintext="$1"
  local mlen=${#plaintext}

for (( idx=0; idx<$mlen; idx++ ))
do
  let "keydx = $idx % $keylen"
  shft=${offsets[keydx]}

  if [ -n "$dflag" ]
  then                    # Дешифрование!
    let "off1 = $(expr index "${alpha1[*]}" ${plaintext:idx:1}) - $shft"
    # Сдвиг назад, для расшифровки.
  else                    # Шифрование!
    let "off1 = $(expr index "${alpha1[*]}" ${plaintext:idx:1}) + $shft"
    # Сдвиг вперед для шифрования.
    test $(( $idx % $GROUPLEN)) = 0 && echo -n " "  # Группы из 5 букв.
    #  Закомментируйте строку выше для вывода в виде строки без пробелов,
    #+ например, при использовании сценария в качестве генератора паролей.
  fi

  ((off1--))   # Нормализация. Зачем она нужна?

      if [ $off1 -lt 0 ]
      then     # Отлов отрицательных индексов.
        let "off1 += $wraplen"
      fi

  ((off1 %= $wraplen))   # Возвращение, если алфавит закончился.

  echo -n "${alpha2[off1]}"

done

  if [ -z "$dflag" ]
  then
    echo " $Enc_suffix"
#   echo "$Enc_suffix"  # Для генератора паролей.
  else
    echo
  fi
} # Окончание функции шифрования/дешифрования.


# int main () {

# Проверка аргументов командной строки.
if [ -z "$1" ]
then
   echo "Usage: $0 ТЕКСТ ДЛЯ ШИФРОВАНИЯ/ДЕШИФРОВАНИЯ"
   exit $E_NOARGS
fi

if [ ${!#} == "$Enc_suffix" ]
#    ^^^^^ Последний аргумент командной строки.
then
  dflag=ON
  echo -n "+"           # "+" это флаг для дешифровки текста, проще ID.
fi

if [ -z "$key" ]
```

```
then
  key="$Default_key"    # "gronsfeldk" см.выше.
fi

keylen=${#key}

for (( idx=0; idx<$keylen; idx++ ))
do  # Подсчет значения смещения для шифрования/дешифрования.
  offsets[idx]=$(expr index "${alpha1[*]}" ${key:idx:1})   # Нормализация.
  ((offsets[idx]--))  #  Необходима, потому что "expr index" начинается с 1,
                      #+ в то время как отсчет массива начинается с 0.
  # Создание массива численных смещений, соответствующих ключу.
  # Для этого есть способы попроще.
done

args=$(echo "$*" | sed -e 's/ //g' | tr A-Z a-z | sed -e 's/[0-9]//g')
# Удаление пробелов и цифр из аргументов командной строки.
# Можно изменить, если нужно, и для удаления знаков пунктуации.

        # Отладка:
        # echo "$args"; exit $DEBUG

: "$args"               # Вызов функции с именем ":".
# : это оператор null, за исключением ... когда он является именем функции!

exit $?    # } Окончание сценария


#   ************************************************************   #
#   Этот сценарий, с несколькими незначительными изменениями,
#+  может работать как генератор паролей, см. выше.
#   Он позволяет легко запомнить пароль, даже само слово «пароль»,
#+  которое шифруется в vrgfotvo29379
#+  достаточно надежный пароль не восприимчивый к словарю.
#   Или, вы можете использовать ваше собственное название,
#+  (конечно, которое легко запомнить!).
#   Например, Bozo Bozeman будет зашифровано в hfnbttdppkt29379.
#   ************************************************************   #
```

Пример A-57. Генератор чисел Bingo

```
#!/bin/bash
# bingo.sh
# Bingo number generator
# Reldate 20Aug12, License: Public Domain

##############################################################################
# Этот сценарий создает числа bingo.
# При нажатии клавиши создается новое число.
# Нажатие на 'q' прекращает сценарий.
# Запущенный сценарий производит не повторяющиеся числа.
# Когда сценарий завершается, он заполняет журнал созданными числами.
##############################################################################

MIN=1       # Самое низкое допустимое число bingo.
MAX=75      # Самое высокое допустимое число bingo.
COLS=15     # Числа в каждой колонке (B I N G O).
SINGLE_DIGIT_MAX=9
```

```
declare -a Numbers
Prefix=(B I N G O)

initialize_Numbers ()
{  # Начинается с нуля.
   # Если выбрано будет увеличиваться.
   local index=0
   until [ "$index" -gt $MAX ]
   do
     Numbers[index]=0
     ((index++))
   done

   Numbers[0]=1   # Флаг ноль, поэтому он не может быть выбран.
}


generate_number ()
{
   local number

   while [ 1 ]
   do
     let "number = $(expr $RANDOM % $MAX)"
     if [ ${Numbers[number]} -eq 0 ]    # Число еще не вызвано.
     then
       let "Numbers[number]+=1"         # Флаг массива.
       break                            # И завершение цикла.
     fi   # Если вызвано, цикл и создание другого числа.
   done
   # Упражнение: Перепишите более элегантно, чем цикл until.

   return $number
}


print_numbers_called ()
{   # Печать вызываемых цифр в аккуратные колонки в журнале.
    # echo ${Numbers[@]}

local pre2=0                # Префикс ноль, так что колонки будут
                            #+ выравнены по одной цифре.

echo "Number Stats"

for (( index=1; index<=MAX; index++))
do
  count=${Numbers[index]}
  let "t = $index - 1"      # Нормализация, массив начинается с индекса 0.
  let "column = $(expr $t / $COLS)"
  pre=${Prefix[column]}
# echo -n "${Prefix[column]} "

if [ $(expr $t % $COLS) -eq 0 ]
then
  echo   # Перевод в конце строки.
fi

  if [ "$index" -gt $SINGLE_DIGIT_MAX ]  # Проверка, что число одно.
  then
```

```
    echo -n "$pre$index#$count "
  else    # Префикс ноль.
    echo -n "$pre$pre2$index#$count "
  fi

done
}


# main () {
RANDOM=$$   # Источник генератора случайных чисел.

initialize_Numbers   # Обнуление отслеживаемого номера массива.

clear
echo "Вызванное число Bingo"; echo

while [[ "$key" != "q" ]]   # Основной цикл.
do
  read -s -n1 -p "Нажмите кнопку для следующего числа [q для выхода] "
  # Обычно 'q' это выход, но не всегда.
  # Нажмите Ctl-C, если не работает q.
  echo

  generate_number; new_number=$?

  let "column = $(expr $new_number / $COLS)"
  echo -n "${Prefix[column]} "   # B-I-N-G-O

  echo $new_number
done

echo; echo

# Закончено ...
print_numbers_called
echo; echo "[#0 = не вызвано . . . #1 = вызвано]"

echo

exit 0
# }


# Конечно, этот сценарий можно улучшить.
# Авторская инструкция:
# www.instructables.com/id/Binguino-An-Arduino-based-Bingo-Number-Generato/
```

В конце этого раздела - обзор основ... и многое другое.

Пример A-58. Рассмотренные основы

```
#!/bin/bash
# basics-reviewed.bash

# Расширение файла == *.bash == специфично для Bash

#   Copyright (c) Michael S. Zick, 2003; All rights reserved.
```

```
#   License: Use in any form, for any purpose.
#   Revision: $ID$
#
#   Для размещения отредактировано М.С.
#   (автором "Advanced Bash Scripting Guide")
#   Исправление и обновления (04/08) Cliff Bamford.


#  Сценарий проверен с версиями Bash 2.04, 2.05a и 2.05b.
#  Не работает с более ранними версиями.
#  Этот демонстрационный сценарий создает одно --международное--
#+ сообщение об ошибке "command not found". См. строку 436.

#  Нынешний майнтейнер Bash, Chet Ramey, исправляет элементы,
#+ введенные в более поздние версии Bash.



        ###-----------------------------------------###
        ###  Для перемещения по страницам           ###
        ###+ Направьте вывод этого сценария в 'more'. ###
        ###                                         ###
        ###  Можно так же перенаправить вывод       ###
        ###+ в файл, для изучения.                  ###
        ###-----------------------------------------###



#  Большинство из следующих пунктов подробно описаны в тексте
#+ "Advanced Bash Scripting Guide", выше.
#  Этот демонстрационный сценарий, просто реорганизованная презентация.
#      -- msz

# Переменные не типизированы, если не указано иное.

#  Переменные именованы. Имена не должны содержать цифры.
#  Дескриптор имени файла (как например: 2>&1)
#+ состоит ТОЛЬКО из цифр.

# Параметры и элементы массива Bash пронумерованы.
# (Параметры в Bash подобны массивам.)

# Имя переменной может быть не определено (пустая ссылка).
unset VarNull

# Имя переменной может быть определено, но быть пустым (нулевое содержимое).
VarEmpty=''                           # Одинарные кавычки, две рядом.

# Имя переменной может быть определено и не пустым.
VarSomething='Буквально'

# Переменная может содержать:
#   * Целое число, как 32-разрядное (или более) целое число
#   * Строку
#  Переменная так же может быть массивом.

#  Строка может содержать вложенные пробелы и может рассматриваться
#+ как имя функции с необязательными аргументами.

#  Имена переменных и имена функций находятся в разных пространствах имен.
```

```
#  Переменная может определяться как массив Bash, явно или косвенно,
#+ синтаксисом оператора присваивания.
#  Явно:
declare -a ArrayVar

# Команда echo является встроенной.
echo $VarSomething

# Команда printf является встроенной.
# %s переводится как: Формат строки
printf %s $VarSomething         # Не указано окончание строки, нет вывода.
echo                            # Умолчание, выводится только с окончанием
                                #+ строки.


# Парсер Bash разделяет слово пробелами.
# Пробел, или его отсутствие, является значимым явлением.
# (Это верно в целом, но есть, конечно, исключения.)


#  Знак ЗНАК_ДОЛЛАРА переводится как: Содержимое-из.
#  Синтаксис расширения способа записи Содержимое-из:
echo ${VarSomething}

#  Синтаксис расширения ${ ... } допускает большее, чем просто необходимость
#+ указания имени переменной.
#  В целом, $VarSomething всегда может быть записана в виде: ${VarSomething}.

# Вызовите этот сценарий с аргументами, чтобы увидеть следующее в действии.

#  За пределами двойных кавычек, специальные символы  @ и * имеют
#+ одинаковое поведение.
#  Могут быть выражены как: Все элементы из.

#  Без указания имени, они относятся к предварительно определенному
#+ параметру массива Bash.

# Ссылки Glob-Pattern
echo $*                         # Все параметры сценария или функции
echo ${*}                       # То же самое

# В Bash отключено расширение имен файлов для Glob-Pattern.
# Активировано только соответствие символов.

# Ссылки Все-элементы-из
echo $@                         # То же, что выше
echo ${@}                       # То же, что выше


#  В двойных кавычках, поведение ссылок Glob-Pattern зависит от
#+ параметра IFS (разделитель полей ввода).
#  В двойных кавычках ссылки Все-элементы-из ведут себя так же.


#  Только имя переменной, содержащая строку, ссылается на все элементы
#+  (символы) строки.


#  Для указания элемента (символа) строки, МОЖЕТ использоваться запись
#+ ссылки синтаксиса расширения  (см. ниже).
```

```
#  Указание только имени массива Bash, это ссылка на индекс
#+ нулевого элемента, а НЕ на ПЕРВЫЙ ОПРЕДЕЛЕННЫЙ,
#+ и не на ПЕРВЫЙ С СОДЕРЖИМЫМ элемент.

#  Дополнительная квалификация необходима для ссылки на другие элементы, это
#+ означает, что ссылка должна быть записана в синтаксисе расширения.
#  Основная форма: ${имя[индекс]}.

#  Можно использовать форму строки: ${имя:индекс}
#+ для массивов Bash ссылающихся на индекс нулевого элемента.


# Массивы Bash реализуются внутренне как связанные списки, а не как
#+ фиксированные области хранения, как в некоторых языках программирования.


#   Характеристики массивов Bash (Bash-Arrays):
#   -------------------------------------------

#   Если не указано иное, индексы массивов Bash начинаются с числового
#+  индекса ноль. Буквально: [0]
#   Это называется индексация с нуля.
###
#   Если не указано иное, Bash-массивы являются индексно упакованными
#+  (последовательность индексов без пропуска индекса).
###
#   Отрицательный индекс недопустим.
###
#   Все элементы Bash-массива могут не быть одного и того же типа.
###
#   Элементы Bash-массива могут быть не определены (пустая ссылка).
#       То есть, Bash-массив может быть "индексно разреженным".
###
#   Элементы Bash-массива могут быть определены и пустыми (нулевое содержание).
###
#   Элементы Bash-массива могут содержать:
#     * Целое число, как 32-разрядное (или более) целое число
#     * Строку
#     * Строку, форматированную так, что она, как кажется, является именем
#     + функции с необязательными аргументами
###
#   Определенные элементы Bash-массива могут быть не определены (не объявлены).
#       То есть, упакованный индекс  Bash-массива может быть заменен
#   +   разреженным индексом Bash-массива.
###
#   Элементы могут добавляться в Bash-массив путем объявления
#+  не определенного ранее элемента.
###
# По этим причинам, я называю их 'Bash-Массивы'.
# Теперь я вернусь к общим терминам 'массива' .
#     -- msz


echo "============================================================"

#  Строки 202 - 334 предоставлены Cliff Bamford. (Спасибо!)
#  Демо --- Взаимодействие кавычек, IFS, echo, * и @  с массивами  ---
#+ всем, что влияет на работу

ArrayVar[0]='zero'                    # 0 обычный
```

```
ArrayVar[1]=one                         # 1 буквальный без кавычек
ArrayVar[2]='two'                       # 2 обычный
ArrayVar[3]='three'                     # 3 обычный
ArrayVar[4]='I am four'                 # 4 обычный с пробелами
ArrayVar[5]='five'                      # 5 обычный
unset ArrayVar[6]                       # 6 не определенный
ArrayValue[7]='seven'                   # 7 обычный
ArrayValue[8]=''                        # 8 определен, но пустой
ArrayValue[9]='nine'                    # 9 обычный


echo '--- Этот массив мы используем для проверки'
echo
echo "ArrayVar[0]='zero'              # 0 обычный"
echo "ArrayVar[1]=one                 # 1 буквальный без кавычек"
echo "ArrayVar[2]='two'               # 2 обычный"
echo "ArrayVar[3]='three'             # 3 обычный"
echo "ArrayVar[4]='I am four'         # 4 обычный с пробелами"
echo "ArrayVar[5]='five'              # 5 обычный"
echo "unset ArrayVar[6]               # 6 не определенный"
echo "ArrayValue[7]='seven'           # 7 обычный"
echo "ArrayValue[8]=''                # 8 определенный, но пустой"
echo "ArrayValue[9]='nine'            # 9 обычный"
echo


echo
echo '---Case0: Без двойных кавычек, по умолчанию IFS, пробел,табуляция,новая
строка ---'
IFS=$'\x20'$'\x09'$'\x0A'               # Точно в этом порядке.
echo 'Здесь: printf %q {${ArrayVar[*]}'
printf %q ${ArrayVar[*]}
echo
echo 'Здесь: printf %q {${ArrayVar[@]}'
printf %q ${ArrayVar[@]}
echo
echo 'Здесь: echo ${ArrayVar[*]}'
echo  ${ArrayVar[@]}
echo 'Здесь: echo {${ArrayVar[@]}'
echo ${ArrayVar[@]}

echo
echo '---Случай1: С двойными кавычками — по умолчанию IFS это пробел — табуляция
- новая строка ---'
IFS=$'\x20'$'\x09'$'\x0A'             #  Те же три байта,
echo 'Здесь: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Здесь: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Здесь: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Здесь: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Случай2: С двойными кавычками - IFS является q'
IFS='q'
echo 'Здесь: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
```

```
echo
echo 'Здесь: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Здесь: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Здесь: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Случай3: С двойными кавычками - IFS является ^'
IFS='^'
echo 'Здесь: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Здесь: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Здесь: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Здесь: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Случай4: С двойными кавычками - IFS является ^ следующим за
пробелом,табуляцией,новой строкой'
IFS=$'^'$'\x20'$'\x09'$'\x0A'       # ^ + пробел  табуляция  новая строка
echo 'Здесь: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Здесь: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Здесь: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Здесь: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Случай5: С двойными кавычками - IFS установлен и пустой '
IFS=''
echo 'Здесь: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Здесь: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Здесь: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Здесь: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Случай6: С двойными кавычками - IFS не установлен'
unset IFS
echo 'Здесь: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Здесь: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
```

```
echo 'Здесь: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Здесь: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Окончание---'
echo "============================================================"; echo



# Возвращаем IFS значение по умолчанию.
# Значение по умолчанию — именно эти три байта.
IFS=$'\x20'$'\x09'$'\x0A'           # Именно в таком порядке.

# Интерпретация выводов выше:
#   Glob-Pattern является I/O; устанавливает значения IFS.
###
#   Все-Элементы-Из не учитывают настройки IFS.
###
#   Обратите внимание на различные вывод использования команды
#+  echo, а кавычки форматируют команду оператора printf.


#  Напомним:
#   Параметры похожи на массивы и имеют аналогичное поведение.
###
#  Приведенные выше примеры свидетельствуют о возможности вариаций.
#  Чтобы сохранить форму разреженного массива, требуется
#+ написать дополнительный сценарий.
###
#  Исходный код Bash имеет обычный для вывода
#+ формат присваивания массива [Индекс]=значение
#  Начиная с версии 2.05b эта процедура не используется,
#+ но это может измениться в будущих версиях.


# Длина строки, измеренная в не пустых элементах (символах):
echo
echo '- - Ссылки без кавычек - -'
echo 'Количество символов не Null: '${#VarSomething}' символов.'

# test='Lit'$'\x00''eral'           # $'\x00' это null символ.
# echo ${#test}                     # Видишь это?


#  Размер массива, измеренный в определенных элементах,
#+ включая элементы содержащие null.
echo
echo 'Определяется количество содержимого: '${#ArrayVar[@]}' элементов.'
# Это НЕ максимальный индекс (4).
# Это НЕ диапазон индексов (1 . . 4 включительно).
# Это ЕСТЬ размер связанного списка.
###
#  Оба - максимальный индекс и диапазон индексов можно
#+ найти, написав дополнительный сценарий.

# Размер строки, измеренный в не пустых элементах (символах):
echo
echo '- - Кавычки, ссылки Glob-Pattern - -'
echo 'Подсчет не пустых символов: '"${#VarSomething}"' символов.'
```

```
#  Размер массива, измеренный в определенных элементах,
#+ включая элементы содержащие null.
echo
echo 'Количество определенных элементов: '"${#ArrayVar[*]}"' элементов.'

#  Объяснение: Подстановка не влияет на работу ${# ... }.
#  Объяснение:
#  Всегда используйте символы Все-Элементы-Из, если
#+ они являются уже назначенными (независимо от IFS).


#  Определение простой функции.
#  Я включил подчеркивания в имени в приведенных ниже примерах,
#+чтобы сделать его отличительным.
###
#  Bash разделяет имена переменных и имена функций
#+ разными пространствами имен.
#  The Mark-One eyeball не подойдет.
###
_simple() {
    echo -n 'SimpleFunc'$@          #  В любом случае символы новой строки
}                                   #+ поглощаются возвращенным результатом.


# Запись ( ... ) вызывает команду или функцию.
# Запись $( ... ) обозначает: Результат-Из.


# Вызов функции _simple
echo
echo '- - Вывод функции _simple - -'
_simple                             # Попробуйте передать аргументы
echo
# или
(_simple)                           # Попробуйте передать аргументы.
echo

echo '- Есть переменная с таким именем? -'
echo $_simple не определена           # Нет переменной с таким именем.

# Результат вызова функции _simple (Сообщение об ошибке)

###
$(_simple)                          # Выдается сообщение об ошибке:
#                           строка 436: SimpleFunc: command not found
#                           ----------------------------------------

echo
###

#  Первое слово результата функции _simple является
#+ допустимой командой Bash, а не именем определенной функции.
###
# Это показывает, что вывод _simple подвергается оценке.
###
# Интерпретация:
#   Функция может использоваться для создания строки команд Bash.


# Простая функция, где первое слово результата ЯВЛЯЕТСЯ командой Bash:
```

```
###
_print() {
    echo -n 'printf %q '$@
}

echo '- - Вывод функции _print - -'
_print parm1 parm2                  # Выводятся НЕ команды.
echo

$(_print parm1 parm2)               #  Выполняется: printf %q parm1 parm2
                                    #  См. Выше примеры различных
                                    #+ возможностей IFS.
echo

$(_print $VarSomething)             # Предсказуемый результат.
echo



# Действие переменных
# -------------------

echo
echo '- - Действие переменных - -'
# Переменная может представлять знаковое целое число, строку или массив.
# Строка может использоваться как имя функции с необязательными аргументами.

# set -vx                           #  Включает, при необходимости,
declare -f funcVar                  #+ пространство имен функций.

funcVar=_print                      # Содержит имя функции.
$funcVar parm1                      # На данный момент - _print.
echo

funcVar=$(_print )                  # Содержит результат функции.
$funcVar                            # Нет ввода, Нет вывода.
$funcVar $VarSomething              # Предсказуемый результат.
echo

funcVar=$(_print $VarSomething)     #  ЗДЕСЬ заменяется $VarSomething.
$funcVar                            #  Расширение является частью содержимого
echo                                #+ переменной.

funcVar="$(_print $VarSomething)"   #  ЗДЕСЬ заменяется $VarSomething.
$funcVar                            #  Расширение является частью содержимого
echo                                #+ переменной.

#  Разница между не заключенным в кавычки и в двойных кавычках версий выше,
#+ может быть видна на примере «protect_literal.sh.
#  Первый случай выше обрабатывается как пара, без кавычек, слов Bash.
#  Второй случай обрабатывается как единое, заключенное в кавычки, слово Bash.


# Задержка замены
# -------------------

echo
echo '- - Задержка замены - -'
funcVar="$(_print '$VarSomething')" # Замены нет, единственное Слово Bash.
eval $funcVar                       # ЗДЕСЬ заменено $VarSomething.
echo
```

```
VarSomething='NewThing'
eval $funcVar                       # ЗДЕСЬ заменено $VarSomething.
echo

# Восстановление исходной настройки испорченной выше.
VarSomething=Буквально

#  Существует пара демонстрационных примеров функций
#+ "protect_literal.sh" и "unprotect_literal.sh".
#  Это функции общего назначения для задержки замены опечаток,
#+ содержащихся в переменных.


# ОБЗОР:
# ------

#  Строка может считаться классическим массивом элементов (символов).
#  Строковая операция применяется ко всем элементам (символам) строки
#+ (в концепции, так или иначе).
###
#  Запись: ${array_name[@]} представляет все элементы
#+ Bash-Массива: array_name.
###
#  Операции синтаксиса расширения строки могут быть применены
#+ ко всем элементам массива.
###
#  Она может рассматриваться как операция Для-Каждого вектора строк.
###
#  Параметры подобны массиву.
#  Инициализация параметров массива для сценария и параметров
#+ массива для функции отличаются только в инициализации ${0},
#+ которая никогда не меняет настройки.
###
#  Индекс ноль, параметра массива сценария, содержит имя сценария.
###
#  Индекс ноль параметра массива функции НЕ содержит имя функции.
#  Имя текущей функции является доступным переменной $FUNCNAME.
###
#  Быстрый обзор следующего списка(быстрый, но не краткий).

echo
echo '- - Проверка (но не изменение) - -'
echo '- пустая ссылка -'
echo -n ${VarNull-'НеОбъявлена'}' '     #  НеОбъявлена
echo ${VarNull}                         # Только новая строка
echo -n ${VarNull:-'НеОбъявлена'}' '    #  НеОбъявлена
echo ${VarNull}                         # Только новая строка

echo '- null contents -'
echo -n ${VarEmpty-'Пусто'}' '          # Только пробел
echo ${VarEmpty}                        # Только новая строка
echo -n ${VarEmpty:-'Пусто'}' '         # Пусто
echo ${VarEmpty}                        # Только новая

echo '- contents -'
echo ${VarSomething-'Содержимое'}        # Буквально
echo ${VarSomething:-'Содержимое'}       # Буквально

echo '- Sparse Array -'
echo ${ArrayVar[@]-'not set'}
```

```
# Искусство времени в ASCII
# Статус            Y==yes, N==no
#                    -       :-
# Не установлено     Y       Y       ${# ... } == 0
# Пусто              N       Y       ${# ... } == 0
# содержимое         N       N       ${# ... } > 0

#  Первая и/или вторая часть проверки может быть
#+ командой или строкой вызова функции.
echo
echo '- - Тест 1 для не определенного - -'
declare -i t
_decT() {
    t=$t-1
}

# Null ссылка, присвоено: t == -1
t=${#VarNull}                           # Результаты в ноль.
${VarNull- _decT }                      # Функция выполнилась, t теперь -1.
echo $t

# Null содержимое, присвоено: t == 0
t=${#VarEmpty}                          # Результаты в ноль.
${VarEmpty- _decT }                     # функция _decT НЕ выполнилась.
echo $t

# Содержимое, присвоено: t == символы не-null чисел
VarSomething='_simple'                  # Присваиваем функции правильное имя.
t=${#VarSomething}                      # не нулевой длины
${VarSomething- _decT }                 # Функция _simple выполнилась.
echo $t                                 # Обратите внимание на
                                        #+ действие Добавить-В.

# Упражнение:  очищаем этот пример.
unset t
unset _decT
VarSomething=Literal

echo
echo '- - Проверки и изменения - -'
echo '- Присвоить, если ссылка null -'
echo -n ${VarNull='NotSet'}' '          # NotSet NotSet
echo ${VarNull}
unset VarNull

echo '- Присвоить, ссылка если null -'
echo -n ${VarNull:='NotSet'}' '         # NotSet NotSet
echo ${VarNull}
unset VarNull

echo '- Не присваивать, если содержимое null -'
echo -n ${VarEmpty='Empty'}' '          # Только пробел
echo ${VarEmpty}
VarEmpty=''

echo '- Присвоить, если содержимое null -'
echo -n ${VarEmpty:='Empty'}' '         # Empty Empty
echo ${VarEmpty}
VarEmpty=''
```

```
echo '- Никаких изменений, если уже есть содержимое -'
echo ${VarSomething='Content'}          # Буквально
echo ${VarSomething:='Content'}         # Буквально


# "Разреженные индексы" массивов Bash
###
#  Bash-массивы это упакованные индексы, начинающиеся
#+ с индекса 0, если не указано иное.
###
#  Одним из способов «указания в противном случае»
#+ была инициализация ArrayVar. Вот другой способ:
###
echo
declare -a ArraySparse
ArraySparse=( [1]=один [2]='' [4]='четыре' )
# [0]=null ссылка, [2]=null содержимое, [3]=null ссылка

echo '- - Список разреженного массива  - -'
# С двойными кавычками, IFS по умолчанию, Glob-Pattern

IFS=$'\x20'$'\x09'$'\x0A'
printf %q "${ArraySparse[*]}"
echo

#  Обратите внимание, что вывод не проводит различия между
#+ "null содержимым"  и  "null ссылкой".
#  Оба выводятся, как будто заэкранированные пробелами.
###
#  Обратите внимание, что вывод НЕ содержит заэкранированные пробелы
#+ "null ссылки(ок)" первого элемента.
###
# Это поведение 2.04, 2.05a и 2.05b было указано и может измениться
#+ в будущих версиях Bash.

#  Для вывода разреженного массива и поддержания отношения [индекс]=значение
#+ без изменений требуется небольшое программирование.
#  Фрагмент возможного кода:
###
# local l=${#ArraySparse[@]}        # Подсчет определенных элементов
# local f=0                         # Подсчет обнаруженных индексов
# local i=0                         # Проверка индексов
(                                   # Последовательность безымянной функции
    for (( l=${#ArraySparse[@]}, f = 0, i = 0 ; f < l ; i++ ))
    do
        # 'если определено то...'
        ${ArraySparse[$i]+ eval echo '\ ['$i']='${ArraySparse[$i]} ; (( f++ )) }
    done
)

#  Читатель, столкнувшийся с фрагментом кода выше, может
#+ захотеть пересмотреть «списки команд» и «многокомандную строку»
#+ в вышеуказанном тексте "Advanced Bash Scripting Guide."
###
#  Примечание:
#  Версия "read -a array_name" команды "read"
#+ начинает заполнение array_name с нулевого индекса.
#  Разреженный массив не определяет значение индекса с нуля.
###
#  Пользователь, которому нужно читать/писать разреженный массив
#+ во внешнем хранилище или коммуникационном сокете, должен придумать
```

```
#+ подходящий код пары чтение/запись.
###
# Упражнение: Очистим это.

unset ArraySparse

echo
echo '- - Условная альтернатива (Но не замена)- -'
echo '- Без альтернативы если ссылка null -'
echo -n ${VarNull+'NotSet'}' '
echo ${VarNull}
unset VarNull

echo '- Без альтернативы если ссылка null -'
echo -n ${VarNull:+'NotSet'}' '
echo ${VarNull}
unset VarNull

echo '- Альтернатива если содержимое null -'
echo -n ${VarEmpty+'Empty'}' '              #  Empty
echo ${VarEmpty}
VarEmpty=''

echo '- Без альтернативы если содержимое null -'
echo -n ${VarEmpty:+'Empty'}' '             # Только пробел
echo ${VarEmpty}
VarEmpty=''

echo '- Альтернатива, если уже есть содержимое -'

# Буквальная альтернатива
echo -n ${VarSomething+'Content'}' '        # Буквальное содержимое
echo ${VarSomething}

# Вызванная функция
echo -n ${VarSomething:+ $(_simple) }' '    # Буквальная SimpleFunc
echo ${VarSomething}
echo

echo '- - Разряженный массив - -'
echo ${ArrayVar[@]+'Empty'}                 # Массив 'Empty'(ies)
echo

echo '- - Проверка 2 на неопределенные - -'

declare -i t
_incT() {
    t=$t+1
}

#  Примечание:
#  Это тот же тест, используемый в фрагменте кода
#+ листинга разреженного массива.

# Null ссылка, присвоено: t == -1
t=${#VarNull}-1                     # Результаты в минус один.
${VarNull+ _incT }                  # Не исполняемая.
echo $t' Null ссылка'

# Null содержимое, присвоено: t == 0
t=${#VarEmpty}-1                    # Результаты в минус один.
```

```
${VarEmpty+ _incT }                 # Исполняемая.
echo $t'  Null содержимое'

# Содержимое, установлено: t == (числовой не null символ)
t=${#VarSomething}-1                # не-null величина минус один
${VarSomething+ _incT }             # Исполняемая.
echo $t'  Contents'

# Упражнение: Очищаем этот пример.
unset t
unset _incT

# ${name?err_msg} ${name:?err_msg}
#  Следуют тем же правилам, но после - всегда выход,
#+ если действие указано после знака вопроса.
#  Действие после вопросительного знака может быть
#+ буквальным или результатом действия.
###
#  ${name?} ${name:?} является только проверкой, возвращать может только
#+ проверенное.



# Операции с элементами
# -----------------

echo
echo '- - Выбор  содержимого элемента - -'

#  Строки, массивы и позиционные параметры

#  Вызовите этот сценарий с несколькими аргументами,
#+ чтобы увидеть выбор параметра.

echo '- Все -'
echo ${VarSomething:0}              # все не-null символы
echo ${ArrayVar[@]:0}               # все элементы с содержимым
echo ${@:0}                         # Все параметры с содержимым
                                    # игнорируя параметр [0]

echo
echo '- Все после -'
echo ${VarSomething:1}              # все не-null символы после [0]
echo ${ArrayVar[@]:1}               # все элементы с содержимым после [0]
echo ${@:2}                         # все параметры с содержимым после [1]

echo
echo '- Диапазон после -'
echo ${VarSomething:4:3}            # Диапазон после
                                    # Три символа после
                                    # символа[3]

echo '- Разреженный массив -'
echo ${ArrayVar[@]:1:2}     #  четыре — Только элемент с содержимым.
                            #  Два элемента после (если много)
                            #  ПЕРВОГО С СОДЕРЖИМЫМ
                            #+ (ПЕРВОЕ С СОДЕРЖИМЫМ это рассматривается как
                            #+ нулевой индекс).
#  Исполняется, так как будто Bash рассматривает ТОЛЬКО элементы
#+ массива с содержимым.
#  printf %q "${ArrayVar[@]:0:3}"    # Попробуй это
```

```
#  В версиях Bash 2.04, 2.05a и 2.05b,
#+ не обрабатываются разреженные массивы, как и ожидалось,
#+ использующие такую запись.
#
#  Нынешний майнтейнер Bash, Chet Ramey, исправил это.


echo '- Не разреженный массив -'
echo ${@:2:2}               # Два параметра следующих за параметром [1]

# Примеры новой жертвы для строки одномерного массива:
stringZ=abcABC123ABCabc
arrayZ=( abcabc ABCABC 123123 ABCABC abcabc )
sparseZ=( [1]='abcabc' [3]='ABCABC' [4]='' [5]='123123' )

echo
echo ' - - Строка Жертва - -'$stringZ'- - '
echo ' - - Строка Жертва - -'${arrayZ[@]}'- - '
echo ' - - Разреженный массив - -'${sparseZ[@]}'- - '
echo ' - [0]==null ссылка, [2]==null ссылка, [4]==null содержимое - '
echo ' - [1]=abcabc [3]=ABCABC [5]=123123 - '
echo ' - подсчет элементов не-null-ссылок: '${#sparseZ[@]}''

echo
echo '- - Удаление префикса (слева) суб-элемента - -'
echo '- - Соответствие Glob-Pattern должно включать первый символ. - -'
echo '- - Глобальный шаблон может быть буквальным или результатом функции. - -'
echo


# Функция, возвращающая простой Glob-Pattern, буквально
_abc() {
    echo -n 'abc'
}

echo '- Короткий префикс (слева) -'
echo ${stringZ#123}                 # Без изменений (не префикс).
echo ${stringZ#$(_abc)}             # ABC123ABCabc
echo ${arrayZ[@]#abc}               # Применяется к каждому элементу.

# echo ${sparseZ[@]#abc}            # Версия-2.05b дампа ядра.
# С тех пор исправлена Chet Ramey.

# -Это было бы точнее- ***Содержимое-индекса-от-начала***
# echo ${#sparseZ[@]#*}             # Это НЕ допустимо в Bash.

echo
echo '- Длинный префикс (слева) -'
echo ${stringZ##1*3}                # Без изменений (не префикс)
echo ${stringZ##a*C}                # abc
echo ${arrayZ[@]##a*c}              # ABCABC 123123 ABCABC

# echo ${sparseZ[@]##a*c}           # Версия-2.05b дампов ядра.
# Исправлено Chet Ramey.

echo
echo '- - Удаление суффикса содержимого элемента (справа) - -'
echo '- - Последний символ должен соответствовать Glob-Pattern. - -'
echo '- - Глобальный шаблон может быть буквальным или результатом функции. - -'
echo
```

```
echo '- Короткий суффикс (справа) -'
echo ${stringZ%1*3}                 # Без изменений (не суффикс).
echo ${stringZ%$(_abc)}             # abcABC123ABC
echo ${arrayZ[@]%abc}               # Применяется для каждого элемента.

# echo ${sparseZ[@]%abc}            # Версия-2.05b дампов ядра.
# Исправлено Chet Ramey.

# -Это было бы точнее- Содержимое-индекса-от-конца
# echo ${#sparseZ[@]%*}             # Это НЕ допустимо в Bash.

echo
echo '- Длинный суффикс (справа) -'
echo ${stringZ%%1*3}                # Без изменений (не суффикс)
echo ${stringZ%%b*c}                # a
echo ${arrayZ[@]%%b*c}              # a ABCABC 123123 ABCABC a

# echo ${sparseZ[@]%%b*c}           # Версия-2.05b дампов ядра.
# Исправлено Chet Ramey.

echo
echo '- - Перемещение содержимого элемента - -'
echo '- - Содержимое элемента в любом месте в строке. - -'
echo '- - Первой спецификацией является Glob-Pattern - -'
echo '- - Glob-Pattern может быть буквенным или Glob-Pattern результата функции.
- -'
echo '- - Второй спецификацией может быть буква или результат функции. - -'
echo '- - Вторая спецификация может быть не указана. Звучит как'
echo '    : Перемещение-в-никуда (Удаление) - -'
echo


# Возврат функции простой, буквальный, Glob-Pattern
_123() {
    echo -n '123'
}

echo '- Замена первого вхождения -'
echo ${stringZ/$(_123)/999}         # Заменяется (123 является компонентом).
echo ${stringZ/ABC/xyz}             # xyzABC123ABCabc
echo ${arrayZ[@]/ABC/xyz}           # Применяется к каждому элементу.
echo ${sparseZ[@]/ABC/xyz}          # Работает, как ожидалось.

echo
echo '- Удаление первого вхождения -'
echo ${stringZ/$(_123)/}
echo ${stringZ/ABC/}
echo ${arrayZ[@]/ABC/}
echo ${sparseZ[@]/ABC/}

#  Замена не обязательно должна быть буквальной, так
#+ как допускается результат вызова функции.
#  Это вообще для всех форм замены.
echo
echo '- Заменить первое вхождение Результат-Из -'
echo ${stringZ/$(_123)/$(_simple)}  # Работает, как ожидалось.
echo ${arrayZ[@]/ca/$(_simple)}     # Применяется к каждому элементу.
echo ${sparseZ[@]/ca/$(_simple)}    # Работает, как ожидалось

echo
echo '- Замена всех вхождений -'
```

```
echo ${stringZ//[b2]/X}             # X-замена b и 2
echo ${stringZ//abc/xyz}            # xyzABC123ABCxyz
echo ${arrayZ[@]//abc/xyz}          # Применяется к каждому элементу.
echo ${sparseZ[@]//abc/xyz}         # Работает, как ожидалось.

echo
echo '- Удаление всех вхождений -'
echo ${stringZ//[b2]/}
echo ${stringZ//abc/}
echo ${arrayZ[@]//abc/}
echo ${sparseZ[@]//abc/}

echo
echo '- - Замена префикса (левого) содержимого элемента - -'
echo '- - Совпадение должно содержать первый символ. - -'
echo

echo '- Замена префикса (левого) содержимого -'
echo ${stringZ/#[b2]/X}             # Без изменений (это не префикс).
echo ${stringZ/#$(_abc)/XYZ}        # XYZABC123ABCabc
echo ${arrayZ[@]/#abc/XYZ}          # Применяется к каждому элементу.
echo ${sparseZ[@]/#abc/XYZ}         # Работает, как ожидалось.

echo
echo '- Удаление префикса (левого) содержимого -'
echo ${stringZ/#[b2]/}
echo ${stringZ/#$(_abc)/}
echo ${arrayZ[@]/#abc/}
echo ${sparseZ[@]/#abc/}

echo
echo '- - Замена суффикса (правого) содержимого элемента  - -'
echo '- - Совпадение должно содержать последний символ. - -'
echo

echo '- Замена суффикса содержимого -'
echo ${stringZ/%[b2]/X}             # Без изменений (Это не суффикс).
echo ${stringZ/%$(_abc)/XYZ}        # abcABC123ABCXYZ
echo ${arrayZ[@]/%abc/XYZ}          # Применяется к каждому элементу.
echo ${sparseZ[@]/%abc/XYZ}         # Работает, как ожидалось.

echo
echo '- Удаление содержимого суффикса -'
echo ${stringZ/%[b2]/}
echo ${stringZ/%$(_abc)/}
echo ${arrayZ[@]/%abc/}
echo ${sparseZ[@]/%abc/}

echo
echo '- - Особые случаи пустого Glob-Pattern - -'
echo

echo '- Все префиксы -'
# пустой шаблон содержимого строки означающий 'префикс'
echo ${stringZ/#/NEW}               # NEWabcABC123ABCabc
echo ${arrayZ[@]/#/NEW}             # Применяется к каждому элементу.
echo ${sparseZ[@]/#/NEW}            # Применяется так же к null-содержимому.
                                    # Это кажется разумно.

echo
echo '- Все суффиксы -'
```

```
# пустой шаблон содержимого строки означающий 'суффикс'
echo ${stringZ/%/NEW}                # abcABC123ABCabcNEW
echo ${arrayZ[@]/%/NEW}              # Применяется к каждому элементу.
echo ${sparseZ[@]/%/NEW}             # Применяется так же к null-содержимому.
                                     # Это кажется разумно.

echo
echo '- - Особый случай Для-каждого Glob-Pattern - -'
echo '- - - - Это красивая мечта - - - -'
echo

_GenFunc() {
    echo -n ${0}                     # Только иллюстрация.
    # На самом деле, это было бы произвольное вычисление.
}

# Все вхождения, соответствующие Любому шаблону.
# В данный момент //*/ не соответствует null-содержимому или null-ссылке.
# /#/ и /%/ не соответствуют null-содержимому, но соответствуют null-ссылке.
echo ${sparseZ[@]//*/$(_GenFunc)}


#  Возможный синтаксис создающий запись параметров,
#+ используемых в конструкции означает:
#   ${1} — Полный элемент
#   ${2} - Префикс, если что-то соответствует содержимому элемента
#   ${3} — Соответствующее содержимое элемента
#   ${4} - Суффикс,  если что-то соответствует содержимому элемента
#
# echo ${sparseZ[@]//*/$(_GenFunc ${3})}   # Здесь так же как ${1}.
# Возможно это будет осуществлено в будущих версиях Bash.


exit 0
```

Пример A-59.Проверка времени выполнения различных команд

```
#!/bin/bash
#  test-execution-time.sh
#  Пример Erik Brandsberg, для проверки времени выполнения
#+ указанных операций.
#  Ссылка в разделе "Оптимизация" главы "Разное".

count=50000
echo "Проверки вычислений"
echo "Вычисления с помощью \$(( ))"
time for (( i=0; i< $count; i++))
do
  result=$(( $i%2 ))
done

echo "Вычисления с помощью *expr*:"
time for (( i=0; i< $count; i++))
do
  result=`expr "$i%2"`
done

echo "Вычисления с помощью *let*:"
```

```
time for (( i=0; i< $count; i++))
do
  let result=$i%2
done

echo
echo "Проверяемые условия проверки"

echo "Проверка с помощью case:"
time for (( i=0; i< $count; i++))
do
  case $(( $i%2 )) in
    0) : ;;
    1) : ;;
  esac
done

echo "Проверка с помощью if [], без кавычек:"
time for (( i=0; i< $count; i++))
do
  if [ $(( $i%2 )) = 0 ]; then
     :
  else
     :
  fi
done

echo "Проверка с помощью if [], с кавычками:"
time for (( i=0; i< $count; i++))
do
  if [ "$(( $i%2 ))" = "0" ]; then
     :
  else
     :
  fi
done

echo "Проверка с помощью if [], с использованием -eq:"
time for (( i=0; i< $count; i++))
do
  if [ $(( $i%2 )) -eq 0 ]; then
     :
  else
     :
  fi
done

exit $?
```

Пример A-60. Ассоциативные массивы против обычных массивов (время выполнения)

```
#!/bin/bash
#  assoc-arr-test.sh
#  Сценарий проверки Benchmark, для сравнения время выполнения
#  индексов числового массива против ассоциативного массива.
#     Благодарность Erik Brandsberg.

count=100000       # Некоторые тесты, ниже, могут занять некоторое время.
```

```
declare simple     # При необходимости можно заменить на 20000.
declare -a array1
declare -A array2
declare -a array3
declare -A array4

echo "===Назначение тестов==="
echo

echo "Назначение простой переменной:"
time for (( i=0; i< $count; i++)); do
        simple=$i$i
done

echo "---"

echo "Назначение записи индекса числового массива:"
time for (( i=0; i< $count; i++)); do
        array1[$i]=$i
done

echo "---"

echo "Перезапись записи индекса числового массива:"
time for (( i=0; i< $count; i++)); do
        array1[$i]=$i
done

echo "---"

echo "Прямое чтение индекса числового массива:"
time for (( i=0; i< $count; i++)); do
        simple=array1[$i]
done

echo "---"

echo "Назначение записи ассоциативного массива:"
time for (( i=0; i< $count; i++)); do
        array2[$i]=$i
done

echo "---"

echo "Перезапись записи ассоциативного массива:"
time for (( i=0; i< $count; i++)); do
        array2[$i]=$i
done

echo "---"

echo "Прямое чтение записи ассоциативного массива:"
time for (( i=0; i< $count; i++)); do
        simple=array2[$i]
done

echo "---"

echo "Назначение случайного числа простой переменной:"
time for (( i=0; i< $count; i++)); do
        simple=$RANDOM
```

```
done

echo "---"

echo "Случайное назначение записи разреженного числового индексного массива в
ячейки по 64k:"
time for (( i=0; i< $count; i++)); do
        array3[$RANDOM]=$i
done

echo "---"

echo "Чтение записи разреженного числового индексного массива:"
time for value in "${array3[@]}"i; do
        simple=$value
done

echo "---"

echo "Случайное назначение записи ассоциативного массива в ячейки по 64k
cells:"
time for (( i=0; i< $count; i++)); do
        array4[$RANDOM]=$i
done

echo "---"

echo "Чтение записи разреженного ассоциативного индексного массива:"
time for value in "${array4[@]}"; do
        simple=$value
done

exit $?
```

Приложение B. Типы ссылок

Следующие типы ссылок предоставляют полезную информацию о некоторых понятиях сценариев.

Таблица B-1. Специальные переменные оболочки

Переменная	Значение
`$0`	Имя сценария
`$1`	Позиционный параметр #1
`$2 - $9`	Позиционные параметры #2 - #9
`${10}`	Позиционный параметр #10
`$#`	Количество позиционных параметров
`"$*"`	Все позиционные параметры (как одно слово) *

Переменная	Значение
"$@"	Все позиционные параметры (как разделенные строки)
${#*}	Количество позиционных параметров
${#@}	Количество позиционных параметров
$?	Возвращаемое значение
$$	ID (PID) процесса сценария
$-	Флаги передаваемые в сценарий (с помощью *set*)
$_	Последний аргумент предыдущей команды
$!	ID (PID) процесса последнего в фоне запущенного задания

***** *Должны быть в кавычках*, в противном случае по умолчанию будет **$@**.

Таблица B-2. Операторы ПРОВЕРКИ: Двоичное сравнение

Оператор	Значение	---	Оператор	Значение
Арифметическое сравнение			Строковое сравнение	
-eq	Равно		**=**	Равно
			==	Равно
-ne	Не равно		**!=**	Не равно
-lt	Меньше чем		**\<**	Меньше чем (ASCII) *
-le	Меньше чем или равно			
-gt	Больше чем		**\>**	Больше чем (ASCII) *
-ge	Больше чем или равно			
			-z	Пустая строка
			-n	Не пустая строка
Арифметическое сравнение	В двойных скобках ((...))			
>	Больше чем			
>=	Больше чем или равно			
<	Меньше чем			
<=	Меньше чем или равно			

*****_Если в двойных скобках_ [[...]] конструкции проверки, _то не нужно экранировать_ **** .

Таблица B-3. Операторы ПРОВЕРКИ: Файлы

Оператор	Проверяется	---	Оператор	Проверяется
-e	Файл существует		**-s**	Файл не *нулевого размера*
-f	Файл это *обычный* файл			
-d	Файл это *директория*		**-r**	Файл имеет права на *чтение*
-h	Файл это *символическая* ссылка		**-w**	Файл имеет права на *запись*

Оператор	Проверяется	---	Оператор	Проверяется
-L	Файл это *символьная* ссылка		**-x**	Файл имеет права на исполнение
-b	Файл это *блочное* устройство			
-c	Файл это *символьное* устройство		**-g**	Установленный флаг *sgid*
-p	Файл это *канал*		**-u**	Установленный флаг *suid*
-S	Файл это *сокет*		**-k**	Установленный "sticky bit"
-t	Файл связан с *терминалом*			
-N	Файл измененный с момента последнего чтения		F1 **-nt** F2	Файл F1 *новее* чем F2 *
-O	Ваш собственный файл		F1 **-ot** F2	Файл F1 *старее* чем F2 *
-G	*Id группы* такой же как и Ваш		F1 **-ef** F2	Файлы F1 и F2 жестко связаны с другим файлом *
!	НЕ (инвертирование смысла проверок выше)			

***** Бинарный оператор (требует двух операндов).

Таблица B-4. Параметры подстановки и расширения

Выражение	Значение
${var}	Значение переменной *var* (то же, что и *$var*)
${var-$DEFAULT}	Если *var* не присвоена, вычисляет выражение как *$DEFAULT* *
${var:-$DEFAULT}	Если *var* не присвоена или пустая, вычисляет выражение как *$DEFAULT* *
${var=$DEFAULT}	Если *var* не присвоена, вычисляет выражение как *$DEFAULT* *
${var:=$DEFAULT}	Если *var* не присвоена или пустая, вычисляет выражение как *$DEFAULT* *
${var+$OTHER}	Если *var* присвоена, вычисляет выражение как *$OTHER*, в противном случае, подобно пустой строке
${var:+$OTHER}	Если *var* присвоена, вычисляет выражение как *$OTHER*, в противном случае, подобно пустой строке
${var?$ERR_MSG}	Если *var* не присвоена, выводит на экран *$ERR_MSG* и завершает сценарий со статусом выхода 1.*
${var:?$ERR_MSG}	Если *var* не присвоена, выводит на экран *$ERR_MSG* и завершает сценарий со статусом выхода 1.*
${!varprefix*}	Сравнение всех ранее объявленных переменных, начиная с *varprefix*

Выражение	Значение
`${!varprefix@}`	Сравнение всех ранее объявленных переменных, начиная с *varprefix*

***** Если *var* присвоена, вычисляет выражение как *$var* без побочных эффектов.

 # Обратите внимание что поведение некоторых операторов выше отличается от более ранних версий Bash.

Таблица B-5. Строковые операции

Выражение	Значение
`${#string}`	Длина *$string*
`${string:position}`	Извлечение содержимого *$string* начиная с *$position*
`${string:position:length}`	Извлечение *$length* содержащихся символов из *$string* начиная с *$position* [нулевая индексация, позиция первого символа 0]
`${string#substring}`	Полоса коротких соответствий *$substring* от начала *$string*
`${string##substring}`	Полоса длинных соответствий *$substring* от начала *$string*
`${string%substring}`	Полоса коротких соответствий *$substring* от конца *$string*
`${string%%substring}`	Полоса длинных соответствий *$substring* от конца *$string*
`${string/substring/replacement}`	Замена *первого* соответствия *$substring* на *$replacement*
`${string//substring/replacement}`	Замена *всех* соответствий *$substring* на *$replacement*
`${string/#substring/replacement}`	Если *$substring* соответствует концу *вначале* *$string*, заменет *$substring* на *$replacement*
`${string/%substring/replacement}`	Если *$substring* соответствует концу *в конце* *$string*, заменяет *$substring* на *$replacement*
`expr match "$string" '$substring'`	Длина сравниваемой *$substring** от начала *$string*
`expr "$string" : '$substring'`	Длина сравниваемой *$substring** от

Выражение	Значение
	начала *$string*
`expr index "$string" $substring`	Числовая позиция в *$string* первого символа *$substring** который соответствует [0 если не соответствует, первый символ считается как позиция 1]
`expr substr $string $position $length`	Извлекает *$length* символов из *$string* начиная с *$position* [0 если не соответствует, первый символ считается как позиция 1]
`expr match "$string" '\($substring\)'`	Извлечение *$substring**, поиск от начала *$string*
`expr "$string" : '\($substring\)'`	Извлечение *$substring** , поиск от начала *$string*
`expr match "$string" '.*\($substring\)'`	Извлечение *$substring**, поиск от конца *$string*
`expr "$string" : '.*\($substring\)'`	Извлечение *$substring**, поиск от конца *$string*

***** Где *$substring* это Регулярное выражение.

Таблица B-6. Различные конструкции

Выражение	Интерпретация
Скобки	
`if [ УСЛОВИЕ ]`	Конструкция проверки
`if [[ УСЛОВИЕ ]]`	Расширяемая конструкция проверки
`Array[1]=элемент1`	Инициализация массива
`[a-z]`	Диапазон символов в регулярном выражении
Фигурные скобки	
`${переменная}`	Подстановка параметров
`${!переменная}`	Косвенная ссылка на переменную
`{ команда1; команда2; ... командаN; }`	Блок кода
`{строка1,строка2,строка3,...}`	Расширение в скобках
`{a..z}`	Расширяемое расширение скобок
`{}`	Замена текста, после **find** и **xargs**
Круглые скобки	
`( команда1; команда2 )`	Группа команд выполняемая в подоболочке
`Array=(элемент1 элемент2 элемент3)`	Инициализация массива
`result=$(КОМАНДА)`	Подстановка команды, новый стиль

Выражение	Интерпретация
`>(КОМАНДА)`	Процесс замены
`<(КОМАНДА)`	Процесс замены
Двойные круглые скобки	
`(( var = 78 ))`	Целочисленная арифметика
`var=$(( 20 + 5 ))`	Целочисленная арифметика, с присвоением переменной
`(( var++ ))`	Увеличение переменной в стиле *Cu*
`(( var-- ))`	Уменьшение переменной в стиле *Cu*
`(( var0 = var1<98?9:21 ))`	Тройная операция в стиле Си
Кавычки	
`"$переменная"`	«Мягкие» кавычки
`'строка'`	«Жесткие» кавычки
Обратные кавычки	
`result=`КОМАНДА``	Подстановка команды, классический стиль

Приложение C. Sed и Awk

Содержание

Это очень краткое введение в утилиты текстовой обработки **sed** и **awk**. Мы рассмотрим только несколько основных команд, но их будет достаточно для понимания простых конструкций **sed** и **awk** внутри сценариев оболочки.

sed: не интерактивный редактор текстовых файлов

awk: поле-ориентированный язык обработки шаблонов с синтаксисом в стиле **Си**

При всех их различиях, эти две утилиты немного похожи синтаксисом вызова, использованием регулярных выражений, чтением ввода из `stdin` по умолчанию и выводом результата в `stdout`. Это прекрасные инструменты UNIX, хорошо работающие вместе. Выходные данные из одной могут быть переданы другой, а их комбинирование придает сценариям мощность Perl.

Одним из важных различий между утилитами является то, что сценарии могут легко передавать аргументы в **sed**, что более сложно для **awk** (см. Пример 36-5 и Пример 28-2).

C.1. Sed

Sed это не интерактивный [1] потоковый редактор (**s**tream **ed**itor). Он получает текстовый ввод, из `stdin` или из файла, выполняет некоторые операции с заданной введенной строкой, одной строкой за раз, а затем выводит результат в `stdout` или в файл. В рамках сценария оболочки, *sed*, обычно, является одним из нескольких компонентов конвейера.

Sed определяет, какими строками ввода, переданными ему, он будет оперировать из *диапазона адресов*. [2] Задается этот диапазон адресов номером строки или шаблоном соответствия. Например, сигнал *sed* `3d` удаляет строку 3 ввода, а `/Windows/d` говорит sed, что нужно удалить каждую строку ввода, содержащую соответствие "Windows".

Из всех операций утилиты *sed* мы сосредоточимся, главным образом, на трех наиболее часто используемых. Это **p**rinting - вывод (в `stdout`), **d**eletion - удаление, и **s**ubstitution — подстановка или замена.

Таблица C-1. Основные операторы sed

Оператор	**Название**	**Действие**
`[address-range]/p`	print	Вывод [указанный диапазон адресов]
`[address-range]/d`	delete	Удаление [указанный диапазон адресов]
`s/шаблон1/шаблон2/`	substitute	Первое соответствие в строке *шаблону1* заменяется *шаблоном2*
`[address-range]/s/шаблон1/шаблон2/`	substitute	Первое соответствие в строке *шаблону1* заменяется *шаблоном2*, в указанном `address-range`
`[address-range]/y/шаблон1/шаблон2/`	transform	Замена любого символа *шаблон1* соответствующим символом *шаблона2* , в

Оператор	Название	Действие
		указанном *address-range* (эквивалент **tr**)
[address] i шаблон Filename	insert	Вставка *шаблона* в указанный адрес в файле Filename. Обычно используется с опцией -i *in-place*.
g	global	Операция с *любым шаблоном* каждого соответствия в строке ввода

Если оператор **g** (global) добавляется в команду *заменить*, замена будет только для *первого* случая соответствия шаблона в каждой строке.

Из командной строки и в сценарии оболочки работа *sed* может потребовать кавычек и определенных параметров.

```
sed -e '/^$/d' $filename
#  Опция -e  вызывает следующие строки, которые будут интерпретироваться как
#+ инструкция для редактирования.
#  (Если передается в sed только одна инструкция, '-e' является
необязательным.)
#  "Строгие" кавычки ('') защищают символы рег. выражения инструкции, в теле
#+ сценария, от истолкования их, как специальных символов.
# (Это оговорено инструкцией расширения рег. выражений  для sed.)
#
# Работает с текстом содержащимся в файле $filename.
```

В некоторых случаях команда редактирования *sed* не будет работать в одинарных кавычках.

```
filename=file1.txt
pattern=BEGIN

  sed "/^$pattern/d" "$filename"  # Работает, как указано.
# sed '/^$pattern/d' "$filename"    даст не ожиданный результат.
#        В данном случае, строгие кавычки (' ... '),
#+      "$pattern" не расширяет "BEGIN".
```

Sed использует опцию -e, для указания, что следующая строка является инструкцией или набором инструкций. Если существует только одна инструкция, содержащаяся в строке, то эта опция может быть опущена.

```
sed -n '/xzy/p' $filename
# Опция -n говорит sed выводить только строки соответствующие шаблону.
# В ином случае выводятся все строки ввода.
# Опция -e не нужна,т.к. здесь только единственная инструкция редактирования.
```

Таблица C-2. Примеры операций sed

Нотация	Действие
`8d`	Удаление *8-й* строки ввода.
`/^$/d`	Удаление *всех* пустых строк.
`1,/^$/d`	Удаление от начала ввода до первой пустой строки, включая первую пустую строку.
`/Jones/p`	Вывод на экран только строк содержащих "*Jones*" (с опцией `-n`).
`s/Windows/Linux/`	Замена найденного первого случая "*Windows*" в каждой строке ввода на "*Linux*".
`s/BSOD/stability/g`	Замена найденного первого случая "*BSOD*" в каждой строке ввода на "*stability*"
`s/ *$//`	Удаление всех пробелов в конце каждой строки.
`s/00*/0/g`	Сжатие всех последовательностей нулей подряд в один ноль.
`echo "Working on it." | sed -e '1i How far are you along?'`	Вывод на экран "*How far are you along?*" первой строкой, "*Working on it*" - второй.
`5i 'Linux is great.' file.txt`	Вставка '*Linux is great.*' в строку *5* файла file.txt.
`/GUI/d`	Удаление всех строк содержащих "GUI".
`s/GUI//g`	Удаление всех случаев "GUI", оставляя остальную часть каждой строки не измененной.

Замена строки нулевой длины на другую эквивалентно удалению строки в строке ввода. Она оставляет оставшуюся часть строки нетронутой. Применение **`s/GUI//`** к строке

```
The most important parts of any application are its GUI and sound effects
```

Даст результат

```
The most important parts of any application are its  and sound effects
```

Обратный слэш принуждает *sed* к замене команды, для продолжения на следующей строке. Это эффект использования *перевода строки* в конце первой строки, как *замены строки*.

```
s/^  */\
/g
```

Эта подстановка заменяет пробел в начале новой строки переводом строки. Конечный результат - замена отступа абзаца пустой строкой между абзацами.

Диапазон адресов, следующих одной или более операций, может потребовать открытой и закрытой фигурных скобок, в соответствующих строках.

```
/[0-9A-Za-z]/,/^$/{
/^$/d
}
```

Удаляется только первая, из каждого последовательного ряда пустых строк. Это может быть полезно для единственного пробела в текстовом файле, но сохраняются пустые строки между абзацами.

Разделение в *sed* обозначается `/`. Однако *sed* позволяет использовать и другие разделители, такие как `%`. Используется, когда `/` является частью заменяемой строки, как в `pathname` файла. См. Пример 11-10 и Пример 16-32.

Быстрый способ установки двойного пробела текстового файла это **`sed G filename`**.

Наглядные примеры *sed* в сценариях, см.:

1. Пример 36-1
2. Пример 36-2
3. Пример 16-3
4. Пример A-2
5. Пример 16-17
6. Пример 16-27
7. Пример A-12
8. Пример A-16
9. Пример A-17
10. Пример 16-32
11. Пример 11-10
12. Пример 16-48
13. Пример A-1
14. Пример 16-14
15. Пример 16-12
16. Пример A-10
17. Пример 19-12
18. Пример 16-19

19. Пример A-29
20. Пример A-31
21. Пример A-24
22. Пример A-43
23. Пример A-55

Для дополнительного изучения *sed* обратитесь к соответствующей ссылке в Библиографии.

Примечания

[1] *Sed* выполняется без вмешательства пользователя.

[2] Если диапазон адресов не указан, значение по умолчанию — *все строки*.

C.2. Awk

Awk[1] — полнофункциональный язык обработки текста с синтаксисом, напоминающим язык Си. Хотя он имеет обширный набор операторов и возможностей, мы здесь охватим лишь немногие из них - наиболее полезные в сценариях оболочки.

Awk делит каждую введенную строку, переданную ей, на *поля*. По умолчанию, поле представляет собой строку символов, разделенных *пробелами*, хотя это можно изменить. *Awk* анализирует и обрабатывает каждое поле в отдельности. Что делает его идеальным для работы со структурированными текстовыми файлами - особенно таблицами - данными, организованными соответствующими частями, такими как строки и столбцы.

Строгие кавычки и *фигурные скобки* вкладывают блоки кода *awk* в сценарий.

```
# $1 это поле #1, $2 это поле #2, и т.д..

echo one two | awk '{print $1}'
# one

echo one two | awk '{print $2}'
#  two

# А что является полем #0 ($0)?
echo one two | awk '{print $0}'
# one two
# Все поля!


awk '{print $3}' $filename
# Выводит поле #3 файла $filename в stdout.

awk '{print $1 $5 $6}' $filename
# Выводит поля #1, #5 и #6 файла $filename.
```

```
awk '{print $0}' $filename
# Выводит весь файл!
# Такой же эффект как:   cat $filename . . . или . . . sed '' $filename
```

Мы только что рассмотрели команду awk *print* в действии. Другой особенностью awk, с которой мы столкнемся, являются *переменные*. Awk обрабатывает переменные подобно сценариям оболочки, хотя еще более гибко.

```
{ total += ${column_number} }
```

Здесь добавляется значение `column_number` кс нарастающим итогом `total>`. Наконец, для вывода «`total`», существует команда блока **END**, выполняющаяся ***после*** обработки сценарием всех входные данных.

```
END { print total }
```

В соответствии **END**, существует и **BEGIN**, для блока кода, который необходимо выполнить ***перед началом*** обработки входных данных *awk*.

В следующем примере демонстрируется, как *awk* может добавить в сценарий оболочки средства синтаксического анализа текста.

Пример C-1. Подсчет вхождений буквы

```
#! /bin/sh
# letter-count2.sh: Подсчет вхождений буквы в текстовом файле.
#
# Сценарий nyal [nyal@voila.fr].
# Используется в ABS Guide с разрешения.
# Комментировано и переформатировано автором ABS Guide.
# Version 1.1: Изменение для работы с gawk 3.1.3.
#              (Будет по-прежнему работать с более ранними версиями.)


INIT_TAB_AWK=""
# Параметр инициализации сценария awk.
count_case=0
FILE_PARSE=$1

E_PARAMERR=85

usage()
{
    echo "Usage: letter-count.sh файл буквы" 2>&1
    # Например:   ./letter-count2.sh filename.txt a b c
    exit $E_PARAMERR  # Слишком мало аргументов, передаваемых сценарию.
}

if [ ! -f "$1" ] ; then
    echo "$1: Нет такого файла." 2>&1
    usage                 # Вывод сообщения пользователю и выход.
fi
```

```
if [ -z "$2" ] ; then
    echo "$2: Не указаны буквы." 2>&1
    usage
fi

shift                       # Буквы указаны.
for letter in `echo $@`     # По одной ...
  do
  INIT_TAB_AWK="$INIT_TAB_AWK tab_search[${count_case}] = \
  \"$letter\"; final_tab[${count_case}] = 0; "
  # Передается в качестве параметра сценария в awk ниже.
  count_case=`expr $count_case + 1`
done

# DEBUG:
# echo $INIT_TAB_AWK;

cat $FILE_PARSE |
# Туннелирование целевого файла следующего сценария в awk.

# ------------------------------------------------------------------------
# Ранняя версия сценария:
# awk -v tab_search=0 -v final_tab=0 -v tab=0 -v \
# nb_letter=0 -v chara=0 -v chara2=0 \

awk \
"BEGIN { $INIT_TAB_AWK } \
{ split(\$0, tab, \"\"); \
for (chara in tab) \
{ for (chara2 in tab_search) \
{ if (tab_search[chara2] == tab[chara]) { final_tab[chara2]++ } } } } \
END { for (chara in final_tab) \
{ print tab_search[chara] \" => \" final_tab[chara] } }"
# ------------------------------------------------------------------------
#  Все сложное - просто ...
#+ циклы for, проверки if и пара специальных функций.

exit $?

# Сравните этот сценарий с letter-count.sh.
```

Более простые примеры awk в сценариях см.:

1. Пример 15-14
2. Пример 20-8
3. Пример 16-32
4. Пример 36-5
5. Пример 28-2
6. Пример 15-20
7. Пример 29-3

8. Пример 29-4
9. Пример 11-3
10. Пример 16-61
11. Пример 9-16
12. Пример 16-4
13. Пример 10-6
14. Пример 36-19
15. Пример 11-9
16. 16. Пример 36-4
17. 17. Пример 16-53
18. 18. Пример T-3

Мы рассмотрели здесь awk, парни, но можно узнать много больше. Смотрите соответствующие ссылки в Библиографии.

Примечания

[1] Название происходит от инициалов ее авторов, **A**ho, **W**einberg и **K**ernighan.

Приложение D. Анализ и управление путями имен (Pathnames)

Emmanual Rouat внес следующий пример синтаксического анализа и преобразования файлов и, в частности, **путей**. В нем используется функциональность *sed*.

```
#!/usr/bin/env bash
#-------------------------------------------------------------
# Управление PATH, LD_LIBRARY_PATH, переменными MANPATH...
# Emmanuel Rouat <no-email>
# (Вдохновленный документацией bash «pathfuncs» и найденными обсуждениями
# переполнение стека:
# http://stackoverflow.com/questions/370047/
# http://stackoverflow.com/questions/273909/#346860 )
# Последнее изменение: Sat Sep 22 12:01:55 CEST 2012
#
# Следующие функции правильно обрабатывают пробелы.
# Я предполагаю, что эти функции находятся в .bash_profile, а не в
# .bashrc.
#
# Модульный аспект этих функций должен сделать их
```

```
# легко расширяемыми для обработки замены PATH, вместо удаления и т.д. ....
#
# См. http://www.catonmat.net/blog/awk-one-liners-explained-part-two/
# (пункт 43) для объяснения удаления «дублированной записи»
# (это красивый трюк!)
#-------------------------------------------------------------

# Выводит $@ (обычно PATH) как список.
function p_show() { local p="$@" && for p; do [[ ${!p} ]] &&
echo -e ${!p//:/\\n}; done }

# Фильтр пустых строк, нескольких/конечных слэшей и повторяющихся записей.
function p_filter()
{ awk '/^[ \t]*$/ {next} {sub(/\/+$/, "");gsub(/\/+/, "/")}!x[$0]++' ;}

# Восстановление списка элементов слова, разделенного ':' (как в PATH).
function p_build() { paste -sd: ;}

# Очистка $1 (типично для PATH) и восстановление ее
function p_clean()
{ local p=${1} && eval ${p}='$(p_show ${p} | p_filter | p_build)' ;}

# Удаление $1 из $2 (при переполнении стека, изменяя).
function p_rm()
{ local d=$(echo $1 | p_filter) p=${2} &&
  eval ${p}='$(p_show ${p} | p_filter | grep -xv "${d}" | p_build)' ;}

#  То же что и предыдущее, но фильтры на основе шаблона (опасно...
#+ не используйте в качестве шаблона 'bin' или '/'  !).
function p_rmpat()
{ local d=$(echo $1 | p_filter) p=${2} && eval ${p}='$(p_show ${p} |
  p_filter | grep -v "${d}" | p_build)' ;}

# Удаление $1 из $2 и добавление очистки.
function p_append()
{ local d=$(echo $1 | p_filter) p=${2} && p_rm "${d}" ${p} &&
  eval ${p}='$(p_show ${p} d | p_build)' ;}

# Удаление $1 из $2 и вставка его очищенного.
function p_prepend()
{ local d=$(echo $1 | p_filter) p=${2} && p_rm "${d}" ${p} &&
  eval ${p}='$(p_show d ${p} | p_build)' ;}

# Некоторые тесты:
echo
MYPATH="/bin:/usr/bin/:/bin://bin/"
p_append "/project//my project/bin" MYPATH
echo "Добавление '/project//my project/bin' в '/bin:/usr/bin/:/bin://bin/'"
echo "(результат должен быть: /bin:/usr/bin:/project/my project/bin)"
echo $MYPATH

echo
MYOTHERPATH="/bin:/usr/bin/:/bin:/project//my project/bin"
p_prepend "/project//my project/bin" MYOTHERPATH
echo "Вставка '/project//my project/bin' \
в '/bin:/usr/bin/:/bin:/project//my project/bin/'"
echo "(результат должен быть: /project/my project/bin:/bin:/usr/bin)"
echo $MYOTHERPATH

echo
p_prepend "/project//my project/bin" FOOPATH  # FOOPATH не существует.
```

```
echo "Вставка '/project//my project/bin' в не объявленную переменную"
echo "(результат должен быть: /project/my project/bin)"
echo $FOOPATH

echo
BARPATH="/a:/b/://b c://a:/my local pub"
p_clean BARPATH
echo "Очистка BARPATH='/a:/b/://b c://a:/my local pub'"
echo "(результат должен быть: /a:/b:/b c:/my local pub)"
echo $BARPATH
```

***

David Wheeler любезно позволил мне использовать его поучительные примеры.

```
Делаем правильно: кратко
David Wheeler
http://www.dwheeler.com/essays/filenames-in-shell.html

Итак, как правильно обрабатывать имена в оболочке? Вот краткий обзор, для
нетерпеливых, которые «просто хотят ответ» о том, как делать это правильно.
Короче говоря: используем двойные кавычки   "$variable" вместо $variable,
устанавливаем IFS вместо новой строки и табуляции, начало (префиксы) всех
символов подстановок/файлов,которые работают правильно, не могут начинаться с
"-" при расширении и использовании одного из нескольких шаблонов. Вот некоторые
из этих шаблонов, работающих правильно  :


 IFS="$(printf '\n\t')"
 # Удаление ПУСТОГО МЕСТА, т.к. файловые имена с пробелами хорошо работают.

 #  Правильное использование символов подстановки:
 #+ всегда используется цикл "for", префикс символа подстановки, проверка
 #+ на наличие:
 for file in ./* ; do          # Используйте "./*" ... НИКОГДА не голое "*" ...
   if [ -e "$file" ] ; then    # Убеждаемся, что совпадение не пустое.
     КОМАНДА ... "$file" ...
   fi
 done

 #  Правильное использование символов подстановки, но требуется нестандартное
 #+ расширение bash.
 shopt -s nullglob  #  Расширение Bash,
                    #+ поэтому это простой шаблон соответствия будет работать.
 for file in ./* ; do        # Используйте "./*", НЕ пользуйтесь голой "*"
   COMMAND ... "$file" ...
 done

 #  Правильная обработка любых файлов;
 #+ может быть громоздко, если COMMAND большая:
 find ... -exec COMMAND... {} \;
 find ... -exec COMMAND... {} \+ # Если несколько файлов в порядке для COMMAND.


 #  Пропуск имен файлов с управляющими символами
 #+ (содержащих табуляцию и перевод строки).
 IFS="$(printf '\n\t')"
 controlchars="$(printf '*[\001-\037\177]*')"
 for file in $(find . ! -name "$controlchars"') ; do
   COMMAND "$file" ...
```

```
done

#  Прекрасно, если имена файлов не содержат табуляции и перевода строки --
#+ остерегайтесь предположений.
IFS="$(printf '\n\t')"
for file in $(find .) ; do
  COMMAND "$file" ...
done

# Нужны нестандартные, но общие расширения в find и xargs:
find . -print0 | xargs -0 COMMAND

# Нужны нестандартные расширения в find и в оболочке (в работе bash).
#  установленные однажды переменные не могут оставаться по окончанию цикла:
find . -print0 | while IFS="" read -r -d "" file ; do ...
  COMMAND "$file" # "$file" используется в кавычках, не $file, как везде.
done

#  Требуются нестандартные расширения, в find и оболочка (в работе bash).
#  Основная система должна включать в себя именованные каналы (FIFO)
#+ или механизм /dev/fd.
#  В этой версии, переменные *do* установлены после окончания цикла
#+ и вы можете читать из stdin.
#+ (Измените 4 на другое число, если нужно fd 4.)

while IFS="" read -r -d "" file <&4 ; do
  COMMAND "$file"   # Используется в кавычках "$file" -- не $file, везде.
done 4< <(find . -print0)


#  Версия именованного канала.
#  Требует нестандартных расширений для find и чтения оболочки (bash OK).
#  Основная система должна содержать именованные каналы (FIFO).
#  Снова, в этой версии, переменные *do* установлены после окончания цикла
#+ и вы можете читать из stdin.
# (Измените 4 на другое число, если fd 4 необходимо.).

mkfifo mypipe

find . -print0 > mypipe &
while IFS="" read -r -d "" file <&4 ; do
  COMMAND "$file" # Используется в кавычках "$file", не $file, как везде.
done 4< mypipe
```

Приложение E. Коды выхода, специальные значения

Таблица E-1. *Зарезервированные* Коды выхода

Число кода выхода	Смысл	Пример	Комментарии
1	Охватывает все общие ошибки	`let "var1 = 1/0"`	Прочие ошибки, такие как «деление на ноль» и другие недопустимые операции
2	Неправильное	`empty_fun`	Отсутствует ключевое слово или

Число кода выхода	Смысл	Пример	Комментарии
	использование встроенных команд оболочки (согласно документации Bash)	`ction() {}`	команда, или проблемы с правами (и код возврата diff ошибка сравнения двоичного файла).
126	Вызываемая команда не может быть выполнена	`/dev/null`	Проблема с правами или команда не является исполняемой
127	"command not found"	`illegal_command`	Возможные проблемы с `$PATH` или типом
128	Недопустимый аргумент для **exit**	`exit 3.14159`	**exit** принимает только целые аргументы в диапазоне 0 - 255 (см. первую сноску)
128+n	Неустранимая ошибка сигнала "n"	*`kill -9 $PPID of script`*	**$?** возвращает 137 (128 + 9)
130	Сценарий завершается нажатием Control-C	*`Ctl-C`*	Control-C это сигнал критической ошибки 2, (130 = 128 + 2, см. выше)
255*	Статус выхода вне диапазона	`exit -1`	**exit** принимает только целые аргументы в диапазоне 0 - 255

Согласно приведенной выше таблице, коды выхода 1-2, 126-165 и 255 [1] имеют особое значение и поэтому пользователям следует избегать указывать эти параметры выхода. Завершение сценария выходом *exit 127*, конечно, приводит к путанице (это код ошибки 'команда не найдена' или определенная пользователем?). Тем не менее, многие сценарии используют *exit 1* в качестве общего решения-на-ошибки. Так код выхода 1 означает любые возможные ошибки, что не особенно полезно при отладке.

Предпринимались попытки систематизировать номера состояния выхода (см. `/usr/include/sysexits.h`), но это предназначалось для программистов Си и C++. Аналогичный стандарт при написании сценариев может быть целесообразен. Автор этого документа предлагает ограничить определенные пользователем коды выхода в диапазоне 64 - 113 (в дополнение к 0, при успешном выходе), для соответствия стандарту Си/C++. Это выделит 50 действительных кодов и сделает устранение неполадок в сценариях более простым. [2] Все пользовательские коды выхода, в прилагаемых примерах в этом документе, соответствуют этому стандарту, за исключением случаев, когда существуют переопределяющие обстоятельства, как в Примере 9-2.

Ввод `$?` из командной строки после завершения сценария даст результат, согласующийся с таблицей выше, но только в Bash или строке sh. Запуск в Си-оболочке или tcsh может дать отличные значения.

Примечания

[1] Выход за диапазон значений выхода может привести к неожиданным кодам выхода. Значение выхода больше, чем 255, возвращает код выхода ***по модулю*** 256. Например,

exit 3809 дает код выхода 225 (3809% 256 =225).

[2] Обновление `/usr/include/sysexits.h` выделяет ранее неиспользуемые коды выхода 64 - 78. Можно ожидать, что диапазон не распределенных кодов возврата в будущем будет дополнительно ограничен. Автор этого документа не будет делать исправления в примерах сценариев, для соответствия меняющимся стандартам. Это не должно вызвать каких-либо проблем, так как нет никакого перекрытия или конфликта в использовании кодов выхода между собранными двоичными файлами C/C++ и сценариями оболочки.

Приложение F. Детальное введение в I/O и перенаправление I/O

написано Stéphane Chazelas, а проверено автором документа

Команда ожидает доступности первых трех **файловых дескрипторов**. Первый, *fd 0* (стандартный ввод, `stdin`), предназначен для чтения. Два других (*fd 1*, `stdout` и *fd 2* `stderr`) - для записи.

Существуют `stdin`, `stdout` и `stderr`, связанные с каждой командой. *`ls 2>&1`* означает временную связь `stderr` команды **`ls`** с тем же «ресурсом», что и `stdout` оболочки.

По соглашению команда считывает входные данные из fd 0 (`stdin`), выводит на экран нормальный вывод fd 1 (`stdout`) и ошибку вывода с fd 2 (`stderr`). Если один из этих трех fd не является открытым, могут возникнуть проблемы:

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

Например, при запуске ***xterm***, она сначала инициализирует себя. Перед запуском оболочки пользователя, ***xterm*** открывает терминальное устройство (`/dev/pts/<n>` или что-то подобное) три раза.

В этот момент Bash наследует эти три файловых дескриптора, и любая команда (дочерний процесс), запущенная Bash, в свою очередь наследует их, за исключением тех случаев, когда вы перенаправляете команду. ***Перенаправление*** означает переназначение одного из дескрипторов файла в другой файл (или канал, или что-то доступное). Файловые дескрипторы могут быть переназначены локально (для команды, группы команд, **подоболочки**, циклов ***while***, ***if***, ***case*** или ***for*** ...), или глобально, на оставшуюся часть оболочки (с помощью команды **`exec`**).

`ls > /dev/null` означает запуск **`ls`** с его fd 1 связанным с `/dev/null`.

```
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    363 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        2u   CHR  136,1         3 /dev/pts/1


bash$ exec 2> /dev/null
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    371 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        2w   CHR    1,3       120 /dev/null


bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    379 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    379 root    1w  FIFO    0,0      7118 pipe
 lsof    379 root    2u   CHR  136,1         3 /dev/pts/1


bash$ echo "$(bash -c 'lsof -a -p $$ -d0,1,2' 2>&1)"
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    426 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    426 root    1w  FIFO    0,0      7520 pipe
```

```
lsof    426 root    2w  FIFO    0,0      7520 pipe
```

Это работает для различных типов перенаправлений.

Упражнение: Проанализируйте следующий сценарий.

```
#! /usr/bin/env bash

mkfifo /tmp/fifo1 /tmp/fifo2
while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 & exec 7> /tmp/fifo1
exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)

exec 3>&1
(
 (
  (
   while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee /dev/stderr \
   | tee /dev/fd/4 | tee /dev/fd/5 | tee /dev/fd/6 >&7 & exec 3> /tmp/fifo2

   echo 1st, в stdout
   sleep 1
   echo 2nd, в stderr >&2
   sleep 1
   echo 3rd, в fd 3 >&3
   sleep 1
   echo 4th, в fd 4 >&4
   sleep 1
   echo 5th, в fd 5 >&5
   sleep 1
   echo 6th, through a pipe | sed 's/.*/PIPE: &, в fd 5/' >&5
   sleep 1
   echo 7th, в fd 6 >&6
   sleep 1
   echo 8th, в fd 7 >&7
   sleep 1
   echo 9th, в fd 8 >&8

  ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
 ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-

rm -f /tmp/fifo1 /tmp/fifo2


# Удачи!

exit 0
```

Приложение G. Опции командной строки

Содержание

Многие исполняемые файлы, двоичные файлы или файлы сценариев, принимают параметры для изменения их поведения во время выполнения. Например: ввод из командной строки `command -o` вызовет ***command*** с опцией `o`.

G.1. Обычные опции командной строки

Со временем свободный стандарт значений флагов параметра командной строки развивался. Утилиты GNU более соответствуют этому «стандарту», чем более старые утилиты UNIX.

Традиционно параметры командной строки UNIX состоят из тире и одной или нескольких букв нижнего регистра. Утилиты GNU добавили *двойное* тире, а затем законченное или составное слово.

Две наиболее широко используемые опции:

- **`-h`**

  `--help`

  Помощь, справка: Выдает сообщение об использовании и выход.

- **`-v`**

  `--version`

  Версия: Выводит версию программы и выход.

Другие общие опции:

- **`-a`**

  `--all`

  Все: Показывает *всю* информацию или действует *всеми* аргументами.

- **`-l`**

  `--list`

  Список: список файлов или аргументов без других действий.

- **`-o`**

  Вывод файла

- **`-q`**

  `--quiet`

  Скрыть: подавляет `stdout`.

- **`-r`**

  `-R`

`--recursive`

Рекурсивность: рекурсивное действие (вниз по дереву директории).

- **`-v`**

  `--verbose`

  Подробно: выводит дополнительные сведения в `stdout` или `stderr`.

- **`-z`**

  `--compress`

  Сжатие: применение сжатия (обычно gzip).

Однако:

- В ***tar*** и ***gawk***:

  `-f`

  `--file`

  Файл: следует имя файла.

- В ***cp***, ***mv***, ***rm***:

  `-f`

  `--force`

  Принудительно: принудительная перезапись целевого файла(ов).

Многие утилиты UNIX и Linux отклоняются от этого «стандарта», поэтому опасно полагать, что данная опция будет вести себя стандартным способом. При сомнении всегда сверяйтесь со справочной страницей команды.

Полная таблица рекомендуемых параметров для утилит GNU доступна на странице стандартов GNU (GNU standards page).

G.2. Опции командной строки Bash

Bash имеет свои опции командной строки. Здесь некоторые из наиболее употребляемых.

- **`-c`**

  Чтение команд из следующей за ней строки и назначение любых аргументов ***позиционным параметрам***.

```
bash$ bash -c 'set a b c d; IFS="+-;"; echo "$*"'
a+b+c+d
```

- **`-r`**

  `--restricted`

  *Запуск оболочки или сценария в **ограниченном** (restricted) **режиме**.*

- **`--posix`**

  *Принуждает Баш сообразовываться с режимом **POSIX**.*

- **`--version`**

  Выводит на экран данные о версии Bash и выход.

- **`--`**

  Окончание опций. Больше ничего в командной строке не является аргументом и не опцией.

Приложение H. Важные файлы

файлы загрузки

Эти файлы содержат алиасы и ***переменные среды*** делающие доступным запущенный Bash, как пользовательскую оболочку, и для всех сценариев Bash, вызванных после инициализации системы.

`/etc/profile`

по умолчанию системная, главным образом устанавливает среды (все оболочки типа Bourne, не только Bash [1])

`/etc/bashrc`

системные функции и алиасы Bash

`$HOME/.bash_profile`

указанные пользователем настройки окружения Bash, в домашней директории каждого пользователя (локальный дубликат `/etc/profile`)

`$HOME/.bashrc`

указанный пользователем файл init Bash в домашней директории каждого пользователя (локальный дубликат `/etc/bashrc`). Этот файл читают только интерактивные оболочки и пользовательские сценарии. См. Приложение M, образец файла `.bashrc`.

logout файла

`$HOME/.bash_logout`

указанная пользователем инструкция файла, находится в домашней директории каждого пользователя. После выхода из оболочки (Bash) выполняет команды, содержащиеся в этом файле.

Файлы данных

`/etc/passwd`

Список всех учетных записей пользователей в системе, их идентификация, их домашние директории, группы, к которым они принадлежат, и их оболочки по умолчанию. Обратите внимание, что пароли пользователей хранятся не в этом файле, [2], а в `/etc/shadow` в зашифрованном виде.

файлы конфигурации системы

`/etc/sysconfig/hwconf`

Список и описание аппаратных устройств. Эта информация находится в текстовой форме и может быть извлечена и изучена.

```
bash$ grep -A 5 AUDIO /etc/sysconfig/hwconf
class: AUDIO
 bus: PCI
 detached: 0
 driver: snd-intel8x0
 desc: "Intel Corporation 82801CA/CAM AC'97 Audio Controller"
 vendorId: 8086
```

Этот файл присутствует в Red Hat и Fedora Core installations, но может отсутствовать в других дистрибутивах.

Примечания

[1] Не применимо к **csh**, **tcsh**, и другим оболочкам не относящимся или не происходящим из классической оболочки Bourne (**sh**).

[2] В более старых версиях UNIX пароли хранятся в `/etc/passwd`, это объясняет и название файла.

Приложение I. Важнейшие системные директории

Системные администраторы и те, кто пишет административные сценарии должны быть хорошо знакомы со следующими системными директориями.

- `/bin`

  Бинарные (исполняемые двоичные) файлы. Основные системные программы и утилиты (такие как **bash**).

- `/usr/bin` [1]

  Дополнительные системные бинарные (двоичные) файлы.

- `/usr/local/bin`

  Различные локальные бинарные файлы для конкретной машины.

- `/sbin`

  Системные бинарные файлы. Основные управляющие системные программы и утилиты (такие как **fsck**).

- `/usr/sbin`

  Дополнительные управляющие системные программы и утилиты.

- `/etc`

  Et cetera. Общесистемные сценарии конфигурации.

  Особый интерес представляют файлы `/etc/fstab` (таблица файловой системы), `/etc/mtab` (таблица примонтированных файловых систем) и `/etc/inittab`.

- `/etc/rc.d`

  Сценарии загрузки, в Red Hat и производных дистрибутивах Linux.

- `/usr/share/doc`

  Документация установленных пакетов.

- `/usr/man`

  Общесистемные справочные страницы

- `/dev`

  Директория устройств. Записи (но не точек монтирования) для физических и виртуальных устройств. См. Глава 29.

- `/proc`

Директория процессов. Содержит информацию и статистические данные о запущенных процессах и параметрах ядра. См. Глава 29.

- `/sys`

  Общесистемная директория устройств. Содержит информацию и статистические данные об устройствах и именах устройств. Недавно добавлено к ядру 2.6.X Linux.

- `/mnt`

  Монтирование. Директория для монтирования разделов жесткого диска, таких как `/mnt/dos` и физических устройств. В новых дистрибутивах Linux директория `/media` взяла на себя монтирование устройств ввода-вывода.

- `/media`

  В новых дистрибутивах Linux монтирование для устройств ввода/вывода, таких как CD/DVD-диски или USB флэш-накопители.

- `/var`

  Переменные (изменяющиеся) системные файлы. Это всеохватывающая директория «Блокнот» для данных, получаемых на работающем компьютере Linux/UNIX.

- `/var/log`

  Общесистемные log файлы.

- `/var/spool/mail`

  Почта пользователя.

- `/lib`

  Общесистемные файлы библиотек

- `/usr/lib`

  Дополнительные общесистемные файлы библиотек

- `/tmp`

  Системные временные файлы

- `/boot`

  Загрузочная системная директория. Здесь находятся ядро, модули связи, системная карта и менеджер загрузки.

Изменение файлов в этой директории, может привести к невозможности загрузки системы.

Примечания

[1] Некоторые ранние системы UNIX имели быстрый диск малого объема (содержащий /, корневой раздел), и второй диск, который был больше, но медленнее (содержащий /usr и другие разделы). Наиболее часто используемые программы и утилиты находились на небольшом, но быстром диске в `/bin`, а остальные на медленном диске в `/usr/bin`.

Это также объясняет разделение между `/sbin` и `/usr/sbin`, `/lib` и `/usr/lib`, и т.д..

Приложение J. Введение в Программное Завершение

Программное завершение в Bash позволяет вводить с клавиатуры части команды, а затем нажав на клавишу **[Tab]** автоматически завершать последовательности команды. [1] Если возможны несколько завершений, то **[Tab]** перечисляет их все. Давайте посмотрим, как она работает.

```
bash$ xtra[Tab]
xtraceroute       xtrapin           xtrapproto
 xtraceroute.real  xtrapinfo         xtrapreset
 xtrapchar         xtrapout          xtrapstats


bash$ xtrac[Tab]
xtraceroute       xtraceroute.real


bash$ xtraceroute.r[Tab]
xtraceroute.real
```

Завершение клавишей **Tab** также работает для переменных и путей имен.

```
bash$ echo $BASH[Tab]
$BASH                 $BASH_COMPLETION      $BASH_SUBSHELL
 $BASH_ARGC            $BASH_COMPLETION_DIR  $BASH_VERSINFO
 $BASH_ARGV            $BASH_LINENO          $BASH_VERSION
 $BASH_COMMAND         $BASH_SOURCE


bash$ echo /usr/local/[Tab]
bin/     etc/     include/ libexec/ sbin/    src/
 doc/     games/   lib/     man/     share/
```

Встроенные в Bash ***complete*** и ***compgen*** делают возможным завершение клавишей **tab** частичное распознавание параметров и опций команды. В самом простом случае мы можем использовать ***complete*** из командной строки для указания краткого списка допустимых параметров.

```
bash$ touch sample_command
bash$ touch file1.txt file2.txt file2.doc file30.txt file4.zzz
bash$ chmod +x sample_command
bash$ complete -f -X '!*.txt' sample_command
```

```
bash$ ./sample[Tab][Tab]
sample_command
file1.txt   file2.txt   file30.txt
```

Опция `-f` выводит *завершения* указанных имен файлов, а `-X` шаблон фильтра.

Для чего-либо более комплексного, мы могли бы написать сценарий, который определяет список допустимых параметров командной строки. Встроенная ***compgen*** расширяет список аргументов для *создания* и *завершения* соответствий.

В качестве примера команды давайте возьмем модифицированную версию сценария *UseGetOpt.sh*. Этот сценарий принимает ряд параметров командной строки с предшествующими им одной или двумя тире. Это соответствующий сценарий завершения, *связывающего* данное имя файла с соответствующей командой.

Пример J-1. Завершающий сценарий для *UseGetOpt.sh*

```
# file: UseGetOpt-2
# UseGetOpt-2.sh завершение параметров

_UseGetOpt-2 ()    #  По соглашению, имя функции
{                  #+ начинается с подчеркивания.
  local cur
  # Указатель на текущее завершение слова.
  # По соглашению, он именуется "cur", но это не строго обходимо.

  COMPREPLY=()   # Переменная массива хранящяя возможные завершения.
  cur=${COMP_WORDS[COMP_CWORD]}

  case "$cur" in
    -*)
    COMPREPLY=( $( compgen -W '-a -d -f -l -t -h --aoption --debug \
                               --file --log --test --help --' -- $cur ) );;
#   Создает соответствующие завершения и загружает их в массив $COMPREPLY.
#   xx) можно еще добавить варианты.
#   yy)
#   zz)
  esac

  return 0
}

complete -F _UseGetOpt-2 -o filenames ./UseGetOpt-2.sh
#        ^^ ^^^^^^^^^^^^  Вызов функции _UseGetOpt-2.
```

Теперь давайте попробуем:

```
bash$ source UseGetOpt-2

bash$ ./UseGetOpt-2.sh -[Tab]
--         --aoption  --debug    --file     --help     --log     --test
 -a         -d         -f         -h         -l         -t
```

```
bash$ ./UseGetOpt-2.sh --[Tab]
--         --aoption  --debug    --file     --help     --log     --test
```

Мы начали с поиска «завершения сценария». Он задает параметры командной строки. [2] В первом случае, при нажатии **[Tab**], после *одного* тире, на выходе оказываются все возможные параметры, начинающиеся с *одного* или *нескольких* тире. Нажатие **[Tab]** после *двух* тире выдает возможные параметры, начинающиеся с *двух* или *более* тире.

Смыслом является только это действие, включить завершение командной строки tab. Оно сокращает нажатия клавиш. [3]

--

Примечания

[1] Работает только из командной строки, а не в сценарии.

[2] Обычно параметры завершения файлов по умолчанию находятся в директории `/etc/profile.d` или в `/etc/bash_completion`. Они автоматически загружаются при запуске системы. Так, после успешного написания сценария, вы, возможно, захотите переместить их (как root, конечно) в одну из этих директорий.

[3] Хорошо документировано, программисты готовы на многочасовые усилия для того, чтобы спасти десять минут «ненужного» труда. Это известно как *оптимизация*.

Приложение K. Локализация

Локализация является не документированной особенностью Bash.

Локализованный сценарий оболочки выводит текст на языке, определенном в локали системы. Пользователь Linux в Берлине, Германия, хотел бы получить вывод сценария на немецком, в то время как его двоюродный брат в Берлине, штат Мэриленд, получить вывод того же сценария на английском языке.

При создании локализованного сценария используйте следующий шаблон, выводящий все сообщения на языке пользователя (сообщения об ошибках, подсказки и т.д.).

```
#!/bin/bash
# localized.sh
#  Сценарий Stéphane Chazelas,
#+ изменения Bruno Haible, исправление ошибок Alfredo Pironti.

.gettext.sh

E_CDERROR=65

error()
{
  printf "$@" >&2
  exit $E_CDERROR
```

```
}

cd $var || error "`eval_gettext \"Can\'t cd to \\\$var.\"`"
#  Три обратных слэша (экраны) перед $var необходимы
#+ "потому что eval_gettext ожидает строку
#+ в которой значения переменных еще не подставлены."
#    -- от Bruno Haible
read -p "`gettext \"Введите значение: \"`" var
#  ...


#  ----------------------------------------------------------------
#  Alfredo Pironti комментирует:

#  Этот сценарий был изменен в пользу синтаксиса "`gettext \"...\"`",
#+ чтобы не использовать синтаксис $"...".
#  Это нормально, но с новой программой localized.sh, команды
#+ 'bash -D имя файла' и 'bash ---dump-po-string имяфайла'
#+ не будут ничего выводить (потому что эти команды
#+ ищут только строки для $' .. . )!
#  ЕДИНСТВЕННЫМ способом, для извлечения строк из нового файла, является
#+ использование программы 'xgettext'. Тем не менее, программа xgettext глючит.

# Обратите внимание, что 'xgettext' имеет и другие глюки.
#
# Фрагмент оболочки:
#    gettext -s "I like Bash"
# будет правильно извлечено, но   . . .
#    xgettext -s "I like Bash"
# ... не удача!
#  'xgettext' будет извлекать "-s", потому что
#+ команда извлекает только первый аргумент после слова «gettext».


#  Символы экранирования:
#
#  Для локализации предложения как
#     echo -e "Hello\tworld!"
#+ вы должны использовать
#     echo -e "`gettext \"Hello\\tworld\"`"
#  "Два символа экранирования" нужны перед `t', потому что
#+ 'gettext' будет искать строку типа: 'Hello\tworld'
#  Потому, что gettext будет читать буквально `\')
#+ и выведет строку типа "Bonjour\tmonde",
#+ т.к. команда 'echo' выведет сообщение на экран правильно.
#
#  Можно не использовать
#     echo "`gettext -e \"Hello\tworld\"`"
#+из-за ошибки xgettext, объясненной выше.



# Давайте локализуем следующий фрагмент оболочки:
#     echo "-h display help and exit"
#
# Сначала сделаем это:
#     echo "`gettext \"-h display help and exit\"`"
#  Здесь 'xgettext' будет работать отлично,
#+ но программа 'gettext' прочитает "-h" как опцию!
```

```
#
# Одним из решений может быть
#     echo "`gettext -- \"-h display help and exit\"`"
#  Здесь 'gettext' будет работать,
#+ но 'xgettext' будет извлекать "--", как указано выше.
#
# Обойти можно используя локализованную строку
#     echo -e "`gettext \"\\0-h display help and exit\"`"
#  Мы добавили \0 (NULL) в начало предложения.
#  Здесь 'gettext' работает правильно, как и 'xgettext.'
#  Кроме того, символ NULL не изменяет поведение команды 'echo'.
#  --------------------------------------------------------------
```

```
bash$ bash -D localized.sh
"Не возможно cd к %s."
 "Введите значение: "
```

В этом листинге указан весь локализованный текст. (Опция `-D` перечисляет строки в двойных кавычках с префиксом `$,` без выполнения сценария.)

```
bash$ bash --dump-po-strings localized.sh
#: a:6
 msgid "Can't cd to %s."
 msgstr ""
 #: a:7
 msgid "Enter the value: "
 msgstr ""
```

Опция `--dump-po-strings` в Bash напоминает опцию `-D`, но использует формат «*po*» **gettext**.

Bruno Haible поясняет:

Начиная с gettext-0.12.2, **xgettext -o - localized.sh** рекомендуется вместо **bash --dump-po-strings localized.sh**, потому что **xgettext** . . .

1. не понимает команды ***gettext*** и ***eval_gettext*** (тогда как **bash --dump-po-strings** не понимает только свой устаревший синтаксис `$"..."`)

2. может извлекать комментарии, размещенные программистом, предназначенные для чтения транслятором.

Этот код оболочки в этом случае не специфичен больше для Баш; он работает так же, как в Bash 1.x , так и в других реализациях `/bin/sh`.

Теперь, создадим файл `language.po` для каждого языка, на который сценарий будет переводить, указав `msgstr`. Alfredo Pironti предлагает следующий пример:

fr.po:

```
#: a:6
msgid "Can't cd to $var."
msgstr "Impossible de se positionner dans le repertoire $var."
#: a:7
msgid "Enter the value: "
msgstr "Entrez la valeur : "

#  Строка выкладывается с именами переменных, а не с синтаксисом %s,
#+ аналогичным программам Си.
#+ Это очень полезная функция, если программист использует имена
#+ переменных, которые имеют смысл!
```

Затем запустите **msgfmt**.

```
msgfmt -o localized.sh.mo fr.po
```

Результирующий файл `localized.sh.mo` поместите в директорию `/usr/local/share/locale/fr/LC_MESSAGES`, а в начале сценария вставьте строки:

```
TEXTDOMAINDIR=/usr/local/share/locale
TEXTDOMAIN=localized.sh
```

Если пользователь на французской системе запустит сценарий, он будет получать сообщения на французском языке.

В более старых версиях Bash или других оболочках, локализации требуется ***gettext***, с опцией `-s`. В этом случае сценарий становится:

```
#!/bin/bash
# localized.sh

E_CDERROR=65

error() {
  local format=$1
  shift
  printf "$(gettext -s "$format")" "$@" >&2
  exit $E_CDERROR
}
cd $var || error "Can't cd to %s." "$var"
read -p "$(gettext -s "Enter the value: ")" var
# ...
```

Переменные `TEXTDOMAIN` и `TEXTDOMAINDIR` должны быть установлены и экспортированы в окружающую среду. Это должно быть сделано в рамках самого сценария.

Этом приложение написано Stéphane Chazelas, с изменениями, предложенными Alfredo Pironti и Bruno Haible, майнтейнера GNU **gettext**.

Приложение L. История команд

Оболочка Bash предоставляет средства командной строки для редактирования и управления *историей команд* пользователя. Это прежде всего удобство, средство сокращения нажатий клавиш.

История команд Bash:

1. **history**
2. **fc**

```
bash$ history
   1  mount /mnt/cdrom
    2  cd /mnt/cdrom
    3  ls
     ...
```

Внутренние переменные, связанные с историей команд Bash:

1. $HISTCMD
2. $HISTCONTROL
3. $HISTIGNORE
4. $HISTFILE
5. $HISTFILESIZE
6. $HISTSIZE
7. $HISTTIMEFORMAT (Bash, вер. 3.0 или позже)
8. !!
9. !$
10. !#
11. !N
12. !-N
13. !STRING
14. !?STRING?
15. ^STRING^string^

К сожалению, инструменты истории Bash не находят применения в сценариях.

```
#!/bin/bash
# history.sh
# (Пустые) попытки использовать команду «history» в сценарии.

history                     # Нет вывода.

var=$(history); echo "$var"  # $var is empty.

#  История команд, по умолчанию, в сценариях отключена.
#  Однако, как отмечает dhw,
#+ set -o history
#+ включает механизм истории.

set -o history
var=$(history); echo "$var"   # 1  var=$(history)
```

```
bash$ ./history.sh
(нет вывода)
```

Сайт Advancing in the Bash Shell дает хорошую интродукцию использования истории команд

Bash.

Приложение M. Образцы файлов `.bashrc` и `.bash_profile`

В файл `~/.bashrc` определяет поведение интерактивных оболочек. Внимательное изучение этого файла может привести к более глубокому пониманию Bash.

Emmanuel Rouat предоставил свой, очень сложный, файл `.bashrc`, написанный для операционной системы Linux. Он приветствует замечания читателей о нем.

Тщательно изучите файл и не стесняйтесь использовать фрагменты кода и функций из него в своем собственном файле `.bashrc` или даже в ваших сценариях.

Пример M-1. Образец файла `.bashrc`

```
# ============================================================== #
#
# СОБСТВЕННЫЙ ФАЙЛ $HOME/.bashrc для bash-3.0 (или поздних)
# Emmanuel Rouat [no-email]
#
# Последнее изменение: Tue Nov 20 22:04:47 CET 2012
```

```
#  Этот файл нормально читается только интерактивной оболочкой.
#+ Здесь место для определения ваших алиасов, функций и других
#+ интерактивных действий, таких как запрос к вводу.
#
#  Большая часть кода предполагает, что вы находитесь на
#+ системе GNU (подобной Linux box) и часто основываетесь
#+ на коде, найденном в Usenet или Интернет.
#
#  См, например:
#  http://tldp.org/LDP/abs/html/index.html
#  http://www.caliban.org/bash
#  http://www.shelldorado.com/scripts/categories.html
#  http://www.dotfiles.org
#
#  Выбор цвета был сделан для оболочки с темным фоном
#+ (белым на черном фоне), а это, как правило, также подходит
#+ для чистой текстовой консоли (когда X сервер не доступен).
#+ Если вы используете белый фон, то вам придется сделать
#+ некоторые изменения для чтения
#
#  Этот файл bashrc немного переполнен.
#  Помните, что это всего лишь пример.
#  Адаптируйте его к вашим потребностям.
#
# ================================================================ #

# --> Комментарии добавленные автором HOWTO.

# Если не запущено интерактивно, то ничего не произойдет
[ -z "$PS1" ] && return


#-----------------------------------------------------------
# Источник глобальных определений (если таковые имеются)
#-----------------------------------------------------------


if [ -f /etc/bashrc ]; then
      . /etc/bashrc   # --> Чтение /etc/bashrc, если существует.
fi


#---------------------------------------------------------------
#  Автоматическая установка $DISPLAY (если еще не установлена).
#  Это работает у меня - у вас может быть иначе . . . .
#  Проблема заключается в том, что различные типы терминалов
#+ дают разные ответы на «who am i» (rxvt в частности может быть
#+ проблемой) - однако, как кажется, в большинстве случаев
#+ этот код работает.
#---------------------------------------------------------------

function get_xserver ()
{
    case $TERM in
        xterm )
            XSERVER=$(who am i | awk '{print $NF}' | tr -d ')''(' )
            # Ane-Pieter Wieringa предлагает следующую альтернативу:
            #  I_AM=$(who am i)
            #  SERVER=${I_AM#*(}
            #  SERVER=${SERVER%*)}
```

```
            XSERVER=${XSERVER%%:*}
            ;;
            aterm | rxvt)
            # Находит код, который здесь работает ...
            ;;
    esac
}

if [ -z ${DISPLAY:=""} ]; then
    get_xserver
    if [[ -z ${XSERVER}  || ${XSERVER} == $(hostname) ||
       ${XSERVER} == "unix" ]]; then
          DISPLAY=":0.0"          # Вывод на экран локального хоста.
    else
       DISPLAY=${XSERVER}:0.0     # Вывод на экран удаленного хоста.
    fi
fi

export DISPLAY

#--------------------------------------------------------------
# Некоторые настройки
#--------------------------------------------------------------

#set -o nounset     # Эти две опции используются для отладки.
#set -o xtrace
alias debug="set -o nounset; set -o xtrace"

ulimit -S -c 0      # Запрет на coredumps.
set -o notify
set -o noclobber
set -o ignoreeof


# Включенные опции:
shopt -s cdspell
shopt -s cdable_vars
shopt -s checkhash
shopt -s checkwinsize
shopt -s sourcepath
shopt -s no_empty_cmd_completion
shopt -s cmdhist
shopt -s histappend histreedit histverify
shopt -s extglob       # Необходимо для программируемого завершения.

# Отключенные опции:
shopt -u mailwarn
unset MAILCHECK        #  Не желаю, что бы моя оболочка предупреждала меня о
                       #+ входящей почте.


#--------------------------------------------------------------
# Приветствие, motd и т.д. ...
#--------------------------------------------------------------

# Определение цвета (взято из Color Bash Prompt HowTo).
# Некоторые цвета на других терминалах могут отличаться.
#  К примеру, я вижу 'четко-красный' на экране как 'оранжевый',
#  поэтому последовательности «зеленый» «четко-красный» «красный»,
#+ я часто использую в моей строке.
```

```
# Нормальные цвета
Black='\e[0;30m'        # Черный
Red='\e[0;31m'          # Красный
Green='\e[0;32m'        # Зеленый
Yellow='\e[0;33m'       # Желтый
Blue='\e[0;34m'         # Синий
Purple='\e[0;35m'       # Фиолетовый
Cyan='\e[0;36m'         # Голубой
White='\e[0;37m'        # Белый

# Четкие
BBlack='\e[1;30m'       # Черный
BRed='\e[1;31m'         # Красный
BGreen='\e[1;32m'       # Зеленый
BYellow='\e[1;33m'      # Желтый
BBlue='\e[1;34m'        # Синий
BPurple='\e[1;35m'      # Фиолетовый
BCyan='\e[1;36m'        # Голубой
BWhite='\e[1;37m'       # Белый

# Фон
On_Black='\e[40m'       # Черный
On_Red='\e[41m'         # Красный
On_Green='\e[42m'       # Зеленый
On_Yellow='\e[43m'      # Желтый
On_Blue='\e[44m'        # Синий
On_Purple='\e[45m'      # Фиолетовый
On_Cyan='\e[46m'        # Голубой
On_White='\e[47m'       # Белый

NC="\e[m"               # Сброс цвета


ALERT=${BWhite}${On_Red} # Четко-белый на красном фоне



echo -e "${BCyan}Это BASH ${BRed}${BASH_VERSION%.*}${BCyan}\
- DISPLAY на ${BRed}$DISPLAY${NC}\n"
date
if [ -x /usr/games/fortune ]; then
    /usr/games/fortune -s     # Делает наш день немного приятней.... :-)
fi

function _exit()              # Выполняемая функция после выхода из оболочки.
{
    echo -e "${Bred}Аста ла виста, бэби${NC}"
}
trap _exit EXIT

#-------------------------------------------------------------
# Shell Prompt — примеры, см.:
#       http://www.debian-administration.org/articles/205
#       http://www.askapache.com/linux/bash-power-prompt.html
#       http://tldp.org/HOWTO/Bash-Prompt-HOWTO
#       https://github.com/nojhan/liquidprompt
#-------------------------------------------------------------
# Текущий формат: [TIME USER@HOST PWD] >
# TIME:
#    Green     == Машина загружается медленно
```

```
#    Orange    == Машина загружается нормально
#    Red       == Машина загружается быстро
#    ALERT     == Машина загружается очень быстро
# USER:
#    Cyan      == обычный пользователь
#    Orange    == пользователь из-под SU
#    Red       == администратор
# HOST:
#    Cyan      == локальная сессия
#    Green     == безопасное удаленное соединение (через ssh)
#    Red       == не безопасное удаленное соединение
# PWD:
#    Green     == больше чем 10% свободного дискового пространства
#    Orange    == меньше чем 10% свободного дискового пространства
#    ALERT     == меньше чем 5% свободного дискового пространства
#    Red       == текущий пользователь не имеет прав на запись
#    Cyan      == текущая ФС нулевого размера (типа /proc)
# >:
#    White     == без фоновых или приостановленных заданий в этой оболочке
#    Cyan      == по крайней мере одно фоновое задание в этой оболочке
#    Orange    == по крайней мере одно приостановленное задание в этой оболочке
#
#    Команда добавляется в файл истории каждый раз, когда вы нажмете enter,
#    поэтому оно доступно для всех оболочек (используя «history -a»).


# Проверка типа соединения:
if [ -n "${SSH_CONNECTION}" ]; then
    CNX=${Green}        # Подключение на удаленной машине через ssh (хорошо).
elif [[ "${DISPLAY%%:0*}" != "" ]]; then
    CNX=${ALERT}        # Подключение на удаленной машине не через ssh (плохо).
else
    CNX=${BCyan}        # Подключение на локальной машине.
fi

# Проверка типа пользователя:
if [[ ${USER} == "root" ]]; then
    SU=${Red}           # Пользователем является root.
elif [[ ${USER} != $(logname) ]]; then
    SU=${BRed}          # Не является зарегистрированным пользователем.
else
    SU=${BCyan}         # Обычный пользователь (ну... большинство из нас).
fi



NCPU=$(grep -c 'processor' /proc/cpuinfo)    # Количество процессоров
SLOAD=$(( 100*${NCPU} ))        # Малая нагрузка if[ -n]thenelifthen[[]]!
=elsefiif[[==]]thenelif[[!=]]thenelsefi
MLOAD=$(( 200*${NCPU} ))        # Средняя нагрузка
XLOAD=$(( 400*${NCPU} ))        # Очень большая нагрузка

# Возвращает системную нагрузку в процентах, т.е., '40', вместо '0.40)'.
function load()
{
    local SYSLOAD=$(cut -d " " -f1 /proc/loadavg | tr -d '.')
    # Системная нагрузка на текущем хосте.
    echo $((10#$SYSLOAD))       # Конвертация в десятичные.
}

# Возвращает цвет, указывающий нагрузку системы.
```

```
программируемой доработки функций  function load_color()
{
    local SYSLOAD=$(load)
    if [ ${SYSLOAD} -gt ${XLOAD} ]; then
        echo -en ${ALERT}
    elif [ ${SYSLOAD} -gt ${MLOAD} ]; then
        echo -en ${Red}
    elif [ ${SYSLOAD} -gt ${SLOAD} ]; then
        echo -en ${BRed}
    else
        echo -en ${Green}
    fi
}

# Возвращает цвет в зависимости от свободного дискового пространства в $PWD.
function disk_color()
{
    if [ ! -w "${PWD}" ] ; then
        echo -en ${Red}
        # Нет прав 'записи' в текущую директорию.
    elif [ -s "${PWD}" ] ; then
        local used=$(command df -P "$PWD" |
                   awk 'END {print $5} {sub(/%/,"")}')
        if [ ${used} -gt 95 ]; then
            echo -en ${ALERT}    # Почти полный диск(>95%).
        elif [ ${used} -gt 90 ]; then
            echo -en ${BRed}     # Свободного дискового пространства почти нет.
        else
            echo -en ${Green}    # Свободного места на диске достаточно.
        fi
    else
        echo -en ${Cyan}
        # Размер текущей директории '0' (типа /proc, /sys и т.п.).
    fi
}

# Возвращает цвет запущенного/приостановленного задания.
function job_color()
{
    if [ $(jobs -s | wc -l) -gt "0" ]; then
        echo -en ${BRed}
    elif [ $(jobs -r | wc -l) -gt "0" ] ; then
        echo -en ${BCyan}
    fi
}

# Добавляет текст в окне терминала (если применимо).


# Теперь конструируем запрос.
PROMPT_COMMAND="history -a"
case ${TERM} in
  *term | rxvt | linux)
        PS1="\[\$(load_color)\][\A\[${NC}\] "
        # Время суток (с указанием нагрузки):
        PS1="\[\$(load_color)\][\A\[${NC}\] "
        # User@Host (с указанием типа соединения):
        PS1=${PS1}"\[${SU}\]\u\[${NC}\]@\[${CNX}\]\h\[${NC}\] "
        # PWD (с информацией о 'дисковом пространстве'):
        PS1=${PS1}"\[\$(disk_color)\]\W]\[${NC}\] "
        # Запрос (with 'job' info):
```

```
        PS1=${PS1}"\[\$(job_color)\]>\[${NC}\] "
        # Задаем название текущего xterm:
        PS1=${PS1}"\[\e]0;[\u@\h] \w\a\]"
        ;;
    *)
        PS1="(\A \u@\h \W) > " # --> PS1="(\A \u@\h \w) > "
                               # -->  Выводим полный путь к файлу текущей
                               #+ --> директории.
        ;;
esac



export TIMEFORMAT=$'\nreal %3R\tuser %3U\tsys %3S\tpcpu %P\n'
export HISTIGNORE="&:bg:fg:ll:h"
export HISTTIMEFORMAT="$(echo -e ${BCyan})[%d/%m %H:%M:%S]$(echo -e ${NC}) "
export HISTCONTROL=ignoredups
export HOSTFILE=$HOME/.hosts    # Помещает список удаленных хостов в ~/.hosts


#============================================================
#
#  АЛИАСЫ И ФУНКЦИИ
#
#  Возможно некоторые функции, определенные здесь, достаточно большие.
#  Если вы хотите сделать этот файл меньше, эти функции могут быть
#+ преобразованы в сценарии и удалены отсюда.
#
#============================================================

#-------------------
Личные алиасы
#-------------------

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
# -> Предотвращает случайное удаление файлов.
alias mkdir='mkdir -p'

alias h='history'
alias j='jobs -l'
alias which='type -a'
alias ..='cd ..'

# Довольно достаточный вывод некоторых PATH переменных:
alias path='echo -e ${PATH//:/\\n}'
alias libpath='echo -e ${LD_LIBRARY_PATH//:/\\n}'


alias du='du -kh'    # Делает вывод более читабельным.
alias df='df -kTh'

#-------------------------------------------------------------
# Семейство 'ls' (предполагается, что вы используете последнее GNU ls).
#-------------------------------------------------------------
# Добавляет цвета в 'ls' для типов файлов и восприятия размеров по умолчанию:
alias ls='ls -h --color'
alias lx='ls -lXB'         #  Сортировка по расширению.
alias lk='ls -lSr'         #  Сортировка по размеру, большие после.
alias lt='ls -ltr'         #  Сортировка по дате, последние внизу.
```

```
alias lc='ls -ltcr'        #  Сортировка по времени изменения,поздние внизу.
alias lu='ls -ltur'        #  Сортировка по времени обращения,позднее внизу.

# Вездесущие "ll": сначала директории, с буквенно-цифровой сортировкой:
alias ll="ls -lv --group-directories-first"
alias lm='ll |more'        #  Канал через 'more'
alias lr='ll -R'           #  Рекурсивное ls.
alias la='ll -A'           #  Показывать скрытые файлы.
alias tree='tree -Csuh'    #  Прекрасная альтернатива 'рекурсивного ls' ...


#-------------------------------------------------------------
# 'less'
#-------------------------------------------------------------

alias more='less'
export PAGER=less
export LESSCHARSET='latin1'
export LESSOPEN='|/usr/bin/lesspipe.sh %s 2>&-'
                # Применяется, если существует lesspipe.sh.
export LESS='-i -N -w  -z-4 -g -e -M -X -F -R -P%t?f%f \
:stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...'

# Цветные справочные страницы LESS (делает Man pages более читабельным).
export LESS_TERMCAP_mb=$'\E[01;31m'
export LESS_TERMCAP_md=$'\E[01;31m'
export LESS_TERMCAP_me=$'\E[0m'
export LESS_TERMCAP_se=$'\E[0m'
export LESS_TERMCAP_so=$'\E[01;44;33m'
export LESS_TERMCAP_ue=$'\E[0m'
export LESS_TERMCAP_us=$'\E[01;32m'


#-------------------------------------------------------------
# Ошибки правописания - очень личные и зависимые от клавиатуры :-)
#-------------------------------------------------------------

alias xs='cd'
alias vf='cd'
alias moer='more'
alias moew='more'
alias kk='ll'


#-------------------------------------------------------------
# Немного забавного
#-------------------------------------------------------------

# Добавляет текст в кадр терминала (если возможно).

function xtitle()
{
    case "$TERM" in function xtitle(){}
    *term* | rxvt)
        echo -en  "\e]0;$*\a" ;;
    *)  ;;
    esac
}


# Алиасы, использующие xtitle
```

```
alias top='xtitle Процессы $ HOST && top'
alias make='xtitle Making $(basename $PWD) ; make'

# .. и функции
function man()
{
    for i ; do
        xtitle The $(basename $1|tr -d .[:digit:]) manual
        command man -a "$i"
    done
}


#----------------------------------------------------------------
# Делает следующие команды автоматически работающими в фоновом режиме:
#----------------------------------------------------------------

function te()  # обертка вокруг xemacs/gnuserv
{
    if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
       gnuclient -q "$@";
    else
       ( xemacs "$@" &);
    fi
}

function soffice() { command soffice "$@" & }
function firefox() { command firefox "$@" & }
function xpdf() { command xpdf "$@" & }


#----------------------------------------------------------------
# Файлы и строки связанные функциями:
#----------------------------------------------------------------


# Поиск файла по шаблону имени:
function ff() { find . -type f -iname '*'"$*"'*' -ls ; }

# Поиск файла с шаблоном $1 в имени и выполнение $2 в нем:
function fe() { find . -type f -iname '*'"${1:-}"'*' \
-exec ${2:-file} {} \;  ; }

#  Поиск по шаблону в наборе файлов и выделение их:
#+ (нужна последняя версия egrep).
function fstr()
{
    OPTIND=1
    local mycase=""
    local usage="fstr: поиск строки в файлах.
Usage: fstr [-i] \"шаблон\" [\"шаблон имени файла\"] "
    while getopts :это opt
    do
        case "$opt" in
           i) mycase="-i " ;;
           *) echo "$usage"; return ;;
        esac}
    done
    shift $(( $OPTIND - 1 ))
    if [ "$#" -lt 1 ]; then
        echo "$usage"
```

```
        return;
    fi
    find . -type f -name "${2:-*}" -print0 | \
xargs -0 egrep --color=always -sn ${case} "$1" 2>&- | more

}


function swap()
{ #  2 файла Swap, если они существуют (из Uzi в bashrc).
    local TMPFILE=tmp.$$

    [ $# -ne 2 ] && echo "swap: необходимы 2 аргумента" && return 1
    [ ! -e $1 ] && echo "swap: $1 не существует" && return 1
    [ ! -e $2 ] && echo "swap: $2 не существует" && return 1

    mv "$1" $TMPFILE
    mv "$2" "$1"
    mv $TMPFILE "$2"
}

function extract()      # Удобная программа извлечения
{
    if [ -f $1 ] ; then
        case $1 in
            *.tar.bz2)   tar xvjf $1     ;;
            *.tar.gz)    tar xvzf $1     ;;
            *.bz2)       bunzip2 $1      ;;
            *.rar)       unrar x $1      ;;
            *.gz)        gunzip $1       ;;
            *.tar)       tar xvf $1      ;;
            *.tbz2)      tar xvjf $1     ;;
            *.tgz)       tar xvzf $1     ;;
            *.zip)       unzip $1        ;;
            *.Z)         uncompress $1   ;;
            *.7z)        7z x $1         ;;
            *)           echo "'$1' не может быть извлечен >extract<" ;;
        esac
    else
        echo "'$1' это не правильный файл!"
    fi
}


# Создание архива (*.tar.gz) из заданной директории.
function maketar() { tar cvzf "${1%%/}.tar.gz"  "${1%%/}/"; }

# Создание ZIP архива файла или папки.
function makezip() { zip -r "${1%%/}.zip" "$1" ; }

# Создание прав доступа вашей директории и файлов.
function sanitize() { chmod -R u=rwX,g=rX,o= "$@" ;}

#--------------------------------------------------------------
# Функции связанные с процессом/системой :
#--------------------------------------------------------------


function my_ps() { ps $@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }
function pp() { my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }
```

```
function killps()   # убивает по имени процесса
{
    local pid pname sig="-TERM"   # сигнал по умолчанию
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
        echo "Usage: killps [-SIGNAL] шаблон"
        return;
    fi
    if [ $# = 2 ]; then sig=$1 ; fi
    for pid in $(my_ps| awk '!/awk/ && $0~pat { print $1 }' pat=${!#} )
    do
        pname=$(my_ps | awk '$1~var { print $5 }' var=$pid )
        if ask "Убить процесс $pid <$pname> сигналом $sig?"
            then kill $sig $pid
        fi
    done
}

function mydf()         # Вывод 'df'.
{                       # Вдохновлено утилитой 'dfc'.
    for fs ; do

        if [ ! -d $fs ]
        then
          echo -e $fs" :Нет такого файла или директории" ; continue
        fi

        local info=( $(command df -P $fs | awk 'END{ print $2,$3,$5 }') )
        local free=( $(command df -Pkh $fs | awk 'END{ print $4 }') )
        local nbstars=$(( 20 * ${info[1]} / ${info[0]} ))
        local out="["
        for ((j=0;j<20;j++)); do
            if [ ${j} -lt ${nbstars} ]; then
               out=$out"*"
            else
               out=$out"-"
            fi
        done
        out=${info[2]}" "$out"] ("$free" свободно на "$fs")"
        echo -e $out
    done
}


function my_ip() # Получение IP адреса ethernet.
{
    MY_IP=$(/sbin/ifconfig eth0 | awk '/inet/ { print $2 } ' |
      sed -e s/addr://)
    echo ${MY_IP:-"Не соединено"}
}

function ii()   # Получение информации связанной с текущим хостом.
{
    echo -e "\nВы вошли на ${BRed}$HOST"
    echo -e "\n${BRed}Добавочная информация:$NC " ; uname -a
    echo -e "\n${BRed}Пользователи вошедшие на:$NC " ; w -hs |
             cut -d " " -f1 | sort | uniq
    echo -e "\n${BRed}Текущая дата :$NC " ; date
    echo -e "\n${BRed}Статус машины :$NC " ; uptime
    echo -e "\n${BRed}Статус памяти :$NC " ; free
    echo -e "\n${BRed}Дисковое пространство :$NC " ; mydf / $HOME
```

```
    echo -e "\n${BRed}Локальный IP Address :$NC" ; my_ip
    echo -e "\n${BRed}Открытые соединения :$NC "; netstat -pan --inet;
    echo
}

#-------------------------------------------------------------
# Различные утилиты:
#-------------------------------------------------------------

function repeat()       # Повтор команды n раз.
{
    local i max
    max=$1; shift;
    for ((i=1; i <= max ; i++)); do  # --> Как синтаксис Си
        eval "$@";
    done
}


function ask()          # Пример применения см. 'killps'.
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}

function corename()   # Получение имени приложения, создавшего corefile.
{
    for file ; do
        echo -n $file : ; gdb --core=$file --batch | head -1
    done
}



#===============================================================================
#
#  ПРОГРАММИРУЕМАЯ ДОРАБОТКА РАЗДЕЛА
#  Большинство из них взято из документации bash 2,05 и из пакета Ian McDonald
#  'Bash completion' (HTTP //www.caliban.org/bash/#completion). На самом деле,
#  для некоторых функций, Вам нужен bash не младше 3.0.
#
#  Обратите внимание, что большинство дистрибутивов linux теперь предоставляют
#  множество доработок 'из коробки' - однако, вам однажды, может потребоваться
#  сделать их самому, поэтому я поместил их сюда в качестве примеров.
#===============================================================================

if [ "${BASH_VERSION%.*}" \< "3.0" ]; then
    echo "Вам необходимо обновиться до версии 3.0 для полной \
          программируемой доработки функций"
    return
fi

shopt -s extglob        # Необходимо.

complete -A hostname   rsh rcp telnet rlogin ftp ping disk
complete -A export     printenv
complete -A variable   export local readonly unset
complete -A enabled    builtin
```

```
complete -A alias      alias unalias
complete -A function   function
complete -A user       su mail finger

complete -A helptopic  help     # В настоящее время, как встроенные команды.
complete -A shopt      shopt
complete -A stopped -P '%' bg
complete -A job -P '%'     fg jobs disown

complete -A directory  mkdir rmdir
complete -A directory   -o default cd

# Сжатие
complete -f -o default -X '*.+(zip|ZIP)'  zip
complete -f -o default -X '!*.+(zip|ZIP)' unzip
complete -f -o default -X '*.+(z|Z)'      compress
complete -f -o default -X '!*.+(z|Z)'     uncompress
complete -f -o default -X '*.+(gz|GZ)'    gzip
complete -f -o default -X '!*.+(gz|GZ)'   gunzip
complete -f -o default -X '*.+(bz2|BZ2)'  bzip2
complete -f -o default -X '!*.+(bz2|BZ2)' bunzip2
complete -f -o default -X '!*.+(zip|ZIP|z|Z|gz|GZ|bz2|BZ2)' extract


# Документы - Postscript,pdf,dvi.....
complete -f -o default -X '!*.+(ps|PS)'  gs ghostview ps2pdf ps2ascii
complete -f -o default -X \
'!*.+(dvi|DVI)' dvips dvipdf xdvi dviselect dvitype
complete -f -o default -X '!*.+(pdf|PDF)' acroread pdf2ps
complete -f -o default -X '!*.@(@(?(e)ps|?(E)PS|pdf|PDF)?\
(.gz|.GZ|.bz2|.BZ2|.Z))' gv ggv
complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
complete -f -o default -X '!*.tex' tex latex slitex
complete -f -o default -X '!*.lyx' lyx
complete -f -o default -X '!*.+(htm*|HTM*)' lynx html2ps
complete -f -o default -X \
'!*.+(doc|DOC|xls|XLS|ppt|PPT|sx?|SX?|csv|CSV|od?|OD?|ott|OTT)' soffice

# Мультимедиа
complete -f -o default -X \
'!*.+(gif|GIF|jp*g|JP*G|bmp|BMP|xpm|XPM|png|PNG)' xv gimp ee gqview
complete -f -o default -X '!*.+(mp3|MP3)' mpg123 mpg321
complete -f -o default -X '!*.+(ogg|OGG)' ogg123
complete -f -o default -X \
'!*.@(mp[23]|MP[23]|ogg|OGG|wav|WAV|pls|\
m3u|xm|mod|s[3t]m|it|mtm|ult|flac)' xmms
complete -f -o default -X '!*.@(mp?(e)g|MP?(E)G|wma|avi|AVI|\
asf|vob|VOB|bin|dat|vcd|ps|pes|fli|viv|rm|ram|yuv|mov|MOV|qt|\
QT|wmv|mp3|MP3|ogg|OGG|ogm|OGM|mp4|MP4|wav|WAV|asx|ASX)' xine



complete -f -o default -X '!*.pl'  perl perl5


#  Это «универсальное» дополнение функции - оно работает, когда команды имеют
#+ режим так называемых "длинных опций", т.е.: 'ls --all' вместо 'ls -a'
#  Для grep необходима опция '-o'
#+ (попробуйте закомментированную версию, если не доступна).

#  Сначала, удалите '=' пробел разделяющий слова
```

```
#+ (Это будет выглядеть как 'ls --color=auto' для правильной работы).

COMP_WORDBREAKS=${COMP_WORDBREAKS/=/}


_get_longopts()
{
  #$1 --help | sed  -e '/--/!d' -e 's/.*--\([^[:space:].,]*\).*/--\1/'| \
  #grep ^"$2" |sort -u ;
    $1 --help | grep -o -e "--[^[:space:].,]*" | grep -e "$2" |sort -u
}

_longopts()
{
    local cur
    cur=${COMP_WORDS[COMP_CWORD]}

    case "${cur:-*}" in
       -*)      ;;
        *)      return ;;
    esac

    case "$1" in
       \~*)     eval cmd="$1" ;;
         *)     cmd="$1" ;;
    esac
    COMPREPLY=( $(_get_longopts ${1} ${cur} ) )
}
complete  -o default -F _longopts configure bash
complete  -o default -F _longopts wget id info a2ps ls recode

_tar()
{
    local cur ext regex tar untar

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    # Если нам нужна опция, возможно возвращение длинных опций.
    case "$cur" in
        -*)     COMPREPLY=( $(_get_longopts $1 $cur ) ); return 0;;
    esac

    if [ $COMP_CWORD -eq 1 ]; then
        COMPREPLY=( $( compgen -W 'c t x u r d A' -- $cur ) )
        return 0
    fi

    case "${COMP_WORDS[1]}" in
        ?(-)c*f)
            COMPREPLY=( $( compgen -f $cur ) )
            return 0
            ;;
        +([^Izjy])f)
            ext='tar'
            regex=$ext
            ;;
        *z*f)
            ext='tar.gz'
            regex='t\(ar\.\)\(gz\|Z\)'
            ;;
```

```
        *[Ijy]*f)
            ext='t?(ar.)bz?(2)'
            regex='t\(ar\.\)bz2\?'
            ;;
        *)
            COMPREPLY=( $( compgen -f $cur ) )
            return 0
            ;;

    esac

    if [[ "$COMP_LINE" == tar*.$ext' '* ]]; then
        # Объединяем файлы в tar файл.
        #
        # Получаем имя tar файла из командной строки.
        tar=$( echo "$COMP_LINE" | \
                        sed -e 's|^.* \([^ ]*'$regex'\) .*$|\1|' )
        # Подумайте, как распаковать и просмотреть его содержимое.
        untar=t${COMP_WORDS[1]//[^Izjyf]/}

        COMPREPLY=( $( compgen -W "$( echo $( tar $untar $tar \
                                2>/dev/null ) )" -- "$cur" ) )
        return 0

    else
        # Файл содержит соответствующие файлы.
        COMPREPLY=( $( compgen -G $cur\*.$ext ) )

    fi

    return 0

}

complete -F _tar -o default tar

_make()
{
    local mdef makef makef_dir="." makef_inc gcmd cur prev i;
    COMPREPLY=();
    cur=${COMP_WORDS[COMP_CWORD]};
    prev=${COMP_WORDS[COMP_CWORD-1]};
    case "$prev" in
        -*f)
            COMPREPLY=($(compgen -f $cur ));
            return 0
            ;;
    esac;
    case "$cur" in
        -*)
            COMPREPLY=($(_get_longopts $1 $cur ));
            return 0
            ;;
    esac;

    # ... make reads
    #          GNUmakefile,
    #     then makefile
    #     then Makefile ...
    if [ -f ${makef_dir}/GNUmakefile ]; then
        makef=${makef_dir}/GNUmakefile
```

```
    elif [ -f ${makef_dir}/makefile ]; then
        makef=${makef_dir}/makefile
    elif [ -f ${makef_dir}/Makefile ]; then
        makef=${makef_dir}/Makefile
    else
       makef=${makef_dir}/*.mk         # Локальное соглашение.
    fi


    #  Прежде чем мы сканировать цели, посмотрите, как было указано
    #+ имя файла Makefile с -f.
    for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
        if [[ ${COMP_WORDS[i]} == -f ]]; then
            # eval для расширения тильды
            eval makef=${COMP_WORDS[i+1]}
            break
        fi
    done
    [ ! -f $makef ] && return 0

    # Сделка с содержащимися Makefiles.
    makef_inc=$( grep -E '^-?include' $makef |
                 sed -e "s,^.* ,"$makef_dir"/," )
    for file in $makef_inc; do
        [ -f $file ] && makef="$makef $file"
    done


    #  Если у нас есть частичное слово для дополнения, ограничьте
    #+ дополнение в соответствиях этого слова.
    if [ -n "$cur" ]; then gcmd='grep "^$cur"' ; else gcmd=cat ; fi

    COMPREPLY=( $( awk -F':' '/^[a-zA-Z0-9][^$#\/\t=]*:([^=]|$)/ \
                               {split($1,A,/ /);for(i in A)print A[i]}' \
                                $makef 2>/dev/null | eval $gcmd  ))

}

complete -F _make -X '+($*|*.[cho])' make gmake pmake




_killall()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    #  Получаем список процессов
    #+ (сначала оценка sed
    #+ заботится о своппинге процессов, потом
    #+ заботится о получении basename процесса).
    COMPREPLY=( $( ps -u $USER -o comm  | \
        sed -e '1,1d' -e 's#[]\[]##g' -e 's#^.*/##'| \
        awk '{if ($0 ~ /^'$cur'/) print $0}' ))

    return 0
}

complete -F _killall killall killps
```

```
# Локальные переменные:
# Режим:shell-script
# sh-shell:bash
# Конец:
```

А вот поучительный фрагмент из файла .bash_profile Andrzej Szelachowski.

Пример M-2. `.bash_profile` file

```
# Из Andrzej Szelachowski ~/.bash_profile:
#  Обратите внимание, что переменной может потребоваться специальная обработка
#+ если она будет экспортироваться.

DARKGRAY='\e[1;30m'
LIGHTRED='\e[1;31m'
GREEN='\e[32m'
YELLOW='\e[1;33m'
LIGHTBLUE='\e[1;34m'
NC='\e[m'

PCT="\`if [[ \$EUID -eq 0 ]]; then T='$LIGHTRED' ; else T='$LIGHTBLUE'; fi;
echo \$T \`"
#  Для "буквальной" подстановки команд переменная должна быть присвоена,
#+ с использованием экранирования и двойных кавычек:
#+       PCT="\` ... \`" . . .
#  В противном случае, значение переменной PCT присваивается только один раз,
#+ когда переменная экспортируется/читается из .bash_profile, и она не будет
#+ меняться  даже после изменении ID пользователя.

PS1="\n$GREEN[\w] \n$DARKGRAY($PCT\t$DARKGRAY)-($PCT\u$DARKGRAY)-($PCT\!
$DARKGRAY)$YELLOW-> $NC"

#  Экранируйте переменные, значение которых изменяется:
#       if [[ \$EUID -eq 0 ]],
#  В противном случае значение переменной EUID будет присвоено только один раз,
#+ как выше.

#  Когда переменная присвоена, она должна вызываться заэкранированной:
#+       echo \$T,
#  В противном случае значение переменной T берется с момента, когда переменная
#+ PCT экспортирована/прочтена из .bash_profile.
#  Поэтому, в этом случае, она была бы null.

#  Когда значение переменной содержит точку с запятой необходимы жесткие
#  кавычки:
#       T='$LIGHTRED',
#  В противном случае, точка с запятой будет интерпретироваться как разделитель
#  команд.

#  Переменные PCT и PS1 могут быть объединены в новую переменную PS1:

PS1="\`if [[ \$EUID -eq 0 ]]; then PCT='$LIGHTRED';
else PCT='$LIGHTBLUE'; fi;
echo '\n$GREEN[\w] \n$DARKGRAY('\$PCT'\t$DARKGRAY)-\
```

```
('\$PCT'\u$DARKGRAY)-('\$PCT'\!$DARKGRAY)$YELLOW-> $NC'\`"

# Хитрость заключается в том, чтобы использовать жесткие кавычки для частей

# старой переменной PS1.
```

Приложение N. Преобразование пакетных файлов DOS, в Shell Scripts

Достаточно большое количество программистов знают, что ПК работал под управлением DOS. Даже уродливый язык пакетных файлов DOS позволял писать некоторые довольно мощные сценарии и приложения, хотя они часто требовали костылей и обходных путей. Иногда по-прежнему возникает необходимость преобразовать старые пакетные файлы DOS в сценариях оболочки UNIX. Это, как правило, не сложно, так как операторы пакетных файлов DOS являются только ограниченным подмножеством эквивалентным оболочки сценариев.

Таблица N-1. Ключевые слова/переменные/операторы пакетных файлов и их эквиваленты оболочки

Оператор пакетных файлов	Эквивалент Shell Script	Значение
%	$	Префикс параметра командной строки
/	-	Флаг опции команды
\	/	Разделитель пути директории
==	=	(Равно) проверка сравнения строки
!==!	!=	(Не равно) проверка сравнения строки
|	|	Канал
@	set +v	Не выводить текущую команду
*	*	Имя файла "символ подстановки"
>	>	Перенаправление файла (перезапись)
>>	>>	Перенаправление файла (добавление)
<	<	Перенаправление stdin
%VAR%	$VAR	Переменная окружения
REM	#	Комментарий
NOT	!	Отрицание следующей проверки
NUL	/dev/null	"черная дыра" для сокрытия вывода команды
ECHO	echo	Вывод на экран (в Bash гораздо больше опций)
ECHO.	echo	Вывод пустой строки

Оператор пакетных файлов	Эквивалент Shell Script	Значение
`ECHO OFF`	`set +v`	Не выводит на экран следующую (ие) команду(ы)
`FOR %%VAR IN (LIST) DO`	for var in [list]; do	Цикл "for"
`:LABEL`	отсутствует (без необходимости)	метка
`GOTO`	отсутствует (используется функция)	Переход к другому месту в сценарии
`PAUSE`	`sleep`	Пауза или интервал ожидания
`CHOICE`	`case` или `select`	Меню выбора
`IF`	`if`	Проверка if
`IF EXIST` *`FILENAME`*	`if [ -e filename ]`	проверка if на наличие файла
`IF !%N==!`	`if [ -z "$N" ]`	Проверка на отсутствие заменяемого параметра "N"
`CALL`	`source` или `.` (оператор точка)	'Включение' другого сценария
`COMMAND /C`	`source` или `.` (оператор точка)	'Включение' другого сценария (то же, что и `CALL`)
`SET`	`export`	Присваивание переменной окружения
`SHIFT`	`shift`	сдвиг влево списка аргументов командной строки
`SGN`	`-lt` или `-gt`	знак (целого числа)
`ERRORLEVEL`	$?	Статус выхода
`CON`	`stdin`	"консоль" (`stdin`)
`PRN`	`/dev/lp0`	(Основное) устройство принтера
`LPT1`	`/dev/lp0`	Первое устройство принтера
`COM1`	`/dev/ttyS0`	Первый последовательный порт

Пакетные файлы обычно содержат команды DOS. Для преобразования пакетного файла в сценарий оболочки они должны быть заменены на их эквиваленты в UNIX.

Таблица N-2. Команды DOS и их эквиваленты в UNIX

Команда DOS	Эквивалент UNIX	Действие
`ASSIGN`	`ln`	Ссылка на файл или директорию
`ATTRIB`	`chmod`	Изменение прав доступа фала
`CD`	`cd`	Смена директории
`CHDIR`	`cd`	Смена директории
`CLS`	`clear`	Очистка экрана

Команда DOS	Эквивалент UNIX	Действие
COMP	diff, comm, cmp	Сравнение файла
COPY	cp	Копирование файла
Ctl-C	Ctl-C	Сброс (сигнала)
Ctl-Z	Ctl-D	EOF (окончание файла)
DEL	rm	Удаление файла(ов)
DELTREE	rm -rf	Рекурсивное удаление директории
DIR	ls -l	Листинг директории
ERASE	rm	Удаление файла(ов)
EXIT	exit	Завершение текущего процесса
FC	comm, cmp	Сравнение файла
FIND	grep	Поиск строк в файлах
MD	mkdir	Создание директории
MKDIR	mkdir	Создание директории
MORE	more	Постраничный фильтр текстового файла
MOVE	mv	Перемещение
PATH	$PATH	Путь для выполнения
REN	mv	Переименование (перемещение)
RENAME	mv	Переименование (перемещение)
RD	rmdir	Удаление директории
RMDIR	rmdir	Удаление директории
SORT	sort	Сортировка файла
TIME	date	Вывод системного времени
TYPE	cat	Вывод файла в stdout
XCOPY	cp	(расширенное) копирование файла

Практически все операторы и команды UNIX и оболочки имеют гораздо больше возможностей и усовершенствований, чем их аналоги DOS и пакетных файлов. Многие командные файлы DOS полагаются на вспомогательные утилиты, такие как ***ask.com***, кривой аналог ***read***.

DOS поддерживает только очень ограниченное и несовместимые подмножество расширений файла специальными символами подстановки, признавая только символы * и ?.

Конвертирование пакетного файла DOS в сценарий оболочки, как правило, не сложно, а

результат часто читается лучше, чем оригинал.

Пример N-1. VIEWDATA.BAT: Пакетный файл DOS

```
REM VIEWDATA

REM ВДОХНОВЛЕНО ПРИМЕРОМ В "DOS POWERTOOLS"
REM                          PAUL SOMERSON


@ECHO OFF

IF !%1==! GOTO VIEWDATA
REM  IF NO COMMAND-LINE ARG...
FIND "%1" C:\BOZO\BOOKLIST.TXT
GOTO EXIT0
REM  PRINT LINE WITH STRING MATCH, THEN EXIT.

:VIEWDATA
TYPE C:\BOZO\BOOKLIST.TXT | MORE
REM  SHOW ENTIRE FILE, 1 PAGE AT A TIME.

:EXIT0
```

Сценарий преобразования является своего рода улучшением. [1]

Пример N-2. *viewdata.sh*: Конвертация VIEWDATA.BAT в Shell Script

```
#!/bin/bash
# viewdata.sh
# Конвертация VIEWDATA.BAT в сценарий оболочки.

DATAFILE=/home/bozo/datafiles/book-collection.data
ARGNO=1

# @ECHO OFF                 Здесь нет необходимости в этой команде.

if [ $# -lt "$ARGNO" ]    # IF !%1==! GOTO VIEWDATA
then
  less $DATAFILE          # TYPE C:\MYDIR\BOOKLIST.TXT | MORE
else
  grep "$1" $DATAFILE     # FIND "%1" C:\MYDIR\BOOKLIST.TXT
fi

exit 0                    # :EXIT0

#  GOTO, метки, дым и зеркала, и прочая чушь не нужны.
#  Преобразованный сценарий короче и чище, чем оригинал.
```

Сайт Ted Davis Shell Scripts on the PC имел набор всеобъемлющих руководств о старомодном искусстве программирования пакетных файлов. К сожалению страница исчезла без следа.

Примечания

[1] Читатели предложили изменения пакетного файла выше, приукрасив его и сделав его более компактным и эффективным. По мнению автора ABS Guide это зря затраченные усилия. Сценарий Bash может получить доступ к файловой системе DOS, или даже к

разделам NTFS (с помощью **ntfs-3g**), для пакетной обработки или операций сценариями.

Приложение O. Упражнения

Содержание

Упражнения, предназначенные для проверки и расширения ваших знаний сценариев. Думайте о них как о вызове, как об интересном способе, чтобы идти дальше по пути тернистого волшебства UNIX .

В грязном переулке трущоб Хобокена, Нью-Джерси, стоит невзрачное приземистое двух этажное кирпичное здание с надписью, на ее стене, вырезанной на мраморной пластине:

`Зал славы Bash Scripting.`

Внутри, среди разных неинтересных пыльных экспонатов, вытравлен на листе латуни короткий, очень короткий, список тех немногих лиц, которые успешно освоили материал Advanced Bash Scripting Guide, о чем свидетельствует их мастерство в этом приложении Упражнения.

(Увы, автора ABS Guide нет среди них. Это, наверное, благодаря злым слухам об отсутствии учетных данных и недостаточном навыке написания сценариев.)

O.1. Анализ сценариев

Изучите следующий сценарий. Запустите его, а затем объясните, что он делает. Аннотируйте сценарий и перепишите его в более компактном и элегантном виде.

```
#!/bin/bash

MAX=10000


  for((nr=1; nr<$MAX; nr++))
  do
```

```
    let "t1 = nr % 5"
    if [ "$t1" -ne 3 ]
    then
      continue
    fi

    let "t2 = nr % 7"
    if [ "$t2" -ne 4 ]
    then
      continue
    fi

    let "t3 = nr % 9"
    if [ "$t3" -ne 5 ]
    then
      continue
    fi

  break   # Что случится, если закомментировать эту строку? Почему?

  done

  echo "Number = $nr"


exit 0
```

Объясните, что делает следующий сценарий. На самом деле это просто канал параметризованной командной строки.

```
#!/bin/bash

DIRNAME=/usr/bin
FILETYPE="shell script"
LOGFILE=logfile

file "$DIRNAME"/* | fgrep "$FILETYPE" | tee $LOGFILE | wc -l

exit 0
```

Исследуйте и объясните следующий сценарий. Подсказка, вы можете ссылаться на списки с **find** и **stat**.

```
#!/bin/bash

# Автор:  Nathan Coulter
# Этот код общедоступен.
# Автор дал разрешение на использование этого фрагмента кода в ABS.

find -maxdepth 1 -type f -printf '%f\000'  | {
   while read -d $'\000'; do
      mv "$REPLY" "$(date -d "$(stat -c '%y' "$REPLY") " '+%Y%m%d%H%M%S'
      )-$REPLY"
   done
```

```
}

# Внимание: Пробуйте этот сценарий в «используемой для упражнений» директории.
# Он как-то повлияет на все файлы в этой директории.
```

Прочитайте следующий фрагмент кода.

```
while read LINE
do
  echo $LINE
done < `tail -f /var/log/messages`
```

Он хотел написать сценарий для отслеживания изменений в системном файле журнала `/var/log/messages`. К сожалению, блок кода выше висит и не делает ничего полезного. Почему? Исправите, что бы он работал. (Подсказка: вместо перенаправления `stdin` в цикл, попробуйте канал.)

Проанализируйте следующие 'однострочник' (здесь разделенный на две строки, для ясности) предоставленный Rory Winston:

```
export SUM=0; for f in $(find src -name "*.java");
do export SUM=$(($SUM + $(wc -l $f | awk '{ print $1 }'))); done; echo $SUM
```

Подсказка: Во-первых, разбейте сценарий на секции. Затем тщательно изучите использование **двойных скобок** для вычислений, команду **export**, команду **find**, команды **wc** и **awk**.

Проанализируйте Пример A-10 и измените его сделав более простым и более логичным. Посмотрите, какие из переменных можно убрать и попробуйте оптимизировать сценарий, для уменьшения время его выполнения.

Изменить сценарий так, что бы он принимал любой обычный текстовый файл ASCII в качестве входных данных для первоначального «создания». Сценарий должен сначала читать символы *`$ROW*$COL`* и устанавливать вхождения гласных, как 'живых' ячеек. Подсказка: не забудьте перевести пробелы, во входном файле, на символы подчеркивания.

O.2. Написание сценариев

Напишите сценарий для выполнения каждой из следующих задач.

ПРОСТЫЕ

Самовоспроизводящийся сценарий

Напишите сценарий, который поддерживает себя, то есть, копирует себя в файл с

именем `backup.sh`.

Подсказка: используйте команду **cat** и соответствующие позиционные параметры.

Листинг домашней директории

Выполните рекурсивный обход директории в домашней директории пользователя и сохраните информацию в файл. Сожмите файл, пусть сценарий предложит пользователю вставить USB флешку, затем нажать ENTER. Наконец, сохраните файл на флэш-диске, когда после анализа вывода **df**, флэш-накопитель надлежащим образом примонтируется. Обратите внимание, что перед извлечением флэш-накопитель нужно будет *отмонтировать*.

Преобразование циклов *for* в циклы *while* и *until*

Преобразуйте циклы ***for*** в Примере 11-1 в циклы ***while***. Подсказка: сохраните данные в *массив* и пройдите по элементам массива.

Теперь, сделав эту трудную работу, конвертируйте циклы примера в циклы ***until***.

Изменение междустрочного интервала текстового файла

Напишите сценарий, который считывает каждую строку заданного файла, а затем выводит строку обратно в `stdout`, но с дополнительными пустыми строками. Должно иметь эффект *двойного межстрочного интервала* файла.

Добавьте весь необходимый код для проверки получения сценарием необходимых аргументов командной строки (имени файла) и определения существования указанного файла.

Когда сценарий будет работать правильно, измените его на установку *тройного* межстрочного интервала в заданном файле.

Наконец, напишите сценарий, удаляющий все пустые строки из целевого файла, оставив *одинарный* межстрочный интервал.

Обратный список

Напишите сценарий, который выводит себя в `stdout`, но в *обратном* порядке.

Автоматическая распаковка файлов

В качестве входных данных задайте список имен файлов, этот сценарий должен запрашивать каждый указанный файл (анализируя вывод команды **file**) для определения используемого в них типа сжатия. Затем сценарий должен автоматически вызывать команду соответствующей распаковки (**gunzip**, **bunzip2**, **unzip**, **uncompress** или любую другую). Если файл не сжат, сценарий должен выдать предупреждающее сообщение, но не принимать никаких иных действий для этого, конкретного, файла.

Уникальный идентификатор системы

Создайте «уникальный» 6-значный шестнадцатеричный идентификатор для вашего компьютера. Не используйте команду ***hostid***. Подсказка: `md5sum /etc/passwd`, а затем выберите первые 6 цифр вывода.

Резервное копирование

Заархивируйте в виде «tarball» (*.tar.gz файлы) все файлы в дереве вашей домашней директории (/home/ваше имя), которые были изменены за последние 24 часа. Подсказка: используйте **find**.

Дополнительно: вы можете использовать это, как основу сценария для резервного копирования.

Проверка, выполняется ли до сих пор процесс

Задав идентификатор процесса (**PID**), как аргумент, этот сценарий должен проверять, в интервалы, определяемые пользователем, выполняется ли еще данный процесс. Вы можете использовать команды **ps** и **sleep**.

Простые числа

Выведите (в `stdout`) все простые числа между 60000 и 63000. Вывод должен быть красиво отформатирован в колонки (Подсказка: используйте **printf**).

Лотерейные номера

Один тур лотереи включает в себя набор пяти различных чисел, в диапазоне 1-50. Напишите сценарий, который создает пять псевдослучайных чисел в этом диапазоне *без повторений*. Сценарий должен дать возможность выводить числа в `stdout` или сохранять их в файл, а также дату и время, когда определенный набор чисел был создан. (Если ваш сценарий последовательно создает *выигрышные* номера лотереи, то можно заработать, оставить оболочки сценариев для тех из нас, кто должен работать для того, что бы жить.)

СРЕДНЕЙ СЛОЖНОСТИ

Целое число или строка

Напишите ***функцию*** сценария, которая определяет, является ли переданный аргумент целым числом или строкой. Функция должна возвращать ИСТИННО (0), если передано целое число и ЛОЖНО (1), если передана строка.

Подсказка: Что возвратится, если `$1` не является целым числом ?

```
expr $1 + 0
```

ASCII в целые числа

Функция ***atoi*** в Си преобразует символ строки в целое число. Напишите функцию сценария оболочки, которая выполняет ту же операцию. Аналогичным образом, напишите функцию сценария оболочки, которая производит обратное, зеркальное, преобразование функции Си ***itoa*** , которая преобразует целое число в символ ASCII.

Управление дисковым пространством

Список, один файл за раз, всех файлов в дереве директории `/home/username` размером более 100K. Предоставьте пользователю возможность удалять или сжимать файл, а затем перейти к выводу следующего файла. Сделать запись в файл журнала имена всех удаленных файлов и время их удаления.

Banner

Имитировать в сценарии функциональность устаревшей команды **banner**.

Удаление не активных акаунтов

Неактивные акаунты засоряют дисковое пространство сетевого сервера и могут стать угрозой для безопасности. Напишите управляющий сценарий (вызываемый *root* или демоном **cron**), который проверяет наличие и удаляет учетные записи пользователей, к которым не обращались в течение последних 90 дней.

Обеспечение соблюдения дисковых квот

Напишите сценарий проверки использования диска пользователями многопользовательской системы. Если пользователь превышает установленный лимит (500 МБ, например) в своей директории `/home/username`, то сценарий автоматически отправляет ему предупреждение по электронной почте '`pigout`'.

Сценарий должен использовать команды **du** и **mail**. Дополнительно он должен позволять устанавливать и обеспечивать соблюдения квот, с помощью команд **quota** и **setquota**.

Информация о вошедших пользователях

Для всех вошедших пользователей, показывать их настоящие имена, время и дату их последнего входа.

Подсказка: используйте **who**, **lastlog** и проанализируйте `/etc/passwd`.

Безопасное удаление

Реализуйте, в виде сценария, команды «безопасного» удаления `sdel.sh`. Имена файлов, переданные, как аргументы командной строки, этому сценарию, не удаляются, а вместо этого *сжимаются* **gzip**, если не сжаты (используйте для проверки **file**), а затем перемещаются в директорию `~/TRASH` (Корзина). После вызова сценарий должен проверять директорию `~/TRASH` на файлы старше 48 часов и окончательно удалять их. (Лучшей альтернативой может быть наличие второго сценария управляющего им,

периодически вызываемого демоном **cron**).

Дополнительно: напишите сценарий так, что бы он мог ***рекурсивно*** обрабатывать файлы и директории. Это даст ему возможность «безопасно удалять» всю структуру директории.

Размен

Как наиболее эффективным способом разменять $ 1,68, используя, в общем, только монеты достоинства (до 25c)? Это 6 монет по 25, 1 монета по 10, пятицентовик и три цента.

Нужно задавать ввод любой произвольной командной строки в долларах и центах (`$*.??`), что бы рассчитать размен, используя минимальное количество монет. Если ваша страна не Соединенные Штаты, вместо долларов вы можете использовать вашу текущую местную валюту. Сценарию необходим анализ вводимого в командную строку, а затем изменение его на наименьшее кратное денежной единицы (центов или чего-то еще). Подсказка: посмотрите Пример 24-8.

Квадратное уравнение

Решите квадратное уравнение вида `Ax^2 + Bx + C = 0`. Создайте сценарий, принимающий в качестве аргументов коэффициенты **A**, **B** и **C** и возвращающий в решении до пяти десятичных знаков.

Подсказка: направляйте коэффициенты каналом в **bc**, по известной формуле, `x = ( -B +/- sqrt( B^2 - 4AC ) ) / 2A`.

Таблица логарифмов

Используя команды **bc** и **printf**, выведите на экран красиво отформатированную таблицу 8-значных натуральных логарифмов в диапазоне 0.00 100.00, с шагом .01.

Подсказка: **bc** потребует опция `-l`, для загрузки математических библиотек.

Таблица Юникода

Используя Пример T-1 в качестве шаблона, напишите сценарий, который выводит в файл полную таблицу Юникода.

Подсказка: Используйте опцию `-e` для **echo**: `echo -e '\uXXXX'`, где XXXX — это числовое обозначение символа Юникода. Необходима версия 4.2 Bash или более поздняя.

Сумма соответствующих чисел

Найдите сумму всех пятизначных цифр (в диапазоне 10000-99999) содержащих ровно две из следующего набор цифр: {4, 5, 6}. Они могут повторяться в числе, и если это так, то они рассчитываются один раз для каждого случая.

Некоторые примеры подходящих чисел 42057, 74638 и 89515.

Счастливые числа

Счастливое число это одна из отдельных цифр, в последовательных добавлениях, добавляемая до 7. Например, 62431 это *счастливое число* (6 + 2 + 4 + 3 + 1 = 16, 1 + 6 = 7). Найдите все *счастливые числа* между 1000 и 10000.

Кости

Заимствуя графику ASCII из Примера A-40, напишите сценарий, который играет в хорошо известную азартную игру *кости*. Сценарий должен принимать ставки от одного или более игроков, бросать кости и отслеживать победы и поражения, а также деньги каждого игрока.

Крестики-нолики

Напишите сценарий, который играет в детскую игру *крестики-нолики* против игрока человека. Сценарий должен выбирать, следует ли давать первый ход человеку. Сценарий должен следовать оптимальной стратегии и поэтому никогда не проигрывать. Чтобы упростить игру, вы можете использовать графику ASCII:

```
o | x |
----------
  | x |
----------
  | o |

Ваш ход, человек (строка, столбец)?
```

Алфавитная строка

Алфавитное (в порядке ASCII) считывание произвольной строки из командной строки.

Анализ

Проанализируйте `/etc/passwd` и выведите его содержимое в красивой, легко читаемой, таблице.

Журнал входа

Произведите анализ `/var/log/messages` и сделайте красиво отформатированный файл входа пользователей и времени входа. Сценарий должен запускаться из-под *root*. (Подсказка: Ищите строку "LOGIN.")

Красивый вывод файла данных

Какие-то базы данных и пакеты таблиц используют сохранение файлов с полями, разделенными запятыми, обычно называемые *разделителями-запятыми* или CSV. Другим приложениям часто необходимо анализировать эти файлы.

Задайте файл данных с запятыми, разделяющими поля:

```
Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989
Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612
...
```

Переформатируйте данные и выведите их в `stdout` в размеченных, равномерно расположенных столбцах.

Обоснование

Задайте ввод текста ASCII из `stdin` или файла, настройте пробелы между словами к правому краю каждой строки, заданной пользователем ширины строки, а затем отправьте вывод в `stdout`.

Список рассылки

Напишите сценарий, используя команду **mail**, который управляет простым списком рассылки. Сценарий ежемесячно автоматически рассылает по e-mail информационный бюллетень компании, считывая из указанного текстового файла, и отправляет его на все адреса в списке рассылки, который сценарий считывает из другого заданного файла.

Создание паролей

Создайте псевдослучайные 8-значные пароли, используя символы в диапазоне [0-9], [A-Z], [a-z]. Каждый пароль должен содержать по крайней мере две цифры.

Мониторинг пользователей

Вы подозреваете, что один конкретный пользователь в сети злоупотребляет привилегиями и, возможно, пытается взломать систему. Напишите сценарий для автоматически отслеживания и регистрации его деятельности, когда он входит в систему. Файл журнала должен сохранять записи предыдущей недели и удалять записи старее семи дней.

Вы можете использовать **last**, **lastlog** и **lastcomm**, для помощи в наблюдении за подозреваемым злодеем.

Проверка на упавшие ссылки

С помощью **lynx**, с опцией `-traversal`, напишите сценарий, который проверяет веб-сайт на упавшие ссылки.

ТРУДНЫЕ

Проверка паролей

Напишите сценарий для проверки правильности паролей. Объектом является флаг «***weak***» или кандидаты на легко угадываемый пароль.

Пробный пароль должен будет вводиться в сценарий, как параметр командной строки. Чтобы считаться приемлемым, пароль должен отвечать следующим минимальным требованиям:

- Минимальная длина в 8 символов
- Должен содержать, по крайней мере, один числовой символ
- Должен содержать по крайней мере один из следующих, не алфавитных, символов: `@, #, $, %, &, *, +, -, =`

Дополнительно:

- Осуществите на стадии проверки проверку словарем на каждую последовательность, по меньшей мере, четырех последовательных букв пароля. Это позволит устранить пароли, содержащие внедренные «слова» находящиеся в стандартном словаре.
- Допустите сценарий для проверки всех паролей в вашей системе. Не находящихся в `/etc/passwd`.

Это упражнение проверяет совершенство *регулярных выражений*.

Перекрестная ссылка

Напишите сценарий, который создает *перекрестную ссылку (согласованную)* на указанный файл. Результатом должен быть список всех вхождений слова в указанном файле, вместе с номерами строк, в которых находится каждое слово. Традиционно, в таких приложениях, будут использоваться конструкции *связанных списков*. Таким образом, в ходе этого упражнения, вы должны исследовать ***массивы***. Пример 16-12, вероятно, не лучший вариант для начала.

Квадратный корень

Напишите сценарий для вычисления квадратного корня чисел с помощью *способа Ньютона*.

Алгоритм, выраженный как фрагмент псевдо-кода Bash :

```
#  Способ (Исаака) Ньютона для быстрого извлечения
#+ квадратных корней.

guess = $argument
```

```
#  $argument это число, из которого нужно найти квадратный корень.
#  $guess это каждое последовательно вычисляемое «предположение»
#+ -- или промежуточное решение -- квадратного корня.
#  Наш первый "guess" квадратного корня это сам аргумент.

oldguess = 0
# $oldguess это предыдущий $guess.

tolerance = .000001
# Точность желаемого вычисления.

loopcnt = 0
# Давайте сделаем несколько проходов цикла.
# Некоторые аргументы потребуют больше проходов цикла, чем другие.


while [ ABS( $guess $oldguess ) -gt $tolerance ]
#       ^^^^^^^^^^^^^^^^^^^^^^^^ Конечно, исправьте синтаксис.

#  "ABS" это (плавающая точка) функция ищущая абсолютное значение
#+ разницы между двумя терминами.
#  Поэтому, до тех пор, пока разница между текущим и предыдущим
#+ промежуточным решением (guess) превышает точность, цикл будет
#+ работать.

do
   oldguess = $guess  # Обновляем $oldguess на предыдущий $guess.

#  ============================================================
   guess = ( $oldguess + ( $argument / $oldguess ) ) / 2.0
#        = 1/2 ( ($oldguess **2 + $argument) / $oldguess )
#  равнозначно:
#        = 1/2 ( $oldguess + $argument / $oldguess )
#  то есть, «усреднение» промежуточного решения и отклонение доли
#+ аргумента (разделение ошибки пополам).
#  Это сводит к точному решению,  удивительно, несколькими проходами
#+  цикла... для аргументов > $tolerance, конечно
#  ============================================================

   (( loopcnt++ ))     # Обновление счетчика циклов.
done
```

Это достаточно простой способ и на первый взгляд кажется достаточно легко преобразовать его в рабочий сценарий Bash. Проблема заключается в том, что Bash не имеет встроенной поддержки чисел с плавающей точкой. Таким образом, автору сценария необходимо использовать **bc** или, возможно, **awk,** для преобразования чисел и произведения вычислений. Может получиться довольно топорно...

Журналирование доступов к файлу

Журнал всех доступов к файлам в `/etc` в течении одного дня. Информация должна включать: имя файла, имя пользователя и время доступа. Если какие-либо файлы изменялись, то они должны быть отмечены. Эти данные должны быть записаны в файл журнала в табличной (разделенной табуляцией) форме.

Мониторинг процессов

Напишите сценарий для постоянного мониторинга всех запущенных процессов и отслеживания количества дочерних процессов порожденных каждым родителем. Если процесс порождает более пяти потомков, то сценарий должен отправлять по электронной почте всю соответствующую информацию системному администратору (или *root*), включая время, родительский PID, PID потомков, и т.д. Сценарий должен добавлять отчет в файл журнала каждые десять минут.

Удаление комментариев

Удалите все комментарии из сценария, имя которого задается в командной строке. Обратите внимание, что что начальная `#!строка` не должна быть удалена.

Удаление тэгов HTML

Удалите все тэги из указанного файла HTML, затем, переформатируйте его в строки длиной от 60 до 75 символов. При необходимости сбросьте абзацы, заблокируйте интервалы и конвертируйте HTML-таблицы в их примерные текстовые эквиваленты.

Конвертация XML

Преобразуйте XML-файл в формат HTML и текстовый формат.

Дополнительно: Сценарий должен конвертировать Docbook/SGML в XML.

Определение спаммеров

Напишите сценарий, который анализирует спам электронной почты, производя поиск DNS IP-адресов в заголовках для идентификации промежуточных хостов и провайдеров (ISP). Сценарий должен пересылать не измененные спам-сообщения ответственным Интернет-провайдерам. Конечно, необходимо будет отфильтровать IP-адрес собственного провайдера.

При необходимости используйте *команды анализа сети*.

Некоторые идеи увидите в Примере 16-41 и Примере A-28 .

Дополнительно: Напишите сценарий, который просматривает список сообщений электронной почты и удаляет спам согласно указанным фильтрам.

Создание Ман-страниц

Напишите сценарий, который автоматизирует процесс создания ман-страниц.

Задайте текстовый файл, который содержит информацию о форматировании ман-страницы, сценарий должен считывать этот файл, а затем вызывать соответствующие команды **groff** для вывода соответствующей ман-страницы в `stdout`. Текстовый файл содержит блоки информации в стандартных разделах ман-страниц, т.е. название, краткий обзор, описание, и т.д.

Для начала будет полезен Пример A-39.

Hex Dump

Создайте hexdump (не десятичный!) двоичного файла, задаваемого как аргумент сценария. Результатом должны быть аккуратные табличные поля, первое поле показывает адреса, каждое из следующих 8 полей 4-х байтное шестнадцатеричное число, а конечное поле - ASCII эквивалент предыдущих 8 полей.

Очевидным ответом на это является расширение сценария hexdump в дизассемблере. С помощью таблицы подстановки, или некоторых других хитрых трюков, преобразуйте шестнадцатеричные значения в коды 80x86 op.

Эмуляция регистра сдвига

Используя Пример 27-15, в качестве образца, напишите сценарий, который эмулирует 64-битный сдвиг регистра как ***массива***. Реализуйте функции для *загрузки* регистра, *сдвига влево*, *сдвига вправо* и его *поворота*. Наконец, напишите функцию, которая представляет содержимое регистра в виде восьми 8-битных символов ASCII.

Расчет детерминантов

Напишите сценарий, который вычисляет детерминанты [1] путем рекурсивного расширения меньшего. Используйте детерминанту 4 x 4 в качестве тестового примера.

Скрытые слова

Напишите генератор головоломки "word-find", сценарий, который скрывает 10 введенных слов в массиве случайных букв 10 x 10. Слова могут быть скрыты вниз, поперек или по диагонали.

Дополнительно: Напишите сценарий, который *решает* головоломку "word-find". Чтобы не усложнять, сценарий должен искать только горизонтальные и вертикальные слова. (Подсказка: относиться к каждой строке и столбцу как к строке, а искать substring.)

Аннанграммы

Введите 4-х буквенную аннаграмму. К примеру, анаграммами *word* являются: *do or rod row word*. В качестве списка ссылок вы можете использовать `/usr/share/dict/linux.words`.

Лестницы слов

"Word ladder" представляет собой последовательность слов, каждое следующее слово в последовательности отличается от предыдущего на одну букву.

Например, "лестница" из *mark* а *vase*:

```
mark --> park --> part --> past --> vast --> vase
         ^           ^       ^      ^           ^
```

Напишите сценарий, который решает головоломку "лестница слов". Учитывая начало и окончание слова, сценарий будет выводить список всех промежуточных шагов в «лестнице». Обратите внимание, что все слова в последовательности должны быть из словаря.

Fog Index

"Fog index" оценивает сложность чтения отрывка текста в виде числа, примерно соответствующему определенному школьному уровню. Например, отрывок с fog index 12 должен быть понятен любому, кто учится 12 лет.

Версия Gunning использует следующий алгоритм для fog index.

1. Выбирается отрывок текста, объемом, по крайней мере, в 100 слов.

2. Подсчитывается количество предложений (часть предложения обрезанное границей отрывка текста считается одним).

3. Высчитывается среднее количество слов в предложении.

   AVE_WDS_SEN = TOTAL_WORDS / SENTENCES

4. Подсчитывается количество «трудных» слов в отрывке — т. е., содержащих, по крайней мере, 3 слога. Доля трудных слов делится на количество всех слов.

   PRO_DIFF_WORDS = LONG_WORDS / TOTAL_WORDS

5. Gunning fog index является суммой двух указанных количеств, умноженных на 0,4 и округленной до ближайшего целого числа.

   G_FOG_INDEX = int (0.4 * (AVE_WDS_SEN + PRO_DIFF_WORDS))

Шаг 4 является самой сложной частью упражнения. Существуют различные алгоритмы подсчета количества слогов в слове.

По большому счету, Gunning fog index не должен считать сложные слова и имена собственные «трудными» словами, это бы чрезвычайно усложнило сценарий.

Расчет Пи с помощью иглы Buffon

Французский математик восемнадцатого века de Buffon придумал новый эксперимент. Многократное бросание иглы длиной n на деревянный пол, состоящий из длинных и узких параллельных досок. Промежутки (пазы), разделяющие доски равной ширины друг от друга, имеют фиксированную ширину d. Подсчитывая общее количество бросков и количество раз, когда игла пересекает паз на полу. Отношение этих двух величин оказывается дробным числом кратным Пи.

В духе Примера 16-50, напишите сценарий, который выполняет способом Монте-Карло моделирование иглы Buffon. Для упрощения, примите длину иглы, равной расстоянию между трещинами, $n = d$.

Подсказка: в действительности существуют две критические переменные: расстояние от центра иглы до ближайшего паза, и угол наклона иглы к этому пазу. Для обработки расчетов можно использовать **bc**.

Шифр Playfair

Реализация в сценарии шифра Playfair (Wheatstone).

Шифр Playfair шифрует текст путем замены *диграмм* (2-х буквенных групп). Традиционным является использование алфавитного квадрата состоящего из 5 x 5 ключевых букв для шифрования и дешифрования.

```
   C O D E S
   A B F G H
   I K L M N
   P Q R T U
   V W X Y Z

Каждая буква алфавита используется один раз, за исключением «I», которая
означает еще и "J". Начинаем с произвольно выбранного ключевого слова «CODES»,
а затем идут все остальные буквы алфавита, в порядке слева направо, пропуская
уже использованные буквы.

Для шифрования, разделите текст сообщения на пары букв (2-х буквенные группы).
Если группа имеет две одинаковые буквы, удалите вторую и сформируйте новую
группу. Если в конце остается одна буква - вставляется символ «null», как
правило «X»

THIS IS A TOP SECRET MESSAGE

TH IS IS AT OP SE CR ET ME SA GE



Для каждой пары есть три варианта.
-----------------------------------------------

1) Обе буквы находятся в одной строке таблицы ключей:
   Каждая буква заменяется буквой находящейся в этой строке справа.
   Если нужно, вернитесь по кругу в лево, к началу строки.

или

2) Обе буквы находятся в той же колонке таблицы ключей:
   Каждая буква заменяется буквой в строке находящейся непосредственно под ней.
   При необходимости, вернитесь по кругу в верхнюю часть колонки.

или

3) Обе буквы находятся в углах ключевой таблицы: Каждая буква заменяется на
   букву в  другом углу прямоугольника, находящейся на той же строке.
```

```
Пара "TH" подпадающая под случай #3.
G H
M N
T U           (Прямоугольник с "T" и "H" в углах)

T --> U
H --> G


Пара "SE" подпадающая под случай #1.
C O D E S     (Строка содержит "S" и "E")

S --> C  (возвращаемся влево, в начало строки)
E --> S

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

Расшифровывается зашифрованный текст в обратном порядке в случаях #1
и #2 (для замещения двигаемся в противоположном направлении). А случае #3,
просто берутся оставшиеся два угла прямоугольника.


Классическая работа Helen Fouche Gaines, ELEMENTARY CRYPTANALYSIS (1939), дает
довольно подробное описание шифра Playfair и способы его решения.
```

Этот сценарий должен иметь три основных раздела

I. Создание таблицы ключей (*key square)*, основывающейся на введенном пользователем ключевом слове.

II. Шифрование текста сообщения.

III.Расшифровка зашифрованного текста.

Сценарий должен более полно использовать ***массивы*** и ***функции***. В качестве образца вы можете использовать Пример A-56.

--

Пожалуйста, не присылайте автору ваши решения этих упражнений. Есть более подходящие способы, что бы впечатлить его вашим умом, например исправления и предложения по улучшению книги.

Примечания

[1] Для всех умных типов, которые не осилили промежуточную алгебру, детерминанта это числовое значение связанное с многомерной *матрицей* (массивом чисел).

```
Пример случая детерминанты 2 x 2:

  |a  b|
  |b  a|
```

```
Решением является a*a - b*b, где "a" и "b" являются числами.
```

Приложение P. История изменений

This document first appeared as a 60-page HOWTO in the late spring of 2000. Since then, it has gone through quite a number of updates and revisions. This book could not have been written without the assistance of the Linux community, and especially of the volunteers

of the Linux Documentation Project.

Here is the e-mail to the LDP requesting permission to submit version 0.1.

```
From thegrendel@theriver.com Sat Jun 10 09:05:33 2000 -0700
Date: Sat, 10 Jun 2000 09:05:28 -0700 (MST)
From: "M. Leo Cooper" <thegrendel@theriver.com>
X-Sender: thegrendel@localhost
To: ldp-discuss@lists.linuxdoc.org
Subject: Permission to submit HOWTO

Dear HOWTO Coordinator,

I am working on and would like to submit to the LDP a HOWTO on the subject
of "Bash Scripting" (shell scripting, using 'bash'). As it happens,
I have been writing this document, off and on, for about the last eight
months or so, and I could produce a first draft in ASCII text format in
a matter of just a few more days.

I began writing this out of frustration at being unable to find a
decent book on shell scripting. I managed to locate some pretty good
articles on various aspects of scripting, but nothing like a complete,
beginning-to-end tutorial.  Well, in keeping with my philosophy, if all
else fails, do it yourself.

As it stands, this proposed "Bash-Scripting HOWTO" would serve as a
combination tutorial and reference, with the heavier emphasis on the
tutorial. It assumes Linux experience, but only a very basic level
of programming skills. Interspersed with the text are 79 illustrative
example scripts of varying complexity, all liberally commented. There
are even exercises for the reader.

At this stage, I'm up to 18,000+ words (124k), and that's over 50 pages of
text (whew!).


I haven't mentioned that I've previously authored an LDP HOWTO, the
"Software-Building HOWTO", which I wrote in Linuxdoc/SGML. I don't know
if I could handle Docbook/SGML, and I'm glad you have volunteers to do
the conversion. You people seem to have gotten on a more organized basis
these last few months. Working with Greg Hankins and Tim Bynum was nice,
but a professional team is even nicer.

Anyhow, please advise.


Mendel Cooper
thegrendel@theriver.com
```

Таблица P-1. История изменений

Релиз	Дата	Комментарий
0.1	14 Jun 2000	Initial release.

Релиз	Дата	Комментарий
0.2	30 Oct 2000	Bugs fixed, plus much additional material and more example scripts.
0.3	12 Feb 2001	Major update.
0.4	08 Jul 2001	Complete revision and expansion of the book.
0.5	03 Sep 2001	Major update: Bugfixes, material added, sections reorganized.
1.0	14 Oct 2001	Stable release: Bugfixes, reorganization, material added.
1.1	06 Jan 2002	Bugfixes, material and scripts added.
1.2	31 Mar 2002	Bugfixes, material and scripts added.
1.3	02 Jun 2002	TANGERINE release: A few bugfixes, much more material and scripts added.
1.4	16 Jun 2002	MANGO release: A number of typos fixed, more material and scripts.
1.5	13 Jul 2002	PAPAYA release: A few bugfixes, much more material and scripts added.
1.6	29 Sep 2002	POMEGRANATE release: Bugfixes, more material, one more script.
1.7	05 Jan 2003	COCONUT release: A couple of bugfixes, more material, one more script.
1.8	10 May 2003	BREADFRUIT release: A number of bugfixes, more scripts and material.
1.9	21 Jun 2003	PERSIMMON release: Bugfixes, and more material.
2.0	24 Aug 2003	GOOSEBERRY release: Major update.
2.1	14 Sep 2003	HUCKLEBERRY release: Bugfixes, and more material.
2.2	31 Oct 2003	CRANBERRY release: Major update.
2.3	03 Jan 2004	STRAWBERRY release: Bugfixes and more material.
2.4	25 Jan 2004	MUSKMELON release: Bugfixes.
2.5	15 Feb 2004	STARFRUIT release: Bugfixes and more material.
2.6	15 Mar 2004	SALAL release: Minor update.
2.7	18 Apr 2004	MULBERRY release: Minor update.
2.8	11 Jul 2004	ELDERBERRY release: Minor update.
3.0	03 Oct 2004	LOGANBERRY release: Major update.
3.1	14 Nov 2004	BAYBERRY release: Bugfix update.
3.2	06 Feb 2005	BLUEBERRY release: Minor update.
3.3	20 Mar 2005	RASPBERRY release: Bugfixes, much material added.
3.4	08 May 2005	TEABERRY release: Bugfixes, stylistic revisions.
3.5	05 Jun 2005	BOXBERRY release: Bugfixes, some material added.
3.6	28 Aug 2005	POKEBERRY release: Bugfixes, some material added.
3.7	23 Oct 2005	WHORTLEBERRY release: Bugfixes, some material added.
3.8	26 Feb 2006	BLAEBERRY release: Bugfixes, some material added.
3.9	15 May 2006	SPICEBERRY release: Bugfixes, some material added.
4.0	18 Jun 2006	WINTERBERRY release: Major reorganization.
4.1	08 Oct 2006	WAXBERRY release: Minor update.
4.2	10 Dec 2006	SPARKLEBERRY release: Important update.
4.3	29 Apr 2007	INKBERRY release: Bugfixes, material added.
5.0	24 Jun 2007	SERVICEBERRY release: Major update.
5.1	10 Nov 2007	LINGONBERRY release: Minor update.
5.2	16 Mar 2008	SILVERBERRY release: Important update.

Релиз	Дата	Комментарий
5.3	11 May 2008	GOLDENBERRY release: Minor update.
5.4	21 Jul 2008	ANGLEBERRY release: Major update.
5.5	23 Nov 2008	FARKLEBERRY release: Minor update.
5.6	26 Jan 2009	WORCESTERBERRY release: Minor update.
6.0	23 Mar 2009	THIMBLEBERRY release: Major update.
6.1	30 Sep 2009	BUFFALOBERRY release: Minor update.
6.2	17 Mar 2010	ROWANBERRY release: Minor update.
6.3	30 Apr 2011	SWOZZLEBERRY release: Major update.
6.4	30 Aug 2011	VORTEXBERRY release: Minor update.
6.5	05 Apr 2012	TUNGSTENBERRY release: Minor update.
6.6	27 Nov 2012	YTTERBIUMBERRY release: Minor update.
10	10 Mar 2014	YTTERBIUMBERRY release: License change.

Приложение Q. Сайты зеркал и загрузки

Последнее обновление этого документа, в виде архива, тарболла ***bzip2***, включая исходник SGML и созданный HTML, могут быть загружены с домашнего сайта автора. Также доступна версия **pdf** (с зеркала сайта). Есть также версия **epub**, любезно предоставленная Craig Barnes и Michael Satke. В журнале изменений указана подробная история изменений. *ABS Guide* даже имеет свою собственную страницу `freshmeat.net/freecode`, для отслеживания важных обновлений, комментариев пользователей и рейтинга популярности проекта.

Основным хостингом сайта этого документа является Linux Documentation Project, который также сопровождает многие другие Руководства и HOWTO.

Большое спасибо Ronny Bangsund за предоставление дискового пространства для размещения этого проекта.

Приложение R. Список To Do (Что еще надо сделать)

- Всеобъемлющий обзор *несовместимости* между Bash и классическим Bourne shell.
- То же, что и выше, но с Korn shell (*ksh*).

Приложение S. Авторские права

Advanced Bash Scripting Guide является настоящим общественным достоянием (PUBLIC DOMAIN). Это имеет следующие смысл и последствия.

```
A.  Все предыдущие релизы Advanced Bash Scripting Guide также предоставляются
    в  общественное достояние.

A1. Все печатные издания, с согласия автора или нет, также предоставляются
    в  общественное достояние. Это юридически переопределяет любые намерения
    или желания издателей. Любое заявление авторских прав является ничтожным
    и недействительным.
    В ЭТОМ ПУНКТЕ НИКАКИХ ИСКЛЮЧЕНИЙ БЫТЬ НЕ МОЖЕТ

A2. Любой выпуск Advanced Bash Scripting Guide, будь то в электронной или
    печатной форме, предоставляются в общественное достояние по прямому
    указанию автора и предыдущего владельца авторских прав, Mendel Cooper.
    Без каких-либо других лиц или организаций, когда-либо имевших
    действительное авторское право.

B.  Как документу Public Domain, предоставляются неограниченные права на
    копирование и распространение. Не может быть никаких ограничений. Если
    кто-нибудь опубликовал, или будет, в будущем, публиковать исходный или
    измененный вариант этого документа, то только дополнительные, оригинальные,
    материалы могут быть защищены авторским правом. Основная работа будет
    оставаться в общественном достоянии.
```

По закону, распространителям и издателям (включая он-лайн издателей), запрещается навязывать каких-либо условия, структуры или положения этого документа для всех предыдущих версий или любых производных версий. Автор утверждает, что он не заключал каких-либо договорных обязательств, которые могли бы изменить вышеупомянутые заявления.

По сути, вы можете свободно распространять эту книгу или любую производную от нее в электронном или печатном виде. Если вы ранее приобрели или имеете распечатанную копию текущего или предыдущего издания, то у вас есть ЗАКОННОЕ право копировать и распространять его, независимо от любого уведомления об авторском праве. Любые уведомления об авторском праве являются недействительными.

Кроме того, автор желает заявить о своем намерении:

```
Если вы скопировали или распространяете эту книгу, просьба НЕ ИСПОЛЬЗОВАТЬ
части, или любую часть,материала в патентах или авторских правах в исках против
сообщества Open Source, его разработчиков, его дистрибьюторов или против
любого, из связанного с ним, программного обеспечения или документации,
включая, но не ограничиваясь, ядра Linux, Open Office, Samba и Wine. Просьба НЕ
ИСПОЛЬЗОВАТЬ любой из материалов, в рамках этой книги, в показаниях в качестве
истца «свидетеля-эксперта» в любом иске против сообщества Open Source, любого
из его разработчиков, его дистрибьюторов или любого, связанного с ними,
программного обеспечения или документации.
```

Лицензия Public Domain по существу не ограничивает любое законное распространение или

использование этой книги. Особенно автор призывает (royalty-free!) использовать ее для учебных целей.

На сегодняшний день, ограниченные права на печать (издание Lulu) были предоставлены одному лицу и никому другому. Ни кто, ни собственно Lulu, не имеет и никогда не имел действительных авторских прав.

Внимание автора привлекло несанкционированная продажа электронных и печатных изданий этой книги на iTunes®, Amazon.com и в других местах. Это незаконные и пиратские издания, произведенные без разрешения автора, и читателям этой книги настоятельно рекомендуется не покупать их. На самом деле, эти пиратские издания теперь законны, но обязательно должны входить в Public Domain, а любые уведомления об авторских правах, содержащиеся в них, являются недействительными и пустыми.

Автор выпустил эту книгу в соответствии с духом ***LDP Manifesto***.

Hyun Jin Cha сделал корейский перевод версии 1.0.11 этой книги. Испанский, португальский, французский, немецкий, итальянский, русский, чешский, китайский, индонезийский, голландский, румынский, болгарский и турецкий переводы, также доступны или в стадии

разработки. Если вы хотите перевести этот документ на другой язык, пожалуйста, не стесняйтесь делать это, в соответствии с условиями, изложенными выше. Автор хотел бы получать уведомления о таких усилиях.

Щедрые читатели, желающие сделать пожертвование автору, могут пожертвовать небольшое количество через через Paypal на мой адрес электронной почты, <thegrendel.abs@gmail.com>. (**Honor Roll** поддерживающих дается в начале истории версий.) Это *не* обязательное требование. ABS Guide является свободным и свободно распространяемым документом для использования сообществом Linux. Тем не менее, в эти трудные времена, оказывать поддержку добровольным проектам и особенно авторам с ограниченными средствами, особенно важно, чем когда-либо.

Приложение T. Таблица ASCII

Традиционно, книги такого рода имеют в приложении таблицу ASCII. В этой книге этого нет. Вместо этого, в ней несколько коротких сценариев, каждый из которых создает полную таблицу ASCII.

Пример T-1. Сценарий создающий таблицу ASCII

```
#!/bin/bash
# ascii.sh
```

```
# ver. 0.2, reldate 26 Aug 2008
# Исправлено автором ABS Guide.

# Оригинальный сценарий Sebastian Arming.
# Используется с разрешения (спасибо!).

exec >ASCII.txt         #  Сохраняем stdout в файл,
                        #+ как в примере сценария
                        #+ reassign-stdout.sh и upperconv.sh.

MAXNUM=256
COLUMNS=5
OCT=8
OCTSQU=64
LITTLESPACE=-3
BIGSPACE=-5

i=1 # Десятичный счетчик
o=1 # Восьмеричный счетчик

while [ "$i" -lt "$MAXNUM" ]; do  #  Нам не нужно рассчитывать последнее
                                  #+ восьмеричное 400.
        paddi="    $i"
        echo -n "${paddi: $BIGSPACE}  "       # Пространство колонки.
        paddo="00$o"
#       echo -ne "\\${paddo: $LITTLESPACE}"   # Оригинал.
        echo -ne "\\0${paddo: $LITTLESPACE}"  # Исправление.
#                  ^
        echo -n "     "
        if (( i % $COLUMNS == 0)); then       # Новая строка.
           echo
        fi
        ((i++, o++))
        #  Восьмеричная запись для 8 это 10, а десятеричное 64
        #+ это восьмеричное  100.
        (( i % $OCT == 0))    && ((o+=2))
        (( i % $OCTSQU == 0)) && ((o+=20))
done

exit $?

# Сравните этот сценарий с примером "pr-asc.sh".
# Он обрабатывает 'непечатные' символы.

# Упражнение:
# Перепишите сценарий для использования десятичных чисел, вместо восьмеричных.
```

Пример T-2. Другой сценарий таблицы ASCII

```
#!/bin/bash
# Автор сценария: Joseph Steinhauser
# Легкое редактирование автором ABS Guide, но без комментариев.
# Используется в ABS Guide с разрешения.

#-----------------------------------------------------------------------
#-- Файл:  ascii.sh    Вывод символов ASCII, создан 10/8/16  (JETS-2012)
```

```
#-------------------------------------------------------------------------
#-- Usage: ascii [oct|dec|hex|help|8|10|16]
#--
#-- Этот сценарий выводит краткое описание кодов символов ASCII от 0 до 127.
#-- Числовые значения могут быть выведены по основанию 10, 8 или 16.
#--
#-- Формат основания: /usr/share/lib/pub/ascii с основанием 10, как умолчание.
#-- Детали см. man ascii ...
#-------------------------------------------------------------------------

[ -n "$BASH_VERSION" ] && shopt -s extglob

case "$1" in
   oct|[Oo]?([Cc][Tt])|8)       Obase=Octal;  Numy=3o;;
   hex|[Hh]?([Ee][Xx])|16|[Xx]) Obase=Hex;    Numy=2X;;
   help|?(-)[h?])        sed -n '2,/^[ ]*$/p' $0;exit;;
   code|[Cc][Oo][Dd][Ee])sed -n '/case/,$p'   $0;exit;;
   *) Obase=Decimal
esac # На самом деле КОДА меньше, чем символов!

printf "\t\t## $Obase ASCII Chart ##\n\n"; FM1="|%0${Numy:-3d}"; LD=-1

AB="nul soh stx etx eot enq ack bel bs tab nl vt np cr so si dle"
AD="dc1 dc2 dc3 dc4 nak syn etb can em sub esc fs gs rs us sp"

for TOK in $AB $AD; do ABR[$((LD+=1))]=$TOK; done;
ABR[127]=del

IDX=0
while [ $IDX -le 127 ] && CHR="${ABR[$IDX]}"
   do (${#CHR}))&& FM2='%-3s'|| FM2=`printf '\\\\%o  ' $IDX`
      printf "$FM1 $FM2" "$IDX" $CHR; (( (IDX+=1)%8))||echo '|'
   done

exit $?
```

Пример T-3. Третий сценарий таблицы ASCII, использующий *awk*

```
#!/bin/bash
# Сценарий таблицы ASCII, использующий awk.
# Автор: Joseph Steinhauser
# В ABS Guide используется с разрешения.


#-------------------------------------------------------------------------
#-- Файл:  ascii     Вывод символов ASCII, создан 10/8/16  (JETS-2010)
#-------------------------------------------------------------------------
#-- Usage: ascii [oct|dec|hex|help|8|10|16]
#--
#-- Этот сценарий выводит краткое описание кодов символов ASCII от 0 до 127.
#-- Числовые значения могут быть выведены по основанию 10, 8 или 16.
#--
#-- Формат основания: /usr/share/lib/pub/ascii по основанию 10, как умолчание.
#-- Детали см. man ascii
#-------------------------------------------------------------------------

[ -n "$BASH_VERSION" ] && shopt -s extglob
```

```
case "$1" in
   oct|[Oo]?([Cc][Tt])|8)       Obase=Octal;  Numy=3o;;
   hex|[Hh]?([Ee][Xx])|16|[Xx]) Obase=Hex;    Numy=2X;;
   help|?(-)[h?])        sed -n '2,/^[ ]*$/p' $0;exit;;
   code|[Cc][Oo][Dd][Ee])sed -n '/case/,$p'   $0;exit;;
   *) Obase=Decimal
esac
export Obase   # На самом деле КОДА меньше, чем символов!

awk 'BEGIN{print "\n\t\t## "ENVIRON["Obase"]" ASCII Chart ##\n"
           ab="soh,stx,etx,eot,enq,ack,bel,bs,tab,nl,vt,np,cr,so,si,dle,"
           ad="dc1,dc2,dc3,dc4,nak,syn,etb,can,em,sub,esc,fs,gs,rs,us,sp"
           split(ab ad,abr,",");abr[0]="nul";abr[127]="del";
           fm1="|%0'"${Numy:- 4d}"' %-3s"
           for(idx=0;idx<128;idx++){fmt=fm1 (++colz%8?"":"|\n")
           printf(fmt,idx,(idx in abr)?abr[idx]:sprintf("%c",idx))} }'

exit $?
```

Указатель

Это указатель/глоссарий/список быстрых ссылок на многие важные темы, затронутые в тексте руководства. Термины расположены в приблизительной сортировке ASCII, измените, в случае необходимости, для повышения ясности.

Обратите внимание, что *команды* указаны в Главе 4.

* * *

^ (**знак вставки**)

- **^** ***Начало строки*** в ***регулярном выражении***
- **^^** **Преобразование в верхний регистр** ***в подстановке параметров***

~ ***Тильда***

- **~** Домашняя директория, соответствует `$HOME`
- **~/** Текущая домашняя директория пользователя
- **~+** Текущая рабочая директория
- **~-** Предыдущая рабочая директория

= **Знак равенства**

- **=** Оператор присваивания переменной
- **=** Оператор сравнения строк

  == Оператор сравнения строк
- **=~** Оператор ***соответствия*** регулярного выражения

Пример сценария

< ***Левая угловая скобка***

- Меньше, чем

  Сравнение строк

  Сравнение целых чисел внутри *двойных круглых скобок*

- Перенаправление

  < `stdin`

  << *Here document*

  <<< *Here string*

  <> *Открытие файла* для чтения и записи

> **Правая угловая скобка**

- Больше, чем

  Сравнение строк

  Сравнение целых чисел внутри *двойных круглых скобок*

- Перенаправление

  > Перенаправление `stdout` в файл

  >> Перенаправление `stdout` в файл, *но добавление*

  i>&j Перенаправление файлового дескриптора i в файловый дескриптор j

  >&j Перенаправление `stdout` в файл дескриптор j

  >&2 Перенаправление `stdout` команды в `stderr`

  2>&1 Перенаправление `stderr` в `stdout`

  &> Перенаправление и `stdout` и `stderr` команды в файл

  :> file *Урезает* файл до нулевой длины

| ***Канал*** (туннель), устройство для передачи вывода команды другой команде или оболочке

|| Логический оператор проверки **ИЛИ**

- (**тире**)

- ***Префикс параметра*** *по умолчанию* при *подстановке параметра*
- ***Префикс*** в опции флага

- Указывает ***перенаправление*** из `stdin` или `stdout`
- `--` (**двойное тире**)

  Префикс для длинных (развернутых) опций команды

  Уменьшение переменной в стиле Си в двойных скобках

`;` (**точка с запятой**)

- Разделитель команд
- `\;` ***Заэкранированная точка с запятой*** завершает команду **find**
- `;;` ***Двойная точка с запятой*** завершает опции в **case**

  Когда требуется...

  если ключевое слово **do** находится на первой строке цикла

  при завершении блока кода в фигурных скобках
- `;;& ;&` ***Завершает*** опции в **case** (версия 4 Bash).

`:` **Двоеточие**

- `:> filename` Урезает файл до нулевой длинны
- ***Команда null***, эквивалентна встроенному **true** (истинно) в Bash
- Используется в ***анонимном here document***
- Используется как ***пустая функция***
- Используется как ***имя функции***

`!` **Оператор отрицания**, инвертирует *код выхода* проверки или команды

- `!=` **не равно** - оператор сравнения строк

`?` (**знак вопроса**)

- Соответствие ничему или одному символу в *расширенном регулярном выражении*
- Один символ *подстановки* в шаблонах подстановки
- Тройной оператор в стиле Си

`//` **Двойной слэш**

`.` (**точка** / точка)

- `.` Загрузка файла (в сценарии), эквивалентна команде **source**
- `.` Соответствует одному символу в регулярном выражении

- **.** Текущая рабочая директория

  ./ Текущая рабочая директория

- **..** Родительская директория

' ... ' (одинарные кавычки) **строгие кавычки**

" ... " (двойные кавычки) **мягкие кавычки**

- ***В двойных кавычках*** обратный слэш (****) является ***символом***

, Оператор запятая

- **,**
- **,,** Преобразование в нижний регистр (в строчные) при *подстановке параметров*

( ) **Круглые скобки**

- **(...)** Группа команд; запуск subshell
- **(...)** Заключенная группа *расширенных регулярных выражений*
- **>(...)**

  <(...) Процесс замены
- **...)** Завершения условия проверки в конструкции **case**
- **((...))** Двойные круглые скобки, арифметическое расширение

[Левая квадратная скобка, конструкция **test**

[ ] **Квадратные скобки**

- Элемент массива
- Заключенный набор символов для сопоставления в *регулярном выражении*
- Конструкция Test (проверка)

[[...]] Двойные квадратные скобки, расширенная конструкция `test` (проверка)

$ Якорь, в регулярном выражении

$ Префикс имени переменной

$(...) ***Команда замены***, присваивает переменную с выходом команды, с помощью нотации *скобок*

` ... ` ***Команда замены***, с помощью нотации *обратных кавычек*

$[...] Целочисленное расширение (устарело)

${ ... } Управление/оценка переменной

- **${var}** Значение переменной
- **${#var}** Длина переменной
- **${#@}** Длина переменной

  ${#*} Количество *позиционных параметров*
- **${parameter?err_msg}** Параметр *отключения* сообщения
- **${parameter-default}** Присвоение параметра по умолчанию

  ${parameter:-default} Присвоение параметра по умолчанию

  ${parameter=default} Присвоение параметра по умолчанию

  ${parameter:=default} Присвоение параметра по умолчанию
- **${parameter+alt_value}** Альтернативное значение параметра, если задано

  ${parameter:+alt_value} Альтернативное значение параметра, если задано
- **${!var}** Косвенная ссылка на переменную, *новая нотация*
- **${!#}** *Конечный* позиционный параметр (Косвенная ссылка на **$#**.)
- **${!varprefix*}** Совпадение с именами всех ранее объявленных переменных, начиная с **varprefix**

  ${!varprefix@} Совпадение с именами всех ранее объявленных переменных, начиная с **varprefix**
- **${string:position}** Извлечение *substring* (содержимого строки)

  ${string:position:length} Извлечение *substring* (содержимого строки)
- **${var#Pattern}** Удаление содержимого строки

  ${var##Pattern} Удаление содержимого строки
- **${var%Pattern}** Удаление содержимого строки

  ${var%%Pattern} Удаление содержимого строки
- **${string/substring/replacement}** Замена содержимого строки

  ${string//substring/replacement} Замена содержимого строки

  ${string/#substring/replacement} Замена содержимого строки

  ${string/%substring/replacement} Замена содержимого строки

$' ... ' Строка расширения, использующая экранированные символы.

**** Символ ***экранирует*** следующее :

- **\< ... \>** З*аэкранированные у*гловые скобки, границы слова в регулярном выражении
- **\{ N \}** З*аэкранированные* «фигурные» скобки, число установленных символов для сравнения в расширенном регулярном выражении
- **\;** **З***аэкранированная т*очка с запятой, завершает команду find
- *Экранирование перевода строки*, для записи многострочной команды

&

- **&>** Перенаправление `stdout` и `stderr` команды в файл
- **>&j** Перенаправление `stdout` в файловый дескриптор `j`

  >&2 Перенаправление `stdout` команды в `stderr`
- **i>&j** Перенаправление файлового дескриптора `i` в файл дескриптора `j`

  2>&1 Перенаправление `stderr` в `stdout`
- *Закрытие файловых дескрипторов*

  n<&- Закрытие ввода файлового дескриптора *n*

  0<&-, **<&-** Закрытие `stdin`

  n>&- Закрытие вывода файлового дескриптора *n*

  1>&-, **>&-** Закрытие `stdout`
- **&&** Логический оператор проверки И
- **Команда &** Запуск задания в *фоновом режиме*

Специальный символ, начало *комментария* сценария

#! Sha-bang, специальная строка начала сценария оболочки

***** **Звездочка**

- Символ подстановки, в globbing
- Любое количество символов в регулярном выражении
- ****** Возведение в степень, арифметический оператор
- ****** Оператор расширенного шаблона сравниваемого файла

% Знак процента

- По модулю, остаток арифметической операции

- Оператор удаления содержимого строки (соответствующего шаблону)

+ Знак плюс

- Символ сравнения, в расширенных регулярных выражениях
- Префикс для альтернативных параметров, в подстановке параметров
- **++** Увеличение переменной в стиле Си, в двойных скобках

* * *

Переменные оболочки

$_ Последний аргумент предыдущей команды

$- Флаги, передаваемые сценарию, с помощью ***set***

$! ID процесса последнего задания в фоновом режиме

$? Статус выхода команды

$@ Все позиционные параметры, как отдельные слова

$* Все позиционные параметры, как единое слово

$$ ID процесса сценария

$# Количество аргументов, передаваемых в функцию или в сам сценарий

$0 Имя файла сценария

$1 Первый аргумент, передаваемый сценарию

$9 Девятый аргумент, передаваемый сценарию

Таблица переменных оболочки

* * * * * *

-a Логическое И в составе проверки сравнения

База данных адресов, пример сценария

Advanced Bash Scripting Guide, где загрузить

Alias

- *Удаление alias, с помощью unalias*

Аннаграммы

And list

- Чтобы поставить по умолчанию аргумент командной строки

&& Логический оператор И

*Заэкранированные у*гловые скобки, **`\< ... \>`** граница слова в регулярном выражении

Анонимный ***here document***, с помощью **`:`** (двоеточия)

Архивирование

- `rpm`
- `tar`

Арифметическое расширение

- Из статуса выхода
- Из другого

Арифметические операторы

- Комбинация операторов, Си-стиль

  `+= -= *= /= %=`

  

  В некоторых контекстах **`+=`** может также функционировать в качестве оператора объединения строк.

Массивы

- Ассоциативные массивы - более эффективны, чем обычные массивы
- Нотация скобок
- Конкатенация, *пример сценария*
- Копирование
- Объявление

  `declare -a array_name`

- Встроенные массивы
- Пустые массивы, пустые элементы, *пример сценария*
- Косвенные ссылки
- Инициализация

  `array=( element1 element2 ... elementN)`

  Пример сценария

  Использование подстановки команд

- Загрузка файла в массив
- Многомерность, имитация

- Размещение и вложение
- Запись и использование
- Количество элементов в массиве

  `${#array_name[@]}`

  `${#array_name[*]}`
- Действия
- Передача массива функции
- Возвращение значения из функции
- Особые свойства, *пример сценария*
- Операции со строками, *пример сценария*
- Удаление элементов массива с помощью ***unset***

Ключи массивов, обнаружение

ASCII

- Определение
- Сценарии для создания таблиц ASCII

`awk`, ориентированный на поля язык обработки текста

- `rand()`, случайная функция
- Манипуляции со строками
- Использование ***`export`*** для передачи переменной во встроенный сценарий awk

* * *

Подсветка, настройка яркости

Обратные кавычки, использование в подстановке команды

Преобразование основания, *пример сценария*

Bash

- Плохая практика создания сценариев
- Обзор основ, *пример сценария*
- Опции командной строки

  Таблица
- Особенности, которых не хватает классическому Bourne shell

Квадратные скобки, **[ ]**

- Элемент *массива*
- Заключенный набор символов, для соответствия *регулярному выражению*
- Конструкция проверки (***test***)

Фигурные скобки, **{}**, используемые в

- Блок кода
- ***find***
- Расширенное регулярное выражение
- Позиционные параметры
- ***xargs***

break команда управления циклом

- Параметр (необязательный)

Команды *встроенные* (***Builtins***) в Bash

- Подпроцесс не являющийся форком

* * *

Конструкция ***case***

- Параметры командной строки, обработка
- Постановка (***Globbing***), с фильтрацией строк

cat, конкатенация файла(ов)

- Злоупотребление
- Сценарии ***cat***
- Менее эффективно, чем перенаправление `stdin`
- Туннелирование вывода в ***read***
- Использование

Символьные устройства

- Проверка ***for***

Контрольная сумма

Дочерний процесс (процесс потомок)

Двоеточие, **`:`** ,эквивалент **`true`** для встроенных команд Bash

Расцвечивание сценариев

- Цикличность цветов фона, пример сценария
- Таблица цветов escape-последовательностей
- Шаблон, цветной текст на цветном фоне

Оператор ***запятая***, ссылки на команды или опции

Опции командной строки

command_not_found_handle () *builtin* функция обработки ошибок (версия Bash 4+)

Подстановка команд

- **`$( ... )`**, предпочтительная нотация
- *Обратные кавычки*
- Расширенный набор инструментов Bash
- *Вызов подоболочки*
- Вложение
- Удаление конечных символов перевода строки
- Присваивание переменной из вывода цикла
- Разбиение слова

Комментирование заголовков, специальное назначение

Комментирование блоков кода

- Применение *анонимного* ***here document***
- Применение конструкции ***if-then***

Соединения и хосты

Составные операторы сравнения

Утилиты сжатия

- `bzip2`
- `compress`
- `gzip`
- `zip`

`continue` команда управления циклом

Символы управления

- `Control-C`, *прерывание*
- `Control-D`, прекращение / выход/ удаление (стирание)
- `Control-G`, **`BEL`** (*звуковой сигнал*)
- `Control-H`, *уничтожение*
- `Control-J`, *новая строка*
- `Control-M`, возврат каретки

Сопроцессы

`cron`, *служба (демон) планирования*

Синтаксис в стиле Си, для обработки переменных

Решатель кроссвордов

Шифрование

Фигурные скобки `{}`

- в команде *`find`*
- в расширенных *регулярных выражениях*
- в *`xargs`*

* * *

Службы, в ОС типа UNIX

`date`

`dc`, утилита для вычислений

`dd`, команда *data duplicator*

- Конвертация
- Копирование необработанных файлов на/с оборудования
- Удаление файлов, *безопасное*
- Нажатие клавиш , захват
- Опции
- Случайный доступ к потоку данных
- *Raspberry Pi*, сценарий для подготовки загрузочной SD карты
- Swap-файлы, инициализация
- *www.linuxquestions.org*

Сценарии для отладки

- Инструменты
- Ловушки на выходе
- Ловушки сигналов

Десятичное число, интерпретация Bash чисел

`declare` встроенная

- опции

Параметры по умолчанию

Директория `/dev`

- `/dev/null` файл псевдо оборудования
- `/dev/urandom` файл псевдо оборудования, для создания псевдослучайных чисел
- `/dev/zero`, файл псевдо оборудования

Файлы оборудования

dialog, утилита для создания диалоговых окон в сценарии

`$DIRSTACK` *стек директории*

Отключенные команды, в *restricted оболочках*

`do` ключевое слово (*keyword*), начинает выполнение команд в цикле

`done` ключевое слово (*keyword)*, завершение выполнения команд в цикле

Пакетные файлы *DOS*, конвертация в сценариях shell

Команды *DOS*, эквивалентность UNIX (**таблица**)

файлы с точкой, "скрытые" установочные и конфигурационные файлы

Двойные скобки **`[[ ... ]]`** конструкции `test`

- и оценка *octal/hex* констант

Двойные круглые скобки **`(( ... ))`** конструкция расширения/оценки

Двойные кавычки **" ... "** *слабые* кавычки

- *Символ обратного слэша* (\) в двойных кавычках

Двойной межстрочный интервал текстового файла, с использованием ***sed***

* * *

`-e` Проверка наличия файла

`echo`

- Передача команд далее по каналу (туннелю)
- Присваивание переменной с помощью подстановки команд
- `/bin/echo`, внешняя команда *`echo`*

`elif`, Сокращение от *`else`* и *`if`*

`else`

Шифрование файлов, с помощью ***openssl***

`esac`, ключевое слово завершающее конструкцию *`case`*

Переменные среды

`-eq` , *равно* сравнение целых чисел в проверке

Сито Эратосфена, алгоритм создания простых чисел

Экранированные символы, особый смысл

- В расширении строки `$' ... '`
- При использовании символов *Unicode*

`/etc/fstab` (монтирование файловой системы) файл

`/etc/passwd` (аккаунт пользователя) файл

`$EUID`, Действующий *ID пользователя*

`eval`, Комбинирование и оценка выражения(ий), расширения переменной

- Отключение, *Пример сценария*
- Принудительная переоценка аргументов
- Внутренние ссылки
- Опасность использования
- Использование *`eval`* для конвертации элементов *массива* в список команд
- Использование *`eval`* для выбора из переменных

Оценка констант *octal/hex в* **`[[ ... ]]`**

`exec` команда, использование в перенаправлении

Упражнения

Выход и Статус выхода

- `exit` команда
- Статус выхода (*код выхода*, значение возвращаемое командой)

  Таблица, Коды выхода с особым смыслом

  Аномалии

  Вне диапазона

  Статус выхода канала

  Определенное *возвращение функции*

  Успешно, **`0`**

  `/usr/include/sysexits.h`, список обычных кодов выхода системных файлов C/C++

`Export`, делает доступными переменные дочерним процессам

- Передача переменной во встроенный сценарий ***awk***

`expr`, Оценка выражения

- Извлечение ***Substring***
- Индекс ***Substring*** (числовая позиция в строке)
- Совпадение ***Substring***

Расширенные *регулярные выражения*

- **`?`** (знак вопроса) соответствие нулю / один символ
- **`( ... )`**Группа выражений
- **`\{ N \}`** "Фигурные" скобки, *экранированные*, количество символов для совпадения
- **`+`** *Символ совпадения*

* * *

`factor`, раскладывает целые числа на просты множители

- Приложение: Создание простых чисел

`false`, возвращение неуспешного статуса выхода (`1`)

Поле, группа символов содержащая данные

Файлы / Архивация

Файловые дескрипторы

- Закрытие

n<&- Закрывает входной дескриптор файла *n*

0<&-, <&- Закрывает `stdin`

n>&- Закрывает выходной дескриптор файла *n*

1>&-, >&- Закрывает `stdout`

- Файлы обрабатываемые *C*, сходство

Шифрование файла

`find`

- **{}** Фигурные скобки
- **\;** Заэкранированная точка с запятой

Фильтр

- Использование — с утилитой обработки файлов в качестве фильтра
- Передача вывода фильтра обратно в тот же фильтр

Числа с плавающей точкой, Bash не поддерживает

`fold`, фильтр для обертки строк текста

Создание еще дочернего процесса

Циклы ***for***

Функции

- Аргументы переданные указанной поцией
- Захват возвращаемого значения функции с помощью *`echo`*
- *Запятая*, как имя функции
- Определение должно предшествовать первому вызову функции
- Статус выхода
- Локальные переменные

  и рекурсия
- Передача массива в функцию
- Передача указателей в функцию
- Позиционные параметры
- Рекурсия
- Перенаправление `stdin` функции

- Возвращение

  Несколько возвращаемых значений из функции, пример сценария

  Возвращение массива из функции

  Ограничения возвращаемого значения, обход
- Сдвиг аргументов передаваемых в функцию
- Необычные имена функций

* * *

Игры и развлечения

- Анаграммы
- Анаграммы, снова
- Создание выигрышных чисел
- Решатель кроссвордов
- Шифрование котировок
- Раздача колоды карт
- Пятнашки
- Скачки
- Рыцарский турнир
- Игра "Life"
- Волшебные квадраты
- Сценарий музыкального проигрывателя
- Nim
- Pachinko
- Perquackey
- Лепестки Розы
- Подкаст
- Стихи
- Создание речи
- Ханойские пагоды

- Подстановка команд
- Сценарий ***ex***
- *Функция*, внедрение в нее
- ***Here strings***

  Вычисление Золотого сечения

  Текстовое начало

  Как `stdin` *цикла*

  Использование ***`read`***
- ***Limit string***

  `!` как *limit string*

  Закрытая *limit string* не может иметь отступа

  Dash опции в *limit string*, `<<-LimitString`
- Побуквенный вывод текста, для генерирования кода программы
- Подстановка параметра

  Отключение подстановки параметра
- Передача параметров
- Временные файлы
- Использование ***vi*** не интерактивно

История команд

`$HOME`, *домашняя директория пользователя*

Решатель домашних заданий

`$HOSTNAME`, *системное имя хоста*

* * *

Параметр `$Id`, в *rcs* (Revision Control System)

`if [ condition ]; then` ... конструкция проверки

- **`if-grep`**, ***if*** и ***grep*** в комбинации

  Исправления для проверки ***if-grep***

`$IFS`, Переменная *Internal field separator*

- Умолчания *для пробелов*

Операторы сравнения целых чисел

`in`, *ключевое слово* предшествующее `[list]` в цикле ***for***

Таблица инициализации, `/etc/inittab`

Встроенные группы, т.е блок кода

Интерактивный сценарий, проверка ***for***

Перенаправление I/O

Внутренние ссылки переменных

- Новая нотация, представленная в версии 2 Bash (пример сценария)

`iptables`, утилита пакетной фильтрации и защиты

- Пример использования
- Пример сценария

Итерации (повторения)

* * *

ID заданий, таблица

`jot`, эмиссия последовательности целых чисел. Эквивалент `seq`.

- Создание случайной последовательности

* * *

Ключевые слова

- ***error, if missing***

`kill`, прекращение процесса по ID процесса

- Опции (`-l`, `-9`)

`killall`, прекращение процесса по *имени*

Сценарий killall в `/etc/rc.d/init.d`

* * *

`lastpipe` опция оболочки

`-le` , *меньше чем или равно* сравнение целых чисел в проверке

`let,` присвоение и выполнение арифметических операций с переменными

- *Операторы увеличения и уменьшения в стиле Си*

Limit string, в ***here document***

`$LINENO`, переменная, указывающее *номер строки*, где она появляется в сценарии

Ссылка, файл (применение команды *`ln`*)

- Вызов сценария с несколькими именами, используя *`ln`*
- Символьные ссылки, *`ln -s`*

Конструкция ***List***

- ***And*** *list*
- ***Or*** *list*

Локальные переменные

- и рекурсия

Локализация

Логические операторы (&&, `||`, etc.)

Фал выхода, `~/.bash_logout` file

Оборудование ***Loopback***, монтирование файла к блочному оборудованию

Циклы

- **`break`** команда управления циклом
- **`continue`** команда управления циклом
- Цикл в двойных скобках в стиле *Cu*

  Цикл ***for***

  Цикл ***while***
- **`do`** (ключевое слово), начинает выполнение команд в цикле
- **`done`** (ключевое слово), завершает цикл
- Циклы ***for***

  `for` `arg` *`in`* `[list]`; *`do`*

  Подстановка команд при создании `[list]`

  Расширение имен файлов в `[list]`

  Несколько параметров в каждом элементе `[list]`

  Опускание `[list]`, умолчания для позиционных параметров

  Параметризация `[list]`

  Перенаправление

- `in`, (ключевое слово) предшествующее `[list]` в цикле ***for***
- Вложенные циклы
- Запуск цикла в *фоновом режиме*, пример сценария
- Необходимость точки с запятой, когда `do` находится на первой строке цикла

  Цикл ***for***

  Цикл ***while***
- Цикл ***until***

  `until [ condition-is-true ]; do`
- Цикл ***while***

  `while [ condition ]; do`

  Функция вызываемая из скобок проверки

  Несколько условий

  Опускание *скобок проверки*

  Перенаправление

  Конструкция `while read`
- Какой тип цикла использовать

Оборудование ***Loopback***

- Директория `/dev`
- Монтирование образа ISO

`-lt` , *меньше чем* сравнение целых чисел в проверке

* * *

`m4`, макропроцессорный язык

`$MACHTYPE`, *Тип машины*

Волшебные числа, метка заголовка файла показывающая его тип

`Makefile`, файл содержащий список зависимостей использующихся командой `make`

`man`, *справочные страницы* (просмотр)

- *Редактор справочных страниц* (сценарий)

`mapfile` встроенная команда, загружает массив текстового файла

Команды вычислений

Метасмысл

Сценарий для изучения кода Морзе

`Modulo`, оператор остатка вычисления

- Приложение: Создание простых чисел

Вычисления платежей, *пример сценария*

* * *

`-n` Проверка, что строка не ***null***

Именованный канал, временный буфер FIFO

- *Пример сценария*

`nc`, ***netcat***, набор сетевых утилит для портов TCP и UDP

`-ne`, *не равно* сравнение целых чисел в проверке

Оператор отрицания, **`!`**, реверсирует проверку

`netstat`, статистика сети

Сетевое программирование

`nl`, фильтр количества строк текста

`Noclobber`, с опцией `-C` в Bash предваряет перезапись файла

Логический оператор ***`НЕ`***, **`!`**

Переменная с присвоенным *`null`*, пустая

* * *

`-o` Логическое ИЛИ объединяющее сравнение проверки

Помутнение

- ***Запятая*** как имя функции
- Домашнее задание

`octal`, числа по основанию `8`

`od`, *восьмеричное содержимое*

`$OLDPWD` Предыдущая рабочая директория

`openssl` утилита шифрования

Оператор

- Значение
- Старшинство

Опции, переданные оболочке или сценарию из командной строки или указанной командой

Or *list*

Логический оператор ИЛИ, **||**

* * *

Подстановка параметров

- *`${parameter+alt_value}`*

  `${parameter:+alt_value}`

  Другие значения параметров, если установлены
- *`${parameter-default}`*

  `${parameter:-default}`

  `${parameter=default}`

  `${parameter:=default}`

  Параметры по умолчанию
- *`${!varprefix*}`*

  `${!varprefix@}`

  Параметр соответствия *имени*
- *`${parameter?err_msg}`*

  Параметр без сообщения об ошибке
- *`${parameter}`*

  Значение *параметра*
- Изменения в `Case` (врсия 4+ Bash).
- *Пример сценария*
- ***Таблица*** *подстановки параметров*

Проблемы родительского/дочернего процесса, дочерний процес не экспортирует переменные в родительский процесс

Круглые скобки

- Группа команд

- Заключенная в скобки группа *расширенных регулярных выражений*
- Двойные круглые скобки, в расширении вычислений

`$PATH`, *путь* (расположение системных двоичных файлов)

- Добавление директорий в `$PATH` с помощью оператора **`+=`** .

Путь к имени файла, `filename` включает в себя полный путь к заданному файлу.

- Проверка пути к имени файла (***pathnames***)

Perl, язык программирования

- комбинирование файлов со сценарием *Bash*
- Включение в сценарий *Bash*

Perquackey — тип аннаграммной игры (сценарий *Quackey*)

Лепестки Розы

PID, *ID процесса*, идентификатор обозначающий запущенный процесс.

Канал (туннель), **`|`** , служит для передачи вывода команды другой команде или оболочке

- Избегание ненужных команд в *канале*
- Встроенные *комментарии*
- Статус выхода канала
- **`Pipefail`**, опция *`set -o pipefail`* показывает статус выхода команд канала
- `$PIPESTATUS`, *статус выхода* последней выполненной команды в канале
- Передача каналом вывода команды сценарию
- Перенаправление `stdin`, вместо использования в канале `cat`

Ловушки

- **`-`** (тире) не является оператором перенаправления
- **`//`** (двойной слэш), поведение команды **`cd`**
- заголовок сценария **`#!/bin/sh`** отключает внешние черты *Bash*
- Злоупотребление ***`cat`***
- *CGI* программирование, с помощью сценариев ***for***
- Закрытая ***limit string*** в ***here document***, отступление
- Перевод строки типа DOS (**`\r\n`**) обрушивает сценарий

- *Заключенный в двойные кавычки символ обратного слэша* (\)
- **eval**, опасность применения
- Недостаток прав на запуск команд в сценарии
- *Статус выхода*, аномалии
- *Статус выхода* арифметического выражения *не* эквивалентно *коду ошибки code*
- Проблемы ***Export***, из процесса потомка в родительский процесс
- Не включенные расширения *Bash*
- Ошибочное заключение в кавычки переменных в скобках проверки
- *GNU* команда **set**, в кросс-платформенных сценариях
- Не правильное использование ***let***: попытка установки строковых переменных
- Несколько объявлений **echo** в функции у которой вывод захватывается
- Объявление переменной ***null***
- Числовые и строковые операторы сравнения не эквивалентны

  = и **-eq** не взаимозаменяемы
- Пропуск точки с запятой терминала, в блоке кода в фигурных скобках
- Передача

  echo в цикле

  echo в ***read*** (тем не менее , эта проблема может быть преодолена)

  tail `-f` в ***grep***
- Пробел перед переменной, непреднамеренные последствия
- Команды с ***suid*** внутри сценария
- Не задокументированные особенности *Bash*, опасность
- Обновление в *Bash* не работающих старых сценариев
- Не инициализированные переменные
- Имена переменных, не подходящие
- Переменные в *subshell*, ограничение сферы применения
- *Subshell* в цикле ***while-read***
- Пробел, ошибки

Указатели

- Файловых дескрипторов
- Функций
- Внутренних ссылок
- Переменных

Переносимость сценариев оболочки

- Установка *`path`* и *`umask`*
- Сценарий *test suite* (Bash версия классического Bourne shell)
- Применение *`whatis`*

Позиционные параметры

- **`$@`**, как *отдельные* слова
- **`$*`**, как *единое* слово
- в функциях

POSIX, *Portable Operating System Interface / UNIX*

- опция `--posix`
- стандарт 1003.2
- Символы классов

`$PPID`, *ID* родительского процесса

Внеочередной, оператор

Предварение строк заголовка файла, *пример сценария*

Простые числа

- Создание простых чисел с помощью команды ***`factor`***
- Создание простых чисел с помощью оператора ***`modulo`***
- Сито Эратосфена, пример сценария

Команда **`printf`**, *форматированный вывод*

Директория `/proc`

- Запуск процессов, описание файлов
- Запись файлов в `/proc`, *опасность*

Процесс

- Дочерний процесс (потомок)
- Родительский процесс
- **ID** процесса (***PID***)

Процесс подстановки

- Сравнение содержимого директорий
- Передача `stdin` команды
- Временно
- Цикл ***while*-*read*** вне *subshell*

Программируемое завершение (расширение табуляции)

Строка приглашения

- `$PS1`, *Главное приглашение*, видимое в командной строке
- `$PS2`, Вторичное приглашение

Псевдо-код, как способ решения проблем

`$PWD`, Текущая рабочая директория

* * *

Quackey, анаграммная игра типа *Perquackey* (сценарий)

Знак вопроса, **?**

- Символ соответствия *в расширенном регулярном выражении*
- *Один символ подстановки* в ***globbing***
- Тройной оператор стиля *Си*

Заключение в кавычки

- Символ строки
- Переменные

  в скобках проверки
- *Пробел*, использование заключение в кавычки для сохранения

* * *

Случайные числа

- `/dev/urandom`

- ***exec*** **<filename**,

  переназначение файловых дескрипторов
- Объяснение начального уровня *перенаправления I/O*
- *Открытие файла* сразу для чтения и для записи

  <>filename
- Ввод в *read* перенаправления из файла
- stderr в stdout

  2>&1
- stdin / stdout, используя -
- stdin *функции*
- stdout в файл

  > ... >>
- stdout *в файловый дескриптор j*

  >&j
- Файловый дескриптор i *в файловый дескриптор j*

  i>&j
- stdout команды в stderr

  >&2
- stdout *и* stderr команды в файл

  &>
- tee, перенаправление в файл вывода команды через канал

Ссылки карт

- Различные конструкции
- Параметр подстановки/расширения
- Специальные переменные оболочки
- Строковые операции
- Операторы проверки

  Двоичное сравнение

  Файлы

Регулярные выражения

- **^** (каретка) Начало строки
- **$** (знак доллара) *Якорь*
- **.** (точка) Соответствие одного символа
- ***** (звездочка) Любое число символов
- **[ ]** (скобки) Заключают набор символов для соответствия
- **** (обратный слэш) Экранирование, интерпретация следующего символа буквально
- **\< ... \>** (Угловые скобки, экранированные) отделяют слово
- Расширенные регулярные выражения

  + *Символ соответствия*

  \{ \} Заэкранированные "фигурные" скобки

  [: :] символ классов POSIX

`$REPLY`, Значение по умолчанию команды `read`

Ограничение оболочки, оболочка (или сценарий) с некоторыми отключенными командами

`return`, команда завершающая функцию

`run-parts`

- Запуск сценария последовательно, без вмешательства пользователя

* * *

Объем переменной, определение

Опции сценария, присвоение из командной строки

Процедуры сценариев , библиотека полезных определений и функций

Вторичное приглашение, **`$PS2`**

Проблемы безопасности

- `nmap`, *карта сети* / сканер портов
- `sudo`
- Команды *`suid`* внутри сценария
- Вирусы, трояны и черви в сценарии
- Написание безопасных сценариев

sed, шаблонный язык программирования

- **Таблица**, основные операторы
- **Таблица**, примеры операторов

`select`, конструкция для создания меню

- **`in` *`list`*** опущены

Семафор

Необходимость точки с запятой, когда ключевое слово `do` находится на первой строке цикла

- Где оканчивается блок кода в фигурных скобках

`seq`, Выделение последовательности целых чисел. Эквивалент `jot`.

`set`, Изменений значений переменных внутри сценария

- `set -u`, Прерывание сценария с сообщением об ошибке, при попытке использовать *не объявленную* переменную.

Сценарий оболочки, определение

Обертка оболочки, сценарий встраивания команды или утилиты

`shift`, Переназначение *позиционных параметров*

`$SHLVL`, У*ровень оболочки*, глубина вложенности оболочки (или сценария)

`shopt`, *Изменение опций оболочки*

Сигнал, сообщение посылаемое процессу

Моделирование

- Броуновское движение
- Доска Гальтона
- Скачки
- *Life*, Игра
- PI, Приближенное значение стрельбы пушечными ядрами
- *Стек* ***Pushdown***

Одинарные кавычки (`' ... '`), жесткие кавычки

Сокет, Узел связи с портом I/O

Сортировка

- Сортировка пузырьков
- Сортировка вставляемого

`source`, выполнить сценарий или, в сценарии, импортировать файл

- Передача позиционных параметров

Спам, имеющие с ним дело

- *Пример сценария*
- *Пример сценария*
- *Пример сценария*
- *Пример сценария*

Специальные символы

Стек

- Определение
- Эмуляция стека *push-down*, пример сценария

Стандартное отклонение, пример сценария

Файлы запуска, Bash

`stdin` и `stdout`

Секундомер, пример сценария

Строки

- **`=~`** Оператор соответствия строки
- Сравнение
- Размер (длина)

  `${#string}`
- Управление
- Управление, с помощью ***awk***
- *Строка* ***Null***, проверка
- Защита строк от расширения и/или переинтерпретации, *пример сценария*

  Незащищенные строки, *пример сценария*
- *strchr()*, эквивалент
- *strlen()*, эквивалент
- Команда `strings` , поиск выводимых строк в двоичном файле или файле данных
- Извлечение *Substring*

`${string:position}`

`${string:position:length}`

Использование *`expr`*

- Индекс *Substring* (числовая позиция в строке)
- Соответствие *Substring*, с помощью `expr`
- Удаление *Substring*

  `${var#Pattern}`

  `${var##Pattern}`

  `${var%Pattern}`

  `${var%%Pattern}`
- Замена *Substring*

  `${string/substring/replacement}`

  `${string//substring/replacement}`

  `${string/#substring/replacement}`

  `${string/%substring/replacement}`

  Пример сценария
- **Таблица** управления *string/substring* и операторы извлечения

Жесткие кавычки `' ... '`

Таблица стилей для написания сценариев

Подоболочка

- Список команд заключенных в кавычки
- Переменные, `$BASH_SUBSHELL` и `$SHLVL`
- Переменные в *subshell*

  Область ограниченная, но...

  ... может осуществляться за пределами subshell?

`su` *Substitute user, вход, как другой пользователь или как администратор*

`suid` (*set user id*) флаг файла

- Команды с *`suid`* внутри сценария, не рекомендуется

Символьные ссылки

Swap-файлы

* * *

Завершение клавишей ***TAB***

Таблица подстановок, пример сценария

`tail`, Вывод в `stdout` строк от конца текстового файла

`tar`, Утилита архивации

`tee`, Перенаправление вывода команд(ы) в файл через *канал*

Терминалы

- `setserial`
- `setterm`
- `stty`
- `tput`
- `wall`

Команда ***`test`***

- Встроенная в Bash (*builtin)*
- Внешняя команда, `/usr/bin/test` (эквивалент `/usr/bin/[`)

Конструкции проверки

Операторы проверки

- **`-a`** Логическое И составного сравнения
- **`-e`** Наличие файла
- **`-eq`** Равно (сравнение целых чисел)
- **`-f`** Файл является *обычным* файлом
- **`-ge`** Больше чем или равно (сравнение целых чисел)
- **`-gt`** Больше чем (сравнение целых чисел)
- **`-le`** Меньше чем или равно (сравнение целых чисел)
- **`-lt`** Меньше чем (сравнение целых чисел)
- **`-n`** Не нулевой длины (сравнение строк)
- **`-ne`** Не равно (сравнение целых чисел)

- **`-o`** Логическое ИЛИ составного сравнения
- **`-u`** Наличие флага *suid*, проверка файла
- **`-z`** Нулевой длины (сравнение строк)
- **`=`** Равно (сравнение строк)

  `==` Равно (сравнение строк)
- **`<`** Меньше чем (сравнение строк)
- **`<`** Меньше чем, (сравнение целых чисел, в двойных круглых скобках)
- **`<=`** Меньше чем или равно, (сравнение целых чисел, *в двойных круглых скобках*)
- **`>`** Больше чем (сравнение строк)
- **`>`** Больше чем, (сравнение целых чисел, *в двойных круглых скобках*)
- **`>=`** Больше чем или равно, (сравнение целых чисел, *в двойных круглых скобках*)
- **`||`** Логическое ИЛИ
- **`&&`** Логическое И
- **`!`** Оператор отрицания, инвертирует статус выхода проверки

  `!=` не равно (сравнение строк)
- ***Таблица*** операторов проверки

  Бинарное сравнение

  Файл

Обработка текста и текстовых файлов

Время / Дата

Время для ввода

- С помощью *`read -t`*
- С помощью *`stty`*
- С помощью цикла устанавливающего время
- С помощью `$TMOUT`

Советы и подсказки для сценариев Bash

- Массивы, как возвратить значение из функции

  Ассоциативный массив более эффективен, чем число-индексированный массив

- Захват возвращаемого значения функции, с помощью *echo*
- *Программирование CGI*, использование для сценариев
- Блоки комментариев

  Используя анонимный ***here documents***

  С помощью конструкций ***if-then***
- Заголовки комментариев, особое назначения
- *Синтаксис стиля Си*, для управления переменными
- Двойной междустрочный интервал текстового файла
- Удаление тире префиксов имен файлов
- Фильтр, передача вывода на тот же самый фильтр
- Обходные пути возвращаемого значения функции
- Исправление проверки ***if-grep***
- Библиотека *функций* и зависимостей
- Избежание присваивания переменным значения ***null***
- Передача *массива* функции
- $PATH, добавление, с помощью оператора **+=** .
- *Добавление* строк в заголовок файла
- Шаблон шкалы выполнения
- Псевдо-код
- rcs
- Перенаправление test в /dev /null для подавления вывода
- Запуск сценариев последовательно, без вмешательства пользователя, с помощью run-parts
- Сценарий как встроенная команда
- *Переносимость* сценария

  Установка *path* и *umask*

  Использование *whatis*
- Установка переменной сценария в блок встроенного кода *sed* или *awk*
- Ускорение выполнения сценария, *отключив Юникод*

- Переменная *Subshell*, доступ из вне *subshell*
- Проверка переменной, чтобы увидеть, содержит ли она только цифры
- Проверка наличия команды, используя `type`
- Отслеживание использования сценария
- Цикл ***while-read*** вне *subshell*
- `Widgets`, вызов из сценария

`$TMOUT`, Время перерыва (ожидания)

Токен, символ для расширения ключевого слова (keyword) или команды

`tput`, Команда управления терминалом

`tr`, Фильтр преобразования символов

- Преобразование текстовых файлов DOS в Unix
- Опции
- Soundex, *пример сценария*
- Варианты

`Trap`, Определение действий при получении сигнала

Тройной оператор, стиль *Cи*, **`var>10?88:99`**

- в конструкции двойных круглых скобок
- в конструкции *`let`*

`true`, *Возвращение успешного* (`0`) статуса выхода

`typeset` встроенная

- Опции

* * *

`$UID`, Число ID пользователя

`unalias`, удаление псевдонима

`uname`, вывод системной информации

Unicode, стандарт кодировки для представления букв и символов

- Отключение *unicode* для оптимизации сценария

Не инициализированные переменные

`uniq`, фильтр для удаления повторяющихся строк из отсортированного файла

`unset`, удаление переменной оболочки

Цикл ***until***

```
until [ condition-is-true ]; do
```

* * *

Переменные

- Операции с массивами
- Присвоение

  Пример сценария

  Пример сценария

  Пример сценария
- Внутренняя переменная *Bash*
- Установка переменных в блоки кода ***sed*** или ***awk***
- Увеличение/уменьшение/тройные операции стиля *Cu*
- Изменение значения переменных внутри сценария с помощью `set`
- `declare`, изменение свойств переменных
- Удаление переменной оболочки с помощью `unset`
- Окружение
- Операторы расширения/замены *Substring*
- Косвенные ссылки

  ```
  eval variable1=\$$variable2
  ```

  Новая нотация

  ```
  ${!variable}
  ```
- Целое число
- Целое число / строка (не типизированные переменные)
- Размер (длина)

  ```
  ${#var}
  ```
- `Lvalue`
- Управление и расширение

- Различия между *именем* и *значением* переменной
- Строка ***Null***, проверка
- Избежание присваиванию переменным значения ***null***
- Заключение в кавычки

  в скобки `test`

  для сохранения пробелов
- `rvalue`
- Установка значения *null*
- ***subshell*** не видимая для родительской оболочки
- Проверка переменной на содержание только чисел T
- Ввод, ограничивающий свойства переменной
- Не объявленность, сообщение об ошибке
- Не объявленность
- Не заключенные в кавычки переменные, *splitting*
- Не установленность
- Не вводимость

* * *

`wait`, приостановка выполнения сценария

- Исправление зависания сценария

Мягкие кавычки " **...** "

Цикл ***while***

`while [ condition ]; do`

- Синтаксис стиля Си
- Вызов `function` в скобках `test`
- Несколько условий
- Опускание скобок `test`
- Конструкция ***while read***

  Избегание ***subshell***

Символы пробелов, пропусков, табуляций, и перевода строк

- `$IFS` по умолчанию
- Ненадлежащее использование
- Закрытие предыдущего ***limit string*** в ***here document***, ошибка
- Комментарии в начале сценария
- *Заключение в кавычки*, сохранение пробелов в строках или переменных
- Символ класса *POSIX* `[:space:]`

`who`, информация о вошедших пользователях

- `w`
- `whoame`
- `logname`

Виджеты

Специальные символы групповых подстановок

- Звездочка `*`
- Конструкция `In [list]`
- Знак вопроса `?`
- Не соответствие файлам с точкой `(dot files)`

Разделение слова

- Определение
- Результат *команды подстановки*

Обертка, оболочка

* * *

`xargs`, Фильтр для группировки аргументов

- Фигурные скобки
- Ограничение переданных аргументов
- Опции
- Аргументы процессов по одному
- Пробел , обработка

* * *

`yes`

- Эмуляция

* * *

`-z` Строка ***null***

Зомби, процесс, который имеет прекращен, но еще не убит его родитель