

Advanced Bash-Scripting Guide

Искусство программирования на языке сценариев командной оболочки

Автор: Mendel Cooper

<thegrendel@theriver.com>

Перевод: Андрей Киселев

<kis_an@mail.ru>

Данное руководство не предполагает наличие у читателя познаний в области программирования на языке сценариев, однако, быстро восполняет этот недостаток . . . *постепенно, шаг за шагом раскрывая мудрость и красоту UNIX*. Это руководство может рассматриваться как учебник, предназначенный для самостоятельного изучения или как справочник по программированию на shell. Руководство снабжено серией хорошо прокомментированных примеров, поскольку лучший путь к изучению языка сценариев -- это написание сценариев.

Последнюю версию документа, в виде [.bz2](#) архива, содержащем исходные тексты в формате SGML и HTML, вы найдете на [домашней страничке автора](#). Там же вы найдете и [change log](#).

Посвящения

Посвящается Аните -- источнику очарования

Содержание

Часть 1. [Введение](#)

1. [Зачем необходимо знание языка Shell?](#)
2. [Для начала о Sha-Bang](#)
 - 2.1. [Запуск сценария](#)
 - 2.2. [Упражнения](#)

Часть 2. [Основы](#)

3. [Служебные символы](#)
4. [Переменные и параметры. Введение.](#)
 - 4.1. [Подстановка переменных](#)
 - 4.2. [Присваивание значений переменным](#)
 - 4.3. [Переменные Bash не имеют типа](#)

- 4.4. [Специальные типы переменных](#)
- 5. [Кавычки](#)
- 6. [Завершение и код завершения](#)
- 7. [Проверка условий](#)
 - 7.1. [Конструкции проверки условий](#)
 - 7.2. [Операции проверки файлов](#)
 - 7.3. [Операции сравнения](#)
 - 7.4. [Вложенные условные операторы if/then](#)
 - 7.5. [Проверка степени усвоения материала](#)
- 8. [Операции и смежные темы](#)
 - 8.1. [Операторы](#)
 - 8.2. [Числовые константы](#)
- Часть 3. [Углубленный материал](#)
 - 9. [К вопросу о переменных](#)
 - 9.1. [Внутренние переменные](#)
 - 9.2. [Работа со строками](#)
 - 9.3. [Подстановка параметров](#)
 - 9.4. [Объявление переменных: **declare** и **typeset**](#)
 - 9.5. [Косвенные ссылки на переменные](#)
 - 9.6. [\\$RANDOM: генерация псевдослучайных целых чисел](#)
 - 9.7. [Двойные круглые скобки](#)
 - 10. [Циклы и ветвления](#)
 - 10.1. [Циклы](#)
 - 10.2. [Вложенные циклы](#)
 - 10.3. [Управление ходом выполнения цикла](#)
 - 10.4. [Операторы выбора](#)
 - 11. [Внутренние команды](#)
 - 11.1. [Команды управления заданиями](#)
 - 12. [Внешние команды, программы и утилиты](#)
 - 12.1. [Базовые команды](#)
 - 12.2. [Более сложные команды](#)
 - 12.3. [Команды для работы с датой и временем](#)
 - 12.4. [Команды обработки текста](#)
 - 12.5. [Команды для работы с файлами и архивами](#)
 - 12.6. [Команды для работы с сетью](#)
 - 12.7. [Команды управления терминалом](#)
 - 12.8. [Команды выполнения математических операций](#)
 - 12.9. [Прочие команды](#)
 - 13. [Команды системного администрирования](#)
 - 14. [Подстановка команд](#)
 - 15. [Арифметические подстановки](#)
 - 16. [Перенаправление ввода/вывода](#)
 - 16.1. [С помощью команды **exec**](#)
 - 16.2. [Перенаправление для блоков кода](#)
 - 16.3. [Область применения](#)
 - 17. [Встроенные документы](#)
- Часть 4. [Материал повышенной сложности](#)
 - 18. [Регулярные выражения](#)
 - 18.1. [Краткое введение в регулярные выражения](#)
 - 18.2. [Globbing -- Подстановка имен файлов](#)
 - 19. [Подоболочки, или Subshells](#)
 - 20. [Ограниченный режим командной оболочки](#)
 - 21. [Подстановка процессов](#)
 - 22. [Функции](#)
 - 22.1. [Сложные функции и сложности с функциями](#)
 - 22.2. [Локальные переменные](#)

- 23. [Псевдонимы](#)
- 24. [Списки команд](#)
- 25. [Массивы](#)
- 26. [Файлы](#)
- 27. [/dev и /proc](#)
 - 27.1. [/dev](#)
 - 27.2. [/proc](#)
- 28. [/dev/zero и /dev/null](#)
- 29. [Отладка сценариев](#)
- 30. [Необязательные параметры \(ключи\)](#)
- 31. [Широко распространенные ошибки](#)
- 32. [Стиль программирования](#)
 - 32.1. [Неофициальные рекомендации по оформлению сценариев](#)
- 33. [Разное](#)
 - 33.1. [Интерактивный и неинтерактивный режим работы](#)
 - 33.2. [Сценарии-обертки](#)
 - 33.3. [Операции сравнения: Альтернативные решения](#)
 - 33.4. [Рекурсия](#)
 - 33.5. ["Цветные" сценарии](#)
 - 33.6. [Оптимизация](#)
 - 33.7. [Разные советы](#)
 - 33.8. [Проблемы безопасности](#)
 - 33.9. [Проблемы переносимости](#)
 - 33.10. [Сценарии командной оболочки под Windows](#)
- 34. [Bash, версия 2](#)
- 35. [Замечания и дополнения](#)
 - 35.1. [От автора](#)
 - 35.2. [Об авторе](#)
 - 35.3. [Инструменты, использовавшиеся при создании книги](#)
 - 35.3.1. [Аппаратура](#)
 - 35.3.2. [Программное обеспечение](#)
 - 35.4. [Благодарности](#)

[Литература](#)

- A. [Дополнительные примеры сценариев](#)
- B. [Маленький учебник по Sed и Awk](#)
 - B.1. [Sed](#)
 - B.2. [Awk](#)
- C. [Коды завершения, имеющие predeterminedный смысл](#)
- D. [Подробное введение в операции ввода-вывода и перенаправление ввода-вывода](#)
- E. [Локализация](#)
- F. [История команд](#)
- G. [Пример файла .bashrc](#)
- H. [Преобразование пакетных \(*.bat\) файлов DOS в сценарии командной оболочки](#)
- I. [Упражнения](#)
 - I.1. [Анализ сценариев](#)
 - I.2. [Создание сценариев](#)
- J. [Авторские права](#)

Перечень таблиц

- 11-1. [Идентификация заданий](#)
- 30-1. [Ключи Bash](#)
- 33-1. [Числовые значения цвета в escape-последовательностях](#)
- B-1. [Основные операции sed](#)
- B-2. [Примеры операций в sed](#)
- C-1. ["Зарезервированные" коды завершения](#)
- H-1. [Ключевые слова/переменные/операторы пакетных файлов DOS и их аналоги](#)

Перечень приложений

- 2-1. [cleanup](#): Сценарий очистки лог-файлов в /var/log
- 2-2. [cleanup](#): Расширенная версия предыдущего сценария.
- 3-1. [Вложенные блоки и перенаправление ввода-вывода](#)
- 3-2. [Сохранение результата исполнения вложенного блока в файл](#)
- 3-3. [Запуск цикла в фоновом режиме](#)
- 3-4. [Резервное архивирование всех файлов, которые были изменены в течение последних суток](#)
- 4-1. [Присваивание значений переменным и подстановка значений переменных](#)
- 4-2. [Простое присваивание](#)
- 4-3. [Присваивание значений переменным простое и замаскированное](#)
- 4-4. [Целое число или строка?](#)
- 4-5. [Позиционные параметры](#)
- 4-6. [wh, whois](#) выяснение имени домена
- 4-7. [Использование команды shift](#)
- 5-1. [Вывод "причудливых" переменных](#)
- 5-2. [Экранированные символы](#)
- 6-1. [завершение / код завершения](#)
- 6-2. [Использование символа ! для логической инверсии кода возврата](#)
- 7-1. [Что есть "истина"?](#)
- 7-2. [Эквиваленты команды test -- /usr/bin/test, \[\], и /usr/bin/\[](#)
- 7-3. [Арифметические выражения внутри \(\(\)\)](#)
- 7-4. [Проверка "битых" ссылок](#)
- 7-5. [Операции сравнения](#)
- 7-6. [Проверка -- является ли строка пустой](#)
- 7-7. [zmost](#)
- 8-1. [Наибольший общий делитель](#)
- 8-2. [Арифметические операции](#)
- 8-3. [Построение сложных условий, использующих && и ||](#)
- 8-4. [Различные представления числовых констант](#)
- 9-1. [\\$IFS и пробельные символы](#)
- 9-2. [Ограничения времени ожидания ввода](#)
- 9-3. [Еще один пример ограничения времени ожидания ввода от пользователя](#)
- 9-4. [Ограничение времени ожидания команды read](#)
- 9-5. [Я -- root?](#)
- 9-6. [arglist](#): Вывод списка аргументов с помощью переменных \$* и \$@
- 9-7. [Противоречия в переменных \\$* и \\$@](#)
- 9-8. [Содержимое \\$* и \\$@, когда переменная \\$IFS -- пуста](#)
- 9-9. [Переменная "подчеркивание"](#)
- 9-10. [Вставка пустых строк между параграфами в текстовом файле](#)
- 9-11. [Преобразование графических файлов из одного формата в другой, с изменением имени файла](#)
- 9-12. [Альтернативный способ извлечения подстрок](#)
- 9-13. [Подстановка параметров и сообщения об ошибках](#)
- 9-14. [Подстановка параметров и сообщение о "порядке использования"](#)
- 9-15. [Длина переменной](#)
- 9-16. [Поиск по шаблону в подстановке параметров](#)
- 9-17. [Изменение расширений в именах файлов:](#)
- 9-18. [Поиск по шаблону при анализе произвольных строк](#)
- 9-19. [Поиск префиксов и суффиксов с заменой по шаблону](#)
- 9-20. [Объявление переменных с помощью инструкции declare](#)
- 9-21. [Косвенные ссылки](#)
- 9-22. [Передача косвенных ссылок в awk](#)

- 9-23. [Генерация случайных чисел](#)
- 9-24. [Выбор случайной карты из колоды](#)
- 9-25. [Имитация бросания кубика с помощью RANDOM](#)
- 9-26. [Переустановка RANDOM](#)
- 9-27. [Получение псевдослучайных чисел с помощью awk](#)
- 9-28. [Работа с переменными в стиле языка C](#)
- 10-1. [Простой цикл for](#)
- 10-2. [Цикл for с двумя параметрами в каждом из элементов списка](#)
- 10-3. [Fileinfo: обработка списка файлов, находящегося в переменной](#)
- 10-4. [Обработка списка файлов в цикле for](#)
- 10-5. [Цикл for без списка аргументов](#)
- 10-6. [Создание списка аргументов в цикле for с помощью операции подстановки команд](#)
- 10-7. [grep для бинарных файлов](#)
- 10-8. [Список всех пользователей системы](#)
- 10-9. [Проверка авторства всех бинарных файлов в текущем каталоге](#)
- 10-10. [Список символических ссылок в каталоге](#)
- 10-11. [Список символических ссылок в каталоге, сохраняемый в файле](#)
- 10-12. [C-подобный синтаксис оператора цикла for](#)
- 10-13. [Работа с командой efax в пакетном режиме](#)
- 10-14. [Простой цикл while](#)
- 10-15. [Другой пример цикла while](#)
- 10-16. [Цикл while с несколькими условиями](#)
- 10-17. [C-подобный синтаксис оформления цикла while](#)
- 10-18. [Цикл until](#)
- 10-19. [Вложенный цикл](#)
- 10-20. [Команды break и continue в цикле](#)
- 10-21. [Прерывание многоуровневых циклов](#)
- 10-22. [Передача управление в начало внешнего цикла](#)
- 10-23. [Живой пример использования "continue N"](#)
- 10-24. [Использование case](#)
- 10-25. [Создание меню с помощью case](#)
- 10-26. [Оператор case допускает использовать подстановку команд вместо анализируемой переменной](#)
- 10-27. [Простой пример сравнения строк](#)
- 10-28. [Проверка ввода](#)
- 10-29. [Создание меню с помощью select](#)
- 10-30. [Создание меню с помощью select в функции](#)
- 11-1. [printf в действии](#)
- 11-2. [Ввод значений переменных с помощью read](#)
- 11-3. [Пример использования команды read без указания переменной для ввода](#)
- 11-4. [Ввод многострочного текста с помощью read](#)
- 11-5. [Обнаружение нажатия на курсорные клавиши](#)
- 11-6. [Чтение командой read из файла через перенаправление](#)
- 11-7. [Смена текущего каталога](#)
- 11-8. [Команда let, арифметические операции.](#)
- 11-9. [Демонстрация команды eval](#)
- 11-10. [Принудительное завершение сеанса](#)
- 11-11. [Шифрование по алгоритму "rot13"](#)
- 11-12. [Замена имени переменной на ее значение, в исходном тексте программы на языке Perl, с помощью eval](#)
- 11-13. [Установка значений аргументов с помощью команды set](#)
- 11-14. [Изменение значений позиционных параметров \(аргументов\)](#)
- 11-15. ["Сброс" переменной](#)
- 11-16. [Передача переменных во вложенный сценарий awk, с помощью export](#)
- 11-17. [Прием опций/аргументов, передаваемых сценарию, с помощью getopt](#)
- 11-18. ["Подключение" внешнего файла](#)

- 11-19. [Пример \(беспольный\) сценария, который подключает себя самого.](#)
- 11-20. [Команда **exec**](#)
- 11-21. [Сценарий, который запускает себя самого](#)
- 11-22. [Ожидание завершения процесса перед тем как продолжить работу](#)
- 11-23. [Сценарий, завершающий себя сам с помощью команды **kill**](#)
- 12-1. [Создание оглавления диска для записи CDR, с помощью команды **ls**](#)
- 12-2. [Badname, удаление файлов в текущем каталоге, имена которых содержат недопустимые символы и пробелы.](#)
- 12-3. [Удаление файла по его номеру **inode**](#)
- 12-4. [Использование команды **xargs** для мониторинга системного журнала](#)
- 12-5. [copydir, копирование файлов из текущего каталога в другое место, с помощью **xargs**](#)
- 12-6. [Пример работы с **expr**](#)
- 12-7. [Команда **date**](#)
- 12-8. [Частота встречаемости отдельных слов](#)
- 12-9. [Какие из файлов являются сценариями?](#)
- 12-10. [Генератор 10-значных случайных чисел](#)
- 12-11. [Мониторинг системного журнала с помощью **tail**](#)
- 12-12. [Сценарий-эмулятор "grep"](#)
- 12-13. [Поиск слов в словаре](#)
- 12-14. [toupper: Преобразование символов в верхний регистр.](#)
- 12-15. [lowercase: Изменение имен всех файлов в текущем каталоге в нижний регистр.](#)
- 12-16. [du: Преобразование текстового файла из формата DOS в формат UNIX.](#)
- 12-17. [rot13: Сверхслабое шифрование по алгоритму rot13.](#)
- 12-18. [Более "сложный" шифр](#)
- 12-19. [Отформатированный список файлов.](#)
- 12-20. [Пример форматирования списка файлов в каталоге](#)
- 12-21. [nl: Самонумерующийся сценарий.](#)
- 12-22. [Пример перемещения дерева каталогов с помощью **cpio**](#)
- 12-23. [Распаковка архива **rpm**](#)
- 12-24. [Удаление комментариев из файла с текстом программы на языке C](#)
- 12-25. [Исследование каталога /usr/X11R6/bin](#)
- 12-26. ["Расширенная" команда **strings**](#)
- 12-27. [Пример сравнения двух файлов с помощью **cmp**.](#)
- 12-28. [Утилиты **basename** и **dirname**](#)
- 12-29. [Проверка целостности файла](#)
- 12-30. [Декодирование файлов](#)
- 12-31. [Сценарий, отправляющий себя самого по электронной почте](#)
- 12-32. [Ежемесячные выплаты по займу](#)
- 12-33. [Перевод чисел из одной системы счисления в другую](#)
- 12-34. [Пример взаимодействия **bc** со "встроенным документом"](#)
- 12-35. [Вычисление числа "пи"](#)
- 12-36. [Преобразование чисел из десятичной в шестнадцатеричную систему счисления](#)
- 12-37. [Разложение числа на простые множители](#)
- 12-38. [Расчет гипотенузы прямоугольного треугольника](#)
- 12-39. [Использование **seq** для генерации списка аргументов цикла **for**](#)
- 12-40. [Использование **getopt** для разбора аргументов командной строки](#)
- 12-41. [Захват нажатых клавиш](#)
- 12-42. [Надежное удаление файла](#)
- 12-43. [Генератор имен файлов](#)
- 12-44. [Преобразование метров в мили](#)
- 12-45. [Пример работы с **m4**](#)
- 13-1. [Установка символа "забоя"](#)
- 13-2. [невидимый пароль: Отключение эхо-вывода на терминал](#)
- 13-3.
- 13-4. [Использование команды **pidof** при остановке процесса](#)

- 13-5. [Проверка образа CD](#)
- 13-6. [Создание файловой системы в обычном файле](#)
- 13-7. [Добавление нового жесткого диска](#)
- 13-8. [Сценарий **killall**, из каталога /etc/rc.d/init.d](#)
- 14-1. [Глупая выходка](#)
- 14-2. [Запись результатов выполнения цикла в переменную](#)
- 16-1. [Перенаправление `stdin` с помощью **exec**](#)
- 16-2. [Перенаправление `stdout` с помощью **exec**](#)
- 16-3. [Одновременное перенаправление устройств, `stdin` и `stdout`, с помощью команды **exec**](#)
- 16-4. [Перенаправление в цикл *while*](#)
- 16-5. [Альтернативная форма перенаправления в цикле *while*](#)
- 16-6. [Перенаправление в цикл *until*](#)
- 16-7. [Перенаправление в цикл *for*](#)
- 16-8. [Перенаправление устройств \(`stdin` и `stdout`\) в цикле *for*](#)
- 16-9. [Перенаправление в конструкции *if/then*](#)
- 16-10. [Файл с именами "names.data", для примеров выше](#)
- 16-11. [Регистрация событий](#)
- 17-1. [**dummyfile**: Создание 2-х строчного файла-заготовки](#)
- 17-2. [**broadcast**: Передача сообщения всем, работающим в системе, пользователям](#)
- 17-3. [Вывод многострочных сообщений с помощью **cat**](#)
- 17-4. [Вывод многострочных сообщений с подавлением символов табуляции](#)
- 17-5. [Встроенные документы и подстановка параметров](#)
- 17-6. [Отключение подстановки параметров](#)
- 17-7. [Передача пары файлов во входящий каталог на "Sunsite"](#)
- 17-8. [Встроенные документы и функции](#)
- 17-9. ["Анонимный" Встроенный Документ](#)
- 17-10. [Блочный комментарий](#)
- 17-11. [Встроенная справка к сценарию](#)
- 19-1. [Область видимости переменных](#)
- 19-2. [Личные настройки пользователей](#)
- 19-3. [Запуск нескольких процессов в подболочках](#)
- 20-1. [Запуск сценария в ограниченном режиме](#)
- 22-1. [Простая функция](#)
- 22-2. [Функция с аргументами](#)
- 22-3. [Наибольшее из двух чисел](#)
- 22-4. [Преобразование чисел в римскую форму записи](#)
- 22-5. [Проверка возможности возврата функциями больших значений](#)
- 22-6. [Сравнение двух больших целых чисел](#)
- 22-7. [Настоящее имя пользователя](#)
- 22-8. [Область видимости локальных переменных](#)
- 22-9. [Использование локальных переменных при рекурсии](#)
- 23-1. [Псевдонимы в сценарии](#)
- 23-2. [**unalias**: Объявление и удаление псевдонимов](#)
- 24-1. [Проверка аргументов командной строки с помощью "И-списка"](#)
- 24-2. [Еще один пример проверки аргументов с помощью "И-списков"](#)
- 24-3. [Комбинирование "ИЛИ-списков" и "И-списков"](#)
- 25-1. [Простой массив](#)
- 25-2. [Форматирование стихотворения](#)
- 25-3. [Некоторые специфичные особенности массивов](#)
- 25-4. [Пустые массивы и пустые элементы](#)
- 25-5. [Копирование и конкатенация массивов](#)
- 25-6. [Старая, добрая: "Пузырьковая" сортировка](#)
- 25-7. [Вложенные массивы и косвенные ссылки](#)
- 25-8. [Пример реализации алгоритма Решето Эратосфена](#)
- 25-9. [Эмуляция структуры "СТЕК" \("первый вошел -- последний вышел"\)](#)

- 25-10. [Исследование математических последовательностей](#)
- 25-11. [Эмуляция массива с двумя измерениями](#)
- 27-1. [Поиск файла программы по идентификатору процесса](#)
- 27-2. [Проверка состояния соединения](#)
- 28-1. [Удаление cookie-файлов](#)
- 28-2. [Создание файла подкачки \(swarfile\), с помощью /dev/zero](#)
- 28-3. [Создание электронного диска](#)
- 29-1. [Сценарий, содержащий ошибку](#)
- 29-2. [Пропущено ключевое слово](#)
- 29-3. [test24](#)
- 29-4. [Проверка условия с помощью функции "assert"](#)
- 29-5. [Ловушка на выходе](#)
- 29-6. [Удаление временного файла при нажатии на Control-C](#)
- 29-7. [Трассировка переменной](#)
- 31-1. [Западня в подоболочке](#)
- 31-2. [Передача вывода от команды **echo** команде **read**, по конвейеру](#)
- 33-1. [сценарий-обертка](#)
- 33-2. [Более сложный пример сценария-обертки](#)
- 33-3. [Сценарий-обертка вокруг сценария **awk**](#)
- 33-4. [Сценарий на языке Perl, встроенный в **Bash**-скрипт](#)
- 33-5. [Комбинирование сценария **Bash** и **Perl** в одном файле](#)
- 33-6. [Сценарий \(бесполезный\), который вызывает себя сам](#)
- 33-7. [Сценарий имеющий практическую ценность\), который вызывает себя сам](#)
- 33-8. ["Цветная" адресная книга](#)
- 33-9. [Вывод цветного текста](#)
- 33-10. [Необычный способ передачи возвращаемого значения](#)
- 33-11. [Необычный способ получения нескольких возвращаемых значений](#)
- 33-12. [Передача массива в функцию и возврат массива из функции](#)
- 33-13. [Игры с анаграммами](#)
- 34-1. [Расширение строк](#)
- 34-2. [Косвенные ссылки на переменные -- новый метод](#)
- 34-3. [Простая база данных, с применением косвенных ссылок](#)
- 34-4. [Массивы и другие хитрости для раздачи колоды карт в четыре руки](#)
- A-1. [**manview**: Просмотр страниц руководств **man**](#)
- A-2. [**mailformat**: Форматирование электронных писем](#)
- A-3. [**rn**: Очень простая утилита для переименования файлов](#)
- A-4. [**blank-rename**: переименование файлов, чьи имена содержат пробелы](#)
- A-5. [**encryptedpw**: Передача файла на ftp-сервер, с использованием пароля](#)
- A-6. [**copy-cd**: Копирование компакт-дисков с данными](#)
- A-7. [Последовательности Коллаца \(Collatz\)](#)
- A-8. [**days-between**: Подсчет числа дней между двумя датами](#)
- A-9. [Создание "словаря"](#)
- A-10. [Расчет индекса "созвучности"](#)
- A-11. ["Игра "Жизнь""](#)
- A-12. [Файл с первым поколением для игры "Жизнь"](#)
- A-13. [**behead**: Удаление заголовков из электронных писем и новостей](#)
- A-14. [**ftpget**: Скачивание файлов по ftp](#)
- A-15. [Указание на авторские права](#)
- A-16. [**password**: Генератор случайного 8-ми символьного пароля](#)
- A-17. [**fifo**: Создание резервных копий с помощью именованных каналов](#)
- A-18. [Генерация простых чисел, с использованием оператора деления по модулю \(остаток от деления\)](#)
- A-19. [**tree**: Вывод дерева каталогов](#)
- A-20. [Функции для работы со строками](#)
- A-21. [Directory information](#)
- A-22. [Объектно ориентированная база данных](#)

G-1. [Пример файла .bashrc](#)

H-1. [VIEWDATA.BAT: пакетный файл DOS](#)

H-2. [viewdata.sh: Результат преобразования VIEWDATA.BAT в сценарий командной оболочки](#)

Часть 1. Введение

Shell -- это командная оболочка. Но это не просто промежуточное звено между пользователем и операционной системой, это еще и мощный язык программирования. Программы на языке shell называют *сценариями*, или *скриптами*. Фактически, из скриптов доступен полный набор команд, утилит и программ UNIX. Если этого недостаточно, то к вашим услугам внутренние команды shell -- условные операторы, операторы циклов и пр., которые увеличивают мощь и гибкость сценариев. Shell-скрипты исключительно хороши при программировании задач администрирования системы и др., которые не требуют для своего создания полновесных языков программирования.

Содержание

1. [Зачем необходимо знание языка Shell?](#)
 2. [Для начала о Sha-Bang](#)
 - 2.1. [Запуск сценария](#)
 - 2.2. [Упражнения](#)
-

Глава 1. Зачем необходимо знание языка Shell?

Знание языка командной оболочки является залогом успешного решения задач администрирования системы. Даже если вы не предполагаете заниматься написанием своих сценариев. Во время загрузки Linux выполняется целый ряд сценариев из `/etc/rc.d`, которые настраивают конфигурацию операционной системы и запускают различные сервисы, поэтому очень важно четко понимать эти скрипты и иметь достаточно знаний, чтобы вносить в них какие либо изменения.

Язык сценариев легок в изучении, в нем не так много специфических операторов и конструкций. [1] Синтаксис языка достаточно прост и прямолинеен, он очень напоминает команды, которые приходится вводить в командной строке. Короткие скрипты практически не нуждаются в отладке, и даже отладка больших скриптов отнимает весьма незначительное время.

Shell-скрипты очень хорошо подходят для быстрого создания прототипов сложных приложений, даже не смотря на ограниченный набор языковых конструкций и определенную "медлительность". Такая метода позволяет детально проработать структуру будущего приложения, обнаружить возможные "ловушки" и лишь затем приступить к кодированию на C, C++, Java, или Perl.

Скрипты возвращают нас к классической философии UNIX -- "разделяй и властвуй" т.е. разделение сложного проекта на ряд простых подзадач. Многие считают такой подход наилучшим или, по меньшей мере, наиболее эстетичным способом решения возникающих проблем, нежели использование нового поколения языков -- "все-в-одном", таких как Perl.

Для каких задач неприменимы скрипты

- для ресурсоемких задач, особенно когда важна скорость исполнения (поиск, сортировка и т.п.)
- для задач, связанных с выполнением математических вычислений, особенно это касается вычислений с плавающей запятой, вычислений с повышенной точностью, комплексных чисел (для таких задач лучше использовать C++ или FORTRAN)
- для кросс-платформенного программирования (для этого лучше подходит язык C)
- для сложных приложений, когда структурирование является жизненной необходимостью (контроль за типами переменных, прототипами функций и т.п.)
- для целевых задач, от которых может зависеть успех предприятия.
- когда во главу угла поставлена безопасность системы, когда необходимо обеспечить целостность системы и защитить ее от вторжения, взлома и вандализма.
- для проектов, содержащих компоненты, очень тесно взаимодействующие между собой.
- для задач, выполняющих огромный объем работ с файлами
- для задач, работающих с многомерными массивами
- когда необходимо работать со структурами данных, такими как связанные списки или деревья
- когда необходимо предоставить графический интерфейс с пользователем (GUI)
- когда необходим прямой доступ к аппаратуре компьютера
- когда необходимо выполнять обмен через порты ввода-вывода или сокет
- когда необходимо использовать внешние библиотеки
- для проприетарных, "закрытых" программ (скрипты представляют из себя исходные тексты программ, доступные для всеобщего обозрения)

Если выполняется хотя бы одно из вышеперечисленных условий, то вам лучше обратиться к более мощным скриптовым языкам программирования, например Perl, Tcl, Python, Ruby или к высокоуровневым компилирующим языкам -- C, C++ или Java. Но даже в этом случае, создание прототипа приложения на языке shell может существенно облегчить разработку.

Название BASH -- это аббревиатура от "Bourne-Again Shell" и игра слов от, ставшего уже классикой, "Bourne Shell" Стефена Бурна (Stephen Bourne). В последние годы BASH достиг такой популярности, что стал стандартной командной оболочкой *de facto* для многих разновидностей UNIX. Большинство принципов программирования на BASH одинаково хорошо применимы и в других командных оболочках, таких как Korn Shell (ksh), от которой Bash позаимствовал некоторые особенности, [2] и C Shell и его производных. (Примечательно, что C Shell не рекомендуется к использованию из-за отдельных

проблем, отмеченных Томом Кристиансеном (Tom Christiansen) в октябре 1993 года на [Usenet post](#)

Далее, в тексте документа вы найдете большое количество примеров скриптов, иллюстрирующих возможности shell. Все примеры -- работающие. Они были протестированы, причем некоторые из них могут пригодиться в повседневной работе. Уважаемый читатель может "поиграть" с рабочим кодом скриптов, сохраняя их в файлы, с именами `scriptname.sh`. [3] Не забудьте выдать этим файлам право на исполнение (`chmod u+rx scriptname`), после чего сценарии можно будет запустить на исполнение и проверить результат их работы. Вам следует помнить, что описание некоторых примеров следует после исходного кода этого примера, поэтому, прежде чем запустить сценарий у себя -- ознакомьтесь с его описанием.

Скрипты были написаны автором книги, если не оговаривается иное.

Глава 2. Для начала о Sha-Bang

В простейшем случае, скрипт -- это ни что иное, как простой список команд системы, записанный в файл. Создание скриптов поможет сохранить ваше время и силы, которые тратятся на ввод последовательности команд всякий раз, когда необходимо их выполнить.

Пример 2-1. cleanup: Сценарий очистки лог-файлов в /var/log

```
# cleanup
# Для работы сценария требуются права root.

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

Здесь нет ничего необычного, это простая последовательность команд, которая может быть набрана в командной строке с консоли или в xterm. Преимущество размещения последовательности команд в скрипте состоит в том, что вам не придется всякий раз набирать эту последовательность вручную. Кроме того, скрипты легко могут быть модифицированы или обобщены для разных применений.

Пример 2-2. cleanup: Расширенная версия предыдущего сценария.

```
#!/bin/bash
# cleanup, version 2
# Для работы сценария требуются права root.

LOG_DIR=/var/log
ROOT_UID=0      # Только пользователь с $UID 0 имеет привилегии root.
LINES=50        # Количество сохраняемых строк по-умолчанию.
E_XCD=66        # Невозможно сменить каталог?
E_NOTROOT=67    # Признак отсутствия root-привилегий.

if [ "$UID" -ne "$ROOT_UID" ]
then
    echo "Для работы сценария требуются права root."
    exit $E_NOTROOT
fi
```

```

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

# Stephane Chazelas предложил следующее,
#+ для проверки корректности аргумента, переданного из командной строки,
#+ правда это достаточно сложно для данного руководства.
#
#     E_WRONGARGS=65 # Не числовой аргумент
#
#     case "$1" in
#         ""          ) lines=50;;
#         *![0-9]*) echo "Usage: `basename $0` file-to-cleanup"; exit $E_WRONGARGS;;
#         *           ) lines=$1;;
#     esac
#
#* Конец проверки корректности аргумента

cd $LOG_DIR

if [ `pwd` != "$LOG_DIR" ] # или   if [ "$PWD" != "$LOG_DIR" ]
                          # Не в /var/log?
then
    echo "Невозможно перейти в каталог $LOG_DIR."
    exit $E_XCD
fi # Проверка каталога перед очисткой лог-файлов.

# более эффективный вариант:
#
# cd /var/log || {
#     echo "Невозможно перейти в требуемый каталог." >&2
#     exit $E_XCD;
# }

tail -$lines messages > mesg.temp # Сохранить последние строки в лог-файле.
mv mesg.temp messages

# cat /dev/null > messages
#* Необходимость этой команды отпала, поскольку очистка выполняется выше.

cat /dev/null > wtmp # команды ': > wtmp' и '> wtmp' имеют тот же эффект.
echo "Лог-файлы очищены."

exit 0
# Возвращаемое значение 0
#+ указывает на успешное завершение работы сценария.

```

Если вы не желаете полностью вычищать системные логи, то выше представлена улучшенная версия предыдущего сценария. Здесь сохраняются последние несколько строк (по-умолчанию -- 50).

Если файл сценария начинается с последовательности `#!`, которая в мире UNIX называется *sha-bang*, то это указывает системе какой интерпретатор следует использовать для исполнения сценария. Это двухбайтовая последовательность, или [\[4\]](#) -- специальный маркер, определяющий тип сценария, в данном случае -- сценарий командной оболочки (см. `man magics`). Более точно, *sha-bang* определяет интерпретатор, который вызывается для исполнения сценария, это может быть

командная оболочка (shell), иной интерпретатор или утилита. [5]

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

Каждая, из приведенных выше сигнатур, приводит к вызову различных интерпретаторов, будь то `/bin/sh` -- командный интерпретатор по-умолчанию (**bash** для Linux-систем), либо иной. [6] При [переносе](#) сценариев с сигнатурой `#!/bin/sh` на другие UNIX системы, где в качестве командного интерпретатора задан другой shell, вы можете лишиться некоторых особенностей, присущих bash. Поэтому такие сценарии должны быть POSIX совместимыми. [7].

Обратите внимание на то, что сигнатура должна указывать правильный путь к интерпретатору, в противном случае вы получите сообщение об ошибке -- как правило это "Command not found".

Сигнатура `#!` может быть опущена, если вы не используете специфичных команд. Во втором примере (см. выше) использование сигнатуры `#!` обязательно, поскольку сценарий использует специфичную конструкцию присваивания значения переменной `lines=50`. Еще раз замечу, что сигнатура `#!/bin/sh` вызывает командный интерпретатор по-умолчанию -- `/bin/bash` в Linux-системах.



В данном руководстве приветствуется модульный подход к построению сценариев. Записывайте, собирайте свою коллекцию участков кода, который может вам встретиться. В конечном итоге вы соберете свою "библиотеку" подпрограмм, которые затем сможете использовать при написании своих сценариев. Например, следующий отрывок сценария проверяет количество аргументов в командной строке:

```
if [ $# -ne Number_of_expected_args ]
then
    echo "Usage: `basename $0` whatever"
    exit $WRONG_ARGS
fi
```

2.1. Запуск сценария

Запустить сценарий можно командой `sh scriptname` [8] или `bash scriptname`. (Не рекомендуется запуск сценария командой `sh <scriptname>`, поскольку это запрещает использование устройства стандартного ввода `stdin` в скрипте). Более удобный вариант -- сделать файл скрипта исполняемым, командой [chmod](#).

Это:

```
chmod 555 scriptname (выдача прав на чтение/исполнение любому
пользователю в системе) [9]
```

или

`chmod +rx scriptname` (выдача прав на чтение/исполнение любому пользователю в системе)

`chmod u+rx scriptname` (выдача прав на чтение/исполнение только "владельцу" скрипта)

После того, как вы сделаете файл сценария исполняемым, вы можете запустить его примерно такой командой `./scriptname`. [10] Если, при этом, текст сценария начинается с корректной сигнатуры ("sha-bang"), то для его исполнения будет вызван соответствующий интерпретатор.

И наконец, завершив отладку сценария, вы можете поместить его в каталог `/usr/local/bin` (естественно, что для этого вы должны обладать правами `root`), чтобы сделать его доступным для себя и других пользователей системы. После этого сценарий можно вызвать, просто напечатав название файла в командной строке и нажав клавишу **[ENTER]**.

2.2. Упражнения

1. Системные администраторы часто создают скрипты для автоматизации своего труда. Подумайте, для выполнения каких задач могут быть написаны сценарии.
2. Напишите сценарий, который выводит [дату](#), [время](#), [список зарегистрировавшихся пользователей](#), и [uptime](#) системы и [сохраняет эту информацию](#) в системном журнале.

Часть 2. Основы

Содержание

3. [Служебные символы](#)
 4. [Переменные и параметры. Введение.](#)
 - 4.1. [Подстановка переменных](#)
 - 4.2. [Присваивание значений переменным](#)
 - 4.3. [Переменные Bash не имеют типа](#)
 - 4.4. [Специальные типы переменных](#)
 5. [Кавычки](#)
 6. [Завершение и код завершения](#)
 7. [Проверка условий](#)
 - 7.1. [Конструкции проверки условий](#)
 - 7.2. [Операции проверки файлов](#)
 - 7.3. [Операции сравнения](#)
 - 7.4. [Вложенные условные операторы if/then](#)
 - 7.5. [Проверка степени усвоения материала](#)
 8. [Операции и смежные темы](#)
 - 8.1. [Операторы](#)
 - 8.2. [Числовые константы](#)
-

Глава 3. Служебные символы

Служебные символы, используемые в текстах сценариев.

#

Комментарии. Строки, начинающиеся с символа # ([за исключением комбинации #!](#)) -- являются комментариями.

```
# Эта строка -- комментарий.
```

Комментарии могут располагаться и в конце строки с исполняемым кодом.

```
echo "Далее следует комментарий." # Это комментарий.
```

Комментариям могут предшествовать [пробелы](#) (пробел, табуляция).

```
# Перед комментарием стоит символ табуляции.
```



Исполняемые команды не могут следовать за комментарием в той же самой строке. Пока что еще не существует способа отделения комментария от "исполняемого кода", следующего за комментарием в той же строке.



Само собой разумеется, экранированный символ # в операторе **echo** не воспринимается как начало комментария. Более того, он может использоваться в [операциях подстановки параметров](#) и в [константных числовых выражениях](#).

```
echo "Символ # не означает начало комментария."  
echo 'Символ # не означает начало комментария.'  
echo Символ \# не означает начало комментария.  
echo А здесь символ # означает начало комментария.
```

```
echo ${ПАТН#*:}          # Подстановка -- не комментарий.  
echo $(( 2#101011 ))    # База системы счисления -- не комментарий.
```

```
# Спасибо, S.C.
```

[Кавычки " ' и \](#) экранируют действие символа #.

В операциях [поиска по шаблону](#) символ # так же не воспринимается как начало комментария.

;

Разделитель команд. [Точка-с-запятой] Позволяет записывать две и более команд в одной строке.

```
echo hello; echo there
```

Следует отметить, что символ ";" иногда так же как и # необходимо [экранировать](#).

::

Ограничитель в операторе выбора [case](#) . [Двойная-точка-с-запятой]

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

команда "точка". Эквивалент команды [source](#) (см. [Пример 11-18](#)). Это [встроенная](#) команда bash.

"точка" может являться частью имени файла . Если имя файла начинается с точки, то это "скрытый" файл, т.е. команда [ls](#) при обычных условиях его не отображает.

```
bash$ touch .hidden-file
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 .hidden-file
```

Если подразумевается имя каталога, то *одна точка* означает текущий каталог и *две точки* -- каталог уровнем выше, или родительский каталог.

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

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

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

Символ *точка* довольно часто используется для обозначения каталога назначения в

операциях копирования/перемещения файлов.

```
bash$ cp /home/bozo/current_work/junk/* .
```

Символ "точка" в операциях поиска. При выполнении [поиска по шаблону](#) , в [регулярных выражениях](#), символ "точка" обозначает одиночный символ.

Двойные кавычки . В строке "STRING", ограниченной двойными кавычками не выполняется интерпретация большинства служебных символов, которые могут находиться в строке. см. [Глава 5](#).

Одинарные кавычки . [Одинарные кавычки] 'STRING' экранирует все служебные символы в строке STRING. Это более строгая форма экранирования. Смотрите так же [Глава 5](#).

Запятая . Оператор **запятая** используется для вычисления серии арифметических выражений. Вычисляются все выражения, но возвращается результат последнего выражения.

```
let "t2 = ((a = 9, 15 / 3))" # Присваивает значение переменной "a" и вычисляет "t2".
```

escape. [обратный слэш] Комбинация \X "экранирует" символ X. Аналогичный эффект имеет комбинация с "одинарными кавычками", т.е. 'X'. Символ \ может использоваться для экранирования кавычек " и '.

Более детальному рассмотрению темы экранирования посвящена [Глава 5](#).

Разделитель, используемый в указании пути к каталогам и файлам. [слэш] Отделяет элементы пути к каталогам и файлам (например /home/bozo/projects/Makefile).

В [арифметических операциях](#) -- это оператор деления.

Подстановка команд. [обратные кавычки] [Обратные кавычки](#) могут использоваться для записи в переменную команды `command`.

пустая команда. [двоеточие] Это эквивалент операции "NOP" (*no op*, нет операции). Может рассматриваться как синоним встроенной команды [true](#). Команда ":" так же является встроенной командой Bash, которая всегда [возвращает](#) "true" (0).

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

Бесконечный цикл:

```
while :  
do  
    operation-1  
    operation-2  
    ...  
    operation-n  
done  
  
# То же самое:  
# while true  
# do  
#     ...  
# done
```

Символ-заполнитель в условном операторе if/then:

```
if condition  
then : # Никаких действий не производится и управление передается дальше  
else  
    take-some-action  
fi
```

Как символ-заполнитель в операциях, которые предполагают наличие двух операндов, см. [Пример 8-2](#) и [параметры по-умолчанию](#).

```
: ${username=`whoami`}  
# ${username=`whoami`} без символа : выдает сообщение об ошибке,  
# если "username" не является командой...
```

Как символ-заполнитель для оператора [вложенного документа](#). См. [Пример 17-9](#).

В операциях с [подстановкой параметров](#) (см. [Пример 9-13](#)).

```
: ${HOSTNAME?} ${USER?} ${MAIL?}  
#Вывод сообщения об ошибке, если одна или более переменных не определены.
```

В операциях [замены подстроки с подстановкой значений переменных](#).

В комбинации с оператором > ([оператор перенаправления вывода](#)), усекает длину файла до нуля. Если указан несуществующий файл -- то он создается.

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

```
# Тот же эффект имеет команда cat /dev/null >data.xxx
# Однако в данном случае не производится создание нового процесса, поскольку ":"
является встроенной командой.
```

См. так же [Пример 12-11](#).

В комбинации с оператором >> -- оператор перенаправления с добавлением в конец файла и обновлением времени последнего доступа (: >> `new_file`). Если задано имя несуществующего файла, то он создается. Эквивалентно команде [touch](#).



Вышеизложенное применимо только к обычным файлам и неприменимо к конвейерам, символическим ссылкам и другим специальным файлам.

Символ # может использоваться для создания комментариев, хотя и не рекомендуется. Если строка комментария начинается с символа #, то такая строка не проверяется интерпретатором на наличие ошибок. Однако в случае оператора : это не так.

: Это комментарий, который генерирует сообщение об ошибке, (`if [$x -eq 3]`).

Символ ":" может использоваться как разделитель полей в `/etc/passwd` и переменной [\\$PATH](#).

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

!

инверсия (или логическое отрицание) используемое в условных операторах.

Оператор ! инвертирует [код завершения](#) команды, к которой он применен. (см. [Пример 6-2](#)). Так же используется для логического отрицания в операциях сравнения, например, операция сравнения "равно" (`=`), при использовании оператора отрицания, преобразуется в операцию сравнения -- "не равно" (`!=`). Символ ! является зарезервированным ключевым словом BASH.

В некоторых случаях символ ! используется для [косвенного обращения к переменным](#).

Кроме того, из *командной строки* оператор ! запускает *механизм историй* Bash (см. [Приложение F](#)). Примечательно, что этот механизм недоступен из сценариев (т.е. исключительно из командной строки).

*

символ-шаблон. [звездочка] Символ * служит "шаблоном" для [подстановки](#) в имена файлов. Одиночный символ * означает любое имя файла в заданном каталоге.

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

В [регулярных выражениях](#) токен * представляет любое количество (в том числе и 0) символов.

*

[арифметический оператор.](#) В арифметических выражениях символ * обозначает операцию умножения.

Двойная звездочка (два символа звездочки, следующих подряд друг за другом -- **), обозначает операцию [возведения в степень](#).

?

Оператор проверки условия. В некоторых выражениях символ ? служит для проверки выполнения условия.

В [конструкциях с двойными скобками](#), символ ? подобен трехместному оператору языка C. См. [Пример 9-28](#).

В выражениях с [подстановкой параметра](#), символ ? [проверяет -- установлена ли переменная](#).

?

символ-шаблон. Символ ? обозначает одиночный символ при [подстановке](#) в имена файлов. В [регулярных выражениях](#) служит для обозначения [одиночного символа](#).

\$

[Подстановка переменной.](#)

```
var1=5
var2=23skidoo

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

Символ \$, предшествующий имени переменной, указывает на то, что будет получено *значение* переменной.

\$

end-of-line (конец строки). В [регулярных выражениях](#), символ "\$" обозначает конец строки.

\${}

[Подстановка параметра.](#)

*, @\$

[параметры командной строки.](#)

\$?

код завершения. [Переменная \\$?](#) хранит [код завершения](#) последней выполненной команды, [функции](#) или сценария.


\$\$

id процесса. [Переменная \\$\\$](#) хранит *id процесса* сценария.

()

группа команд.

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

 Команды, заключенные в *круглые скобки* исполняются в дочернем процессе -- [subshell-e](#).

Переменные, создаваемые в дочернем процессе не видны в "родительском" сценарии. Родительский процесс-сценарий, [не может обращаться к переменным, создаваемым в дочернем процессе](#).

```
a=123
( a=321; )

echo "a = $a"    # a = 123
# переменная "a" в скобках подобна локальной переменной.
```

инициализация массивов.

```
Array=(element1 element2 element3)
```

```
{xxx,yyy,zzz,...}
```

Фигурные скобки.

```
grep Linux file*.{txt,htm*}
# Поиск всех вхождений слова "Linux"
# в файлах "fileA.txt", "file2.txt", "fileR.html", "file-87.htm", и пр.
```

Команда интерпретируется как список команд, разделенных точкой с запятой, с вариациями, представленными в *фигурных скобках*. [11] При интерпретации имен файлов ([подстановка](#)) используются параметры, заключенные в фигурные скобки.

 Использование *неэкранированных или неоквазыченных* пробелов внутри фигурных скобок недопустимо.

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

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

```
}
```

Блок кода. [фигурные скобки] Известен так же как "вложенный блок", эта конструкция, фактически, создает анонимную функцию. Однако, в отличие от обычных [функций](#), переменные, создаваемые во вложенных блоках кода, доступны объемлющему сценарию.

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

```
a=123
{ a=321; }
echo "a = $a"    # a = 321    (значение, присвоенное во вложенном блоке кода)

# Спасибо, S.C.
```

Код, заключенный в фигурные скобки, может выполнять [перенаправление ввода-вывода](#).

Пример 3-1. Вложенные блоки и перенаправление ввода-вывода

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

File=/etc/fstab

{
read line1
read line2
} < $File

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

exit 0
```

Пример 3-2. Сохранение результата исполнения вложенного блока в файл

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

# Запрашивает описание rpm-архива, список файлов, и проверяется возможность
установки.
# Результат сохраняется в файле.
#
# Этот сценарий иллюстрирует порядок работы со вложенными блоками кода.

SUCCESS=0
E_NOARGS=65

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

{
    echo
    echo "Описание архива:"
```

```

rpm -qri $1          # Запрос описания.
echo
echo "Список файлов:"
rpm -qpl $1         # Запрос списка.
echo
rpm -i --test $1    # Проверка возможности установки.
if [ "$?" -eq $SUCCESS ]
then
    echo "$1 может быть установлен."
else
    echo "$1 -- установка невозможна!"
fi
echo
} > "$1.test"      # Перенаправление вывода в файл.

echo "Результаты проверки rpm-архива находятся в файле $1.test"

# За дополнительной информацией по ключам команды rpm см. man rpm.

exit 0

```



В отличие от групп команд в (круглых скобках), описанных выше, вложенные блоки кода, заключенные в {фигурные скобки} исполняются в пределах того же процесса, что и сам скрипт (т.е. не вызывают запуск дочернего процесса -- [subshell](#)). [12]

} \;

pathname -- полное имя файла (т.е. путь к файлу и его имя). Чаще всего используется совместно с командой [find](#).



Обратите внимание на то, что символ ";", которым завершается ключ -exec команды **find**, экранируется обратным слэшем. Это необходимо, чтобы предотвратить его интерпретацию.

[]

test.

[Проверка истинности](#) выражения, заключенного в квадратные скобки [].

Примечательно, что [является частью встроенной команды **test** (и ее синонимом), и не имеет никакого отношения к "внешней" утилите /usr/bin/test.

[[]]

test.

Проверка истинности выражения, заключенного между [[]] ([зарезервированное слово](#) интерпретатора).

См. описание конструкции [\[\[...\]\]](#) ниже.

[]

элемент массива.

При работе с [массивами](#) в квадратных скобках указывается порядковый номер того элемента массива, к которому производится обращение.

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

[]

диапазон символов.

В [регулярных выражениях](#), в квадратных скобках задается [диапазон искомых символов](#).

(())

двойные круглые скобки.

Вычисляется целочисленное выражение, заключенное между двойными круглыми скобками (()).

См. обсуждение, посвященное [конструкции \(\(...\)\)](#).

> &> >& >> <

[перенаправление.](#)

Конструкция `scriptname >filename` перенаправляет вывод `scriptname` в файл `filename`. Если файл `filename` уже существовал, то его прежнее содержимое будет утеряно.

Конструкция `command &>filename` перенаправляет вывод команды `command`, как со `stdout`, так и с `stderr`, в файл `filename`.

Конструкция `command >&2` перенаправляет вывод со `stdout` на `stderr`.

Конструкция `scriptname >>filename` добавляет вывод `scriptname` к файлу `filename`. Если задано имя несуществующего файла, то он создается.

[подстановка процесса.](#)

(command) >

<(command)

В [операциях сравнения](#), символы "<" и ">" обозначают операции [сравнения строк](#).

[А так же](#) -- операции [сравнения целых чисел](#). См. так же [Пример 12-6](#).

<<

перенаправление ввода на [встроенный документ](#).

<, >

[Посимвольное ASCII-сравнение.](#)

```
veg1=carrots  
veg2=tomatoes
```



```

if [[ "$veg1" < "$veg2" ]]
then
    echo "Не смотря на то, что в словаре слово $veg1 предшествует слову $veg2,"
    echo "это никак не отражает мои кулинарные предпочтения."
else
    echo "Интересно. Каким словарем вы пользуетесь?"
fi

```

<, >

границы отдельных слов в регулярных выражениях.

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

|

конвейер. Передает вывод предыдущей команды на вход следующей или на вход командного интерпретатора shell. Этот метод часто используется для связывания последовательности команд в единую цепочку.

```

echo ls -l | sh
# Передает вывод "echo ls -l" командному интерпретатору shell,
#+ тот же результат дает простая команда "ls -l".

```

```

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

```

Конвейеры (еще их называют каналами) -- это классический способ взаимодействия процессов, с помощью которого stdout одного процесса перенаправляется на stdin другого. Обычно используется совместно с командами вывода, такими как [cat](#) или [echo](#), от которых поток данных поступает в "фильтр" (команда, которая на входе получает данные, преобразует их и обрабатывает).

```
cat $filename | grep $search_word
```

В конвейер могут объединяться и сценарии на языке командной оболочки.

```

#!/bin/bash
# uppercase.sh : Преобразование вводимых символов в верхний регистр.

tr 'a-z' 'A-Z'
# Диапазоны символов должны быть заключены в кавычки
#+ чтобы предотвратить порождение имен файлов от однобуквенных имен файлов.

exit 0

```

А теперь попробуем объединить в конвейер команду **ls -l** с этим сценарием.

```

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

```



Выход stdout каждого процесса в конвейере должен читаться на входе stdin последующим, в конвейере, процессом. Если этого не делается, то поток данных *блокируется*, в результате конвейер будет работать не так как ожидается.

```
cat file1 file2 | ls -l | sort
# Вывод команды "cat file1 file2" будет утерян.
```

Конвейер исполняется в [дочернем процессе](#), а посему -- не имеет доступа к переменным сценария.

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

Если одна из команд в конвейере завершается аварийно, то это приводит к аварийному завершению работы всего конвейера.

>|

принудительное перенаправление, даже если установлен ключ [noclobber option](#).

||

логическая операция OR (логическое ИЛИ). В [опрециях проверки условий](#), оператор || возвращает 0 (success), если один из операндов имеет значение true (ИСТИНА).

&

Выполнение задачи в фоне. Команда, за которой стоит &, будет исполняться в фоновом режиме.

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

В сценариях 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 & # Запуск цикла в фоне.
      # Иногда возможны случаи выполнения этого цикла после второго цикла.
```

```

echo # Этот 'echo' иногда не отображается на экране.

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

echo # Этот 'echo' иногда не отображается на экране.

# =====

# Ожидается, что данный сценарий выведет следующую последовательность:
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20

# Иногда возможен такой вариант:
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (Второй 'echo' не был выполнен. Почему?)

# Изредка возможен такой вариант:
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (Первый 'echo' не был выполнен. Почему?)

# Крайне редко встречается и такое:
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# Второй цикл начал исполняться раньше первого.

exit 0

```



Команда, исполняемая в пределах сценария в фоне, может подвесить сценарий, ожидая нажатия клавиши. К счастью, это легко ["лечится"](#).

&&

Логическая операция AND (логическое И). В [операциях проверки условий](#), оператор && возвращает 0 (success) тогда, и только тогда, когда оба операнда имеют значение true (ИСТИНА).

-

префикс ключа. С этого символа начинаются опциональные ключи команд.

COMMAND - [Option1] [Option2] [...]

ls -al

sort -dfu \$filename

set -- \$variable

```

if [ $file1 -ot $file2 ]
then
    echo "Файл $file1 был создан раньше чем $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
```

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

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
# Перемещение полного дерева файлов и подкаталогов из одной директории в другую
# [спасибо Алану Коксу (Alan Cox) <a.cox@swansea.ac.uk>, за небольшие поправки]

# 1) cd /source/directory      Переход в исходный каталог, содержимое которого
будет перемещено
# 2) &&                        "И-список": благодаря этому все последующие команды
будут выполнены
#                               только тогда, когда 'cd' завершится успешно
# 3) tar cf - .                ключом 'c' архиватор 'tar' создает новый архив,
#                               ключом 'f' (file) и последующим '-' задается файл
архива -- stdout,
#                               в архив помещается текущий каталог ('.') с
вложенными подкаталогами.
# 4) |                          конвейер с ...
# 5) ( ... )                   subshell-ом (дочерним экземпляром командной
оболочки)
# 6) cd /dest/directory        Переход в каталог назначения.
# 7) &&                        "И-список", см. выше
# 8) tar xpvf -                Разархивирование ('x'), с сохранением атрибутов
"владельца" и прав доступа ('p') к файлам,
#                               с выдачей более подробных сообщений на stdout
('v'),
#                               файл архива -- stdin ('f' с последующим '-').
#
#                               Примечательно, что 'x' -- это команда, а 'p', 'v' и
'f' -- ключи
# Во как!

# Более элегантный вариант:
#   cd source-directory
#   tar cf - . | (cd ../target-directory; tar xzf -)
#
# cp -a /source/directory /dest      имеет тот же эффект.
```

```
bunzip2 linux-2.4.3.tar.bz2 | tar xvf -
# --разархивирование tar-файла-- | --затем файл передается утилите "tar"--
# Если у вас утилита "tar" не поддерживает работу с "bunzip2",
# тогда придется выполнять работу в два этапа, с использованием конвейера.
# Целью данного примера является разархивирование тарбола (tar.bz2) с исходными
текстами ядра.
```

Обратите внимание, что в этом контексте "-" - не самостоятельный оператор Bash, а скорее опция, распознаваемая некоторыми утилитами UNIX (такими как **tar**, **cat** и т.п.), которые выводят результаты своей работы в `stdout`.

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

В случае, когда ожидается имя файла, тогда "-" перенаправляет вывод на stdout (вспомните пример с `tar cf`) или принимает ввод с stdin.

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

Сама по себе команда [file](#) без параметров завершается с сообщением об ошибке.

Добавим символ "-" и получим более полезный результат. Это заставит командный интерпретатор ожидать ввода от пользователя.

```
bash$ file -
abc
standard input:                ASCII text

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

Теперь команда принимает ввод пользователя со stdin и анализирует его.

Используя передачу stdout по конвейеру другим командам, можно выполнять довольно эффектные трюки, например [вставка строк в начало файла](#).

С помощью команды [diff](#) -- находить различия между одним файлом и частью другого:

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

И наконец пример использования служебного символа "-" с командой [tar](#).

Пример 3-4. Резервное архивирование всех файлов, которые были изменены в течение последних суток

```
#!/bin/bash

# Резервное архивирование (backup) всех файлов в текущем каталоге,
# которые были изменены в течение последних 24 часов
#+ в тарболл (tarball) (.tar.gz - файл).

BACKUPFILE=backup
archive=${1:-$BACKUPFILE}
# На случай, если имя архива в командной строке не задано,
#+ т.е. по-умолчанию имя архива -- "backup.tar.gz"

tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Каталог $PWD заархивирован в файл \"$archive.tar.gz\"."


# Stephane Chazelas заметил, что вышеприведенный код будет "падать"
#+ если будет найдено слишком много файлов
#+ или если имена файлов будут содержать символы пробела.

# Им предложен альтернативный код:
# -----
# find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
#     используется версия GNU утилиты "find".

# find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
```

```
# более универсальный вариант, хотя и более медленный,  
# зато может использоваться в других версиях UNIX.  
# -----
```

```
exit 0
```


 Могут возникнуть конфликтные ситуации между оператором перенаправления "-" и именами файлов, начинающимися с символа "-". Поэтому сценарий должен проверять имена файлов и предаварять их префиксом пути, например, ./ - FILENAME, \$PWD/ - FILENAME или \$PATHNAME/ - FILENAME.

Если значение переменной начинается с символа "-", то это тоже может быть причиной появления ошибок.

```
var="-n"  
echo $var  
# В данном случае команда приобретет вид "echo -n" и ничего не выведет.
```

-

предыдущий рабочий каталог. [дефис] Команда **cd** - выполнит переход в предыдущий рабочий каталог, путь к которому хранится в [переменной окружения \\$OLDPWD](#).

 Не путайте оператор "-" (предыдущего рабочего каталога) с оператором "-" (переназначения). Еще раз напомним, что интерпретация символа "-" зависит от контекста, в котором он употребляется.

-

Минус. Знак минус в [арифметических операциях](#).

=

Символ "равно". [Оператор присваивания](#)

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

В зависимости от [контекста применения](#), символ "=" может выступать в качестве оператора [сравнения](#).

+

Плюс. Оператор сложения в [арифметических операциях](#).

В зависимости от [контекста применения](#), символ + может выступать как оператор [регулярного выражения](#).

+

Ключ (опция). Дополнительный флаг для ключей (опций) команд.

Отдельные внешние и [встроенные](#) команды используют символ "+" для разрешения некоторой опции, а символ "-" -- для запрещения.

%

модуль. Модуль (остаток от деления) -- [арифметическая операция](#).

В зависимости от [контекста применения](#), символ % может выступать в качестве [шаблона](#).

~

домашний каталог. [тильда] Соответствует содержимому внутренней переменной [\\$HOME](#). `~bozo` -- домашний каталог пользователя bozo, а команда `ls ~bozo` выведет содержимое его домашнего каталога. `~/` -- это домашний каталог текущего пользователя, а команда `ls ~/` выведет содержимое домашнего каталога текущего пользователя.

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

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

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

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

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

~+

текущий рабочий каталог. Соответствует содержимому внутренней переменной [\\$PWD](#).

~-

предыдущий рабочий каталог. Соответствует содержимому внутренней переменной [\\$OLDPWD](#).

^

начало-строки. В [регулярных выражениях](#) символ "^" задает начало строки текста.

Управляющий символ

изменяет поведение терминала или управляет выводом текста. Управляющий символ набирается с клавиатуры как комбинация **CONTROL + <клавиша>**.

- Ctl-C

Завершение выполнения процесса.

- **Ctl-D**

Выход из командного интерпретатора (log out) (аналог команды [exit](#)).

"EOF" (признак конца файла). Этот символ может выступать в качестве завершающего при вводе с stdin.

- **Ctl-G**

"BEL" (звуковой сигнал -- "звонок").

- **Ctl-H**

Backspace -- удаление предыдущего символа.

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

a="^H^H"                # Два символа Ctl-H (backspace).
echo "abcdef"          # abcdef
echo -n "abcdef$a "   # abcd f
# Пробел в конце ^    ^ двойной шаг назад.
echo -n "abcdef$a"    # abcdef
# Пробела в конце нет      backspace не работает (почему?).
# Результаты могут получиться совсем не те, что вы ожидаете.
echo; echo
```

- **Ctl-J**

Возврат каретки.

- **Ctl-L**

Перевод формата (очистка экрана (окна) терминала). Аналогична команде [clear](#).

- **Ctl-M**

Перевод строки.

- **Ctl-U**

Стирание строки ввода.

- **Ctl-Z**

Приостановка процесса.

Пробельный символ

используется как разделитель команд или переменных. В качестве пробельного символа могут выступать -- собственно пробел (space), символ табуляции, символ перевода строки, символ возврата каретки или комбинация из вышеперечисленных символов. В некоторых случаях, таких как [присваивание значений переменным](#), использование пробельных символов недопустимо.

Пустые строки никак не обрабатываются командным интерпретатором и могут свободно использоваться для визуального выделения отдельных блоков сценария.

[\\$IFS](#) -- переменная специального назначения. Содержит символы-разделители полей, используемые некоторыми командами. По-умолчанию -- пробельные символы.

Глава 4. Переменные и параметры. Введение.

Переменные -- это одна из основ любого языка программирования. Они участвуют в арифметических операциях, в синтаксическом анализе строк и совершенно необходимы для абстрагирования каких либо величин с помощью символических имен. Физически переменные представляют собой ни что иное как участки памяти, в которые записана некоторая информация.

4.1. Подстановка переменных

Когда интерпретатор встречается в тексте сценария *имя* переменной, то он вместо него подставляет *значение* этой переменной. Поэтому ссылки на переменные называются *подстановкой переменных*.

\$

Необходимо всегда помнить о различиях между *именем* переменной и ее *значением*. Если `variable1` -- это имя переменной, то `$variable1` -- это ссылка на ее *значение*. "Чистые" имена переменных, без префикса \$, могут использоваться только при объявлении переменных, при присваивании переменной некоторого значения, при *удалении (сбросе)*, при [экспорте](#) и в особых случаях -- когда переменная представляет собой название [сигнала](#) (см. [Пример 29-5](#)). Присваивание может производиться с помощью символа = (например: `var1=27`), инструкцией [read](#) и в заголовке цикла (`for var2 in 1 2 3`).

Заключение ссылки на переменную в двойные кавычки (" ") никак не сказывается на работе механизма подстановки. Этот случай называется "частичные кавычки", иногда можно встретить название "нестрогие кавычки". Одиночные кавычки (' ') заставляют интерпретатор воспринимать ссылку на переменную как простой набор символов, потому что в одинарных кавычках операции подстановки не производятся. Этот случай называется "полные", или "строгие" кавычки. Дополнительную информацию вы найдете в [Глава 5](#).

Примечательно, что написание `$variable` фактически является упрощенной формой написания `${variable}`. Более строгая форма записи `${variable}` может с успехом использоваться в тех случаях, когда применение упрощенной формы записи порождает сообщения о синтаксических ошибках (см. [Section 9.3](#), ниже).

Пример 4-1. Присваивание значений переменным и подстановка значений переменных

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

```

# Присваивание значений переменным и подстановка значений переменных

a=375
hello=$a

#-----
# Использование пробельных символов
# с обеих сторон символа "=" присваивания недопустимо.

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

# Если записать "VARIABLE= value",
#+ то интерпретатор попытается установить переменную окружения "VARIABLE" в ""
#+ и выполнить команду "value".
#-----

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

echo $hello
echo ${hello} # Идентично предыдущей строке.

echo "$hello"
echo "${hello}"

echo

hello="A B C D"
echo $hello    # A B C D
echo "$hello"  # A B C D
# Здесь вы сможете наблюдать различия в выводе echo $hello и echo "$hello".
# Заключение ссылки на переменную в кавычки сохраняет пробельные символы.

echo

echo '$hello'  # $hello
# Внутри одинарных кавычек не производится подстановка значений переменных,
#+ т.е. "$" интерпретируется как простой символ.

# Обратите внимание на различия, существующие между этими типами кавычек.

hello=        # Запись пустого значения в переменную.
echo "\$hello (пустое значение) = $hello"
# Обратите внимание: запись пустого значения -- это не то же самое,
#+ что сброс переменной, хотя конечный результат -- тот же (см. ниже).

# -----

# Допускается присваивание нескольких переменных в одной строке,
#+ если они отделены пробельными символами.
# Внимание! Это может снизить читабельность сценария и оказаться непереносимым.

var1=variable1 var2=variable2 var3=variable3
echo
echo "var1=$var1 var2=$var2 var3=$var3"

# Могут возникнуть проблемы с устаревшими версиями "sh".

# -----

echo; echo

numbers="один два три"
other_numbers="1 2 3"
# Если в значениях переменных встречаются пробелы,

```

```

# то использование кавычек обязательно.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"    # other_numbers = 1 2 3
echo

echo "uninitialized_variable = $uninitialized_variable"
# Неинициализированная переменная содержит "пустое" значение.
uninitialized_variable=    # Объявление неинициализированной переменной
                          #+ (то же, что и присваивание пустого значения, см.
выше).
echo "uninitialized_variable = $uninitialized_variable"
                          # Переменная содержит "пустое" значение.

uninitialized_variable=23    # Присваивание.
unset uninitialized_variable    # Сброс.
echo "uninitialized_variable = $uninitialized_variable"
                          # Переменная содержит "пустое" значение.

echo

exit 0

```



Неинициализированная переменная хранит "пустое" значение - не ноль!
Использование неинициализированных переменных может приводить к ошибкам разного рода в процессе исполнения.

Не смотря на это в арифметических операциях допускается использовать неинициализированные переменные.

```

echo "$uninitialized"          # (пустая строка)
let "uninitialized += 5"       # Прибавить 5.
echo "$uninitialized"         # 5

```

```

# Заключение:
# Неинициализированные переменные не имеют значения, однако
#+ в арифметических операциях за значение таких переменных принимается число
0.
# Это недокументированная (и возможно непереносимая) возможность.

```

См. так же [Пример 11-19](#).

4.2. Присваивание значений переменным

=

оператор присваивания (*пробельные символы до и после оператора -- недопустимы*)



Не путайте с операторами сравнения `=` и `-eq`!

Обратите внимание: символ `=` может использоваться как в качестве оператора присваивания, так и в качестве оператора сравнения, конкретная интерпретация зависит от контекста применения.

Пример 4-2. Простое присваивание

```

#!/bin/bash
# Явные переменные

echo

```

```

# Когда перед именем переменной не употребляется символ '$'?
# В операциях присваивания.

# Присваивание
a=879
echo "Значение переменной \"a\" -- $a."

# Присваивание с помощью ключевого слова 'let'
let a=16+5
echo "Значение переменной \"a\" теперь стало равным: $a."

echo

# В заголовке цикла 'for' (своего рода неявное присваивание)
echo -n "Значения переменной \"a\" в цикле: "
for a in 7 8 9 11
do
    echo -n "$a "
done

echo
echo

# При использовании инструкции 'read' (тоже одна из разновидностей присваивания)
echo -n "Введите значение переменной \"a\" "
read a
echo "Значение переменной \"a\" теперь стало равным: $a."

echo

exit 0

```

Пример 4-3. Присваивание значений переменным простое и замаскированное

```

#!/bin/bash

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

# Теперь немного более сложный вариант (подстановка команд).

a=`echo Hello!` # В переменную 'a' попадает результат работы команды 'echo'
echo $a
# Обратите внимание на восклицательный знак (!) в подставляемой команде
#+ этот вариант не будет работать при наборе в командной строке,
#+ поскольку здесь используется механизм "истории команд" BASH
# Однако, в сценариях, механизм истории команд запрещен.

a=`ls -l`     # В переменную 'a' записывается результат работы команды 'ls -l'
echo $a      # Кавычки отсутствуют, удаляются лишние пробелы и пустые строки.
echo
echo "$a"    # Переменная в кавычках, все пробелы и пустые строки сохраняются.
              # (См. главу "Кавычки.")

exit 0

```

Присваивание переменных с использованием $\$(...)$ (более современный метод, по сравнению с [обратными кавычками](#))

```

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

```

4.3. Переменные Bash не имеют типа

В отличие от большинства других языков программирования, Bash не производит разделения переменных по "типам". По сути, переменные Bash являются строковыми переменными, но, в зависимости от контекста, Bash допускает целочисленную арифметику с переменными. Определяющим фактором здесь служит содержимое переменных.

Пример 4-4. Целое число или строка?

```
#!/bin/bash
# int-or-string.sh: Целое число или строка?

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

b=${a/23/BB}         # замена "23" на "BB".
echo "b = $b"         # Происходит трансформация числа в строку.
declare -i b          # b = BB35
echo "b = $b"         # Явное указание типа здесь не поможет.
                    # b = BB35

let "b += 1"          # BB35 + 1 =
echo "b = $b"         # b = 1
echo

c=BB34
echo "c = $c"         # c = BB34
d=${c/BB/23}         # замена "BB" на "23".
echo "d = $d"         # Переменная $d становится целочисленной.
                    # d = 2334
let "d += 1"         # 2334 + 1 =
echo "d = $d"         # d = 2335
echo

# А что происходит с "пустыми" переменными?
e=""
echo "e = $e"        # e =
let "e += 1"         # Арифметические операции допускают использование "пустых"
переменных?
echo "e = $e"        # e = 1
echo                 # "Пустая" переменная становится целочисленной.

# А что происходит с необъявленными переменными?
echo "f = $f"        # f =
let "f += 1"         # Арифметические операции допустимы?
echo "f = $f"        # f = 1
echo                 # Необъявленная переменная трансформируется в целочисленную.

# Переменные Bash не имеют типов.

exit 0
```

Отсутствие типов -- это и благословение и проклятие. С одной стороны -- отсутствие типов делает сценарии более гибкими (чтобы повеситься -- достаточно иметь веревку!) и

облегчает чтение кода. С другой -- является источником потенциальных ошибок и поощряет привычку к "неряшливому" программированию.

Бремя отслеживания типа той или иной переменной полностью лежит на плечах программиста. Bash не будет делать это за вас!


4.4. Специальные типы переменных

локальные переменные


переменные, область видимости которых ограничена [блоком кода](#) или телом функции (см так же [локальные переменные в функциях](#))

переменные окружения

переменные, которые затрагивают командную оболочку и порядок взаимодействия с пользователем

-  В более общем контексте, каждый процесс имеет некоторое "окружение" (среду исполнения), т.е. набор переменных, к которым процесс может обращаться за получением определенной информации. В этом смысле командная оболочка подобна любому другому процессу.

Каждый раз, когда запускается командный интерпретатор, для него создаются переменные, соответствующие переменным окружения. Изменение переменных или добавление новых переменных окружения заставляет оболочку обновить свои переменные, и все дочерние процессы (и команды, исполняемые ею) наследуют это окружение.


-  Пространство, выделяемое под переменные окружения, ограничено. Создание слишком большого количества переменных окружения или одной переменной, которая занимает слишком большое пространство, может привести к возникновению определенных проблем.

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

```
bash$ du
bash: /usr/bin/du: Argument list too long
```

(Спасибо S. C. за вышеприведенный пример и пояснения.)

Если сценарий изменяет переменные окружения, то они должны "экспортироваться", т.е. передаваться окружению, локальному по отношению к сценарию. Эта функция возложена на команду [export](#).

-  Сценарий может **экспортировать** переменные только дочернему процессу, т.е. командам и процессам запускаемым из данного сценария. Сценарий, запускаемый из командной строки *не может* экспортировать переменные "на верх" командной оболочке. [Дочерний процесс](#) не может

экспортировать переменные родительскому процессу.

позиционные параметры

аргументы, передаваемые скрипту из командной строки -- \$0, \$1, \$2, \$3..., где \$0 -- это название файла сценария, \$1 -- это первый аргумент, \$2 -- второй, \$3 -- третий и так далее. [\[13\]](#) Аргументы, следующие за \$9, должны заключаться в фигурные скобки, например: \${10}, \${11}, \${12}.

Специальные переменные [\\$*](#) и [\\$@](#) содержат все позиционные параметры (аргументы командной строки).

Пример 4-5. Позиционные параметры

```
#!/bin/bash

# Команда вызова сценария должна содержать по меньшей мере 10 параметров,
# например
# ./scriptname 1 2 3 4 5 6 7 8 9 10
MINPARAMS=10

echo

echo "Имя файла сценария: \"$0\"."
# Для текущего каталога добавит ./
echo "Имя файла сценария: \"`basename $0`\"."
# Добавит путь к имени файла (см. 'basename')

echo

if [ -n "$1" ]           # Проверяемая переменная заключена в кавычки.
then
    echo "Параметр #1: $1"   # необходимы кавычки для экранирования символа #
fi

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

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

# ...

if [ -n "${10}" ] # Параметры, следующие за $9 должны заключаться в фигурные
скобки
then
    echo "Параметр #10: ${10}"
fi

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

if [ $# -lt "$MINPARAMS" ]
then
    echo
    echo "Количество аргументов командной строки должно быть не менее
$MINPARAMS !"
fi
```

```
echo
exit 0
```

Скобочная нотация позиционных параметров дает довольно простой способ обращения к *последнему* аргументу, переданному в сценарий из командной строки. Такой способ подразумевает использование [косвенной адресации](#).

```
args=$#           # Количество переданных аргументов.
lastarg=${!args} # Обратите внимание: lastarg=${!$#} неприменимо.
```

В сценарии можно предусмотреть различные варианты развития событий, в зависимости от имени сценария. Для этого сценарий должен проанализировать аргумент \$0 -- имя файла сценария. Это могут быть и имена символических ссылок на файл сценария.

- ❗ Если сценарий ожидает передачи аргументов в командной строке, то при их отсутствии он получит "пустые" переменные, что может вызвать нежелательный побочный эффект. Один из способов борьбы с подобными ошибками -- добавить дополнительный символ в обеих частях операции присваивания, где используются аргументы командной строки.

```
variable1_=$1_
# Это предотвратит появление ошибок, даже при отсутствии входного аргумента.

critical_argument01=$variable1_

# Дополнительные символы всегда можно "убрать" позднее.
# Это может быть сделано примерно так:
variable1=${variable1_/_/} # Побочный эффект возникает только если имя
переменной
# $variable1_ будет начинаться с символа "_".
# Здесь используется один из вариантов подстановки параметров, обсуждаемых в
Главе 9.
# Отсутствие шаблона замены приводит к удалению.

# Более простой способ заключается
#+ в обычной проверке наличия позиционного параметра.
if [ -z $1 ]
then
    exit $POS_PARAMS_MISSING
fi
```

Пример 4-6. `wh`, [whois](#) выяснение имени домена

```
#!/bin/bash

# Команда 'whois domain-name' выясняет имя домена на одном из 3 серверов:
#                               ripe.net, cw.net, radb.net

# Разместите этот скрипт под именем 'wh' в каталоге /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-cw
# ln -s /usr/local/bin/wh /usr/local/bin/wh-radb
```



```

if [ -z "$1" ]
then
  echo "Порядок использования: `basename $0` [domain-name]"
  exit 65
fi

case `basename $0` in
# Проверка имени скрипта и, соответственно, имени сервера
  "wh"      ) whois $1@whois.ripe.net;;
  "wh-ripe") whois $1@whois.ripe.net;;
  "wh-radb") whois $1@whois.radb.net;;
  "wh-cw"   ) whois $1@whois.cw.net;;
  *        ) echo "Порядок использования: `basename $0` [domain-name]";;
esac

exit 0

---
```

Команда **shift** "сдвигает" позиционные параметры, в результате чего параметры "сдвигаются" на одну позицию влево.

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

Прежний аргумент \$1 теряется, но аргумент \$0 (*имя файла сценария*) *остаётся без изменений*. Если вашему сценарию передается большое количество входных аргументов, то команда **shift** позволит вам получить доступ к аргументам, с порядковым номером больше 9, без использования [{фигурных скобок}](#).

Пример 4-7. Использование команды **shift**

```

#!/bin/bash
# Использование команды 'shift' с целью перебора всех аргументов командной
строки.

# Назовите файл с этим сценарием, например "shft",
#+ и вызовите его с набором аргументов, например:
#      ./shft a b c def 23 skidoo

until [ -z "$1" ] # До тех пор пока не будут разобраны все входные аргументы...
do
  echo -n "$1 "
  shift
done

echo                # Дополнительная пустая строка.

exit 0
```



Команда **shift** может применяться и к входным аргументам [функций](#). См. [Пример 33-10](#).

Глава 5. Кавычки

Кавычки, ограничивающие строки с обеих сторон, служат для предотвращения интерпретации специальных символов, которые могут находиться в строке. (Символ называется "специальным", если он несет дополнительную смысловую нагрузку, например символ шаблона -- *.)

```
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]*: No such file or directory
```



Некоторые программы и утилиты могут вызываться с дополнительными параметрами, содержащими специальными символами, поэтому очень важно предотвратить интерпретацию передаваемых параметров командной оболочкой, позволяя сделать это вызываемой программой.

```
bash$ grep '[Пп]ервая' *.txt
file1.txt:Это первая строка в file1.txt.
file2.txt:Это Первая строка в file2.txt.
```

Примечательно, что "не окавыченный" вариант команды `grep [Пп]ервая *.txt` будет правильно исполняться в `Bash`, но не в `tcsh`.

Вообще, желательно использовать двойные кавычки (" ") при обращении к переменным. Это предотвратит интерпретацию специальных символов, которые могут содержаться в именах переменных, за исключением `$`, ``` (обратная кавычка) и `\` (escape -- обратный слэш). [14] То, что символ `$` попал в разряд исключений, позволяет выполнять обращение к переменным внутри строк, ограниченных двойными кавычками ("*variable*"), т.е. выполнять подстановку значений переменных (см. [Пример 4-1](#), выше).

Двойные кавычки могут быть использованы для предотвращения разбиения строки на слова. [15] Заключение строки в кавычки приводит к тому, что она передается как один аргумент, даже если она содержит [пробельные символы](#) - разделители.

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

COMMAND "This is $variable1"    # Исполнение COMMAND с одним входным аргументом:
# "This is a variable containing five words"

variable2=""                   # Пустая переменная.

COMMAND $variable2 $variable2 $variable2      # Исполнение COMMAND без аргументов.
COMMAND "$variable2" "$variable2" "$variable2" # Исполнение COMMAND с 3 "пустыми"
аргументами.
COMMAND "$variable2 $variable2 $variable2"    # Исполнение COMMAND с 1 аргументом
(и 2 пробелами).

# Спасибо S.C.
```



Заключение в кавычки аргументов команды `echo` необходимо только в том случае, когда разбиение на отдельные слова сопряжено с определенными трудностями.

Пример 5-1. Вывод "причудливых" переменных

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

```
# weirdvars.sh: Вывод "причудливых" переменных

var="' ([\\{}\\$\\'"
echo $var      # ' ([\\{}$"'
echo "$var"    # ' ([\\{}$"'      Никаких различий.


echo

IFS='\'
echo $var      # ' ([ {}$"'      \ символ-разделитель преобразован в пробел.
echo "$var"    # ' ([\\{}$"'

# Примеры выше предоставлены S.C.

exit 0
```

Одиночные кавычки (' ') схожи по своему действию с двойными кавычками, только не допускают обращение к переменным, поскольку специальный символ "\$" внутри одинарных кавычек воспринимается как обычный символ. Внутри одиночных кавычек, *любой* специальный символ, за исключением ', интерпретируется как простой символ. Одиночные кавычки ("строгие, или полные кавычки") следует рассматривать как более строгий вариант чем двойные кавычки ("нестрогие, или неполные кавычки").

 Поскольку внутри одиночных кавычек даже экранирующий (\) символ воспринимается как обычный символ, попытка вывести одиночную кавычку внутри строки, ограниченной одинарными кавычками, не даст желаемого результата.


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

echo

# Обходной метод.
echo 'Why can\'\'t I write \''\'s between single quotes'
# |-----| |-----| |-----|
# Три строки, ограниченных одинарными кавычками,
# и экранированные одиночные кавычки между ними.

# Пример любезно предоставлен Stephane Chazelas.
```

Экранирование -- это способ заключения в кавычки одиночного символа. Экранирующий (escape) символ (\) сообщает интерпретатору, что следующий за ним символ должен восприниматься как обычный символ.

 С отдельными командами и утилитами, такими как [echo](#) и [sed](#), экранирующий символ может применяться для получения обратного эффекта - когда обычные символы при экранировании приобретают специальное значение.

Специальное назначение некоторых экранированных символов

используемых совместно с **echo** и **sed**

\n

перевод строки (новая строка)

\r

перевод каретки

\t

табуляция

\v

вертикальная табуляция

\b

забой (backspace)

\a

"звонок" (сигнал)

\Oxx

ASCII-символ с кодом *0xx* в восьмеричном виде)

Пример 5-2. Экранированные символы

```
#!/bin/bash
# escaped.sh: экранированные символы

echo; echo

echo "\v\v\v\v"      # Вывод последовательности символов \v\v\v\v.
# Для вывода экранированных символов следует использовать ключ -e.
echo "======"
echo "ВЕРТИКАЛЬНАЯ ТАБУЛЯЦИЯ"
echo -e "\v\v\v\v"   # Вывод 4-х вертикальных табуляций.
echo "======"

echo "КАВЫЧКИ"
echo -e "\042"       # Выводит символ " (кавычки с восьмеричным кодом ASCII 42).
echo "======"

# Конструкция '$\X' делает использование ключа -e необязательным.
echo; echo "НОВАЯ СТРОКА И ЗВОНОК"
echo $\n'           # Перевод строки.
echo $\a'           # Звонок (сигнал).

echo "======"
echo "КАВЫЧКИ"
# Bash версии 2 и выше допускает использование конструкции '$\nnn'.
# Обратите внимание: здесь под '\nnn' подразумевается восьмеричное значение.
echo $\t \042 \t'   # Кавычки (") окруженные табуляцией.

# В конструкции '$\xhhh' допускается использовать и шестнадцатеричные значения.
echo $\t \x22 \t'   # Кавычки (") окруженные табуляцией.
# Спасибо Greg Keraunen, за это примечание.
# Ранние версии Bash допускали употребление конструкции в виде '\x022'.
echo "======"
echo

# Запись ASCII-символов в переменную.
# -----
quote=$'\042'      # запись символа " в переменную.
echo "$quote Эта часть строки ограничена кавычками, $quote а эта -- нет."

echo

# Конкатенация ASCII-символов в переменную.
triple_underscore=$'\137\137\137' # 137 -- это восьмеричный код символа '_'.

```

```

echo "$triple_underline ПОДЧЕРКИВАНИЕ $triple_underline"

echo

ABC=${'\101\102\103\010'}           # 101, 102, 103 это A, B и C соответственно.
echo $ABC

echo; echo

escape=${'\033'}                     # 033 -- восьмеричный код экранирующего
символа.
echo "\"escape\" выводится как $escape"
#                                     вывод отсутствует.

echo; echo

exit 0

```

Еще один пример использования конструкции '\$ ' ' ' вы найдете в [Пример 34-1](#).

"

кавычки

```

echo "Привет"                        # Привет
echo "Он сказал: \"Привет\"."        # Он сказал: "Привет".

```

\\$

символ доллара (если за комбинацией символов \\$ следует имя переменной, то она не будет разыменована)

```

echo "\$variable01" # выведет $variable01

```

\\

обратный слэш

```

echo "\\\" # выведет \

```



Поведение символа \ сильно зависит от того экранирован ли он, ограничен ли кавычками или находится внутри конструкции [подстановки команды](#) или во [вложенном документе](#).

```

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

# Подстановка команды
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

# Встроенный документ
cat <<EOF
\z
EOF                  # \z

cat <<EOF
\\z
EOF                  # \z

# Эти примеры предоставил Stephane Chazelas.

```

Отдельные символы в строке, которая записывается в переменную, могут быть экранированы, исключение составляет сам экранирующий символ.

```

variable=\
echo "$variable"
# Не работает - дает сообщение об ошибке:
# test.sh: : command not found
# В "чистом" виде экранирующий (escape) символ не может быть записан в переменную.
#
# Фактически, в данном примере, происходит экранирование символа перевода строки
#+ в результате получается такая команда:  variable=echo "$variable"
#+                                           ошибочное присваивание

variable=\
23skidoo
echo "$variable"      # 23skidoo
# Здесь все в порядке, поскольку вторая строка
#+ является нормальным, с точки зрения присваивания,
выражением.

variable=\
#      \^      За escape-символом следует пробел
echo "$variable"      # пробел

variable=\\
echo "$variable"      # \

variable=\\\
echo "$variable"
# Не работает - сообщение об ошибке:
# test.sh: \: command not found
#
# Первый escape-символ экранирует второй, а третий оказывается неэкранированным,
#+ результат тот же, что и в первом примере.

variable=\\\
echo "$variable"      # \\
# Второй и четвертый escape-символы экранированы.
# Это нормально.

```

Экранирование пробелов предотвращает разбиение списка аргументов командной строки на отдельные аргументы.

```

file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
# Список файлов как аргумент(ы) командной строки.

```

```

# Добавить два файла в список и вывести список.
ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list

echo "-----"

# Что произойдет, если экранировать пробелы в списке?
ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
# Ошибка: первые три файла будут "слиты" воедино
# и переданы команде 'ls -l' как один аргумент
# потому что два пробела, разделяющие аргументы (слова) -- экранированы.

```

Кроме того, `escape`-символ позволяет писать многострочные команды. Обычно, каждая команда занимает одну строку, но `escape`-символ позволяет *экранировать символ перевода строки*, в результате чего одна команда может занимать несколько строк.

```

(cd /source/directory && tar cf - . ) | \
(cd /dest/directory && tar xpvf -)
# Команда копирования дерева каталогов.
# Разбита на две строки для большей удобочитаемости.

# Альтернативный вариант:
tar cf - -C /source/directory . |
tar xpvf - -C /dest/directory
# См. примечание ниже.
# (Спасибо Stephane Chazelas.)

```



Если строка сценария заканчивается символом создания конвейера `|`, то необходимость в применении символа `\`, для экранирования перевода строки, отпадает. Тем не менее, считается хорошим тоном, всегда использовать символ `"\"` в конце промежуточных строк многострочных команд.

```

echo "foo
bar"
#foo
#bar

echo

echo 'foo
bar' # Никаких различий.
#foo
#bar

echo

echo foo\
bar # Перевод строки экранирован.
#foobar

echo

echo "foo\
bar" # Внутри "нестрогих" кавычек символ "\" интерпретируется как экранирующий.
#foobar

echo

echo 'foo\
bar' # В "строгих" кавычках обратный слэш воспринимается как обычный символ.
#foo\
#bar

```

Глава 6. Завершение и код завершения

...эта часть Bourne shell покрыта мраком, тем не менее все пользуются ею.

Chet Ramey

Команда **exit** может использоваться для завершения работы сценария, точно так же как и в программах на языке C. Кроме того, она может возвращать некоторое значение, которое может быть проанализировано вызывающим процессом.

Каждая команда возвращает *код завершения* (иногда код завершения называют *возвращаемым значением*). В случае успеха команда должна возвращать 0, а в случае ошибки -- ненулевое значение, которое, как правило, интерпретируется как код ошибки. Практически все команды и утилиты UNIX возвращают 0 в случае успешного завершения, но имеются и исключения из правил.

Аналогичным образом ведут себя функции, расположенные внутри сценария, и сам сценарий, возвращая код завершения. Код, возвращаемый функцией или сценарием, определяется кодом возврата последней команды. Команде `exit` можно явно указать код возврата, в виде: `exit nnn`, где `nnn` -- это код возврата (число в диапазоне 0 - 255).



Когда работа сценария завершается командой **exit** без параметров, то код возврата сценария определяется кодом возврата последней исполненной командой.

Код возврата последней команды хранится в специальной переменной `$?`. После исполнения кода функции, переменная `$?` хранит код завершения последней команды, исполненной в функции. Таким способом в Bash передается "значение, возвращаемое" функцией. После завершения работы сценария, код возврата можно получить, обратившись из командной строки к переменной `$?`, т.е. это будет код возврата последней команды, исполненной в сценарии.

Пример 6-1. завершение / код завершения

```
#!/bin/bash

echo hello
echo $?      # код возврата = 0, поскольку команда выполнилась успешно.

lskdf       # Несуществующая команда.
echo $?     # Ненулевой код возврата, поскольку команду выполнить не удалось.

echo

exit 113    # Явное указание кода возврата 113.
            # Проверить можно, если набрать в командной строке "echo $?"
            # после выполнения этого примера.

# В соответствии с соглашениями, 'exit 0' указывает на успешное завершение,
#+ в то время как ненулевое значение означает ошибку.
```

Переменная `$?` особенно полезна, когда необходимо проверить результат исполнения

команды (см. [Пример 12-27](#) и [Пример 12-13](#)).



Символ `!`, может выступать как логическое "НЕ" для инверсии [кода возврата](#).

Пример 6-2. Использование символа `!` для логической инверсии кода возврата

```
true # встроенная команда "true".
echo "код возврата команды \"true\" = $?" # 0

! true
echo "код возврата команды \"! true\" = $?" # 1
# Обратите внимание: символ "!" от команды необходимо отделять пробелом.
# !true вызовет сообщение об ошибке "command not found"

# Спасибо S.C.
```



В отдельных случаях коды возврата должны иметь [предопределенные значения](#) и не должны задаваться пользователем.

Глава 7. Проверка условий

практически любой язык программирования включает в себя условные операторы, предназначенные для проверки условий, чтобы выбрать тот или иной путь развития событий в зависимости от этих условий. В Bash, для проверки условий, имеется команда `test`, различного вида скобочные операторы и условный оператор `if/then`.

7.1. Конструкции проверки условий

- Оператор `if/then` проверяет -- является ли [код завершения](#) списка команд 0 (поскольку 0 означает "успех"), и если это так, то выполняет одну, или более, команд, следующие за словом `then`.
- Существует специальная команда -- `[` ([левая квадратная скобка](#)). Она является синонимом команды `test`, и является [встроенной](#) командой (т.е. более эффективной, в смысле производительности). Эта команда воспринимает свои аргументы как выражение сравнения или как файловую проверку и возвращает код завершения в соответствии с результатами проверки (0 -- истина, 1 -- ложь).
- Начиная с версии 2.02, Bash предоставляет в распоряжение программиста конструкцию `[[...]]` *расширенный вариант команды test*, которая выполняет сравнение способом более знакомым программистам, пишущим на других языках программирования. Обратите внимание: `[[` -- это [зарезервированное слово](#), а не команда.

Bash исполняет `[[$a -lt $b]]` как один элемент, который имеет код возврата.

Круглые скобки `((...))` и предложение `let...` так же возвращают код 0, если результатом арифметического выражения является ненулевое значение. Таким образом, [арифметические выражения](#) могут участвовать в операциях сравнения.

Предложение `let "1<2"` возвращает 0 (так как результат сравнения `"1<2" -- "1"`, или "истина")
`((0 && 1))` возвращает 1 (так как результат операции `"0 && 1" -- "0"`, или "ложь")

- Условный оператор **if** проверяет код завершения любой команды, а не только результат выражения, заключенного в квадратные скобки.

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

if grep -q Bash file
then echo "Файл содержит, как минимум, одно слово Bash."
fi

if COMMAND_WHOSE_EXIT_STATUS_IS_0_UNLESS_ERROR_OCCURRED
then echo "Команда выполнена успешно."
else echo "Обнаружена ошибка при выполнении команды."
fi
```

- Оператор **if/then** допускает наличие вложенных проверок.

```
if echo "Следующий *if* находится внутри первого *if*."

    if [[ $comparison = "integer" ]]
    then (( a < b ))
    else
        [[ $a < $b ]]
    fi

then
    echo '$a меньше $b'
fi
```

Это детальное описание конструкции "if-test" любезно предоставлено Stephane Chazelas.

Пример 7-1. Что есть "истина"?

```
#!/bin/bash

echo

echo "Проверяется \"0\""
if [ 0 ] # ноль
then
    echo "0 -- это истина."
else
    echo "0 -- это ложь."
fi # 0 -- это истина.

echo

echo "Проверяется \"1\""
if [ 1 ] # единица
then
    echo "1 -- это истина."
```

```

else
    echo "1 -- это ложь."
fi
    # 1 -- это ложь.

echo

echo "Testing \"-1\"""
if [ -1 ]      # минус один
then
    echo "-1 -- это истина."
else
    echo "-1 -- это ложь."
fi
    # -1 -- это истина.

echo

echo "Проверяется \\"NULL\"""
if [ ]      # NULL (пустое условие)
then
    echo "NULL -- это истина."
else
    echo "NULL -- это ложь."
fi
    # NULL -- это ложь.

echo

echo "Проверяется \\"хуз\"""
if [ хуз ]   # строка
then
    echo "Случайная строка -- это истина."
else
    echo "Случайная строка -- это ложь."
fi
    # Случайная строка -- это истина.

echo

echo "Проверяется \\"$хуз\"""
if [ $хуз ]   # Проверка, если $хуз это null, но...
                # только для неинициализированных переменных.
then
    echo "Неинициализированная переменная -- это истина."
else
    echo "Неинициализированная переменная -- это ложь."
fi
    # Неинициализированная переменная -- это ложь.

echo

echo "Проверяется \\"-n $хуз\"""
if [ -n "$хуз" ]      # Более корректный вариант.
then
    echo "Неинициализированная переменная -- это истина."
else
    echo "Неинициализированная переменная -- это ложь."
fi
    # Неинициализированная переменная -- это ложь.

echo

хуз=      # Инициализирована пустым значением.

echo "Проверяется \\"-n $хуз\"""
if [ -n "$хуз" ]
then
    echo "Пустая переменная -- это истина."
else
    echo "Пустая переменная -- это ложь."
fi
    # Пустая переменная -- это ложь.

```

```

echo

# Кргда "ложь" истинна?

echo "Проверяется \"false\""
if [ "false" ]           # это обычная строка "false".
then
    echo "\"false\" -- это истина." #+ и она истинна.
else
    echo "\"false\" -- это ложь."
fi                       # "false" -- это истина.

echo

echo "Проверяется \"\$false\"" # Опять неинициализированная переменная.
if [ "$false" ]
then
    echo "\"\$false\" -- это истина."
else
    echo "\"\$false\" -- это ложь."
fi                       # "$false" -- это ложь.
                        # Теперь мв получили ожидаемый результат.

echo

exit 0

```

Упражнение. Объясните результаты, полученные в [Пример 7-1](#).

```

if [ condition-true ]
then
    command 1
    command 2
    ...
else
    # Необязательная ветка (можно опустить, если в ней нет необходимости).
    # Дополнительный блок кода,
    # исполняемый в случае, когда результат проверки -- "ложь".
    command 3
    command 4
    ...
fi

```



Когда *if* и *then* располагаются в одной строке, то конструкция *if* должна завершаться точкой с запятой. И *if*, и *then* -- это [зарезервированные слова](#). Зарезервированные слова начинают инструкцию, которая должна быть завершена прежде, чем в той же строке появится новая инструкция.

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

Else if и elif

elif

`elif` -- это краткая форма записи конструкции `else if`. Применяется для построения многоярусных инструкций `if/then`.


```
if [ condition1 ]
then
```

```

    command1
    command2
    command3
elif [ condition2 ]
# То же самое, что и else if
then
    command4
    command5
else
    default-command
fi

```

Конструкция `if test condition-true` является точным эквивалентом конструкции `if [condition-true]`, где левая квадратная скобка `[` выполняет те же действия, что и команда `test`. Закрывающая правая квадратная скобка `]` не является абсолютно необходимой, однако, более новые версии Bash требуют ее наличие.

 Команда `test` -- это [встроенная](#) команда Bash, которая выполняет проверки файлов и производит сравнение строк. Таким образом, в Bash-скриптах, команда `test` *не* вызывает внешнюю (`/usr/bin/test`) утилиту, которая является частью пакета `sh-utils`. Аналогично, `[` не производит вызов утилиты `/usr/bin/`, которая является символической ссылкой на `/usr/bin/test`.

```

bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '['
[[ is a shell keyword
bash$ type ']'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found

```

Пример 7-2. Эквиваленты команды `test -- /usr/bin/test, [, и /usr/bin/`

```

#!/bin/bash

echo

if test -z "$1"
then
    echo "Аргументы командной строки отсутствуют."
else
    echo "Первый аргумент командной строки: $1."
fi

echo

if /usr/bin/test -z "$1"      # Дает тот же результат, что и встроенная команда
"test".
then
    echo "Аргументы командной строки отсутствуют."
else
    echo "Первый аргумент командной строки: $1."
fi

echo

if [ -z "$1" ]              # Функционально идентично вышеприведенному блоку кода.
#   if [ -z "$1"           эта конструкция должна работать, но...

```

```

#+ Bash выдает сообщение об отсутствующей закрывающей скобке.
then
    echo "Аргументы командной строки отсутствуют."
else
    echo "Первый аргумент командной строки: $1."
fi

echo


if /usr/bin/[ -z "$1"          # Функционально идентично вышеприведенному блоку кода.
# if /usr/bin/[ -z "$1" ]    # Работает, но выдает сообщение об ошибке.
then
    echo "Аргументы командной строки отсутствуют."
else
    echo "Первый аргумент командной строки: $1."
fi

echo

exit 0

```

Конструкция `[[]]` более универсальна, по сравнению с `[]`. Этот *расширенный вариант команды `test`* перекочевал в Bash из *ksh88*.


-  Внутри этой конструкции не производится никакой дополнительной интерпретации имен файлов и не производится разбиение аргументов на отдельные слова, но допускается подстановка параметров и команд.


```

file=/etc/passwd

if [[ -e $file ]]
then
    echo "Файл паролей найден."
fi

```

-  Конструкция `[[...]]` более предпочтительна, нежели `[...]`, поскольку поможет избежать некоторых логических ошибок. Например, операторы `&&`, `||`, `<` и `>` внутри `[[]]` вполне допустимы, в то время как внутри `[]` порождают сообщения об ошибках.

-  Строго говоря, после оператора **if**, ни команда **test**, ни квадратные скобки (`[]` или `[[]]`) не являются обязательными.

```

dir=/home/bozo

if cd "$dir" 2>/dev/null; then # "2>/dev/null" подавление вывода сообщений об
    echo "Переход в каталог $dir выполнен."
else
    echo "Невозможно перейти в каталог $dir."
fi

```

Инструкция `if COMMAND` возвращает код возврата команды `COMMAND`.

Точно так же, условие, находящееся внутри квадратных скобок может быть проверено без использования оператора **if**.

```

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

home=/home/bozo

```

```
[ -d "$home" ] || echo "каталог $home не найден."
```

Внутри `(())` производится вычисление арифметического выражения. Если результатом вычислений является ноль, то возвращается 1, или "ложь". Ненулевой результат дает код возврата 0, или "истина". То есть полная противоположность инструкциям `test` и `[]`, обсуждавшимся выше.

Пример 7-3. Арифметические выражения внутри `(())`

```
#!/bin/bash
# Проверка арифметических выражений.

# Инструкция (( ... )) вычисляет арифметические выражения.
# Код возврата противоположен коду возврата инструкции [ ... ] !

(( 0 ))
echo "Код возврата \"(( 0 ))\": $?."          # 1

(( 1 ))
echo "Код возврата \"(( 1 ))\": $?."          # 0

(( 5 > 4 ))
echo "Код возврата \"(( 5 > 4 ))\": $?."      # true
                                                # 0

(( 5 > 9 ))
echo "Код возврата \"(( 5 > 9 ))\": $?."      # false
                                                # 1

(( 5 - 5 ))
echo "Код возврата \"(( 5 - 5 ))\": $?."      # 0
                                                # 1

(( 5 / 4 ))
echo "Код возврата \"(( 5 / 4 ))\": $?."      # Деление, все в порядке
                                                # 0

(( 1 / 2 ))
echo "Код возврата \"(( 1 / 2 ))\": $?."      # Результат деления < 1.
                                                # Округляется до 0.
                                                # 1

(( 1 / 0 )) 2>/dev/null
echo "Код возврата \"(( 1 / 0 ))\": $?."      # Деление на 0.
                                                # 1

# Для чего нужна инструкция "2>/dev/null" ?
# Что произойдет, если ее убрать?
# Попробуйте убрать ее и выполнить сценарий.

exit 0
```

7.2. Операции проверки файлов

Возвращает true если...

-e

файл существует

-f

обычный файл (не каталог и не файл устройства)

-s

ненулевой размер файла

-d

файл является каталогом

-b

файл является блочным устройством (floppy, cdrom и т.п.)

-c

файл является символьным устройством (клавиатура, модем, звуковая карта и т.п.)

-p

файл является каналом

-h

файл является символической ссылкой

-L

файл является символической ссылкой

-S

файл является сокетом

-t

файл ([дескриптор](#)) связан с терминальным устройством

Этот ключ может использоваться для проверки -- является ли файл стандартным устройством ввода `stdin` ([-t 0]) или стандартным устройством вывода `stdout` ([-t 1]).

-r

файл доступен для чтения (*пользователю, запустившему сценарий*)

-w

файл доступен для записи (*пользователю, запустившему сценарий*)

-x

файл доступен для исполнения (*пользователю, запустившему сценарий*)

-g

set-group-id (sgid) флаг для файла или каталога установлен

Если для каталога установлен флаг `sgid`, то файлы, создаваемые в таком каталоге, наследуют идентификатор группы каталога, который может не совпадать с

идентификатором группы, к которой принадлежит пользователь, создавший файл. Это может быть полезно для каталогов, в которых хранятся файлы, общедоступные для группы пользователей.

-u

set-user-id (suid) флаг для файла установлен

Установленный флаг `suid` приводит к изменению привилегий запущенного процесса на привилегии владельца исполняемого файла. Исполняемые файлы, владельцем которых является `root`, с установленным флагом `set-user-id` запускаются с привилегиями `root`, даже если их запускает обычный пользователь. [16] Это может оказаться полезным для некоторых программ (таких как `pppd` и `cdrecord`), которые осуществляют доступ к аппаратной части компьютера. В случае отсутствия флага `suid`, программы не смогут быть запущены рядовым пользователем, не обладающим привилегиями `root`.

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

Файл с установленным флагом `suid` отображается с включенным флагом `s` в поле прав доступа.

-k

флаг `sticky bit` (бит фиксации) установлен

Общеизвестно, что флаг "sticky bit" -- это специальный тип прав доступа к файлам. Программы с установленным флагом "sticky bit" остаются в системном кэше после своего завершения, обеспечивая тем самым более быстрый запуск программы. [17] Если флаг установлен для каталога, то это приводит к ограничению прав на запись. Установленный флаг "sticky bit" отображается в виде символа `t` в поле прав доступа.

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

Если пользователь не является владельцем каталога, с установленным "sticky bit", но имеет право на запись в каталог, то он может удалять только те файлы в каталоге, владельцем которых он является. Это предотвращает удаление и перезапись "чужих" файлов в общедоступных каталогах, таких как `/tmp`.

-O

вы являетесь владельцем файла

-G

вы принадлежите к той же группе, что и файл

-N

файл был модифицирован с момента последнего чтения

f1 -nt f2

файл `f1` более новый, чем `f2`

f1 -ot f2

файл *f1* более старый, чем *f2*

f1 -ef f2

файлы *f1* и *f2* являются "жесткими" ссылками на один и тот же файл

!

"НЕ" -- логическое отрицание (инверсия) результатов всех вышеприведенных проверок (возвращается true если условие отсутствует).

Пример 7-4. Проверка "битых" ссылок

```
#!/bin/bash
# broken-link.sh
# Автор Lee Bigelow <ligelowbee@yahoo.com>
# Используется с его разрешения.

#Сценарий поиска "битых" ссылок и их вывод в "окавыченном" виде
#таким образом они могут передаваться утилите xargs для дальнейшей обработки :)
#например. broken-link.sh /somedir /someotherdir|xargs rm
#
#На всякий случай приведу лучший метод:
#
#find "somedir" -type l -print0|\
#xargs -r0 file|\
#grep "broken symbolic"|
#sed -e 's/^\ |: *broken symbolic.*$/"/g'
#
#но это не чисто BASH-евский метод, а теперь сам сценарий.
#Внимание! будьте осторожны с файловой системой /proc и циклическими ссылками!
#####

#Если скрипт не получает входных аргументов,
#то каталогом поиска является текущая директория
#В противном случае, каталог поиска задается из командной строки
#####
[ $# -eq 0 ] && directories=`pwd` || directories=$@

#Функция linkchk проверяет каталог поиска
#на наличие в нем ссылок на несуществующие файлы, и выводит их имена.
#Если анализируемый файл является каталогом,
#то он передается функции linkcheck рекурсивно.
#####
linkchk () {
    for element in $1/*; do
        [ -h "$element" -a ! -e "$element" ] && echo "\"$element\""
        [ -d "$element" ] && linkchk $element
        # Само собой, '-h' проверяет символические ссылки, '-d' -- каталоги.
    done
}

#Вызов функции linkchk для каждого аргумента командной строки,
#если он является каталогом. Иначе выводится сообщение об ошибке
#и информация о порядке пользования скриптом.
#####
for directory in $directories; do
    if [ -d $directory ]
    then linkchk $directory
    else
        echo "$directory не является каталогом"
        echo "Порядок использования: $0 dir1 dir2 ..."
    fi
fi
```

done

exit 0

[Пример 28-1](#), [Пример 10-7](#), [Пример 10-3](#), [Пример 28-3](#) и [Пример А-2](#) так же иллюстрируют операции проверки файлов.

7.3. Операции сравнения

сравнение целых чисел

-eq

равно

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

-ne

не равно

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

-gt

больше

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

-ge

больше или равно

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

-lt

меньше

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

-le

меньше или равно

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

<

меньше (внутри [двойных круглых скобок](#))

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

<=

меньше или равно (внутри двойных круглых скобок)

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

>

больше (внутри двойных круглых скобок)

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

>=

больше или равно (внутри двойных круглых скобок)

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

сравнение строк

=

равно

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

==

равно

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

Синоним оператора =.

```
[[ $a == z* ]] # истина, если $a начинается с символа "z" (сравнение по шаблону)
```

```
[[ $a == "z*" ]] # истина, если $a равна z*
```

```
[ $a == z* ] # имеют место подстановка имен файлов и разбиение на слова
```

```
[ "$a" == "z*" ] # истина, если $a равна z*
```

```
# Спасибо S.C.
```

!=

не равно

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

Этот оператор используется при поиске по шаблону внутри [\[\[...\]\]](#).

<

меньше, в смысле величины ASCII-кодов

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

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

Обратите внимание! Символ "<" необходимо экранировать внутри [].

>

больше, в смысле величины ASCII-кодов

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

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

Обратите внимание! Символ ">" необходимо экранировать внутри [].

См. [Пример 25-6](#) относительно применения этого оператора сравнения.

-z

строка "пустая", т.е. имеет нулевую длину

-n

строка не "пустая".



Оператор - n требует, чтобы строка была заключена в кавычки внутри квадратных скобок. Как правило, проверка строк, не заключенных в кавычки, оператором ! - z, или просто указание строки без кавычек внутри квадратных скобок (см. [Пример 7-6](#)), проходит нормально, однако это небезопасная, с точки зрения отказоустойчивости, практика. *Всегда* заключайте проверяемую строку в кавычки. [\[18\]](#)

Пример 7-5. Операции сравнения

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

```
a=4  
b=5
```

```
# Здесь переменные "a" и "b" могут быть как целыми числами, так и строками.  
# Здесь наблюдается некоторое размывание границ  
#+ между целочисленными и строковыми переменными,  
#+ поскольку переменные в Bash не имеют типов.
```

```
# Bash выполняет целочисленные операции над теми переменными,  
#+ которые содержат только цифры  
# Будьте внимательны!
```

```
echo
```

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

```
echo
```

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

```

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

# Оба варианта, "-ne" и "!=", работают правильно.

echo

exit 0

```

Пример 7-6. Проверка -- является ли строка пустой

```

#!/bin/bash
# str-test.sh: Проверка пустых строк и строк, не заключенных в кавычки,
# Используется конструкция if [ ... ]

# Если строка не инициализирована, то она не имеет никакого определенного значения.
# Такое состояние называется "null" (пустая) (это не то же самое, что ноль).

if [ -n $string1 ] # $string1 не была объявлена или инициализирована.
then
echo "Строка \"$string1\" не пустая."
else
echo "Строка \"$string1\" пустая."
fi
# Неверный результат.
# Выводится сообщение о том, что $string1 не пустая,
#+не смотря на то, что она не была инициализирована.

echo

# Попробуем еще раз.

if [ -n "$string1" ] # На этот раз, переменная $string1 заключена в кавычки.
then
echo "Строка \"$string1\" не пустая."
else
echo "Строка \"$string1\" пустая."
fi # Внутри квадратных скобок заключайте строки в кавычки!

echo

if [ $string1 ] # Опустим оператор -n.
then
echo "Строка \"$string1\" не пустая."
else
echo "Строка \"$string1\" пустая."
fi
# Все работает прекрасно.
# Квадратные скобки -- [ ], без посторонней помощи определяют, что строка пустая.
# Тем не менее, хорошим тоном считается заключать строки в кавычки ("string1").
#
# Как указывает Stephane Chazelas,
# if [ $string 1 ] один аргумент "]"
# if [ "$string 1" ] два аргумента, пустая "$string1" и "]"

echo

```

```

string1=initialized

if [ $string1 ]          # Опять, попробуем строку без ничего.
then
    echo "Строка \"$string1\" не пустая."
else
    echo "Строка \"$string1\" пустая."
fi
# И снова получим верный результат.
# И опять-таки, лучше поместить строку в кавычки ("string1"), поскольку...

string1="a = b"

if [ $string1 ]          # И снова, попробуем строку без ничего..
then
    echo "Строка \"$string1\" не пустая."
else
    echo "Строка \"$string1\" пустая."
fi
# Строка без кавычек дает неверный результат!

exit 0
# Спвсибо Florian Wisser, за предупреждение.

```

Пример 7-7. zmost

```

#!/bin/bash

#Просмотр gz-файлов с помощью утилиты 'most'

NOARGS=65
NOTFOUND=66
NOTGZIP=67

if [ $# -eq 0 ] # то же, что и: if [ -z "$1" ]
# $1 должен существовать, но может быть пустым: zmost "" arg2 arg3
then
    echo "Порядок использования: `basename $0` filename" >&2
    # Сообщение об ошибке на stderr.
    exit $NOARGS
    # Код возврата 65 (код ошибки).
fi

filename=$1

if [ ! -f "$filename" ] # Кавычки необходимы на тот случай, если имя файла содержит
пробелы.
then
    echo "Файл $filename не найден!" >&2
    # Сообщение об ошибке на stderr.
    exit $NOTFOUND
fi

if [ ${filename##*.} != "gz" ]
# Квадратные скобки нужны для выполнения подстановки значения переменной
then
    echo "Файл $1 не является gz-файлом!"
    exit $NOTGZIP
fi

zcat $1 | most

# Используется утилита 'most' (очень похожа на 'less').
# Последние версии 'most' могут просматривать сжатые файлы.
# Можно вставить 'more' или 'less', если пожелаете.

```

```
exit $? # Сценарий возвращает код возврата, полученный по конвейеру.  
# На самом деле команда "exit $?" не является обязательной,  
# так как работа скрипта завершится здесь в любом случае,
```

построение сложных условий проверки

-a

логическое И (and)

exp1 -a exp2 возвращает true, если оба выражения, и *exp1*, и *exp2* истинны.

-o

логическое ИЛИ (or)

exp1 -o exp2 возвращает true, если хотябы одно из выражений, *exp1* или *exp2* истинно.

Они похожи на операторы Bash **&&** и **||**, употребляемые в [двойных квадратных скобках](#).

```
[[ condition1 && condition2 ]]
```

Операторы **-o** и **-a** употребляются совместно с командой **test** или внутри одинарных квадратных скобок.

```
if [ "$exp1" -a "$exp2" ]
```

Чтобы увидеть эти операторы в действии, смотрите [Пример 8-3](#) и [Пример 25-11](#).

7.4. Вложенные условные операторы if/then

Операторы проверки условий **if/then** могут быть вложенными друг в друга. Конечный результат будет таким же как если бы результаты всех проверок были объединены оператором **&&**.

```
if [ condition1 ]  
then  
  if [ condition2 ]  
  then  
    do-something # Только если оба условия "condition1" и "condition2" истинны.  
  fi  
fi
```

См. [Пример 34-4](#) -- пример использования вложенных операторов *if/then*.

7.5. Проверка степени усвоения материала

Для запуска X-сервера может быть использован файл `xinitrc`. Этот файл содержит некоторое число операторов *if/then*. Ниже приводится отрывок из этого файла.

```
if [ -f $HOME/.Xclients ]; then
    exec $HOME/.Xclients
elif [ -f /etc/X11/xinit/Xclients ]; then
    exec /etc/X11/xinit/Xclients
else
    # failsafe settings.  Although we should never get here
    # (we provide fallbacks in Xclients as well) it can't hurt.
    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 действия условных операторов в вышеприведенном отрывке, затем просмотрите файл `/etc/X11/xinit/xinitrc` и проанализируйте его. Возможно вам придется обратиться к разделам, посвященным [grep](#), [sed](#) и [регулярным выражениям](#).

Глава 8. Операции и смежные темы

8.1. Операторы

присваивание

variable assignment

Инициализация переменной или изменение ее значения

=

Универсальный оператор присваивания, пригоден как для сравнения целых чисел, так и для сравнения строк.

```
var=27
category=minerals # Пробелы до и после оператора "=" -- недопустимы.
```



Пусть вас не смущает, что оператор присваивания ("`=`"), по своему внешнему виду, совпадает с оператором сравнения (`=`).

Здесь знак "`=`" выступает в качестве оператора сравнения

```
if [ "$string1" = "$string2" ]
# if [ "X$string1" = "X$string2" ] более отказоустойчивый вариант,
# предохраняет от "сваливания" по ошибке в случае, когда одна из переменных
# пуста.
# (добавленные символы "X" компенсируют друг друга.)
```

```
then
  command
fi
```

арифметические операторы

+

сложение

-

вычитание

*

умножение

/

деление

**

возведение в степень

В Bash, начиная с версии 2.02, был введен оператор возведения в степень --
"***".

```
let "z=5**3"
echo "z = $z"    # z = 125
```

%

модуль (деление по модулю), возвращает остаток от деления

```
bash$ echo `expr 5 % 3`
2
```

Этот оператор может применяться в алгоритмах генерации псевдослучайных чисел в заданном диапазоне (см. [Пример 9-23](#) и [Пример 9-25](#)), для форматирования вывода на экран (см. [Пример 25-10](#) и [Пример A-7](#)), и даже для генерации простых чисел (см. [Пример A-18](#)). На удивление часто операцию деления по модулю можно встретить в различных численных алгоритмах.

Пример 8-1. Наибольший общий делитель

```
#!/bin/bash
# gcd.sh: поиск наибольшего общего делителя
#           по алгоритму Эвклида

# Под "наибольшим общим делителем" (нод) двух целых чисел
#+ понимается наибольшее целое число, которое делит оба делимых без остатка.
```

```

# Алгоритм Эвклида выполняет последовательное деление.
# В каждом цикле,
#+ делимое <--- делитель
#+ делитель <--- остаток
#+ до тех пор, пока остаток не станет равным нулю (остаток = 0).
#+ The gcd = dividend, on the final pass.
#
# Замечательное описание алгоритма Эвклида можно найти
# на сайте Jim Loy, http://www.jimloy.com/number/euclids.htm.

# -----
# Проверка входных параметров
ARGS=2
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
    echo "Порядок использования: `basename $0` первое-число второе-число"
    exit $E_BADARGS
fi
# -----

gcd ()
{
    dividend=$1          # Начальное присваивание.
    divisor=$2          # В сущности, не имеет значения
                        #+ какой из них больше.
                        # Почему?

    remainder=1        # Если переменные неинициализировать,
                        #+ то работа сценария будет прервана по ошибке
                        #+ в первом же цикле.

    until [ "$remainder" -eq 0 ]
    do
        let "remainder = $dividend % $divisor"
        dividend=$divisor      # Повторить цикл с новыми исходными данными
        divisor=$remainder
    done                      # алгоритм Эвклида
}                             # последнее $dividend и есть нод.

gcd $1 $2

echo; echo "НОД чисел $1 и $2 = $dividend"; echo

# Упражнение :
# -----
# Вставьте дополнительную проверку входных аргументов,
#+ и предусмотрите завершение работы сценария с сообщением об ошибке, если
#+ входные аргументы не являются целыми числами.

exit 0

```

+=

"плюс-равно" (увеличивает значение переменной на заданное число)

let "var += 5" значение переменной var будет увеличено на 5.

-=

"минус-равно" (уменьшение значения переменной на заданное число)

`*=`

"умножить-равно" (умножить значение переменной на заданное число, результат записать в переменную)

`let "var *= 4"` значение переменной `var` будет увеличено в 4 раза.

`/=`

"слэш-равно" (уменьшение значения переменной в заданное число раз)

`%=`

"процент-равно" (найти остаток от деления значения переменной на заданное число, результат записать в переменную)

Арифметические операторы очень часто используются совместно с командами [expr](#) и [let](#).

Пример 8-2. Арифметические операции

```
#!/bin/bash
# От 1 до 6 пятью различными способами.

n=1; echo -n "$n "

let "n = $n + 1" # let "n = n + 1" тоже допустимо
echo -n "$n "

: $((n = $n + 1))
# оператор ":" обязателен, поскольку в противном случае, Bash будет
#+ интерпретировать выражение "$((n = $n + 1))" как команду.
echo -n "$n "

n=$(( $n + 1 ))
echo -n "$n "

: $[ n = $n + 1 ]
# оператор ":" обязателен, поскольку в противном случае, Bash будет
#+ интерпретировать выражение "$[ n = $n + 1 ]" как команду.
# Не вызывает ошибки даже если "n" содержит строку.
echo -n "$n "

n=$[ $n + 1 ]
# Не вызывает ошибки даже если "n" содержит строку.
#* Старайтесь избегать употребления такой конструкции,
#+ поскольку она уже давно устарела и не переносима.
echo -n "$n "; echo

# Спасибо Stephane Chazelas.


exit 0
```



Целые числа в Bash фактически являются знаковыми *длинными* целыми (32-бит), с диапазоном изменений от -2147483648 до 2147483647. Если в результате какой либо операции эти пределы будут превышены, то результат получится ошибочным.

```
a=2147483646
echo "a = $a" # a = 2147483646
let "a+=1" # Увеличить "a" на 1.
```

```
echo "a = $a"      # a = 2147483647
let "a+=1"         # увеличить "a" еще раз, с выходом за границы диапазона.
echo "a = $a"      # a = -2147483648
#                # ОШИБКА! (выход за границы диапазона)
```

 Bash ничего не знает о существовании чисел с плавающей запятой. Такие числа, из-за наличия символа десятичной точки, он воспринимает как строки.

```
a=1.5

let "b = $a + 1.3" # Ошибка.
# t2.sh: let: b = 1.5 + 1.3: syntax error in expression (error token is ".5 + 1.3")

echo "b = $b"      # b=1
```

Для работы с числами с плавающей запятой в сценариях можно использовать утилиту-калькулятор [bc](#).

битовые операции. Битовые операции очень редко используются в сценариях командного интерпретатора. Их главное назначение, на мой взгляд, установка и проверка некоторых значений, читаемых из портов ввода-вывода и сокетов. "Битовые операции" гораздо более уместны в компилирующих языках программирования, таких как C и C++.

битовые операции

<<

сдвигает на 1 бит влево (умножение на 2)

<<=

"сдвиг-влево-равно"

let "var <<= 2" значение переменной var сдвигается влево на 2 бита (умножается на 4)

>>

сдвиг вправо на 1 бит (деление на 2)

>>=

"сдвиг-вправо-равно" (имеет смысл обратный <<=)

&

по-битовое И (AND)

&=

"по-битовое И-равно"

|

по-битовое ИЛИ (OR)

|=

"по-битовое ИЛИ-равно"

~

по-битовая инверсия

!

По-битовое отрицание

^

по-битовое ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)

^=

"по-битовое ИСКЛЮЧАЮЩЕЕ-ИЛИ-равно"

логические операции

&&

логическое И (and)

```
if [ $condition1 ] && [ $condition2 ]
# То же самое, что: if [ $condition1 -a $condition2 ]
# Возвращает true если оба операнда condition1 и condition2 истинны...

if [[ $condition1 && $condition2 ]] # То же верно
# Обратите внимание: оператор && не должен использоваться внутри [ ... ].
```



оператор &&, в зависимости от контекста, может так же использоваться в [И-списках](#) для построения составных команд.

||

логическое ИЛИ (or)

```
if [ $condition1 ] || [ $condition2 ]
# То же самое, что: if [ $condition1 -o $condition2 ]
# Возвращает true если хотя бы один из операндов истинен...

if [[ $condition1 || $condition2 ]] # Also works.
# Обратите внимание: оператор || не должен использоваться внутри [ ... ].
```



Bash производит проверку [кода возврата](#) КАЖДОГО из операндов в логических выражениях.

Пример 8-3. Построение сложных условий, использующих && и ||

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

```
a=24
b=47
```

```

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
    echo "Первая проверка прошла успешно."
else
    echo "Первая проверка не прошла."
fi

# OKA:  if [ "$a" -eq 24 && "$b" -eq 47 ]
#        пытается выполнить ' [ "$a" -eq 24 '
#        и терпит неудачу наткнувшись на ']'.
#
#   if [[ $a -eq 24 && $b -eq 24 ]] это правильный вариант
#   (в строке 17 оператор "&&" имеет иной смысл, нежели в строке 6.)
#   Спасибо Stephane Chazelas.

if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
    echo "Вторая проверка прошла успешно."
else
    echo "Вторая проверка не прошла."
fi

# Опции -a и -o предоставляют
#+ альтернативный механизм проверки условий.
# Спасибо Patrick Callahan.

if [ "$a" -eq 24 -a "$b" -eq 47 ]
then
    echo "Третья проверка прошла успешно."
else
    echo "Третья проверка не прошла."
fi

if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
    echo "Четвертая проверка прошла успешно."
else
    echo "Четвертая проверка не прошла."
fi

a=rhino
b=crocodile
if [ "$a" = rhino ] && [ "$b" = crocodile ]
then
    echo "Пятая проверка прошла успешно."
else
    echo "Пятая проверка не прошла."
fi

exit 0

```

Операторы && и || могут использоваться и в арифметических вычислениях.

```

bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
1 0 1 0

```

прочие операции

запятая

С помощью оператора **запятая** можно связать несколько арифметических в одну последовательность. При разборе таких последовательностей, командный интерпретатор вычисляет все выражения (которые могут иметь побочные эффекты) в последовательности и возвращает результат последнего.

```
let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
echo "t1 = $t1"           # t1 = 11

let "t2 = ((a = 9, 15 / 3))" # Выполняется присваивание "a" = 9,
                           #+ а затем вычисляется "t2".
echo "t2 = $t2    a = $a"   # t2 = 5    a = 9
```

Оператор запятая чаще всего находит применение в [циклах for](#). См. [Пример 10-12](#).

8.2. Числовые константы

Интерпретатор командной оболочки воспринимает числа как десятичные, в противном случае числу должен предшествовать специальный префикс, либо число должно быть записано в особой нотации. Числа, начинающиеся с символа *0*, считаются *восьмеричными*. если числу предшествует префикс *0x*, то число считается *шестнадцатиричным*. Число, в записи которого присутствует символ *#*, расценивается как запись числа с указанием основы счисления в виде *ОСНОВА#ЧИСЛО*.

Пример 8-4. Различные представления числовых констант

```
#!/bin/bash
# numbers.sh: Различные представления числовых констант.

# Десятичное: по-умолчанию
let "dec = 32"
echo "десятичное число = $dec"           # 32
# Впрочем-то ничего необычного.

# Восьмеричное: числа начинаются с '0' (нуля)
let "oct = 032"
echo "восьмеричное число = $oct"         # 26
# Результат печатается в десятичном виде.
# -----

# Шестнадцатиричное: числа начинаются с '0x' или '0X'
let "hex = 0x32"
echo "шестнадцатиричное число = $hex"   # 50
# Результат печатается в десятичном виде.

# Другие основы счисления: ОСНОВА#ЧИСЛО
# ОСНОВА должна быть между 2 и 64.
# для записи ЧИСЛА должен использоваться соответствующий ОСНОВЕ диапазон символов,
# см. ниже.

let "bin = 2#111100111001101"
echo "двоичное число = $bin"            # 31181

let "b32 = 32#77"
echo "32-ричное число = $b32"           # 231
```



```

let "b64 = 64#@_"
echo "64-ричное число = $b64"                # 4094
#
# Нотация ОСНОВА#ЧИСЛО может использоваться на ограниченном
#+ диапазоне основ счисления (от 2 до 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 замечание:
# -----
# Использование символов, для записи числа, выходящих за диапазо,
#+ соответствующий ОСНОВЕ счисления
#+ будет приводить к появлению сообщений об ошибках.

let "bad_oct = 081"
# numbers.sh: let: oct = 081: value too great for base (error token is "081")
#           Для записи восьмеричных чисел допускается использовать
#+           только цифры в диапазоне 0 - 7.

exit 0      # Спасибо Rich Bartell и Stephane Chazelas, за разъяснения.

```

Часть 3. Углубленный материал

Содержание

9. [К вопросу о переменных](#)
 - 9.1. [Внутренние переменные](#)
 - 9.2. [Работа со строками](#)
 - 9.2.1. [Использование awk при работе со строками](#)
 - 9.2.2. [Дальнейшее обсуждение](#)
 - 9.3. [Подстановка параметров](#)
 - 9.4. [Объявление переменных: **declare** и **typeset**](#)
 - 9.5. [Косвенные ссылки на переменные](#)
 - 9.6. [\\$RANDOM: генерация псевдослучайных целых чисел](#)
 - 9.7. [Двойные круглые скобки](#)
10. [Циклы и ветвления](#)
 - 10.1. [Циклы](#)
 - 10.2. [Вложенные циклы](#)
 - 10.3. [Управление ходом выполнения цикла](#)
 - 10.4. [Операторы выбора](#)
11. [Внутренние команды](#)
 - 11.1. [Команды управления заданиями](#)
12. [Внешние команды, программы и утилиты](#)
 - 12.1. [Базовые команды](#)
 - 12.2. [Более сложные команды](#)
 - 12.3. [Команды для работы с датой и временем](#)
 - 12.4. [Команды обработки текста](#)
 - 12.5. [Команды для работы с файлами и архивами](#)
 - 12.6. [Команды для работы с сетью](#)
 - 12.7. [Команды управления терминалом](#)
 - 12.8. [Команды выполнения математических операций](#)
 - 12.9. [Прочие команды](#)
13. [Команды системного администрирования](#)
14. [Подстановка команд](#)
15. [Арифметические подстановки](#)

16. [Перенаправление ввода/вывода](#)
 - 16.1. [С помощью команды `exec`](#)
 - 16.2. [Перенаправление для блоков кода](#)
 - 16.3. [Область применения](#)
 17. [Встроенные документы](#)
-

Глава 9. К вопросу о переменных

Правильное использование переменных может придать сценариям дополнительную мощь и гибкость, а для этого необходимо изучить все тонкости и нюансы.

9.1. Внутренние переменные

[Встроенные](#) переменные

`$BASH`

путь к исполняемому файлу *Bash*

```
bash$ echo $BASH
/bin/bash
```

`$BASH_VERSINFO[n]`

это [массив](#), состоящий из 6 элементов, и содержащий информацию о версии Bash. Очень похожа на переменную `$BASH_VERSION`, описываемую ниже.

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] = 2           # Major version no.
# BASH_VERSINFO[1] = 05        # Minor version no.
# BASH_VERSINFO[2] = 8         # Patch level.
# BASH_VERSINFO[3] = 1         # Build version.
# BASH_VERSINFO[4] = release   # Release status.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu # Architecture
# (same as $MACHTYPE).
```

`$BASH_VERSION`

версия Bash, установленного в системе

```
bash$ echo $BASH_VERSION
2.04.12(1)-release
```

```
tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```

Проверка переменной `$BASH_VERSION` -- неплохой метод проверки типа командной оболочки, под которой исполняется скрипт. Переменная [\\$SHELL](#) не всегда дает правильный ответ.

\$DIRSTACK

содержимое вершины стека каталогов (который управляется командами [pushd](#) и [popd](#))

Эта переменная соответствует команде [dirs](#), за исключением того, что **dirs** показывает полное содержимое всего стека каталогов.

\$EDITOR

заданный по-умолчанию редактор, вызываемый скриптом, обычно **vi** или **emacs**.

\$EUID

"эффективный" идентификационный номер пользователя (Effective User ID)

Идентификационный номер пользователя, права которого были получены, возможно с помощью команды [su](#).



Значение переменной `$EUID` необязательно должно совпадать с содержимым переменной [\\$UID](#).

\$FUNCNAME

имя текущей функции

```
xyz23 ()
{
  echo "Исполняется функция $FUNCNAME." # Исполняется функция xyz23.
}

xyz23

echo "FUNCNAME = $FUNCNAME"           # FUNCNAME =
                                       # Пустое (Null) значение за пределами
функций.
```

\$GLOBIGNORE

Перечень шаблонных символов, которые будут проигнорированы при выполнении [подстановки имен файлов \(globbing\)](#).

\$GROUPS

группы, к которым принадлежит текущий пользователь

Это список групп (массив) идентификационных номеров групп для текущего пользователя, как это записано в `/etc/passwd`.

```
root# echo $GROUPS
0
```

```
root# echo ${GROUPS[1]}
1
```

```
root# echo ${GROUPS[5]}
6
```

\$HOME

домашний каталог пользователя, как правило это `/home/username` (см. [Пример 9-13](#))

\$HOSTNAME

Сетевое имя хоста устанавливается командой [hostname](#) во время исполнения инициализирующих сценариев на загрузке системы. Внутренняя переменная `$HOSTNAME` Bash получает свое значение посредством вызова функции `gethostname()`. См. так же [Пример 9-13](#).

\$HOSTTYPE

тип машины

Подобно [\\$MACHINE](#), идентифицирует аппаратную архитектуру.

```
bash$ echo $HOSTTYPE
i686
```

\$IFS

разделитель полей во вводимой строке (IFS -- Input Field Separator)

По-умолчанию -- [пробельный символ](#) (пробел, табуляция и перевод строки), но может быть изменен, например, для разбора строк, в которых отдельные поля разделены запятыми. Обратите внимание: при составлении содержимого переменной `$*`, Bash использует первый символ из `$IFS` для разделения аргументов. См. [Пример 5-1](#).

```
bash$ echo $IFS | cat -vte
$
```

```
bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
```


диапазонах символов в квадратных скобках. Например,, **ls [A-M]*** выведет как `File1.txt`, так и `file1.txt`. Возврат к общепринятому стандарту поведения шаблонов в квадратных скобках выполняется установкой переменной `LC_COLLATE` в значение `C` командой `export LC_COLLATE=C` в файле `/etc/profile` и/или `~/ .bashrc`.

`$LC_STYPE`

Эта внутренняя переменная определяет кодировку символов. Используется в операциях [подстановки](#) и поиске по шаблону.

`$LINENO`

Номер строки исполняемого сценария. Эта переменная имеет смысл только внутри исполняемого сценария и чаще всего применяется в отладочных целях.

```
# *** BEGIN DEBUG BLOCK ***
last_cmd_arg=$_ # Запомнить.

echo "Строка $LINENO: переменная \"v1\" = $v1"
echo "Последний аргумент командной строки = $last_cmd_arg"
# *** END DEBUG BLOCK ***
```

`$MACHTYPE`

аппаратная архитектура

Идентификатор аппаратной архитектуры.

```
bash$ echo $MACHTYPE
i686
```

`$OLDPWD`

прежний рабочий каталог ("OLD-Print-Working-Directory")

`$OSTYPE`

тип операционной системы

```
bash$ echo $OSTYPE
linux
```

`$PATH`

путь поиска, как правило включает в себя каталоги `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, и т.д.

Когда командный интерпретатор получает команду, то он автоматически пытается отыскать соответствующий исполняемый файл в указанном списке каталогов (в переменной `$PATH`). Каталоги, в указанном списке, должны отделяться друг от друга двоеточиями. Обычно, переменная `$PATH` инициализируется в `/etc/profile` и/или в `~/ .bashrc` (см. [Глава 26](#)).

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

Инструкция `PATH=${PATH}:/opt/bin` добавляет каталог `/opt/bin` в конец текущего пути поиска. Иногда может оказаться целесообразным, внутри сценария, временно добавить какой-либо каталог к пути поиска. По завершении работы скрипта, эти изменения будут утеряны (вспомните о том, что невозможно изменить переменные окружения вызывающего процесса).



Текущий "рабочий каталог", `.` / `/`, обычно не включается в `$PATH` из соображений безопасности.

`$PIPESTATUS`

Код возврата [канала \(конвейера\)](#). Интересно, что это не то же самое, что [код возврата](#) последней исполненной команды.

```
bash$ echo $PIPESTATUS
0
```

```
bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $PIPESTATUS
141
```

```
bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $?
127
```



Переменная `$PIPESTATUS` может давать неверные значения при вызове из командной строки.

```
tcsh% bash

bash$ who | grep nobody | sort
bash$ echo ${PIPESTATUS[*]}
0
```

Если поместить эти строки в сценарий и исполнить его, то будут выведены верные значения `0 1 0`.

Спасибо Wayne Pollock за замечания и предоставленный пример.

`$PPID`

Переменная `$PPID` хранит PID (идентификатор) родительского процесса. [\[19\]](#)

Сравните с командой [pidof](#).

`$PS1`

`prompt`, приглашение командной строки.

\$PS2

Вторичное приглашение командной строки, выводится тогда, когда от пользователя ожидается дополнительный ввод. Отображается как ">".

\$PS3

Третичное приглашение (prompt), выводится тогда, когда пользователь должен сделать выбор в операторе [select](#) (см. [Пример 10-29](#)).

\$PS4

Приглашение (prompt) четвертого уровня, выводится в начале каждой строки вывода тогда, когда сценарий вызывается с [ключом](#) -x. Отображается как "+".

\$PWD

рабочий (текущий) каталог

Аналог встроенной команды [pwd](#).

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

```
E_WRONG_DIRECTORY=73
```

```
clear # Очистка экрана.
```

```
TargetDirectory=/home/bozo/projects/GreatAmericanNovel
```

```
cd $TargetDirectory
```

```
echo "Удаление файлов в каталоге $TargetDirectory."
```

```
if [ "$PWD" != "$TargetDirectory" ]
```

```
then # Защита от случайного удаления файлов не в том каталоге.
```

```
  echo "Неверный каталог!"
```

```
  echo "Переменная $PWD указывает на другой каталог!"
```

```
  exit $E_WRONG_DIRECTORY
```

```
fi
```

```
rm -rf *
```

```
rm [A-Za-z0-9]* # удалить "скрытые" файлы (начинающиеся с ".")
```

```
# rm -f [^.]* ..?* удалить файлы, чьи имена начинаются с нескольких точек.
```

```
# (short -s dotglob; rm -f *) тоже работает верно.
```

```
# Спасибо S.C. за замечание.
```

```
# Имена файлов могут содержать любые символы из диапазона 0-255, за исключением  
"/".
```

```
# Оставляю вопрос удаления файлов с "необычными" символами для самостоятельного  
изучения.
```

```
# Здесь можно вставить дополнительные действия, по мере необходимости.
```

```
echo
```

```
echo "Конец."
```

```
echo "Файлы, из каталога $TargetDirectory, удалены."
```

```
echo
```

```
exit 0
```

\$REPLY

переменная по-умолчанию, куда записывается ввод пользователя, выполненный с помощью команды [read](#) если явно не задана другая переменная. Так же может использоваться в операторе [select](#), для построения меню выбора.

```
#!/bin/bash

echo
echo -n "Ваше любимое растение? "
read

echo "Ваше любимое растение: $REPLY."
# REPLY хранит последнее значение, прочитанное командой "read" тогда, и только тогда
#+ когда команде "read" не передается имя переменной.

echo
echo -n "Ваш любимый фрукт? "
read fruit
echo "Ваш любимый фрукт $fruit."
echo "но..."
echo "Значение переменной \${REPLY} осталось равным $REPLY."
# Переменная $REPLY не была перезаписана потому, что
# следующей команде "read", в качестве аргумента была передана переменная $fruit

echo

exit 0
```

\$SECONDS

Время паботы сценария в секундах.

```
#!/bin/bash
# Автор: Mendel Cooper
# Дополнен переводчиком.
#

TIME_LIMIT=10
INTERVAL=1

echo
echo "Для прерывания работы сценария, ранее чем через $TIME_LIMIT секунд,
нажмите Control-C."
echo

while [ "$SECONDS" -le "$TIME_LIMIT" ]
do
# Оригинальный вариант сценария содержал следующие строки
# if [ "$SECONDS" -eq 1 ]
# then
#     units=second
# else
#     units=seconds
# fi
#
# Однако, из-за того, что в русском языке для описания множественного числа
# существует большее число вариантов, чем в английском,
# переводчик позволил себе смелость несколько подправить сценарий
# (прошу ногами не бить! ;-))
# === НАЧАЛО БЛОКА ИЗМЕНЕНИЙ, ВНЕСЕННЫХ ПЕРЕВОДЧИКОМ ===

let "last_two_sym = $SECONDS - $SECONDS / 100 * 100" # десятки и единицы
if [ "$last_two_sym" -ge 11 -a "$last_two_sym" -le 19 ]
then
    units="секунд" # для чисел, которые заканчиваются на
```

```

"...надцать"
else
  let "last_sym = $last_two_sym - $last_two_sym / 10 * 10" # единицы
  case "$last_sym" in
    "1" )
      units="секунду"          # для чисел, заканчивающихся на 1
      ;;
    "2" | "3" | "4" )
      units="секунды"         # для чисел, заканчивающихся на 2, 3 и 4
      ;;
    * )
      units="секунд"          # для всех остальных (0, 5, 6, 7, 8, 9)
      ;;
  esac
fi
# === КОНЕЦ БЛОКА ИЗМЕНЕНИЙ, ВНЕСЕННЫХ ПЕРЕВОДЧИКОМ ===

echo "Сценарий отработал $SECONDS $units."
# В случае перегруженности системы, скрипт может перескакивать через
отдельные
#+ значения счетчика
sleep $INTERVAL
done

echo -e "\a" # Сигнал!

exit 0

```

\$SHELLOPTS

список допустимых [опций](#) интерпретатора shell. Переменная доступна только для чтения.

```

bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs

```

\$SHLVL

Уровень вложенности shell. Если в командной строке

```
echo $SHLVL
```

дает 1, то в сценарии значение этой переменной будет больше на 1, т.е. 2.

\$TMOUT

Если переменная окружения *\$TMOUT* содержит ненулевое значение, то интерпретатор будет ожидать ввод не более чем заданное число секунд, что, в первичном приглашении (см. описание PS1 выше), может привести к автоматическому завершению сеанса работы.



К сожалению это возможно только во время ожидания ввода с консоли или в окне терминала. А как было бы здорово, если бы можно было использовать эту внутреннюю переменную, скажем в комбинации с командой [read](#)! Но в данном контексте эта переменная абсолютно не применима и потому фактически бесполезна в сценариях. (Есть сведения

о том, что в *ksh* время ожидания ввода командой **read** можно ограничить.)

Организация ограничения времени ожидания ввода от пользователя в сценариях возможна, но это требует довольно сложных хитростей. Как один из вариантов, можно предложить организовать прерывание цикла ожидания по сигналу. Но это потребует написания функции обработки сигналов командой `trap` (см. [Пример 29-5](#)).

Пример 9-2. Ограничения времени ожидания ввода

```
#!/bin/bash
# timed-input.sh

# TMOUТ=3                бесполезно в сценариях

TIMELIMIT=3 # Три секунды в данном случае, но может быть установлено и другое
значение

PrintAnswer()
{
    if [ "$answer" = TIMEOUT ]
    then
        echo $answer
    else # Чтобы не спутать разные варианты вывода.
        echo "Ваше любимое растение $answer"
        kill $! # "Прибить" ненужную больше функцию TimerOn, запущенную в фоновом
процессе.
        # $! -- PID последнего процесса, запущенного в фоне.
    fi
}

TimerOn()
{
    sleep $TIMELIMIT && kill -s 14 $$ &
    # Ждать 3 секунды, после чего выдать sigalarm сценарию.
}

Int14Vector()
{
    answer="TIMEOUT"
    PrintAnswer
    exit 14
}

trap Int14Vector 14 # переназначить процедуру обработки прерывания от таймера
(14)

echo "Ваше любимое растение? "
TimerOn
read answer
PrintAnswer

# По общему признанию, это не очень хороший способ ограничения времени
ожидания,
#+ однако опция "-t" команды "read" упрощает задачу.
# См. "t-out.sh", ниже.

# Если вам нужно что-то более элегантное...
#+ подумайте о написании программы на С или С++,
#+ с использованием соответствующих библиотечных функций, таких как 'alarm' и
'setitimer'.

exit 0
```

В качестве альтернативы можно использовать [stty](#).

Пример 9-3. Еще один пример ограничения времени ожидания ввода от пользователя

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

# Автор: Stephane Chazelas,
# дополнен автором документа.

INTERVAL=5          # предел времени ожидания

timeout_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. man stty.
}

echo; echo -n "Как Вас зовут? Отвечайте быстрее! "
timeout_read $INTERVAL your_name

# Такой прием может не работать на некоторых типах терминалов.
# Максимальное время ожидания зависит от терминала.
# (чаще всего это 25.5 секунд).

echo

if [ ! -z "$your_name" ] # Если имя было введено...
then
  echo "Вас зовут $your_name."
else
  echo "Вы не успели ответить."
fi

echo

# Алгоритм работы этого сценария отличается от "timed-input.sh".
# Каждое нажатие на клавишу вызывает сброс счетчика в начальное состояние.

exit 0
```

Возможно самый простой способ -- использовать опцию -t команды [read](#).

Пример 9-4. Ограничение времени ожидания команды read

```
#!/bin/bash
# t-out.sh

TIMELIMIT=4        # 4 секунды

read -t $TIMELIMIT variable <&1

echo

if [ -z "$variable" ]
then
  echo "Время ожидания истекло."
else
  echo "variable = $variable"
fi

exit 0
```

\$UID

user id number

UID (идентификатор) текущего пользователя, в соответствии с `/etc/passwd`

Это реальный UID текущего пользователя, даже если он временно приобрел права другого пользователя с помощью [su](#). Переменная \$UID доступна только для чтения.

Пример 9-5. Я -- root?

```
#!/bin/bash
# am-i-root.sh:   Root я, или не root?

ROOT_UID=0   # $UID root-а всегда равен 0.

if [ "$UID" -eq "$ROOT_UID" ] # Настоящий "root"?
then
    echo "- root!"
else
    echo "простой пользователь (но мамочка вас тоже любит)!"
fi

exit 0

# ===== #
# Код, приведенный ниже, никогда не отработает,
#+ поскольку работа сценария уже завершилась выше

# Еще один способ отличить root-а от не root-а:

ROOTUSER_NAME=root

username=`id -nu`           # Или...   username=`whoami`
if [ "$username" = "$ROOTUSER_NAME" ]
then
    echo "Рутти-тутти. - root!"
else
    echo "Вы - лишь обычный юзер."
fi
```

См. также [Пример 2-2](#).



Переменные \$ENV, \$LOGNAME, \$MAIL, \$TERM, \$USER и \$USERNAME, не являются [встроенными](#) переменными Bash. Тем не менее, они часто инициализируются как [переменные окружения](#) в одном из [стартовых файлов](#) Bash. Переменная \$SHELL, командная оболочка пользователя, может задаваться в `/etc/passwd` или в сценарии "init" и она тоже не является встроенной переменной 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 (аргументы)

\$0, \$1, \$2 и т.д.

аргументы передаются... из командной строки в сценарий, функциям или команде [set](#) (см. [Пример 4-5](#) и [Пример 11-13](#))

\$#

количество аргументов командной строки [\[20\]](#), или позиционных параметров (см. [Пример 33-2](#))

\$*

Все аргументы в виде одной строки (слова)

\$@

То же самое, что и \$*, но при этом каждый параметр представлен как отдельная строка (слово), т.е. параметры не подвергаются какой либо интерпретации.

Пример 9-6. arglist: Вывод списка аргументов с помощью переменных \$* и @\$

```
#!/bin/bash
# Вызовите сценарий с несколькими аргументами, например: "один два три".

E_BADARGS=65

if [ ! -n "$1" ]
then
    echo "Порядок использования: `basename $0` argument1 argument2 и т.д."
    exit $E_BADARGS
fi

echo

index=1

echo "Список аргументов в переменной \"\$*\":\"
for arg in "$*" # Работает некорректно, если "$*" не ограничена кавычками.
do
    echo "Аргумент #$index = $arg"
    let "index+=1"
done
# $* воспринимает все аргументы как одну строку.
echo "Полный список аргументов выглядит как одна строка."

echo

index=1

echo "Список аргументов в переменной \"@$\": \"
for arg in "$@"
do
    echo "Аргумент #$index = $arg"
    let "index+=1"
done
# @$ воспринимает аргументы как отдельные строки (слова).
echo "Список аргументов выглядит как набор различных строк (слов).\"

echo
```

```
exit 0
```

После команды **shift** (сдвиг), первый аргумент, в переменной \$@, теряется, а остальные сдвигаются на одну позицию "вниз" (или "влево", если хотите).

```
#!/bin/bash
# Вызовите сценарий в таком виде: ./scriptname 1 2 3 4 5

echo "$@"      # 1 2 3 4 5
shift
echo "$@"      # 2 3 4 5
shift
echo "$@"      # 3 4 5

# Каждая из команд "shift" приводит к потере аргумента $1,
# но остальные аргументы остаются в "$@".
```

Специальная переменная \$@ может быть использована для выбора типа ввода в сценария. Команда **cat "\$@"** позволяет выполнять ввод как со стандартного устройства ввода `stdin`, так и из файла, имя которого передается сценарию из командной строки. См. [Пример 12-17](#) и [Пример 12-18](#).



Переменные \$* и \$@, в отдельных случаях, могут содержать противоречивую информацию! Это зависит от содержимого переменной [\\$IFS](#).

Пример 9-7. Противоречия в переменных \$* и \$@

```
#!/bin/bash

# Демонстрация противоречивости содержимого внутренних переменных "$*" и "$@",
#+ которая проявляется при изменении порядка заключения параметров в кавычки.
# Демонстрация противоречивости, проявляющейся при изменении
#+ содержимого переменной IFS.

set -- "Первый один" "второй" "третий:один" "" "Пятый: :один"
# Установка аргументов $1, $2, и т.д.

echo

echo 'IFS по-умолчанию, переменная "$*'
c=0
for i in "$*"          # в кавычках
do echo "$((c+=1)): [$i]" # Эта строка остается без изменений во всех циклах.
                        # Вывод аргументов.
done
echo ---

echo 'IFS по-умолчанию, переменная $*'
c=0
for i in $*           # без кавычек
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS по-умолчанию, переменная "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---
```

```

echo 'IFS по-умолчанию, переменная $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

IFS=:
echo 'IFS=":", переменная "$*'
c=0
for i in "$*"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", переменная $*'
c=0
for i in $*
do echo "$((c+=1)): [$i]"
done
echo ---

var=$*
echo 'IFS=":", переменная "$var" (var=$*)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", переменная $var (var=$*)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo 'IFS=":", переменная $var (var="$*")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", переменная "$var" (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", переменная "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", переменная $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

var=$@
echo 'IFS=":", переменная $var (var=$@)'
c=0

```



```

for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", переменная "$var" (var=$@)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

var="$@"
echo 'IFS=":", переменная "$var" (var="$@")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", переменная $var (var="$@")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done

echo

# Попробуйте запустить этот сценарий под ksh или zsh -y.

exit 0

# Это сценарий написан Stephane Chazelas,
# Незначительные изменения внесены автором документа.

```



Различия между `$@` и `$*` наблюдаются только тогда, когда они помещаются в двойные кавычки.

Пример 9-8. Содержимое `$*` и `$@`, когда переменная `$IFS` -- пуста

```

#!/bin/bash

# Если переменная $IFS инициализирована "пустым" значением,
# то "$*" и "$@" содержат аргументы не в том виде, в каком ожидается.

mecho ()          # Вывод аргументов.
{
echo "$1,$2,$3";
}

IFS=""           # Инициализация "пустым" значением.
set a b c       # Установка аргументов.

mecho "$*"      # abc,,
mecho $*        # a,b,c

mecho $@        # a,b,c
mecho "$@"     # a,b,c

# Поведение переменных $* и $@, при "пустой" $IFS, зависит
# от версии командной оболочки, Bash или sh.
# Поэтому, было бы неразумным пользоваться этой "фичей" в своих сценариях.

# Спасибо S.C.

exit 0

```

Прочие специальные переменные

\$-

Список флагов, переданных сценарию (командой [set](#)). См. [Пример 11-13](#).



Эта конструкция изначально была введена в *ksh*, откуда перекочевала в *Bash* и, похоже, работает в *Bash* не совсем надежно. Единственное возможное применение -- [проверка - запущен ли сценарий в интерактивном режиме](#).

\$!

PID последнего, запущенного в фоне, процесса

```
LOG=$0.log
```

```
COMMAND1="sleep 100"
```

```
echo "Запись в лог всех PID фоновых процессов, запущенных из сценария: $0" >>  
"$LOG"
```

```
# Таким образом возможен мониторинг и удаление процессов по мере необходимости.  
echo >> "$LOG"
```

```
# Команды записи в лог.
```

```
echo -n "PID of \"${COMMAND1}\": " >> "$LOG"
```

```
${COMMAND1} &
```

```
echo $! >> "$LOG"
```

```
# PID процесса "sleep 100": 1506
```

```
# Спасибо Jacques Lederer за предложенный пример.
```

\$_

Специальная переменная, содержит последний аргумент предыдущей команды.

Пример 9-9. Переменная "подчеркивание"

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

```
echo $_
```

```
# /bin/bash
```

```
# Для запуска сценария был вызван /bin/bash.
```

```
du >/dev/null
```

```
# Подавление вывода.
```

```
echo $_
```

```
# du
```

```
ls -al >/dev/null
```

```
# Подавление вывода.
```

```
echo $_
```

```
# -al (последний аргумент)
```

```
:
```

```
echo $_
```

```
# :
```

\$?

[Код возврата](#) команды, [функции](#) или скрипта (см. [Пример 22-3](#))

\$\$

PID самого процесса-сценария. Переменная \$\$ часто используется при генерации "уникальных" имен для временных файлов (см. [Пример А-14](#), [Пример 29-6](#), [Пример 12-23](#) и [Пример 11-23](#)). Обычно это проще чем вызов [mktemp](#).

9.2. Работа со строками

Bash поддерживает на удивление большое количество операций над строками. К сожалению, этот раздел Bash испытывает недостаток унификации. Одни операции являются подмножеством операций [подстановки параметров](#), а другие -- совпадают с функциональностью команды UNIX -- [expr](#). Это приводит к противоречиям в синтаксисе команд и перекрытию функциональных возможностей, не говоря уже о возникающей путанице.

Длина строки

```
#{#string}
expr length $string
expr "$string" : '.*'
```

```
stringZ=abcABC123ABCabc
echo ${#stringZ}           # 15
echo `expr length $string` # 15
echo `expr "$string" : '.*'` # 15
```

Пример 9-10. Вставка пустых строк между параграфами в текстовом файле

```
#!/bin/bash
# paragraph-space.sh

# Вставка пустых строк между параграфами в текстовом файле.
# Порядок использования: $0 <FILENAME

MINLEN=45      # Возможно потребуется изменить это значение.
# Строки, содержащие количество символов меньше, чем $MINLEN
#+ принимаются за последнюю строку параграфа.

while read line # Построчное чтение файла от начала до конца...
do
  echo "$line"  # Вывод строки.

  len=${#line}
  if [ "$len" -lt "$MINLEN" ]
  then echo     # Добавление пустой строки после последней строки параграфа.
  fi
done

exit 0
```

Длина подстроки в строке (подсчет совпадающих символов ведется с начала строки)

```
expr match "$string" '$substring'
```

где *\$substring* -- [регулярное выражение](#).

```
expr "$string" : '$substring'
```

где *\$substring* -- регулярное выражение.

```
stringZ=abcABC123ABCabc  
#      |-----|
```

```
echo `expr match "$stringZ" 'abc[A-Z]*.2'` # 8  
echo `expr "$stringZ" : 'abc[A-Z]*.2'`     # 8
```

Index

```
expr index $string $substring
```

Номер позиции первого совпадения в *\$string* с первым символом в *\$substring*.

```
stringZ=abcABC123ABCabc  
echo `expr index "$stringZ" C12`           # 6  
# позиция символа C.  
  
echo `expr index "$stringZ" 1c`           # 3  
# символ 'c' (в #3 позиции) совпал раньше, чем '1'.
```

Эта функция довольно близка к функции *strchr()* в языке C.

Извлечение подстроки

```
${string:position}
```

Извлекает подстроку из *\$string*, начиная с позиции *\$position*.

Если строка *\$string* -- "" или "@", то извлекается [позиционный параметр](#) (аргумент), [\[21\]](#) с номером *\$position*.

```
${string:position:length}
```

Извлекает *\$length* символов из *\$string*, начиная с позиции *\$position*.

```
stringZ=abcABC123ABCabc  
#      0123456789.....  
#      Индексация начинается с 0.  
  
echo ${stringZ:0}           # abcABC123ABCabc  
echo ${stringZ:1}           # bcABC123ABCabc  
echo ${stringZ:7}           # 23ABCabc  
  
echo ${stringZ:7:3}         # 23A  
# Извлекает 3 символа.  
  
# Возможна ли индексация с "правой" стороны строки?  
  
echo ${stringZ:-4}          # abcABC123ABCabc  
# По-умолчанию выводится полная строка.  
# Однако . . .  
  
echo ${stringZ:(-4)}        # Cabc
```

```

echo ${stringZ: -4}                # Cabc
# Теперь выводится правильно.
# Круглые скобки или дополнительный пробел "экранируют" параметр позиции.

# Спасибо Dan Jacobson, за разъяснения.

```

Если `$string -- "*" или "@"`, то извлекается до `$length` позиционных параметров (аргументов), начиная с `$position`.

```

echo ${*:2}                # Вывод 2-го и последующих аргументов.
echo {@:2}                 # То же самое.

echo ${*:2:3}              # Вывод 3-х аргументов, начиная со 2-го.

```

`expr substr $string $position $length`

Извлекает `$length` символов из `$string`, начиная с позиции `$position`.

```

stringZ=abcABC123ABCabc
#      123456789.....
#      Индексация начинается с 1.

echo `expr substr $stringZ 1 2`      # ab
echo `expr substr $stringZ 4 3`      # ABC

```

`expr match "$string" '\($substring\)`

Находит и извлекает первое совпадение `$substring` в `$string`, где `$substring` -- это [регулярное выражение](#).

`expr "$string" : '\($substring\)`

Находит и извлекает первое совпадение `$substring` в `$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 вышеприведенные операции дают один и тот же результат.

```

`expr match "$string" '.*\($substring\)`

Находит и извлекает первое совпадение `$substring` в `$string`, где `$substring` -- это регулярное выражение. Поиск начинается с конца `$string`.

`expr "$string" : '.*\($substring\)`

Находит и извлекает первое совпадение `$substring` в `$string`, где `$substring` -- это регулярное выражение. Поиск начинается с конца `$string`.

```

stringZ=abcABC123ABCabc
#          =====

echo `expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\) '`      # ABCabc
echo `expr "$stringZ" : '.*\([.....]\) '`                          # ABCabc

```

Удаление части строки

`${string#substring}`

Удаление самой короткой, из найденных, подстроки *\$substring* в строке *\$string*. Поиск ведется с начала строки

`${string##substring}`

Удаление самой длинной, из найденных, подстроки *\$substring* в строке *\$string*. Поиск ведется с начала строки

```

stringZ=abcABC123ABCabc
#          |----|
#          |-----|

echo ${stringZ#a*C}      # 123ABCabc
# Удаление самой короткой подстроки.

echo ${stringZ##a*C}    # abc
# Удаление самой длинной подстроки.

```

`${string%substring}`

Удаление самой короткой, из найденных, подстроки *\$substring* в строке *\$string*. Поиск ведется с конца строки

`${string%%substring}`

Удаление самой длинной, из найденных, подстроки *\$substring* в строке *\$string*. Поиск ведется с конца строки

```

stringZ=abcABC123ABCabc
#          ||
#          |-----|

echo ${stringZ%b*c}      # abcABC123ABCa
# Удаляется самое короткое совпадение. Поиск ведется с конца $stringZ.

echo ${stringZ%%b*c}    # a
# Удаляется самое длинное совпадение. Поиск ведется с конца $stringZ.

```

Пример 9-11. Преобразование графических файлов из одного формата в другой, с изменением имени файла

```

#!/bin/bash
# cvt.sh:
# Преобразование всех файлов в заданном каталоге,
#+ из графического формата MacPaint, в формат "pbm".

```

```

# Используется утилита "macptorbm", входящая в состав пакета "netpbm",
#+ который сопровождается Brian Henderson (bryanh@giraffe-data.com).
# Netpbm -- стандартный пакет для большинства дистрибутивов Linux.

OPERATION=macptorbm
SUFFIX=pbm      # Новое расширение файла.

if [ -n "$1" ]
then
  directory=$1      # Если каталог задан в командной строке при вызове сценария
else
  directory=$PWD    # Иначе просматривается текущий каталог.
fi

# Все файлы в каталоге, имеющие расширение ".mac", считаются файлами
#+ формата MacPaint.

for file in $directory/* # Подстановка имен файлов.
do
  filename=${file%.*c}  # Удалить расширение ".mac" из имени файла
                        #+ ( с шаблоном '*.c' совпадают все подстроки
                        #+ начинающиеся с '.' и заканчивающиеся 'c',
  $OPERATION $file > "$filename.$SUFFIX"
                        # Преобразование с перенаправлением в файл с новым
именем
  rm -f $file          # Удаление оригинального файла после преобразования.
  echo "$filename.$SUFFIX" # Вывод на stdout.
done

exit 0

# Упражнение:
# -----
# Сейчас этот сценарий конвертирует *все* файлы в каталоге
# Измените его так, чтобы он конвертировал *только* те файлы,
#+ которые имеют расширение ".mac".

```

Замена подстроки

`${string/substring/replacement}`

Замещает первое вхождение *\$substring* строкой *\$replacement*.

`${string//substring/replacement}`

Замещает все вхождения *\$substring* строкой *\$replacement*.

```
stringZ=abcABC123ABCabc
```

```
echo ${stringZ/abc/xyz}
```

```
# xyzABC123ABCabc
```

```
# Замена первой подстроки 'abc' строкой 'xyz'.
```

```
echo ${stringZ//abc/xyz}
```

```
# xyzABC123ABCxyz
```

```
# Замена всех подстрок 'abc' строкой 'xyz'.
```

`${string/#substring/replacement}`

Подстановка строки *\$replacement* вместо *\$substring*. Поиск ведется с начала строки *\$string*.

`${string/%substring/replacement}`

Подстановка строки *\$replacement* вместо *\$substring*. Поиск ведется с конца строки *\$string*.

```
stringZ=abcABC123ABCabc
```

```
echo ${stringZ/#abc/XYZ}      # XYZABC123ABCabc  
                              # Поиск ведется с начала строки
```

```
echo ${stringZ/%abc/XYZ}     # abcABC123ABCXYZ  
                              # Поиск ведется с конца строки
```

9.2.1. Использование awk при работе со строками

В качестве альтернативы, Bash-скрипты могут использовать средства [awk](#) при работе со строками.

Пример 9-12. Альтернативный способ извлечения подстрок

```
#!/bin/bash  
# substring-extraction.sh  
  
String=23skidoo1  
#      012345678   Bash  
#      123456789   awk  
# Обратите внимание на различия в индексации:  
# Bash начинает индексацию с '0'.  
# Awk  начинает индексацию с '1'.  
  
echo ${String:2:4} # с 3 позиции (0-1-2), 4 символа  
                  # skid  
  
# В эквивалент в awk: substr(string,pos,length).  
echo | awk '  
{ print substr("'"${String}"'",3,4)      # skid  
}'  
,  
# Передача пустого "echo" по каналу в awk, означает фиктивный ввод,  
#+ делая, тем самым, ненужным предоставление имени файла.  
  
exit 0
```

9.2.2. Дальнейшее обсуждение

Дополнительную информацию, по работе со строками, вы найдете в разделе [Section 9.3](#) и в [секции](#), посвященной команде [expr](#). Примеры сценариев:

1. [Пример 12-6](#)
2. [Пример 9-15](#)
3. [Пример 9-16](#)
4. [Пример 9-17](#)

9.3. Подстановка параметров

Работа с переменными и/или подстановка их значений

`${parameter}`

То же самое, что и `$parameter`, т.е. значение переменной `parameter`. В отдельных случаях, при возникновении неоднозначности интерпретации, корректно будет работать только такая форма записи: `${parameter}`.

Может использоваться для конкатенации (слияния) строковых переменных.

```
your_id=${USER}-on-${HOSTNAME}
echo "$your_id"
#
echo "Старый \${PATH} = $PATH"
PATH=${PATH}:/opt/bin #Добавление /opt/bin в $PATH.
echo "Новый \${PATH} = $PATH"
```

`${parameter-default}`, `${parameter:-default}`

Если параметр отсутствует, то используется значение по-умолчанию.

```
echo ${username-`whoami`}
# Вывод результата работы команды `whoami`, если переменная $username не
установлена.
```



Формы записи `${parameter-default}` и `${parameter:-default}` в большинстве случаев можно считать эквивалентными. Дополнительный символ `:` имеет значение только тогда, когда `parameter` определен, но имеет "пустое" (null) значение.

```
#!/bin/bash

username0=
# переменная username0 объявлена, но инициализирована "пустым" значением.
echo "username0 = ${username0-`whoami`}"
# Вывод после символа "=" отсутствует.

echo "username1 = ${username1-`whoami`}"
# Переменная username1 не была объявлена.
# Выводится имя пользователя, выданное командой `whoami`.

username2=
# переменная username2 объявлена, но инициализирована "пустым" значением.
echo "username2 = ${username2:-`whoami`}"
# Выводится имя пользователя, выданное командой `whoami`, поскольку
#+здесь употребляется конструкция ":-", а не "-".

exit 0
```

Параметры по-умолчанию очень часто находят применение в случаях, когда сценарию необходимы какие либо входные аргументы, передаваемые из командной строки, но такие аргументы не были переданы.

```
DEFAULT_FILENAME=generic.data
filename=${1:-$DEFAULT_FILENAME}
# Если имя файла не задано явно, то последующие операторы будут работать
#+ с файлом "generic.data".
#
```

см. так же [Пример 3-4](#), [Пример 28-2](#) и [Пример A-7](#).

Сравните этот подход с [методом списков *and list*](#), для задания параметров командной строки по-умолчанию .

`${parameter=default}`, `${parameter:=default}`

Если значения параметров не заданы явно, то они принимают значения по-умолчанию.

Оба метода задания значений по-умолчанию до определенной степени идентичны. Символ `:` имеет значение только когда *\$parameter* был инициализирован "пустым" (null) значением, [\[22\]](#) как показано выше.

```
echo ${username=`whoami`}
# Переменная "username" принимает значение, возвращаемое командой `whoami`.
```

`${parameter+alt_value}`, `${parameter:+alt_value}`

Если параметр имеет какое либо значение, то используется *alt_value*, иначе -- null ("пустая" строка).

Оба варианта до определенной степени идентичны. Символ `:` имеет значение только если *parameter* объявлен и "пустой", см. ниже.

```
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 = $a"      # a =
# Вывод отличается от a=${param5+xyz}

param6=123
a=${param6+xyz}
echo "a = $a"      # a = xyz

```

`${parameter?err_msg}, ${parameter:?err_msg}`

Если `parameter` инициализирован, то используется его значение, в противном случае -- выводится `err_msg`.

Обе формы записи можно, до определенной степени, считать идентичными. Символ `:` имеет значение только когда `parameter` инициализирован "пустым" значением, см. ниже.

Пример 9-13. Подстановка параметров и сообщения об ошибках

```

#!/bin/bash

# Проверка отдельных переменных окружения.
# Если переменная, к примеру $USER, не установлена,
#+ то выводится сообщение об ошибке.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
echo
echo "Имя машины: $HOSTNAME."
echo "Ваше имя: $USER."
echo "Ваш домашний каталог: $HOME."
echo "Ваш почтовый ящик: $MAIL."
echo
echo "Если перед Вами появилось это сообщение,"
echo "то это значит, что все критические переменные окружения установлены."
echo
echo

# -----

# Конструкция ${variablename?} так же выполняет проверку
#+ наличия переменной в сценарии.

ThisVariable=Value-of-ThisVariable
# Обратите внимание, в строковые переменные могут быть записаны
#+ символы, которые запрещено использовать в именах переменных.
: ${ThisVariable?}
echo "Value of ThisVariable is $ThisVariable".
echo
echo

: ${ZZXy23AB?"Переменная ZXy23AB не инициализирована."}
# Если ZXy23AB не инициализирована,
#+ то сценарий завершается с сообщением об ошибке.

# Текст сообщения об ошибке можно задать свой.
# : ${ZXy23AB?"Переменная ZXy23AB не инициализирована."}

# То же самое: dummy_variable=${ZXy23AB?}
# dummy_variable=${ZXy23AB?"Переменная ZXy23AB не инициализирована."}
#

```

```
# echo ${ZZXy23AB?} >/dev/null
```

echo "Это сообщение не будет напечатано, поскольку сценарий завершится раньше."

```
HERE=0
```

```
exit $HERE # Сценарий завершит работу не здесь.
```

Пример 9-14. Подстановка параметров и сообщение о "порядке использования"

```
#!/bin/bash
# usage-message.sh
```

```
: ${1?"Порядок использования: $0 ARGUMENT"}
# Сценарий завершит свою работу здесь, если входные аргументы отсутствуют,
#+ со следующим сообщением.
# usage-message.sh: 1: Порядок использования: usage-message.sh ARGUMENT
```

```
echo "Эти две строки появятся, только когда задан аргумент в командной строке."
echo "Входной аргумент командной строки = \"$1\""
```

```
exit 0 # Точка выхода находится здесь, только когда задан аргумент командной строки.
```

```
# Проверьте код возврата в обоих случаях, с и без аргумента командной строки.
# Если аргумент задан, то код возврата будет равен 0.
# Иначе -- 1.
```

Подстановка параметров и/или экспансия. Следующие выражения могут служить дополнениями оператора **match** команды **expr**, применяемой к строкам (см. [Пример 12-6](#)). Как правило, они используются при разборе имен файлов и каталогов.

Длина переменной / Удаление подстроки

`${#var}`

String length (число символов в переменной `var`). В случае [массивов](#), команда **`${#array}`** возвращает длину первого элемента массива.



Исключения:

- **`${#*}`** и **`${#@}`** возвращает *количество аргументов (позиционных параметров)*.
- Для массивов, **`${#array[*]}`** и **`${#array[@]}`** возвращает количество элементов в массиве.

Пример 9-15. Длина переменной

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

```
E_NO_ARGS=65
```

```
if [ $# -eq 0 ] # Для работы скрипта необходим хотя бы один входной параметр.
then
    echo "Вызовите сценарий с одним или более параметром командной строки."
    exit $E_NO_ARGS
fi
```

```
var01=abcdEFGH28ij
```

```

echo "var01 = ${var01}"
echo "Length of var01 = ${#var01}"

echo "Количество входных параметров = ${#@}"
echo "Количество входных параметров = ${#*}"

exit 0

```

`${var#Pattern}, ${var##Pattern}`

Удаляет из переменной `$var` наименьшую/наибольшую подстроку, совпадающую с шаблоном `$Pattern`. Поиск ведется с начала строки `$var`.

Пример использования из [Пример А-8](#):

```

# Функция из сценария "days-between.sh".
# Удаляет нули, стоящие в начале аргумента-строки.

strip_leading_zero () # Ведущие нули, которые согут находиться в номере
дня/месяца,
    # лучше удалить
    val=${1#0}        # В противном случае Bash будет интерпретировать числа
    return $val      # как восьмеричные (POSIX.2, sect 2.9.2.1).
}

```

Другой пример:

```

echo `basename $PWD`      # Имя текущего рабочего каталога.
echo "${PWD##*/}"        # Имя текущего рабочего каталога.
echo
echo `basename $0`       # Имя файла-сценария.
echo $0                  # Имя файла-сценария.
echo "${0##*/}"         # Имя файла-сценария.
echo
filename=test.data
echo "${filename##*}"    # data
                        # Расширение файла.

```

`${var%Pattern}, ${var%%Pattern}`

Удаляет из переменной `$var` наименьшую/наибольшую подстроку, совпадающую с шаблоном `$Pattern`. Поиск ведется с конца строки `$var`.

Bash [версии 2](#) имеет ряд дополнительных возможностей.

Пример 9-16. Поиск по шаблону в подстановке параметров

```

#!/bin/bash
# Поиск по шаблону в операциях подстановки параметров # ## % %%.

var1=abcd12345abc6789
pattern1=a*c # * (символ шаблона), означает любые символы между a и c.

echo
echo "var1 = $var1"          # abcd12345abc6789
echo "var1 = ${var1}"        # abcd12345abc6789 (альтернативный вариант)
echo "Число символов в ${var1} = ${#var1}"
echo "pattern1 = $pattern1"  # a*c (между 'a' и 'c' могут быть любые символы)
echo

```

```

echo '${var1#$pattern1} =' "${var1#$pattern1}" # d12345abc6789
# Наименьшая подстрока, удаляются первые 3 символа abcd12345abc6789
# ^^^^^^ | - |
echo '${var1##$pattern1} =' "${var1##$pattern1}" # 6789
# Наибольшая подстрока, удаляются первые 12 символов abcd12345abc6789
# ^^^^^^^ |-----|

echo; echo

pattern2=b*9 # все, что между 'b' и '9'
echo "var1 = $var1" # abcd12345abc6789
echo "pattern2 = $pattern2"
echo

echo '${var1%pattern2} =' "${var1%pattern2}" # abcd12345a
# Наименьшая подстрока, удаляются последние 6 символов abcd12345abc6789
# ^^^^^^^^^^ |----|
echo '${var1%%pattern2} =' "${var1%%pattern2}" # a
# Наибольшая подстрока, удаляются последние 12 символов abcd12345abc6789
# ^^^^^^^^^^^ |-----|

# Запомните, # и ## используются для поиска с начала строки,
# % и %% используются для поиска с конца строки.

echo

exit 0

```

Пример 9-17. Изменение расширений в именах файлов:

```

#!/bin/bash

# rfe
# ---

# Изменение расширений в именах файлов.
#
# rfe old_extension new_extension
#
# Пример:
# Изменить все расширения *.gif в именах файлов на *.jpg, в текущем каталоге
# rfe gif jpg

ARGS=2
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
    echo "Порядок использования: `basename $0` old_file_suffix new_file_suffix"
    exit $E_BADARGS
fi

for filename in *.$1
# Цикл прохода по списку имен файлов, имеющих расширение равное первому аргументу.
do
    mv $filename ${filename%$1}$2
    # Удалить первое расширение и добавить второе,
done

exit 0

```

Подстановка значений переменных / Замена подстроки

Эти конструкции перекочевали в Bash из *ksh*.

`${var:pos}`

Подставляется значение переменной *var*, начиная с позиции *pos*.

`${var:pos:len}`

Подставляется значение переменной *var*, начиная с позиции *pos*, не более *len* символов. См. [Пример A-16](#).

`${var/Pattern/Replacement}`

Первое совпадение с шаблоном *Pattern*, в переменной *var* замещается подстрокой *Replacement*.

Если подстрока *Replacement* отсутствует, то найденное совпадение будет удалено.

`${var//Pattern/Replacement}`

Глобальная замена. Все найденные совпадения с шаблоном *Pattern*, в переменной *var*, будут замещены подстрокой *Replacement*.

Как и в первом случае, если подстрока *Replacement* отсутствует, то все найденные совпадения будут удалены.

Пример 9-18. Поиск по шаблону при анализе произвольных строк

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-}
echo "var1 (все, от начала строки по первый символ \"-\", включительно,
удаляется) = $t"
# t=${var1#*-} то же самое,
#+ поскольку оператор # ищет кратчайшее совпадение,
#+ а * соответствует любым предшествующим символам, включая пустую строку.
# (Спасибо S. C. за разъяснения.)

t=${var1##*-}
echo "Если var1 содержит \"-\", то возвращается пустая строка... var1 = $t"

t=${var1%*-}
echo "var1 (все, начиная с последнего \"-\" удаляется) = $t"

echo

# -----
path_name=/home/bozo/ideas/thoughts.for.today
# -----
echo "path_name = $path_name"
t=${path_name##*/}
echo "Из path_name удален путь к файлу = $t"
# В данном случае, тот же эффект можно получить так: t=`basename $path_name`
# t=${path_name%/*}; t=${t##*/} более общее решение,
#+ но имеет некоторые ограничения.
# Если $path_name заканчивается символом перевода строки, то `basename
$path_name` не будет работать,
#+ но для данного случая вполне применимо.
# (Спасибо S.C.)

t=${path_name%/*.*}
```

```

# Тот же эффект дает      t=`dirname $path_name`
echo "Из $path_name удалено имя файла = $t"
# Этот вариант будет терпеть неудачу в случаях: "../", "/foo////", # "foo/",
"/".
# Удаление имени файла, особенно когда его нет,
#+ использование dirname имеет свои особенности.
# (Спасибо S.C.)

echo

t=${path_name:11}
echo "Из $path_name удалены первые 11 символов = $t"
t=${path_name:11:5}
echo "Из $path_name удалены первые 11 символов, выводится 5 символов = $t"

echo

t=${path_name/bozo/clown}
echo "В $path_name подстрока \"bozo\" заменена на \"clown\" = $t"
t=${path_name/today/}
echo "В $path_name подстрока \"today\" удалена = $t"
t=${path_name//o/O}
echo "В $path_name все символы \"o\" переведены в верхний регистр, = $t"
t=${path_name//o/}
echo "Из $path_name удалены все символы \"o\" = $t"

exit 0

```

`${var/#Pattern/Replacement}`

Если в переменной *var* найдено совпадение с *Pattern*, причем совпадающая подстрока расположена в начале строки (префикс), то оно заменяется на *Replacement*. Поиск ведется с начала строки

`${var/%Pattern/Replacement}`

Если в переменной *var* найдено совпадение с *Pattern*, причем совпадающая подстрока расположена в конце строки (суффикс), то оно заменяется на *Replacement*. Поиск ведется с конца строки

Пример 9-19. Поиск префиксов и суффиксов с заменой по шаблону

```

#!/bin/bash
# Поиск с заменой по шаблону.

v0=abc1234zip1234abc      # Начальное значение переменной.
echo "v0 = $v0"          # abc1234zip1234abc
echo

# Поиск совпадения с начала строки.
v1=${v0/#abc/ABCDEF}     # abc1234zip1234abc
# |-|
echo "v1 = $v1"          # ABCDE1234zip1234abc
# |---|

# Поиск совпадения с конца строки.
v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
#                               |-|
echo "v2 = $v2"          # abc1234zip1234ABCDEF
#                               |----|

echo

# -----
# Если совпадение находится не с начала/конца строки,
#+ то замена не производится.

```



```

# -----
v3=${v0/#123/000}      # Совпадение есть, но не в начале строки.
echo "v3 = $v3"        # abc1234zip1234abc
                        # ЗАМЕНА НЕ ПРОИЗВОДИТСЯ!
v4=${v0/%123/000}      # Совпадение есть, но не в конце строки.
echo "v4 = $v4"        # abc1234zip1234abc
                        # ЗАМЕНА НЕ ПРОИЗВОДИТСЯ!

exit 0

```

`${!varprefix*}`, `${!varprefix@}`

Поиск по шаблону всех, ранее объявленных переменных, имена которых начинаются с *varprefix*.

```

xyz23=whatever
xyz24=

a=${!xyz*}             # Подстановка имен объявленных переменных, которые начинаются с
"xyz".
echo "a = $a"          # a = xyz23 xyz24
a=${!xyz@}             # То же самое.
echo "a = $a"          # a = xyz23 xyz24

# Эта возможность была добавлена в Bash, в версии 2.04.

```

9.4. Объявление переменных: `declare` и `typeset`

Инструкции **`declare`** и **`typeset`** являются [встроенными](#) инструкциями (они абсолютно идентичны друг другу и являются синонимами) и предназначена для наложения ограничений на переменные. Это очень слабая попытка контроля над типами, которая имеется во многих языках программирования. Инструкция **`declare`** появилась в Bash, начиная с версии 2. Кроме того, инструкция **`typeset`** может использоваться и в ksh-сценариях.

ключи инструкций `declare/typeset`

`-r readonly` (только для чтения)

```
declare -r var1
```

(`declare -r var1` аналогично объявлению `readonly var1`)

Это грубый эквивалент констант (`const`) в языке C. Попытка изменения таких переменных завершается сообщением об ошибке.

`-i integer`

```

declare -i number
# Сценарий интерпретирует переменную "number" как целое число.

number=3
echo "number = $number"      # number = 3

```

```
number=three
echo "number = $number"      # number = 0
# Строка "three" интерпретируется как целое число.
```

Примечательно, что допускается выполнение некоторых арифметических операций над переменными, объявленными как `integer`, не прибегая к инструкциям [expr](#) или [let](#).

-a *array*

```
declare -a indices
```

Переменная `indices` объявляется массивом.

-f *functions*

```
declare -f
```

Инструкция `declare -f`, без аргументов, приводит к выводу списка ранее объявленных функций в сценарии.

```
declare -f function_name
```

Инструкция `declare -f function_name` выводит имя функции `function_name`, если она была объявлена ранее.

-x [export](#)

```
declare -x var3
```

Эта инструкция объявляет переменную, как доступную для экспорта.

`var=$value`

```
declare -x var3=373
```

Инструкция **declare** допускает совмещение объявления и присваивания значения переменной одновременно.

Пример 9-20. Объявление переменных с помощью инструкции `declare`

```
#!/bin/bash

func1 ()
{
  echo Это функция.
}

declare -f      # Список функций, объявленных выше.
```

```

echo

declare -i var1    # var1 -- целочисленная переменная.
var1=2367
echo "переменная var1 объявлена как $var1"
var1=var1+1       # Допустимая арифметическая операция над целочисленными
переменными.
echo "переменная var1 увеличена на 1 = $var1."
# Допустимая операция для целочисленных переменных
echo "Возможно ли записать дробное число 2367.1 в var1?"
var1=2367.1       # Сообщение об ошибке, переменная не изменяется.
echo "значение переменной var1 осталось прежним = $var1"

echo

declare -r var2=13.36      # инструкция 'declare' допускает установку свойств
переменной                #+ и одновременно присваивать значение.
echo "var2 declared as $var2" # Допускается ли изменять значение readonly переменных?
var2=13.37                # Сообщение об ошибке и завершение работы сценария.

echo "значение переменной var2 осталось прежним $var2" # Эта строка никогда не будет
выполнена.

exit 0                    # Сценарий завершит работу выше.

```

9.5. Косвенные ссылки на переменные

Предположим, что значение одной переменной -- есть имя второй переменной. Возможно ли получить значение второй переменной через обращение к первой? Например, Пусть `a=letter_of_alphabet` и `letter_of_alphabet=z`, тогда вопрос будет звучать так: "Возможно ли получить значение `z`, обратившись к переменной `a`". В действительности это возможно и это называется *косвенной ссылкой*. Для этого необходимо прибегнуть к несколько необычной нотации `eval var1=\$$var2`.

Пример 9-21. Косвенные ссылки

```

#!/bin/bash
# Косвенные ссылки на переменные.

a=letter_of_alphabet
letter_of_alphabet=z

echo

# Прямое обращение к переменной.
echo "a = $a"

# Косвенное обращение к переменной.
eval a=\$$a
echo "А теперь a = $a"

echo

# Теперь попробуем изменить переменную, на которую делается ссылка.

t=table_cell_3
table_cell_3=24
echo "\"table_cell_3\" = $table_cell_3"
echo -n "разыменованное (получение ссылки) \"t\" = "; eval echo \$$t
# В данном, простом, случае,
# eval t=\$$t; echo "\"t\" = $t"

```

```

# дает тот же результат (почему?).

echo

t=table_cell_3
NEW_VAL=387
table_cell_3=$NEW_VAL
echo "Значение переменной \"table_cell_3\" изменено на $NEW_VAL."
echo "Теперь \"table_cell_3\" = $table_cell_3"
echo -n "разыменование (получение ссылки) \"t\" = "; eval echo \$$t
# инструкция "eval" принимает два аргумента "echo" и "\$$t" (назначает равным
$table_cell_3)
echo

# (Спасибо S.C. за разъяснения.)

# Еще один способ -- нотация ${!t}, будет обсуждаться в разделе "Bash, версия 2".
# Так же, см. пример "ex78.sh".

exit 0

```

Пример 9-22. Передача косвенных ссылок в *awk*

```

#!/bin/bash

# Другая версия сценария "column totaler"
# который суммирует заданную колонку (чисел) в заданном файле.
# Здесь используются косвенные ссылки.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # Проверка количества входных аргументов.
then
    echo "Порядок использования: `basename $0` filename column-number"
    exit $E_WRONGARGS
fi

filename=$1
column_number=$2

##### До этой строки идентично первоначальному варианту сценария #####

# Многострочные скрипты awk вызываются конструкцией   awk ' ..... '

# Начало awk-сценария.
# -----
awk "

{ total += \${column_number} # косвенная ссылка
}
END {
    print total
}

" "$filename"
# -----
# Конец awk-сценария.

# Косвенные ссылки делают возможным бесконфликтное
# обращение к переменным shell внутри вложенных сценариев awk.
# Спасибо Stephane Chazelas.

exit 0

```



Такой метод обращения к переменным имеет свои особенности. Если переменная, на которую делается ссылка, меняет свое значение, то переменная которая ссылается, должна быть должным образом разыменована, т.е. олжна быть выполнена операция получения ссылки, как это делается в примере выше. К счастью, нотация `${!variable}`, введенная в Bash, начиная с [версии 2](#) (см. [Пример 34-2](#)) позволяет выполнять косвенные ссылки более интуитивно понятным образом.

9.6. \$RANDOM: генерация псевдослучайных целых чисел

`$RANDOM` -- внутренняя функция Bash (не константа), которая возвращает *псевдослучайные* целые числа в диапазоне 0 - 32767. Функция `$RANDOM` *не* должна использоваться для генерации ключей шифрования.

Пример 9-23. Генерация случайных чисел

```
#!/bin/bash

# $RANDOM возвращает различные случайные числа при каждом обращении к ней.
# Диапазон изменения: 0 - 32767 (16-битовое целое со знаком).

MAXCOUNT=10
count=1

echo
echo "$MAXCOUNT случайных чисел:"
echo "-----"
while [ "$count" -le $MAXCOUNT ]      # Генерация 10 ($MAXCOUNT) случайных чисел.
do
    number=$RANDOM
    echo $number
    let "count += 1" # Нарастить счетчик.
done
echo "-----"

# Если вам нужны случайные числа не превышающие определенного числа,
# воспользуйтесь оператором деления по модулю (остаток от деления).

RANGE=500

echo

number=$RANDOM
let "number %= $RANGE"
echo "Случайное число меньше $RANGE --- $number"

echo

# Если вы желаете ограничить диапазон "снизу",
# то просто производите генерацию псевдослучайных чисел в цикле до тех пор,
# пока не получите число большее нижней границы.

FLOOR=200

number=0 # инициализация
while [ "$number" -le $FLOOR ]
do
    number=$RANDOM
done
echo "Случайное число, большее $FLOOR --- $number"
echo
```

```

# Эти два способа могут быть скомбинированы.
number=0 #initialize
while [ "$number" -le $FLOOR ]
do
    number=$RANDOM
    let "number %= $RANGE" # Ограничение "сверху" числом $RANGE.
done
echo "Случайное число в диапазоне от $FLOOR до $RANGE --- $number"
echo

# Генерация случайных "true" и "false" значений.
BINARY=2
number=$RANDOM
T=1

let "number %= $BINARY"
# let "number >>= 14"      дает более равномерное распределение
# (сдвиг вправо смещает старший бит на нулевую позицию, остальные биты обнуляются).
if [ "$number" -eq $T ]
then
    echo "TRUE"
else
    echo "FALSE"
fi

echo

# Можно имитировать бросание 2-х игровых кубиков.
SPOTS=7 # остаток от деления на 7 дает диапазон 0 - 6.
ZERO=0
die1=0
die2=0

# Кубики "выбрасываются" отдельно.

while [ "$die1" -eq $ZERO ] # Пока на "кубике" ноль.
do
    let "die1 = $RANDOM % $SPOTS" # Имитировать бросок первого кубика.
done

while [ "$die2" -eq $ZERO ]
do
    let "die2 = $RANDOM % $SPOTS" # Имитировать бросок второго кубика.
done

let "throw = $die1 + $die2"
echo "Результат броска кубиков = $throw"
echo

exit 0

```

Пример 9-24. Выбор случайной карты из колоды

```

#!/bin/bash
# pick-card.sh

# Пример выбора случайного элемента массива.

# Выбор случайной карты из колоды.

Suites="Треф
Бубей
Червей
Пик"

```

```

Denominations="2
3
4
5
6
7
8
9
10
Валет
Дама
Король
Туз"

suite=($Suites)          # Инициализация массивов.
denomination=($Denominations)

num_suites=${#suite[*]}  # Количество элементов массивов.
num_denominations=${#denomination[*]}

echo -n "${denomination[$((RANDOM%num_denominations))]} "
echo ${suite[$((RANDOM%num_suites))]}

# $bozo sh pick-cards.sh
# Валет Треф

# Спасибо "jipe," за пояснения по работе с $RANDOM.
exit 0

```



Jipe подсказал еще один способ генерации случайных чисел из заданного диапазона.

```

# Генерация случайных чисел в диапазоне 6 - 30.
rnumber=$((RANDOM%25+6))

# Генерируется случайное число из диапазона 6 - 30,
#+ но при этом число должно делиться на 3 без остатка.
rnumber=$(( (RANDOM%30/3+1)*3))

# Упражнение: Попробуйте разобраться с выражением самостоятельно.

```

Насколько случайны числа, возвращаемые функцией \$RANDOM? Лучший способ оценить "случайность" генерируемых чисел -- это написать сценарий, который будет имитировать бросание игрального кубика достаточно большое число раз, а затем выведет количество выпадений каждой из граней...

Пример 9-25. Имитация бросания кубика с помощью RANDOM

```

#!/bin/bash
# Случайные ли числа возвращает RANDOM?

RANDOM=$$          # Инициализация генератора случайных чисел числом PID процесса-
сценария.

PIPS=6            # Кубик имеет 6 граней.
MAXTHROWS=600    # Можете увеличить, если не знаете куда девать свое время.
throw=0          # Счетчик бросков.

zeroes=0         # Обнулить счетчики выпадения отдельных граней.
ones=0           # т.к. неинициализированные переменные - "пустые", и не равны нулю!.
twos=0
threes=0

```

```

fours=0
fives=0
sixes=0

print_result ()
{
echo
echo "единиц   =   $ones"
echo "двоек   =   $twos"
echo "троек   =   $threes"
echo "четверок =   $fours"
echo "пятерок =   $fives"
echo "шестерок =   $sixes"
echo
}

update_count()
{
case "$1" in
0) let "ones += 1";; # 0 соответствует грани "1".
1) let "twos += 1";; # 1 соответствует грани "2", и так далее
2) let "threes += 1";;
3) let "fours += 1";;
4) let "fives += 1";;
5) let "sixes += 1";;
esac
}

echo

while [ "$throw" -lt "$MAXTHROWS" ]
do
let "die1 = RANDOM % $PIPS"
update_count $die1
let "throw += 1"
done

print_result

# Количество выпадений каждой из граней должно быть примерно одинаковым, если считать
RANDOM достаточно случайным.
# Для $MAXTHROWS = 600, каждая грань должна выпасть примерно 100 раз (плюс-минус 20).
#
# Имейте ввиду, что RANDOM - это генератор ПСЕВДОСЛУЧАЙНЫХ чисел,

# Упражнение:
# -----
# Перепишите этот сценарий так, чтобы он имитировал 1000 бросков монеты.
# На каждом броске возможен один из двух вариантов выпадения - "ОРЕЛ" или "РЕШКА".

exit 0

```

Как видно из последнего примера, неплохо было бы производить переустановку начального числа генератора случайных чисел RANDOM перед тем, как начать работу с ним. Если используется одно и то же начальное число, то генератор RANDOM будет выдавать одну и ту же последовательность чисел. (Это совпадает с поведением функции *random()* в языке C.)

Пример 9-26. Переустановка RANDOM

```

#!/bin/bash
# seeding-random.sh: Переустановка переменной RANDOM.

MAXCOUNT=25      # Длина генерируемой последовательности чисел.

random_numbers ()

```



```

{
count=0
while [ "$count" -lt "$MAXCOUNT" ]
do
    number=$RANDOM
    echo -n "$number "
    let "count += 1"
done
}

echo; echo

RANDOM=1          # Переустановка начального числа генератора случайных чисел RANDOM.
random_numbers

echo; echo

RANDOM=1          # То же самое начальное число...
random_numbers  # ...в результате получается та же последовательность чисел.
                #
                # В каких случаях может оказаться полезной генерация совпадающих
серий?

echo; echo

RANDOM=2          # Еще одна попытка, но с другим начальным числом...
random_numbers  # получим другую последовательность.

echo; echo

# RANDOM=$$ в качестве начального числа выбирается PID процесса-сценария.
# Вполне допустимо взять в качестве начального числа результат работы команд 'time'
или 'date'.

# Немного воображения...
SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
# Псевдослучайное число забирается
#+ из системного генератора псевдослучайных чисел /dev/urandom ,
#+ затем конвертируется в восьмеричное число командой "od",
#+ и наконец "awk" возвращает единственное число для переменной SEED.
RANDOM=$SEED
random_numbers

echo; echo

exit 0

```



Системный генератор `/dev/urandom` дает последовательность псевдослучайных чисел с равномерным распределением, чем `$RANDOM`. Команда `dd if=/dev/urandom of=t bs=1 count=XX` создает файл, содержащий последовательность псевдослучайных чисел. Эти числа требуют дополнительной обработки, например с помощью команды [od](#) (этот пример используется в примере выше) или [dd](#) (см. [Пример 12-42](#)).

Есть и другие способы генерации псевдослучайных последовательностей в сценариях. Для этого достаточно удобные средства.

Пример 9-27. Получение псевдослучайных чисел с помощью [awk](#)

```

#!/bin/bash
# random2.sh: Генерация псевдослучайных чисел в диапазоне 0 - 1.
# Используется функция rand() из awk.

AWKSCRIPT=' { srand(); print rand() } '
# Команды/параметры, передаваемые awk
# Обратите внимание, функция srand() переустанавливает начальное число генератора случайных чисел.

```

```
echo -n "Случайное число в диапазоне от 0 до 1 = "  
echo | awk "$AWKSCRIPT"  
  
exit 0
```

```
# Упражнения:  
# -----
```

```
# 1) С помощью оператора цикла выведите 10 различных случайных чисел.  
#     (Подсказка: вам потребуется вызвать функцию "srand()"  
#     в каждом цикле с разными начальными числами.  
#     Что произойдет, если этого не сделать?)
```

```
# 2) Заставьте сценарий генерировать случайные числа в диапазоне 10 - 100  
#     используя целочисленный множитель, как коэффициент масштабирования
```

```
# 3) То же самое, что и во втором упражнении,  
#     но на этот раз случайные числа должны быть целыми.
```

9.7. Двойные круглые скобки

Эта конструкция во многом похожа на инструкцию [let](#), внутри ((...)) вычисляются арифметические выражения и возвращается их результат. В простейшем случае, конструкция `a=$((5 + 3))` присвоит переменной "a" значение выражения "5 + 3", или 8. Но, кроме того, двойные круглые скобки позволяют работать с переменными в стиле языка C.

Пример 9-28. Работа с переменными в стиле языка C

```
#!/bin/bash  
# Работа с переменными в стиле языка C.  
  
echo  
  
(( a = 23 )) # Присвоение переменной в стиле C, с обеих сторон от "=" стоят пробелы.  
echo "a (начальное значение) = $a"  
  
(( a++ ))    # Пост-инкремент 'a', в стиле C.  
echo "a (после a++) = $a"  
  
(( a-- ))    # Пост-декремент 'a', в стиле C.  
echo "a (после a--) = $a"  
  
(( ++a ))    # Пред-инкремент 'a', в стиле C.  
echo "a (после ++a) = $a"  
  
(( --a ))    # Пред-декремент 'a', в стиле C.  
echo "a (после --a) = $a"  
  
echo  
  
(( t = a<45?7:11 )) # Трехместный оператор в стиле языка C.  
echo "If a < 45, then t = 7, else t = 11."  
echo "t = $t "      # Да!  
  
echo  
  
# См. так же описание ((...)) в циклах "for" и "while".
```

Эта конструкция доступна в Bash, начиная с версии 2.04.

exit 0

См. так же [Пример 10-12](#).

Глава 10. Циклы и ветвления

Управление ходом исполнения -- один из ключевых моментов структурной организации сценариев на языке командной оболочки. Циклы и преходы являются теми инструментальными средствами, которые обеспечивают управление порядком исполнения команд.

10.1. Циклы

Цикл -- это блок команд, который исполняется многократно до тех пор, пока не будет выполнено условие выхода из цикла.

циклы **for**

for (in)

Это одна из основных разновидностей циклов. И она значительно отличается от аналога в языке C.

```
for arg in [list]  
do  
    команда (ы)...  
done
```



На каждом проходе цикла, переменная-аргумент цикла *arg* последовательно, одно за другим, принимает значения из списка *list*.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"  
# На первом проходе, $arg = $var1  
# На втором проходе, $arg = $var2  
# На третьем проходе, $arg = $var3  
# ...  
# На N-ном проходе, $arg = $varN
```

Элементы списка заключены в кавычки для того, чтобы предотвратить возможное разбиение их на отдельные аргументы (слова).

Элементы списка могут включать в себя шаблонные символы.

Есл ключевое слово **do** находится в одной строке со словом **for**, то после списка аргументов (перед **do**) необходимо ставить точку с запятой.

```
for arg in [list]; do
```

Пример 10-1. Простой цикл for

```
#!/bin/bash
# Список планет.

for planet in Меркурий Венера Земля Марс Юпитер Сатурн Уран Нептун Плутон
do
    echo $planet
done

echo

# Если 'список аргументов' заключить в кавычки, то он будет восприниматься как
единственный аргумент .
for planet in "Меркурий Венера Земля Марс Юпитер Сатурн Уран Нептун Плутон"
do
    echo $planet
done

exit 0
```



Каждый из элементов [списка] может содержать несколько аргументов. Это бывает полезным при обработке групп параметров. В этом случае, для принудительного разбора каждого из аргументов в списке, необходимо использовать инструкцию **set** (см. [Пример 11-13](#)).

Пример 10-2. Цикл for с двумя параметрами в каждом из элементов списка

```
#!/bin/bash
# Список планет.

# Имя каждой планеты ассоциировано с расстоянием от планеты до Солнца (млн.
миль).

for planet in "Меркурий 36" "Венера 67" "Земля 93" "Марс 142" "Юпитер 483"
do
    set -- $planet # Разбиение переменной "planet" на множество аргументов
(позиционных параметров).
    # Конструкция "--" предохраняет от неожиданностей, если $planet "пуста" или
начинается с символа "-".

    # Если каждый из аргументов потребуется сохранить, поскольку на следующем
проходе они будут "забиты" новыми значениями,
    # То можно поместить их в массив,
    #     original_params=("$@")

    echo "$1      в $2,000,000 миль от Солнца"
    #----две табуляции----к параметру $2 добавлены нули
done

# (Спасибо S.C., за разъяснения.)

exit 0
```

В качестве списка, в цикле **for**, можно использовать переменную.

Пример 10-3. *Fileinfo*: обработка списка файлов, находящегося в переменной

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

```

FILES="/usr/sbin/privatepw
/usr/sbin/pwck
/usr/sbin/go500gw
/usr/bin/fakefile
/sbin/mkreiserfs
/sbin/ypbind"      # Список интересующих нас файлов.
                   # В список добавлен фиктивный файл /usr/bin/fakefile.

echo

for file in $FILES
do

    if [ ! -e "$file" ]      # Проверка наличия файла.
    then
        echo "Файл $file не найден."; echo
        continue          # Переход к следующей итерации.
    fi

    ls -l $file | awk '{ print $8 "          размер: " $5 }' # Печать 2 полей.
    whatis `basename $file` # Информация о файле.
    echo
done

exit 0

```

В [списке] цикла **for** могут быть использованы имена файлов, которые в свою очередь могут содержать символы-шаблоны.

Пример 10-4. Обработка списка файлов в цикле **for**

```

#!/bin/bash
# list-glob.sh: Создание список файлов в цикле for с использованием
# операции подстановки имен файлов ("globbing").

echo

for file in *
do
    ls -l "$file" # Список всех файлов в $PWD (текущем каталоге).
    # Напоминаю, что символу "*" соответствует любое имя файла,
    # однако, в операциях подстановки имен файлов ("globbing"),
    # имеются исключения -- имена файлов, начинающиеся с точки.

    # Если в каталоге нет ни одного файла, соответствующего шаблону,
    # то за имя файла принимается сам шаблон.
    # Чтобы избежать этого, используйте ключ nullglob
    # (short -s nullglob).
    # Спасибо S.C.
done

echo; echo

for file in [jx]*
do
    rm -f $file # Удаление файлов, начинающихся с "j" или "x" в $PWD.
    echo "Удален файл \"$file\"".
done

echo

exit 0

```

Если [список] в цикле **for** не задан, то в качестве оного используется переменная **\$@** -- список аргументов командной строки. Оень остроумно эта особенность проиллюстрирована в [Пример A-18](#).

Пример 10-5. Цикл for без списка аргументов

```
#!/bin/bash

# Попробуйте вызвать этот сценарий с аргументами и без них и посмотреть на
# результаты.

for a
do
    echo -n "$a "
done

# Список аргументов не задан, поэтому цикл работает с переменной '$@'
#+ (список аргументов командной строки, включая пробельные символы).

echo

exit 0
```

При создании списка аргументов, в цикле for допускается пользоваться [подстановкой команд](#). См. [Пример 12-39](#), [Пример 10-10](#) и [Пример 12-33](#).

Пример 10-6. Создание списка аргументов в цикле for с помощью операции подстановки команд

```
#!/bin/bash
# ущЫь for гЯ [гащгыЯЭ], гЯкФСЮЮйЭ г аЯЭЯниР аЯФгдСЮЯзыщ ыЯЭСЮФ.

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
```

Более сложный пример использования подстановки команд при создании списка аргументов цикла.

Пример 10-7. [grep](#) для бинарных файлов

```
#!/bin/bash
# bin-grep.sh: Поиск строк в двоичных файлах.

# замена "grep" для бинарных файлов.
# Аналогично команде "grep -a"

E_BADARGS=65
E_NOFILE=66

if [ $# -ne 2 ]
then
    echo "Порядок использования: `basename $0` string filename"
    exit $E_BADARGS
fi

if [ ! -f "$2" ]
then
    echo "Файл \"$2\" не найден."
    exit $E_NOFILE
fi

for word in $( strings "$2" | grep "$1" )
```

```

# Инструкция "strings" возвращает список строк в двоичных файлах.
# Который затем передается по конвейеру команде "grep", для выполнения поиска.
do
    echo $word
done

# Как указывает S.C., вышеприведенное объявление цикла for может быть упрощено
# strings "$2" | grep "$1" | tr -s "$IFS" '\n*'

# Попробуйте что нибудь подобное: ./bin-grep.sh mem /bin/ls"

exit 0

```

Еще один пример.

Пример 10-8. Список всех пользователей системы

```

#!/bin/bash
# userlist.sh

PASSWORD_FILE=/etc/passwd
n=1 # Число пользователей

for name in $(awk 'BEGIN{FS=":"}{print $1}' < "$PASSWORD_FILE" )
# Разделитель полей = : ^^^^^^
# Вывод первого поля ^^^^^^^^^
# Данные берутся из файла паролей ^^^^^^^^^^^^^^^^^^^^^^^
do
    echo "Пользователь #$n = $name"
    let "n += 1"
done

# Пользователь #1 = root
# Пользователь #2 = bin
# Пользователь #3 = daemon
# ...
# Пользователь #30 = bozo

exit 0

```

И заключительный пример использования подстановки команд при создании [списка].

Пример 10-9. Проверка авторства всех бинарных файлов в текущем каталоге

```

#!/bin/bash
# findstring.sh:
# Поиск заданной строки в двоичном файле.

directory=/usr/local/bin/
fstring="Free Software Foundation" # Поиск файлов от FSF.

for file in $( find $directory -type f -name '*' | sort )
do
    strings -f $file | grep "$fstring" | sed -e "s%$directory%"
    # Команде "sed" передается выражение (ключ -e),
    #+ для того, чтобы изменить обычный разделитель "/" строки поиска и строки
замены
    #+ поскольку "/" - один из отфильтровываемых символов.
    # Использование такого символа порождает сообщение об ошибке (попробуйте).
done

exit 0

# Упражнение:

```

```
# -----
# Измените сценарий таким образом, чтобы он брал
#+ $directory и $fstring из командной строки.
```

Результат работы цикла **for** может передаваться другим командам по конвейеру.

Пример 10-10. Список символических ссылок в каталоге

```
#!/bin/bash
# symlinks.sh: Список символических ссылок в каталоге.

directory=${1-`pwd`}
# По-умолчанию в текущем каталоге,
# Блок кода, который выполняет аналогичные действия.
# -----
# ARGS=1 # Ожидается один аргумент командной строки.
#
# if [ $# -ne "$ARGS" ] # Если каталог поиска не задан...
# then
#   directory=`pwd` # текущий каталог
# else
#   directory=$1
# fi
# -----

echo "символические ссылки в каталоге \"$directory\""

for file in "$( find $directory -type l )" # -type l = символические ссылки
do
  echo "$file"
done | sort # В противном случае получится неотсортированный список.

# Как отмечает Dominik 'Aeneas' Schnitzer,
#+ в случае отсутствия кавычек для $( find $directory -type l )
#+ сценарий "подавится" именами файлов, содержащими пробелы.

exit 0
```

Вывод цикла может быть [перенаправлен](#) со `stdout` в файл, ниже приводится немного модифицированный вариант предыдущего примера, демонстрирующий эту возможность.

Пример 10-11. Список символических ссылок в каталоге, сохраняемый в файле

```
#!/bin/bash
# symlinks.sh: Список символических ссылок в каталоге.

OUTFILE=symlinks.list # файл со списком

directory=${1-`pwd`}
# По-умолчанию -- текущий каталог,

echo "символические ссылки в каталоге \"$directory\"" > "$OUTFILE"
echo "-----" >> "$OUTFILE"

for file in "$( find $directory -type l )" # -type l = символические ссылки
do
  echo "$file"
done | sort >> "$OUTFILE" # перенаправление вывода
# ^^^^^^^^^^^^^^^^^ в файл.

exit 0
```

Оператор цикла **for** имеет и альтернативный синтаксис записи -- очень похожий на

синтаксис оператора for в языке C. Для этого используются двойные круглые скобки.

Пример 10-12. C-подобный синтаксис оператора цикла for

```
#!/bin/bash
# Два варианта оформления цикла.

echo

# Стандартный синтаксис.
for a in 1 2 3 4 5 6 7 8 9 10
do
    echo -n "$a "
done

echo; echo

# +=====+

# А теперь C-подобный синтаксис.

LIMIT=10

for ((a=1; a <= LIMIT ; a++)) # Двойные круглые скобки и "LIMIT" без "$".
do
    echo -n "$a "
done # Конструкция заимствована из 'ksh93'.

echo; echo

# +=====+

# Попробуем и C-шный оператор "запятая".

for ((a=1, b=1; a <= LIMIT ; a++, b++)) # Запятая разделяет две операции,
которые выполняются совместно.
do
    echo -n "$a-$b "
done

echo; echo

exit 0
```

См. так же [Пример 25-10](#), [Пример 25-11](#) и [Пример А-7](#).

А сейчас пример сценария, который может найти "реальное" применение.

Пример 10-13. Работа с командой efa в пакетном режиме

```
#!/bin/bash

EXPECTED_ARGS=2
E_BADARGS=65

if [ $# -ne $EXPECTED_ARGS ]
# Проверка наличия аргументов командной строки.
then
    echo "Порядок использования: `basename $0` phone# text-file"
    exit $E_BADARGS
fi

if [ ! -f "$2" ]
```

```

then
    echo "Файл $2 не является текстовым файлом"
    exit $E_BADARGS
fi

fax make $2          # Создать fax-файлы из текстовых файлов.

for file in $(ls $2.0*) # Все файлы, получившиеся в результате преобразования.
                    # Используется шаблонный символ в списке.
do
    fil="$fil $file"
done

efax -d /dev/ttyS3 -o1 -t "T$1" $fil # отправить.

# Как указывает S.C., в цикл for может быть вставлена сама команда отправки в
# виде:
#   efax -d /dev/ttyS3 -o1 -t "T$1" $2.0*
# но это не так поучительно [;-)].

exit 0

```

while

Оператор `while` проверяет условие перед началом каждой итерации и если условие истинно (если [код возврата](#) равен 0), то управление передается в тело цикла. В отличие от циклов [for](#), циклы `while` используются в тех случаях, когда количество итераций заранее не известно.

```

while [condition]
do
    command...
done

```

Как и в случае с циклами `for/in`, при размещении ключевого слова **do** в одной строке с объявлением цикла, необходимо вставлять символ ";" перед **do**.

```

while [condition]; do

```

Обратите внимание: в отдельных случаях, таких как использование конструкции [getopts](#) совместно с оператором **while**, синтаксис несколько отличается от приводимого здесь.

Пример 10-14. Простой цикл while

```

#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
do
    echo -n "$var0 "          # -n подавляет перевод строки.
    var0=`expr $var0 + 1`    # допускается var0=$((var0+1)).
done

echo

```

```
exit 0
```

Пример 10-15. Другой пример цикла while

```
#!/bin/bash

echo

while [ "$var1" != "end" ]      # возможна замена на while test "$var1" != "end"
do
    echo "Введите значение переменной #1 (end - выход) "
    read var1                  # Конструкция 'read $var1' недопустима (почему?).
    echo "переменная #1 = $var1" # кавычки обязательны, потому что имеется символ
    "#".
    # Если введено слово 'end', то оно тоже выводится на экран.
    # потому, что проверка переменной выполняется в начале итерации (перед
    вводом).
    echo
done

exit 0
```

Оператор **while** может иметь несколько условий. Но только последнее из них определяет возможность продолжения цикла. В этом случае синтаксис оператора цикла должен быть несколько иным.

Пример 10-16. Цикл while с несколькими условиями

```
#!/bin/bash

var1=unset
previous=$var1

while echo "предыдущее значение = $previous"
do
    echo
    previous=$var1      # запомнить предыдущее значение
    [ "$var1" != end ]
    # В операторе "while" присутствуют 4 условия, но только последнее
    управляет циклом.
    # *последнее* условие - единственное, которое вычисляется.
done
echo "Введите значение переменной #1 (end - выход) "
read var1
echo "текущее значение = $var1"
done

# попробуйте самостоятельно разобраться в сценарии works.

exit 0
```

Как и в случае с **for**, цикл **while** может быть записан в C-подобной нотации, с использованием двойных круглых скобок (см. так же [Пример 9-28](#)).

Пример 10-17. C-подобный синтаксис оформления цикла while

```
#!/bin/bash
# wh-loop.sh: Цикл перебора от 1 до 10.

LIMIT=10
a=1

while [ "$a" -le $LIMIT ]
do
    echo -n "$a "
```

```

    let "a+=1"
done          # Пока ничего особенного.

echo; echo

# +=====+

# А теперь оформим в стиле языка C.

((a = 1))     # a=1
# Двойные скобки допускают наличие лишних пробелов в выражениях.

while (( a <= LIMIT )) # В двойных скобках символ "$" перед переменными
опускается.
do
    echo -n "$a "
    ((a += 1)) # let "a+=1"
    # Двойные скобки позволяют наращивание переменной в стиле языка C.
done

echo

# Теперь, программисты, пишущие на C, могут чувствовать себя в Bash как дома.

exit 0

```



Стандартное устройство ввода `stdin`, для цикла **while**, можно [перенаправить на файл](#) с помощью команды перенаправления `<` в конце цикла.

until

Оператор цикла **until** проверяет условие в начале каждой итерации, но в отличие от **while** итерация возможна только в том случае, если условие ложно.

```

until [condition-is-true]
do
    command...
done

```

Обратите внимание: оператор **until** проверяет условие завершения цикла ПЕРЕД очередной итерацией, а не после, как это принято в некоторых языках программирования.

Как и в случае с циклами `for/in`, при размещении ключевого слова **do** в одной строке с объявлением цикла, необходимо вставлять символ ";" перед **do**.

```

until [condition-is-true]; do

```

Пример 10-18. Цикл until

```

#!/bin/bash

until [ "$var1" = end ] # Проверка условия производится в начале итерации.
do
    echo "Введите значение переменной #1 "
    echo "(end - выход)"
    read var1

```

```
    echo "значение переменной #1 = $var1"
done

exit 0
```

10.2. Вложенные циклы

Цикл называется вложенным, если он размещается внутри другого цикла. На первом проходе, внешний цикл вызывает внутренний, который исполняется до своего завершения, после чего управление передается в тело внешнего цикла. На втором проходе внешний цикл опять вызывает внутренний. И так до тех пор, пока не завершится внешний цикл. Само собой, как внешний, так и внутренний циклы могут быть прерваны командой **break**.

Пример 10-19. Вложенный цикл

```
#!/bin/bash
# Вложенные циклы "for".

outer=1          # Счетчик внешнего цикла.

# Начало внешнего цикла.
for a in 1 2 3 4 5
do
    echo "Итерация #$outer внешнего цикла."
    echo "-----"
    inner=1      # Сброс счетчика вложенного цикла.

    # Начало вложенного цикла.
    for b in 1 2 3 4 5
    do
        echo "Итерация #$inner вложенного цикла."
        let "inner+=1" # Увеличить счетчик итераций вложенного цикла.
    done
    # Конец вложенного цикла.

    let "outer+=1" # Увеличить счетчик итераций внешнего цикла.
    echo          # Пустая строка для отделения итераций внешнего цикла.
done
# Конец внешнего цикла.

exit 0
```

Демонстрацию вложенных циклов "while" вы найдете в [Пример 25-6](#), а вложение цикла "while" в "until" -- в [Пример 25-8](#).

10.3. Управление ходом выполнения цикла

break, continue

Для управления ходом выполнения цикла служат команды **break** и **continue** [23] и точно соответствуют своим аналогам в других языках программирования. Команда **break** прерывает исполнение цикла, в то время как **continue** передает управление в начало цикла, минуя все последующие команды в теле цикла.

Пример 10-20. Команды break и continue в цикле

```

#!/bin/bash

LIMIT=19 # Верхний предел

echo
echo "Печать чисел от 1 до 20 (исключая 3 и 11)."
```

a=0

```

while [ $a -le "$LIMIT" ]
do
  a=$((a+1))

  if [ "$a" -eq 3 ] || [ "$a" -eq 11 ] # Исключить 3 и 11
  then
    continue # Переход в начало цикла.
  fi

  echo -n "$a "
done

# Упражнение:
# Почему число 20 тоже выводится?

echo; echo

echo Печать чисел от 1 до 20, но взгляните, что происходит после вывода числа 2
#####

# Тот же цикл, только 'continue' заменено на 'break'.

a=0

while [ "$a" -le "$LIMIT" ]
do
  a=$((a+1))

  if [ "$a" -gt 2 ]
  then
    break # Завершение работы цикла.
  fi

  echo -n "$a "
done

echo; echo; echo

exit 0
```

Команде **break** может быть передан необязательный параметр. Команда **break** без параметра прерывает тот цикл, в который она вставлена, а **break N** прерывает цикл, стоящий на N уровней выше (причем 1-й уровень -- это уровень текущего цикла, прим. перев.).

Пример 10-21. Прерывание многоуровневых циклов

```

#!/bin/bash
# break-levels.sh: Прерывание циклов.

# "break N" прерывает исполнение цикла, стоящего на N уровней выше текущего.

for outerloop in 1 2 3 4 5
do
  echo -n "Группа $outerloop:  "

  for innerloop in 1 2 3 4 5
```

```

do
    echo -n "$innerloop "

    if [ "$innerloop" -eq 3 ]
    then
        break # Попробуйте "break 2",
              # тогда будут прерываться как вложенный, так и внешний циклы
    fi
done

echo
done

echo

exit 0

```

Команда **continue**, как и команда **break**, может иметь необязательный параметр. В простейшем случае, команда **continue** передает управление в начало текущего цикла, а команда **continue N** прерывает исполнение текущего цикла и передает управление в начало внешнего цикла, отстоящего от текущего на N уровней (причем 1-й уровень -- это уровень текущего цикла, прим. перев.).

Пример 10-22. Передача управление в начало внешнего цикла

```

#!/bin/bash
# Команда "continue N" передает управление в начало внешнего цикла, отстоящего
от текущего на N уровней.

for outer in I II III IV V          # внешний цикл
do
    echo; echo -n "Группа $outer: "

    for inner in 1 2 3 4 5 6 7 8 9 10 # вложенный цикл
    do
        if [ "$inner" -eq 7 ]
        then
            continue 2 # Передача управления в начало цикла 2-го уровня.
                       # попробуйте убрать параметр 2 команды "continue"
        fi

        echo -n "$inner " # 8 9 10 никогда не будут напечатаны.
    done

done

echo; echo

# Упражнение:
# Подумайте, где реально можно использовать "continue N" в сценариях.

exit 0

```

Пример 10-23. Живой пример использования "continue N"

```

# Albert Reiner привел пример использования "continue N":
# -----

# Допустим, у меня есть большое количество задач, обрабатывающие некоторые
данные,
#+ которые хранятся в некоторых файлах, с именами, задаваемыми по шаблону,
#+ в заданном каталоге.
#+ Есть несколько машин, которым открыт доступ к этому каталогу
#+ и я хочу распределить обработку информации между машинами.
#+ тогда я обычно для каждой машины пишу нечто подобное:

```

```

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
done
break
done

# Конкретная реализация цикла, особенно sleep N, зависит от конкретных
# применений,
#+ но в общем случае он строится по такой схеме:

while true
do
  for job in {шаблон}
  do
    {файл уже обработан или обрабатывается} && continue
    {пометить файл как обрабатываемый, обработать, пометить как обработанный}
    continue 2
  done
done
break          # Или что нибудь подобное `sleep 600`, чтобы избежать завершения.
done

# Этот сценарий завершит работу после того как все данные будут обработаны
#+ (включая данные, которые поступили во время обработки). Использование
#+ соответствующих lock-файлов позволяет вести обработку на нескольких машинах
#+ одновременно, не производя дублирующих вычислений [которые, в моем случае,
#+ выполняются в течении нескольких часов, так что для меня это очень важно].
#+ Кроме того, поскольку поиск необработанных файлов всегда начинается с
#+ самого начала, можно задавать приоритеты в именах файлов. Конечно, можно
#+ обойтись и без `continue 2`, но тогда придется ввести дополнительную
#+ проверку -- действительно ли был обработан тот или иной файл
#+ (чтобы перейти к поиску следующего необработанного файла).

```



Конструкция **continue N** довольно сложна в понимании и применении, поэтому, вероятно лучше будет постараться избегать ее использования.

10.4. Операторы выбора

Инструкции **case** и **select** технически не являются циклами, поскольку не предусматривают многократное исполнение блока кода. Однако, они, как и циклы, управляют ходом исполнения программы, в зависимости от начальных или конечных условий.

case (in) / esac

Конструкция **case** эквивалентна конструкции **switch** в языке C/C++. Она позволяет выполнять тот или иной участок кода, в зависимости от результатов проверки условий. Она является, своего рода, краткой формой записи большого количества

операторов if/then/else и может быть неплохим инструментом при создании разного рода меню.

```
case "$variable" in
```

```
"$condition1" )  
  command...  
;;
```

```
"$condition2" )  
  command...  
;;
```

```
esac
```



- Заключать переменные в кавычки необязательно, поскольку здесь не производится разбиения на отдельные слова.
- Каждая строка с условием должна завершаться правой (закрывающей) круглой скобкой).
- Каждый блок команд, обрабатывающих по заданному условию, должен завершаться двумя символами точка-с-запятой ;;.
- Блок **case** должен завершаться ключевым словом **esac** (case записанное в обратном порядке).

Пример 10-24. Использование case

```
#!/bin/bash  
  
echo; echo "Нажмите клавишу и затем клавишу Return."  
read Keypress  
  
case "$Keypress" in  
  [a-z]  ) echo "буква в нижнем регистре";;  
  [A-Z]  ) echo "Буква в верхнем регистре";;  
  [0-9]  ) echo "Цифра";;  
  *      ) echo "Знак пунктуации, пробел или что-то другое";;  
esac # Допускается указывать диапазоны символов в [квадратных скобках].  
  
# Упражнение:  
# -----  
# Сейчас сценарий считывает нажатую клавишу и завершается.  
# Измените его так, чтобы сценарий продолжал отвечать на нажатия клавиш,  
# но завершался бы только после ввода символа "X".  
# Подсказка: заключите все в цикл "while".  
  
exit 0
```

Пример 10-25. Создание меню с помощью case

```
#!/bin/bash  
  
# Грубый пример базы данных  
  
clear # Очистка экрана
```

```

echo "          Список"
echo "          -----"
echo "Выберите интересующую Вас персону:"
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 взята в кавычки.

    "E" | "e" )
# Пользователь может ввести как заглавную, так и строчную букву.
echo
echo "Roland Evans"
echo "4321 Floppy Dr."
echo "Hardscrabble, CO 80753"
echo "(303) 734-9874"
echo "(303) 734-9892 fax"
echo "revans@zzy.net"
echo "Старый друг и партнер по бизнесу"
;;
# Обратите внимание: блок кода, анализирующий конкретный выбор, завершается
# двумя символами "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 "Подружка"
echo "День рождения: 11 февраля"
;;

# Информация о Smith и Zane будет добавлена позднее.

    * )
# Выбор по-умолчанию.
# "Пустой" ввод тоже обрабатывается здесь.
echo
echo "Нет данных."
;;

esac

echo

# Упражнение:
# -----
# Измените этот сценарий таким образом, чтобы он не завершал работу
#+ после вывода информации о персоне, а переходил на ожидание нового
#+ ввода от пользователя.

exit 0

```

Очень хороший пример использования **case** для анализа аргументов, переданных из командной строки.

```

#!/bin/bash

case "$1" in

```

```

") echo "Порядок использования: ${0##*/} <filename>"; exit 65;; # Параметры
командной строки отсутствуют,
# или первый параметр --
"пустой".
# Обратите внимание на ${0##*/} это подстановка параметра ${var##pattern}. В
результате получается $0.

-*) FILENAME=./$1;; # Если имя файла (аргумент $1) начинается с "-",
# то заменить его на ./$1
# тогда параметр не будет восприниматься как ключ команды.

* ) FILENAME=$1;; # В противном случае -- $1.
esac

```

Пример 10-26. Оператор case допускает использовать подстановку команд вместо анализируемой переменной

```

#!/bin/bash
# Подстановка команд в "case".

case $( arch ) in # команда "arch" возвращает строку, описывающую аппаратную
архитектуру.
i386 ) echo "Машина на базе процессора 80386";;
i486 ) echo "Машина на базе процессора 80486";;
i586 ) echo "Машина на базе процессора Pentium";;
i686 ) echo "Машина на базе процессора Pentium2 или выше";;
*    ) echo "Машина на другом типе процессора";;
esac

exit 0

```

Оператор **case** допускает использование шаблонных конструкций.

Пример 10-27. Простой пример сравнения строк

```

#!/bin/bash
# match-string.sh: простое сравнение строк

match_string ()
{
    MATCH=0
    NOMATCH=90
    PARAMS=2 # Функция требует два входных аргумента.
    BAD_PARAMS=91

    [ $# -eq $PARAMS ] || return $BAD_PARAMS

    case "$1" in
"$2") return $MATCH;;
* ) return $NOMATCH;;
    esac
}

a=one
b=two
c=three
d=two

match_string $a # неверное число аргументов
echo $? # 91

match_string $a $b # не равны

```

```

echo $?          # 90

match_string $b $d # равны
echo $?          # 0

exit 0

```

Пример 10-28. Проверка ввода

```

#!/bin/bash
# isalpha.sh: Использование "case" для анализа строк.

SUCCESS=0
FAILURE=-1

isalpha () # Проверка - является ли первый символ строки символом алфавита.
{
if [ -z "$1" ] # Вызов функции без входного аргумента?
then
return $FAILURE
fi

case "$1" in
[a-zA-Z]*) return $SUCCESS;; # Первый символ - буква?
*) return $FAILURE;;
esac
} # Сравните с функцией "isalpha ()" в языке C.

isalpha2 () # Проверка - состоит ли вся строка только из символов алфавита.
{
[ $# -eq 1 ] || return $FAILURE

case $1 in
*(!a-zA-Z)*|") return $FAILURE;;
*) return $SUCCESS;;
esac
}

isdigit () # Проверка - состоит ли вся строка только из цифр.
{
# Другими словами - является ли строка целым числом.
[ $# -eq 1 ] || return $FAILURE

case $1 in
*(!0-9)*|") return $FAILURE;;
*) return $SUCCESS;;
esac
}

check_var () # Интерфейс к isalpha
{
if isalpha "$@"
then
echo "\"$*\\" начинается с алфавитного символа."
if isalpha2 "$@"
then # Дальнейшая проверка не имеет смысла, если первый символ не буква.
echo "\"$*\\" содержит только алфавитные символы."
else
echo "\"$*\\" содержит по меньшей мере один не алфавитный символ."
fi
else
echo "\"$*\\" начинсется с не алфавитного символа ."
# Если функция вызвана без входного параметра,
#+ то считается, что строка содержит "не алфавитной" символ.
fi
}

```

```

echo
}

digit_check () # Интерфейс к isdigit ().
{
if isdigit "$@"
then
    echo "\"$*\\" содержит только цифры [0 - 9].\"
else
    echo "\"$*\\" содержит по меньшей 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 # Вызов без параметра, что произойдет?
#
digit_check $g
digit_check $h
digit_check $i

exit 0 # Сценарий дополнен S.C.

# Упражнение:
# -----
# Напишите функцию 'isfloat ()', которая проверяла бы вещественные числа.
# Подсказка: Эта функция подобна функции 'isdigit ()',
#+ надо лишь добавить анализ наличия десятичной точки.

```

select

Оператор **select** был заимствован из Korn Shell, и является еще одним инструментом, используемым при создании меню.

```

select variable [in list]
do
    command...
break
done

```

Этот оператор предлагает пользователю выбрать один из представленных вариантов. Примечательно, что **select** по-умолчанию использует в качестве приглашения к вводу (prompt) -- PS3 (#?), который легко изменить.

Пример 10-29. Создание меню с помощью select

```
#!/bin/bash

PS3='Выберите ваш любимый овощ: ' # строка приглашения к вводу (prompt)

echo

select vegetable in "бобы" "морковь" "картофель" "лук" "брюква"
do
    echo
    echo "Вы предпочитаете $vegetable."
    echo ";-))"
    echo
    break # если 'break' убрать, то получится бесконечный цикл.
done

exit 0
```

Если в операторе **select** список *in list* не задан, то в качестве списка будет использоваться список аргументов ($\$@$), передаваемый сценарию или функции.

Сравните это с поведением оператора цикла

```
for variable [in list]
```

в котором не задан список аргументов.

Пример 10-30. Создание меню с помощью select в функции

```
#!/bin/bash

PS3='Выберите ваш любимый овощ: '

echo

choice_of()
{
    select vegetable
    # список выбора [in list] отсутствует, поэтому 'select' использует входные
    # аргументы функции.
    do
        echo
        echo "Вы предпочитаете $vegetable."
        echo ";-))"
        echo
        break
    done
}

choice_of бобы рис морковь редис томат шпинат
#          $1  $2  $3      $4    $5    $6
#          передача списка выбора в функцию choice_of()

exit 0
```

См. так же [Пример 34-3](#).

Глава 11. Внутренние команды

Внутренняя команда -- это **команда**, которая встроена непосредственно в Bash. Команды делаются встроенными либо из соображений производительности -- встроенные команды исполняются быстрее, чем внешние, которые, как правило, запускаются в дочернем процессе, либо из-за необходимости прямого доступа к внутренним структурам командного интерпретатора.

Действие, когда какая либо команда или сама командная оболочка инициирует (*порождает*) новый подпроцесс, что бы выполнить какую либо работу, называется *ветвлением* (*forking*) процесса. Новый процесс называется "дочерним" (или "потомком"), а породивший его процесс -- "родительским" (или "предком"). В результате и *потомок* и *предок* продолжают исполняться одновременно -- параллельно друг другу.

В общем случае, *встроенные команды* Bash, при исполнении внутри сценария, не порождают новый подпроцесс, в то время как вызов внешних команд, как правило, приводит к созданию нового подпроцесса.

Внутренние команды могут иметь внешние аналоги. Например, внутренняя команда Bash -- **echo** имеет внешний аналог `/bin/echo` и их поведение практически идентично.

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

```
echo "Эта строка выводится внутренней командой \"echo\"."
/bin/echo "А эта строка выводится внешней командой the /bin/echo."
```

Ключевое слово (keyword) -- это *зарезервированное* слово, синтаксический элемент (token) или оператор. Ключевые слова имеют специальное назначение для командного интерпретатора, и фактически являются элементами синтаксиса языка командной оболочки. В качестве примера можно привести "for", "while", "do", "!", которые являются ключевыми (или зарезервированными) словами. Подобно *встроенным командам*, ключевые слова жестко зашиты в Bash, но в отличие от встроенных команд, ключевые слова не являются командами как таковыми, хотя при этом могут являться их составной частью. [\[24\]](#)

Ввод/вывод

echo

выводит (на stdout) выражение или содержимое переменной (см. [Пример 4-1](#)).

```
echo Hello
echo $a
```

Для вывода экранированных символов, **echo** требует наличие ключа -e. См. [Пример 5-2](#).


Обычно, командв **echo** выводит в конце символ перевода строки. Подавить вывод это символа можно ключом -n.



Команда **echo** может использоваться для передачи информации по

конвейеру другим командам.

```
if echo "$VAR" | grep -q txt # if [[ $VAR = *txt* ]]
then
  echo "$VAR содержит подстроку \"txt\""
fi
```

-  Кроме того, команда **echo**, в комбинации с [подстановкой команд](#) может участвовать в операции присвоения значения переменной.

```
a=`echo "HELLO" | tr A-Z a-z`
```

См. так же [Пример 12-15](#), [Пример 12-2](#), [Пример 12-32](#) и [Пример 12-33](#).

Следует запомнить, что команда **echo `command`** удалит все символы перевода строки, которые будут выведены командой *command*.

Переменная [\\$IFS](#) обычно содержит символ перевода строки `\n`, как один из вариантов [пробельного](#) символа. Bash разобьет вывод команды *command*, по пробельным символам, на аргументы и передаст их команде **echo**, которая выведет 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
362 Nov 7 2000 seconds.au
```

-  Это встроенная команда Bash и имеет внешний аналог `/bin/echo`.


```
bash$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

printf

printf -- команда форматированного вывода, расширенный вариант команды **echo** и ограниченный вариант библиотечной функции `printf()` в языке C, к тому же синтаксис их несколько отличается друг от друга.

printf *format-string... parameter...*

Это встроенная команда Bash. Имеет внешний аналог `/bin/printf` или `/usr/bin/printf`. За более подробной информацией обращайтесь к страницам справочного руководства **man 1 printf** по системным командам.

-  Старые версии Bash могут не поддерживать команду **printf**.

Пример 11-1. printf в действии


```

#!/bin/bash
# printf demo

# От переводчика:
# Считаю своим долгом напомнить, что в качестве разделителя дробной и целой
# частей в вещественных числах, может использоваться символ "запятая"
# (в русских локалях), поэтому данный сценарий может выдавать сообщение
# об ошибке (у меня так и произошло) при выводе числа PI.
# Тогда попробуйте заменить в определении числа PI десятичную точку
# на запятую -- это должно помочь. ;- )

PI=3,14159265358979
DecimalConstant=31373
Message1="Поздравляю,"
Message2="Землянин."

echo

printf "Число пи с точностью до 2 знака после запятой = %1.2f" $PI
echo
printf "Число пи с точностью до 9 знака после запятой = %1.9f" $PI # Даже
округляет правильно.

printf "\n" # Перевод строки,

printf "Константа = \t%d\n" $DecimalConstant # Вставлен символ табуляции (\t)

printf "%s %s \n" $Message1 $Message2

echo

# =====#
# Эмуляция функции 'sprintf' в языке C.
# Запись форматированной строки в переменную.

echo

Pi12=$(printf "%1.12f" $PI)
echo "Число пи с точностью до 12 знака после запятой = $Pi12"

Msg=`printf "%s %s \n" $Message1 $Message2`
echo $Msg; echo $Msg

exit 0

```

Одно из полезных применений команды **printf** -- форматированный вывод сообщений об ошибках

```

E_BADDIR=65

var=nonexistent_directory

error()
{
    printf "$@" >&2
    # Форматированный вывод аргументов на stderr.
    echo
    exit $E_BADDIR
}

cd $var || error $"Невозможно перейти в каталог %s." "$var"

# Спасибо S.C.

```

"Читает" значение переменной с устройства стандартного ввода -- `stdin`, в интерактивном режиме это означает клавиатуру. Ключ `-a` позволяет записывать значения в массивы (см. [Пример 25-3](#)).

Пример 11-2. Ввод значений переменных с помощью `read`

```
#!/bin/bash

echo -n "дите значение переменной 'var1': "
# Ключ -n подавляет вывод символа перевода строки.

read var1
# Обратите внимание -- перед именем переменной отсутствует символ '$'.

echo "var1 = $var1"

echo

# Одной командой 'read' можно вводить несколько переменных.
echo -n "дите значения для переменных 'var2' и 'var3' (через пробел или
табуляцию): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
# Если было введено значение только одной переменной, то вторая останется
"пустой".

exit 0
```

Если команде `read` не было передано ни одной переменной, то ввод будет осуществлен в переменную [\\$REPLY](#).

Пример 11-3. Пример использования команды `read` без указания переменной для ввода

```
#!/bin/bash

echo

# ----- #
# Первый блок кода.
echo -n "Введите значение: "
read var
echo "\"var\" = \"$var\""
# Здесь нет ничего неожиданного.
# ----- #

echo

echo -n "Введите другое значение: "
read          # Команда 'read' употребляется без указания переменной для
ввода,
              #+ тем не менее...
              #+ По-умолчанию ввод осуществляется в переменную $REPLY.
var="$REPLY"
echo "\"var\" = \"$var\""
# Эта часть сценария эквивалентна первому блоку, выделенному выше.

echo

exit 0
```

Обычно, при вводе в окне терминала с помощью команды `read`, символ `\` служит для экранирования символа перевода строки. Ключ `-r` заставляет интерпретировать символ `\` как обычный символ.

Пример 11-4. Ввод многострочного текста с помощью read

```
#!/bin/bash

echo

echo "Введите строку, завершающуюся символом \\", и нажмите ENTER."
echo "Затем введите вторую строку, и снова нажмите ENTER."
read var1      # При чтении, символ "\" экранирует перевод строки.
               # первая строка \
               # вторая строка

echo "var1 = $var1"
# var1 = первая строка вторая строка

# После ввода каждой строки, завершающейся символом "\",
# вы можете продолжать ввод на другой строке.

echo; echo

echo "Введите другую строку, завершающуюся символом \\", и нажмите ENTER."
read -r var2   # Ключ -r заставляет команду "read" воспринимать "\"
               # как обычный символ.
               # первая строка \

echo "var2 = $var2"
# var2 = первая строка \

# Ввод данных прекращается сразу же после первого нажатия на клавишу ENTER.

echo

exit 0
```

Команда **read** имеет ряд очень любопытных опций, которые позволяют выводить подсказку - приглашение ко вводу (prompt), и даже читать данные не дожидаясь нажатия на клавишу **ENTER**.

```
# Чтение данных, не дожидаясь нажатия на клавишу ENTER.

read -s -n1 -p "Нажмите клавишу " keypress
echo; echo "Была нажата клавиша "\"$keypress\""."

# -s -- подавляет эхо-вывод, т.е. ввод с клавиатуры не отображается на экране.
# -n N -- ввод завершается автоматически, сразу же после ввода N-го символа.
# -p -- задает вид строки подсказки - приглашения к вводу (prompt).

# Использование этих ключей немного осложняется тем, что они должны следовать в определенном порядке.
```

Ключ **-n**, кроме всего прочего, позволяет команде **read** обнаруживать нажатие *курсорных* и некоторых других служебных клавиш.

Пример 11-5. Обнаружение нажатия на курсорные клавиши

```
#!/bin/bash
# arrow-detect.sh: Обнаружение нажатия на курсорные клавиши, и не только...
# Спасибо Sandro Magi за то что показал мне -- как.

# -----
# Коды клавиш.
arrowup='\[A'
arrowdown='\[B'
arrowrt='\[C'
```

```

arrowleft='\[D'
insert='\[2'
delete='\[3'
# -----

SUCCESS=0
OTHER=65

echo -n "Нажмите на клавишу... "
# Может потребоваться нажать на ENTER, если была нажата клавиша
# не входящая в список выше.
read -n3 key          # Прочитать 3 символа.

echo -n "$key" | grep "$arrowup" #Определение нажатой клавиши.
if [ "$?" -eq $SUCCESS ]
then
    echo "Нажата клавиша \"."
    exit $SUCCESS
fi

echo -n "$key" | grep "$arrowdown"
if [ "$?" -eq $SUCCESS ]
then
    echo "Нажата клавиша \"
    exit $SUCCESS
fi

echo -n "$key" | grep "$arrowrt"
if [ "$?" -eq $SUCCESS ]
then
    echo "Нажата клавиша \"O\"."
    exit $SUCCESS
fi

echo -n "$key" | grep "$arrowleft"
if [ "$?" -eq $SUCCESS ]
then
    echo "Нажата клавиша \"."
    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 какая-то другая клавиша."

exit $OTHER

# Упражнения:
# -----
# 1) Упростите сценарий, заменив множество if-ов
#+   одной конструкцией 'case'.
# 2) Добавьте определение нажатий на клавиши "Home", "End", "PgUp" и "PgDn".

```

Ключ - t позволяет ограничивать время ожидания ввода командой **read** (см. [Пример 9-4](#)).

Команда **read** может считывать значения для переменных из файла, [перенаправленного](#) на `stdin`. Если файл содержит не одну строку, то переменной будет присвоена только первая строка. Если команде **read** будет передано несколько переменных, то первая строка файла будет разбита, по пробелам, на несколько подстрок, каждая из которых будет записана в свою переменную. Будьте осторожны!

Пример 11-6. Чтение командой `read` из файла через [перенаправление](#)

```
#!/bin/bash

read var1 <data-file
echo "var1 = $var1"
# Первая строка из "data-file" целиком записывается в переменную var1

read var2 var3 <data-file
echo "var2 = $var2   var3 = $var3"
# Обратите внимание!
# Поведение команды "read" далеко от ожидаемого!
# 1) Произошел возврат к началу файла.
# 2) Вместо того, чтобы последовательно читать строки из файла,
#    по числу переменных, первая строка файла была разбита на подстроки,
#    разделенные пробелами, которые и были записаны в переменные.
# 3) В последнюю переменную была записана вся оставшаяся часть строки.
# 4) Если команде "read" будет передано большее число переменных, чем подстрок
#    в первой строке файла, то последние переменные останутся "пустыми".

echo "-----"

# Эта проблема легко разрешается с помощью цикла:
while read line
do
    echo "$line"
done <data-file
# Спасибо Heiner Steven за разъяснения.

echo "-----"

# Разбор строки, разделенной на поля
# Для задания разделителя полей, используется переменная $IFS,

echo "Список всех пользователей:"
IFS=$IFS; IFS=:      # В файле /etc/passwd, в качестве разделителя полей
                    # используется символ ":" .
while read name passwd uid gid fullname ignore
do
    echo "$name ($fullname)"
done </etc/passwd   # перенаправление ввода.
IFS=$IFS           # Восстановление предыдущего состояния переменной $IFS.
# Эту часть кода написал Heiner Steven.

# Если переменная $IFS устанавливается внутри цикла,
#+ то отпадает необходимость сохранения ее первоначального значения
#+ во временной переменной.
# Спасибо Dim Segebart за разъяснения.
echo "-----"
echo "Список всех пользователей:"

while IFS=: read name passwd uid gid fullname ignore
do
    echo "$name ($fullname)"
done </etc/passwd   # перенаправление ввода.

echo
echo "Значение переменной \ $IFS осталось прежним: $IFS"
```

exit 0



Передача информации, выводимой командой [echo](#), по конвейеру команде **read**, [будет вызывать ошибку](#).

Тем не менее, передача данных по конвейеру от [cat](#), кажется срабатывает.

```
cat file1 file2 |
while read line
do
echo $line
done
```

Файловая система

cd

Уже знакомая нам команда **cd**, изменяющая текущий каталог, может быть использована в случаях, когда некоторую команду необходимо запустить только находясь в определенном каталоге.

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

[взято из [упоминавшегося ранее](#) примера]

Команда **cd** с ключом **-P** (physical) игнорирует символические ссылки.

Команда "**cd -**" выполняет переход в каталог [\\$OLDPWD](#) -- предыдущий рабочий каталог.



Неожиданным образом выполняется команда **cd**, если ей передать, в качестве каталога назначения, два слэша.

```
bash$ cd //
bash$ pwd
//
```

Само собой разумеется, это должен был бы быть каталог **/**. Эта проблема наблюдается как в командной строке, так и в сценариях.

pwd

Выводит название текущего рабочего каталога (Print Working Directory) (см. [Пример 11-7](#)). Кроме того, имя текущего каталога хранится во внутренней переменной [\\$PWD](#).

pushd, popd, dirs

Этот набор команд является составной частью механизма "закладок" на каталоги и позволяет перемещаться по каталогам вперед и назад в заданном порядке. Для хранения имен каталогов используется стек (LIFO -- "последний вошел, первый вышел").

pushd dir-name -- помещает имя текущего каталога в стек и осуществляет переход в каталог *dir-name*.

popd -- выталкивает, находящееся на вершине стека, имя каталога и одновременно осуществляет переход в каталог, оказавшийся на вершине стека.

dirs -- выводит содержимое стека каталогов (сравните с переменной [\\$DIRSTACK](#)). В случае успеха, обе команды -- **pushd** и **popd** автоматически вызывают **dirs**.

Эти команды могут оказаться весьма полезными, когда в сценарии нужно производить частую смену каталогов, но при этом не хочется жестко "зашивать" имена каталогов. Обратите внимание: содержимое стека каталогов постоянно хранится в переменной-массиве -- \$DIRSTACK.

Пример 11-7. Смена текущего каталога

```
#!/bin/bash

dir1=/usr/local
dir2=/var/spool

pushd $dir1
# Команда 'dirs' будет вызвана автоматически (на stdout будет выведено
содержимое стека).
echo "Выполнен переход в каталог `pwd`." # Обратные одиночные кавычки.

# Теперь можно выполнить какие либо действия в каталоге 'dir1'.
pushd $dir2
echo "Выполнен переход в каталог `pwd`."

# Теперь можно выполнить какие либо действия в каталоге 'dir2'.
echo "На вершине стека находится: $DIRSTACK."
popd
echo "Возврат в каталог `pwd`."

# Теперь можно выполнить какие либо действия в каталоге 'dir1'.
popd
echo "Возврат в первоначальный рабочий каталог `pwd`."

exit 0
```

Переменные

let

Команда **let** производит арифметические операции над переменными. В большинстве случаев, ее можно считать упрощенным вариантом команды [expr](#).

Пример 11-8. Команда let, арифметические операции.

```
#!/bin/bash

echo

let a=11          # То же, что и 'a=11'
let a=a+5        # Эквивалентно "a = a + 5"
                  # (Двойные кавычки и дополнительные пробелы делают код более
удобочитаемым)
echo "11 + 5 = $a"

let "a <<= 3"     # Эквивалентно let "a = a << 3"
echo "\"\$a\" (=16) после сдвига влево на 3 разряда = $a"

let "a /= 4"     # Эквивалентно let "a = a / 4"
echo "128 / 4 = $a"

let "a -= 5"     # Эквивалентно let "a = a - 5"
```

```

echo "32 - 5 = $a"

let "a = a * 10" # Эквивалентно let "a = a * 10"
echo "27 * 10 = $a"

let "a %= 8"      # Эквивалентно let "a = a % 8"
echo "270 mod 8 = $a (270 / 8 = 33, остаток = $a)"

echo

exit 0

```

eval

```
eval arg1 [arg2] ... [argN]
```

Транслирует список аргументов, из списка, в команды.

Пример 11-9. Демонстрация команды eval

```

#!/bin/bash

y=`eval ls -l` # Подобно y=`ls -l`
echo $y        # но символы перевода строки не выводятся, поскольку имя
переменной не в кавычках.
echo
echo "$y"      # Если имя переменной записать в кавычках -- символы перевода
строки сохраняются.

echo; echo

y=`eval df`    # Аналогично y=`df`
echo $y        # но без символов перевода строки.

# Когда производится подавление вывода символов LF (перевод строки), то анализ
#+ результатов различными утилитами, такими как awk, можно сделать проще.

exit 0

```

Пример 11-10. Принудительное завершение сеанса

```

#!/bin/bash

y=`eval ps ax | sed -n '/ppp/p' | awk '{ print $1 }'`
# Выяснить PID процесса 'ppp'.

kill -9 $y # "Прихлопнуть" его

# Предыдущие строки можно заменить одной строкой
# kill -9 `ps ax | awk '/ppp/ { print $1 }'

chmod 666 /dev/ttyS3
# Завершенный, по сигналу SIGKILL, ppp изменяет права доступа
# к последовательному порту. Вернуть их в первоначальное состояние.

rm /var/lock/LCK..ttyS3 # Удалить lock-файл последовательного порта.

exit 0

```

Пример 11-11. Шифрование по алгоритму "rot13"

```

#!/bin/bash
# Реализация алгоритма шифрования "rot13" с помощью 'eval'.
# Сравните со сценарием "rot13.sh".

```



```

setvar_rot_13()          # Криптование по алгоритму "rot13"
{
    local varname=$1 varvalue=$2
    eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
}

setvar_rot_13 var "foobar" # Пропустить слово "foobar" через rot13.
echo $var                 # sbbone

echo $var | tr a-z n-za-m # foobar
                        # Расшифровывание.

# Пример предоставил Stephane Chazelas.

exit 0

```

Rory Winston представил следующий пример, как образец практического использования команды **eval**.

Пример 11-12. Замена имени переменной на ее значение, в исходном тексте программы на языке Perl, с помощью eval

В программе "test.pl", на языке Perl:

```

...
my $WEBSITE = <WEBSITE_PATH>;
...

```

Эта попытка подстановки значения переменной вместо ее имени:

```

$export WEBSITE_PATH=/usr/local/webroot
$sed 's/<WEBSITE_PATH>/$WEBSITE_PATH/' < test.pl > out

```

даст такой результат:

```

my $WEBSITE = $WEBSITE_PATH;

```

Тем не менее:

```

$export WEBSITE_PATH=/usr/local/webroot
$eval sed 's/<WEBSITE_PATH>/$WEBSITE_PATH/' < test.pl > out
#
====

```

Этот вариант дал желаемый результат -- имя переменной, в тексте программы, благополучно было заменено на ее значение:

```

my $WEBSITE = /usr/local/webroot

```



Команда **eval** может быть небезопасна. Если существует приемлемая альтернатива, то желательно воздерживаться от использования **eval**. Так, **eval \$COMMANDS** исполняет код, который записан в переменную *COMMANDS*, которая, в свою очередь, может содержать весьма неприятные сюрпризы, например **rm -rf ***. Использование команды **eval**, для исполнения кода неизвестного происхождения, крайне опасно.

set

Команда **set** изменяет значения внутренних переменных сценария. Она может использоваться для переключения [опций \(ключей, флагов\)](#), определяющих поведение скрипта. Еще одно применение -- сброс/установка [позиционных параметров \(аргументов\)](#), значения которых будут восприняты как результат работы команды (**set `command`**).

Пример 11-13. Установка значений аргументов с помощью команды set

```

#!/bin/bash

```

```

# script "set-test"

# Вызовите сценарий с тремя аргументами командной строки,
# например: "./set-test one two three".

echo
echo "Аргументы перед вызовом set `uname -a` :"
echo "Аргумент #1 = $1"
echo "Аргумент #2 = $2"
echo "Аргумент #3 = $3"

set `uname -a` # Изменение аргументов
               # значения которых берутся из результата работы `uname -a`

echo $_

echo "Аргументы после вызова set `uname -a` :"
# $1, $2, $3 и т.д. будут переустановлены в соответствии с выводом
#+ команды `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

```

Вызов **set** без параметров просто выводит список инициализированных переменных [окружения](#).

```

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

```

Если команда **set** используется с ключом "--", после которого следует переменная, то значение переменной переносится в позиционные параметры (аргументы). Если имя переменной отсутствует, то эта команда приводит к сбросу позиционных параметров.

Пример 11-14. Изменение значений позиционных параметров (аргументов)

```

#!/bin/bash

variable="one two three four five"

set -- $variable
# Значения позиционных параметров берутся из "$variable".

first_param=$1
second_param=$2
shift; shift      # сдвиг двух первых параметров.
remaining_params="$*"

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 --
# Позиционные параметры сбрасываются, если не задано имя переменной.

first_param=$1
second_param=$2
echo "первый параметр = $first_param"           # (пустое значение)
echo "второй параметр = $second_param"         # (пустое значение)

exit 0

```

См. так же [Пример 10-2](#) и [Пример 12-40](#).

unset

Команда **unset** удаляет переменную, фактически -- устанавливает ее значение в *null*. Обратите внимание: эта команда не может сбрасывать позиционные параметры (аргументы).

```

bash$ unset PATH

bash$ echo $PATH

bash$

```

Пример 11-15. "Сброс" переменной

```

#!/bin/bash
# unset.sh: Сброс переменной.

variable=hello           # Инициализация.
echo "variable = $variable"

unset variable           # Сброс.
                        # Тот же эффект дает variable=
echo "(unset) variable = $variable" # $variable = null.

exit 0

```

export

Команда **export** экспортирует переменную, делая ее доступной дочерним процессам. К сожалению, невозможно экспортировать переменную родительскому процессу. В качестве примера использования команды **export** можно привести [сценарии инициализации системы](#), вызываемые в процессе загрузки, которые инициализируют и экспортируют [переменные окружения](#), делая их доступными для пользовательских процессов.

Пример 11-16. Передача переменных во вложенный сценарий [awk](#), с помощью `export`

```
#!/bin/bash

# Еще одна версия сценария "column totaler" (col-totaler.sh)
# который суммирует заданную колонку (чисел) в заданном файле.
# Здесь используются переменные окружения, которые передаются сценарию 'awk'.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # Проверка количества входных аргументов.
then
    echo "Порядок использования: `basename $0` filename column-number"
    exit $E_WRONGARGS
fi

filename=$1
column_number=$2


#==== До этой строки идентично первоначальному варианту сценария ====#

export column_number
# Экспорт номера столбца.

# Начало awk-сценария.
# -----
awk '{ total += $ENVIRON["column_number"]
}
END { print total }' $filename
# -----
# Конец awk-сценария.

# Спасибо Stephane Chazelas.

exit 0
```

 Допускается объединение инициализации и экспорта переменной в одну инструкцию: **export var1=xxx**.

Однако, как заметил Greg Keraunen, в некоторых ситуациях такая комбинация может давать иной результат, нежели отдельная инициализация и экспорт.

```
bash$ export var=(a b); echo ${var[0]}
(a b)
bash$ var=(a b); export var; echo ${var[0]}
a
```

declare, typeset

Команды [declare](#) и [typeset](#) задают и/или накладывают ограничения на переменные.

readonly

То же самое, что и [declare -r](#), делает переменную доступной только для чтения, т.е. переменная становится подобна константе. При попытке изменить значение такой переменной выводится сообщение об ошибке. Эта команда может расцениваться как квалификатор типа **const** в языке C.

getopts

Мощный инструмент, используемый для разбора аргументов, передаваемых сценарию из командной строки. Это встроенная команда Bash, но имеется и ее "внешний" аналог [/usr/bin/getopt](#), а так же программистам, пишущим на C, хорошо знакома похожая библиотечная функция **getopt**. Она позволяет обрабатывать серии опций, объединенных в один аргумент [25] и дополнительные аргументы, передаваемые сценарию (например, `scriptname -abc -e /usr/local`).

С командой **getopts** очень тесно взаимосвязаны скрытые переменные. `$OPTIND` -- указатель на аргумент (*OPT*ion *IND*ex) и `$OPTARG` (*OPT*ion *ARG*ument) -- дополнительный аргумент опции. Символ двоеточия, следующий за именем опции, указывает на то, что она имеет дополнительный аргумент.

Обычно **getopts** упаковывается в цикл [while](#), в каждом проходе цикла извлекается очередная опция и ее аргумент (если он имеется), обрабатывается, затем уменьшается на 1 скрытая переменная `$OPTIND` и выполняется переход к началу новой итерации.



1. Опциям (ключам), передаваемым в сценарий из командной строки, должен предшествовать символ "минус" (-) или "плюс" (+). Этот префикс (- или +) позволяет **getopts** отличать опции (ключи) от прочих аргументов. Фактически, **getopts** не будет обрабатывать аргументы, если им не предшествует символ - или +, выделение опций будет прекращено как только встретится первый аргумент.
2. Типичная конструкция цикла **while** с **getopts** несколько отличается от стандартной из-за отсутствия квадратных скобок, проверяющих условие продолжения цикла.
3. Пример **getopts**, заменившей устаревшую, и не такую мощную, внешнюю команду [getopt](#).

```
while getopts ":abcde:fg" Option
# Начальное объявление цикла анализа опций.
# a, b, c, d, e, f, g -- это возможные опции (ключи).
# Символ : после опции 'e' указывает на то, что с данной опцией может идти
# дополнительный аргумент.
do
  case $Option in
    a ) # Действия, предусмотренные опцией 'a'.
    b ) # Действия, предусмотренные опцией 'b'.
    ...
    e) # Действия, предусмотренные опцией 'e', а так же необходимо обработать
$OPTARG,
      # в которой находится дополнительный аргумент этой опции.
    ...
    g ) # Действия, предусмотренные опцией 'g'.
  esac
done
shift $(( $OPTIND - 1 ))
# Перейти к следующей опции.

# Все не так сложно, как может показаться ;-)

```

Пример 11-17. Прием опций/аргументов, передаваемых сценарию, с помощью

getopts

```
#!/bin/bash
# ex33.sh

# Обработка опций командной строки с помощью 'getopts'.

# Попробуйте вызвать этот сценарий как:
# 'scriptname -mn'
# 'scriptname -oq qOption' (qOption может быть любой произвольной строкой.)
# 'scriptname -qXXX -r'
#
# 'scriptname -qr' - Неожиданный результат: "r" будет воспринят как
дополнительный аргумент опции "q"
# 'scriptname -q -r' - То же самое, что и выше
# Если опция ожидает дополнительный аргумент ("flag:"), то следующий параметр
# в командной строке, будет воспринят как дополнительный аргумент этой опции.

NO_ARGS=0
E_OPTERROR=65

if [ $# -eq "$NO_ARGS" ] # Сценарий вызван без аргументов?
then
    echo "Порядок использования: `basename $0` options (-mnopqrs)"
    exit $E_OPTERROR      # Если аргументы отсутствуют -- выход с сообщением
                        # о порядке использования скрипта
fi
# Порядок использования: scriptname -options
# Обратите внимание: дефис (-) обязателен

while getopts ":mnopq:rs" Option
do
    echo $OPTIND
    case $Option in
        m      ) echo "Сценарий #1: ключ -m-";;
        n | o ) echo "Сценарий #2: ключ -$Option-";;
        p      ) echo "Сценарий #3: ключ -p-";;
        q      ) echo "Сценарий #4: ключ -q-, с аргументом \"\$OPTARG\"";;
        # Обратите внимание: с ключом 'q' должен передаваться дополнительный
    аргумент,
        # в противном случае отработает выбор "по-умолчанию".
        r | s ) echo "Сценарий #5: ключ -$Option-";;
        *      ) echo "Выбран недопустимый ключ.";; # ПО-УМОЛЧАНИЮ
    esac
done
shift $((OPTIND - 1))
# Переход к очередному параметру командной строки.

exit 0
```

Управление сценарием

source, . ([точка](#))

Когда эта команда вызывается из командной строки, то это приводит к запуску указанного сценария. Внутри сценария, команда `source file-name` загружает файл `file-name`. Таким образом она очень напоминает директиву препроцессора языка C/C++ -- `#include`. Может найти применение в ситуациях, когда несколько сценариев пользуются одним файлом с данными или библиотекой функций.

Пример 11-18. "Подключение" внешнего файла

```
#!/bin/bash

. data-file      # Загрузка файла с данными.
```

```

# Тот же эффект дает "source data-file", но этот вариант более переносим.

# Файл "data-file" должен находиться в текущем каталоге,
#+ т.к. путь к нему не указан.

# Теперь, выведем некоторые переменные 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) = $sum"
echo "message1 (из data-file): \"$message1\""
# Обратите внимание:           кавычки экранированы

print_message Вызвана функция вывода сообщений, находящаяся в data-file.

exit 0

```

Файл data-file для [Пример 11-18](#), представленного выше, должен находиться в том же каталоге.

```

# Этот файл подключается к сценарию.
# Подключаемые файлы могут содержать объявления переменных, функций и т.п.
# Загружаться может командой 'source' или '.' .

# Инициализация некоторых переменных.

variable1=22
variable2=474
variable3=5
variable4=97

message1="Привет! Как поживаете?"
message2="Досвидания!"

print_message ()
{
# Вывод сообщения переданного в эту функцию.

  if [ -z "$1" ]
  then
    return 1
    # Ошибка, если аргумент отсутствует.
  fi

  echo

  until [ -z "$1" ]
  do
    # Цикл по всем аргументам функции.
    echo -n "$1"
    # Вывод аргумента с подавлением символа перевода строки.
    echo -n " "
    # Вставить пробел, для разделения выводимых аргументов.
    shift
    # Переход к следующему аргументу.
  done

  echo

  return 0
}

```

Сценарий может подключить даже самого себя, только этому едва ли можно найти какое либо практическое применение.

Пример 11-19. Пример (бесполезный) сценария, который подключает себя самого.

```
#!/bin/bash
# self-source.sh: сценарий, который рекурсивно подключает себя самого."
# Из "Бестолковые трюки", том II.

MAXPASSCNT=100 # Максимальное количество проходов.

echo -n "$pass_count "
# На первом проходе выведет два пробела,
#+ т.к. $pass_count еще не инициализирована.

let "pass_count += 1"
# Операция инкремента неинициализированной переменной $pass_count
#+ на первом проходе вполне допустима.
# Этот прием срабатывает в Bash и pdksh, но,
#+ при переносе сценария в другие командные оболочки,
#+ он может оказаться неработоспособным или даже опасным.
# Лучшим выходом из положения, будет присвоить переменной $pass_count
#+ значение 0, если она неинициализирована.

while [ "$pass_count" -le $MAXPASSCNT ]
do
. $0 # "Подключение" самого себя.
# ./$0 (истинная рекурсия) в данной ситуации не работает.
done

# Происходящее здесь фактически не является рекурсией как таковой,
#+ т.к. сценарий как бы "расширяет" себя самого
#+ (добавляя новый блок кода)
#+ на каждом проходе цикла 'while',
#+ командой 'source' в строке 22.
#
# Само собой разумеется, что первая строка (!), вновь подключенного сценария,
#+ интерпретируется как комментарий, а не как начало нового сценария (sha-bang)

echo

exit 0 # The net effect is counting from 1 to 100.
# Very impressive.

# Упражнение:
# -----
# Напишите сценарий, который использовал бы этот трюк для чего либо полезного.
```

exit

Безусловное завершение работы сценария. Команде **exit** можно передать целое число, которое будет возвращено вызывающему процессу как [код завершения](#). Вообще, считается хорошей практикой завершать работу сценария, за исключением простейших случаев, командой `exit 0`, чтобы проинформировать родительский процесс об успешном завершении.



Если сценарий завершается командой **exit** без аргументов, то в качестве кода завершения сценария принимается код завершения последней выполненной команды, не считая самой команды **exit**.

exec

Это встроенная команда интерпретатора shell, заменяет текущий процесс новым процессом, запускаемым командой `exec`. Обычно, когда командный интерпретатор встречает эту команду, то он [порождает](#) дочерний процесс, чтобы исполнить команду. При использовании встроенной команды **exec**, оболочка не порождает еще

один процесс, а заменяет текущий процесс другим. Для сценария это означает его завершение сразу после исполнения команды **ехес**. По этой причине, если вам встретится **ехес** в сценарии, то, скорее всего это будет последняя команда в сценарии.

Пример 11-20. Команда ехес

```
#!/bin/bash

ехес echo "Завершение \"$0\"." # Это завершение работы сценария.

# -----
# Следующие ниже строки никогда не будут исполнены
echo "Эта строка никогда не будет выведена на экран."

ехит 99 # Сценарий завершит работу не здесь.
        # Проверьте код завершения сценария
        #+ командой 'echo $?'.
        # Он точно не будет равен 99.
```

Пример 11-21. Сценарий, который запускает себя самого

```
#!/bin/bash
# self-ехес.sh

ехо

ехо "Эта строка в сценарии единственная, но она продолжает выводиться раз за разом."
ехо "PID остался равным $$."
# Демонстрация того, что команда ехес не порождает дочерний процесс.

ехо "===== Для завершения - нажмите Ctl-C ====="

sleep 1

ехес $0 # Запуск очередного экземпляра этого же сценария
        #+ который замещает предыдущий.

ехо "Эта строка никогда не будет выведена!" # Почему?

ехит 0
```

Команда **ехес** так же может использоваться для перенаправления. Так, команда **ехес <zzz - file** заменит стандартное устройство ввода (stdin) файлом zzz - file (см. [Пример 16-1](#)).



Ключ -ехес команды [find](#) -- это не то же самое, что встроенная команда **ехес**.

shopt

Эта команда позволяет изменять ключи (опции) оболочки на лету (см. [Пример 23-1](#) и [Пример 23-2](#)). Ее часто можно встретить в [стартовых файлах](#), но может использоваться и в обычных сценариях. Требуется Bash [версии 2](#) или выше.

```
shopt -s cdspell
# Исправляет незначительные орфографические ошибки в именах каталогов в команде 'cd'

cd /hрme # Oops! Имелось ввиду '/home'.
pwd      # /home
        # Shell исправил опечатку.
```

Команды

true

Команда возвращает код завершения -- ноль, или успешное завершение, и ничего больше.

```
# Бесконечный цикл
while true # вместо ":"
do
  operation-1
  operation-2
  ...
  operation-n
  # Следует предусмотреть способ завершения цикла.
done
```

false

Возвращает [код завершения](#), свидетельствующий о неудаче, и ничего более.

```
# Цикл, который никогда не будет исполнен
while false
do
  # Следующий код не будет исполнен никогда.
  operation-1
  operation-2
  ...
  operation-n
done
```

type [cmd]

Очень похожа на внешнюю команду [which](#), **type cmd** выводит полный путь к "cmd". В отличие от **which**, **type** является внутренней командой Bash. С опцией -a не только различает ключевые слова и внутренние команды, но и определяет местоположение внешних команд с именами, идентичными внутренним.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
[ is /usr/bin/[[
```

hash [cmds]

Запоминает путь к заданной команде (в хэш-таблице командной оболочки), благодаря чему, при повторном обращении к ней, оболочка или сценарий уже не будет искать путь к команде в \$PATH. При вызове команды **hash** без аргументов, просто выводит содержимое хэш-таблицы. С ключом -r -- очищает хэш-таблицу.

help

help COMMAND -- выводит краткую справку по использованию внутренней команды COMMAND. Аналог команды [whatis](#), только для внутренних команд.

```
bash$ help exit
exit: exit [n]
      Exit the shell with a status of N.  If N is omitted, the exit status
      is that of the last command executed.
```

11.1. Команды управления заданиями

Некоторые из нижеследующих команд принимают, в качестве аргумента, "идентификатор задания". См. [таблицу](#) в конце главы.

jobs

Выводит список заданий, исполняющихся в фоне. Команда **ps** более информативна.



Задания и процессы легко спутать. Некоторые [внутренние команды](#), такие как **kill**, **disown** и **wait** принимают в качестве параметра либо номер задания, либо номер процесса. Команды **fg**, **bg** и **jobs** принимают только номер задания.

```
bash$ sleep 100 &
[1] 1384
```

```
bash $ jobs
[1]+  Running                  sleep 100 &
```

"1" -- это номер задания (управление заданиями осуществляет текущий командный интерпретатор), а "1384" -- номер процесса (управление процессами осуществляется системой). Завершить задание/процесс ("прихлопнуть") можно либо командой **kill %1**, либо **kill 1384**.

Спасибо S.C.

disown

Удаляет задание из таблицы активных заданий командной оболочки.

fg, bg

Команда **fg** переводит задание из фона на передний план. Команда **bg** перезапускает приостановленное задание в фоновом режиме. Если эти команды были вызваны без указания номера задания, то они воздействуют на текущее исполняющееся задание.

wait

Останавливает работу сценария до тех пор пока не будут завершены все фоновые задания или пока не будет завершено задание/процесс с указанным номером задания/PID процесса. Возвращает [код завершения](#) указанного задания/процесса.

Вы можете использовать команду **wait** для предотвращения преждевременного завершения сценария до того, как завершит работу фоновое задание.

Пример 11-22. Ожидание завершения процесса перед тем как продолжить работу

```
#!/bin/bash

ROOT_UID=0 # Только пользователь с $UID = 0 имеет привилегии root.
E_NOTROOT=65
E_NOPARAMS=66

if [ "$UID" -ne "$ROOT_UID" ]
then
    echo "Для запуска этого сценария вы должны обладать привилегиями root."
    exit $E_NOTROOT
fi

if [ -z "$1" ]
then
    echo "Порядок использования: `basename $0` имя-файла"
    exit $E_NOPARAMS
fi

echo "Обновляется база данных 'locate'..."
echo "Это может занять продолжительное время."
updatedb /usr & # Должна запускаться с правами root.


wait
# В этом месте сценарий приостанавливает свою работу до тех пор, пока не
отработает 'updatedb'.
# Желательно обновить базу данных перед тем как выполнить поиск файла.

locate $1

# В худшем случае, без команды wait, сценарий завершил бы свою работу до того,
# как завершила бы работу утилита 'updatedb',
# сделав из нее "осиротевший" процесс.

exit 0
```

Команда **wait** может принимать необязательный параметр -- номер задания/процесса, например, **wait %1** или **wait \$PPID**. См. таблицу [идентификации заданий](#).

 При запуске команды в фоне из сценария может возникнуть ситуация, когда сценарий приостанавливает свою работу до тех пор, пока не будет нажата клавиша **ENTER**. Это, кажется, происходит с командами, делающими вывод на `stdout`. Такое поведение может вызывать раздражение у пользователя.

```
#!/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
-
```

Разместив команду **wait**, после запуска фонового задания, можно предотвратить такое поведение сценария.

```
#!/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 в файл или даже на устройство `/dev/null` также снимает эту проблему.

suspend

Действует аналогично нажатию на комбинацию клавиш **Control+Z**, за исключением того, что она приостанавливает работу командной оболочки.

logout

Завершает сеанс работы командной оболочки, можно указать необязательный [код завершения](#).

times

Выдает статистику исполнения команд в единицах системного времени, в следующем виде:

```
0m0.020s 0m0.020s
```

Имеет весьма ограниченную сферу применения, так как сценарии крайне редко подвергаются профилированию.

kill

Принудительное завершение процесса путем передачи ему соответствующего сигнала (см. [Пример 13-4](#)).

Пример 11-23. Сценарий, завершающий себя сам с помощью команды kill

```
#!/bin/bash
# self-destruct.sh

kill $$ # Сценарий завершает себя сам.
        # Надеюсь вы еще не забыли, что "$$" -- это PID сценария.


echo "Эта строка никогда не будет выведена."
# Вместо него на stdout будет выведено сообщение "Terminated".
```

```

exit 0


# Какой код завершения вернет сценарий?
#
# sh self-destruct.sh
# echo $?
# 143
#
# 143 = 128 + 15
#                               сигнал TERM

```

-  Команда `kill -l` выведет список всех [сигналов](#). Команда `kill -9 --` это "жесткий kill", она используется, как правило, для завершения зависших процессов, которые упорно отказываются "умирать", отвергая простой `kill`. Иногда достаточно подать команду `kill -15`. "Процессы-зомби", т.е. процессы, ["родители"](#) которых уже завершили работу, не могут быть "убиты" таким способом (невозможно "убить" "мертвого"), рано или поздно с ними "расправится" процесс `init`.

command

Директива **command COMMAND** запрещает использование псевдонимов и функций с именем "COMMAND".

-  Это одна из трех директив командного интерпретатора, которая влияет на обработку команд. Другие две -- [builtin](#) и [enable](#).

builtin

Конструкция **builtin BUILTIN_COMMAND** запускает [внутреннюю команду](#) "BUILTIN_COMMAND", на время запрещая использование функций и внешних системных команд с тем же именем.

enable

Либо запрещает, либо разрешает вызов внутренних команд. Например, **enable -n kill** запрещает использование внутренней команды [kill](#), в результате, когда интерпретатор встретит команду `kill`, то он вызовет внешнюю команду `kill`, т.е. `/bin/kill`.

Команда `enable -a` выведет список всех внутренних команд, указывая для каждой -- действительно ли она разрешена. Команда `enable -f filename` загрузит [внутренние команды](#) как разделяемую библиотеку (DLL) из указанного объектного файла. [\[26\]](#).

autoload

Перенесена в Bash из *ksh*. Если функция объявлена как **autoload**, то она будет загружена из внешнего файла в момент первого вызова. [\[27\]](#) Такой прием помогает экономить системные ресурсы.

Обратите внимание: **autoload** не является частью ядра Bash. Ее необходимо загрузить с помощью команды **enable -f** (см. выше).

Таблица 11-1. Идентификация заданий

Нотация	Описание
%N	Номер задания [N]
%S	Вызов (командная строка) задания, которая начинается со строки S
%?S	Вызов (командная строка) задания, которая содержит строку S
%%	"текущее" задание (последнее задание приостановленное на переднем плане или запущенное в фоне)
%+	"текущее" задание (последнее задание приостановленное на переднем плане или запущенное в фоне)
%-	Последнее задание
#!	Последний фоновый процесс

Глава 12. Внешние команды, программы и утилиты

Благодаря стандартизации набора команд UNIX-систем, сценарии, на языке командной оболочки, могут быть легко перенесены из системы в систему практически без изменений. Мощь сценариев складывается из наборов системных команд и директив командной оболочки с простыми программными конструкциями.

12.1. Базовые команды

Первая команда, с которой сталкиваются новички

ls

Команда вывода "списка" файлов. Многие недооценивают всю мощь этой скромной команды. Например, с ключом `-R`, рекурсивный обход дерева каталогов, командв **ls** выводит содержимое каталогов в виде древовидной структуры. Вот еще ряд любопытных ключей (опций) команды **ls**: `-S` -- сортировка по размеру файлов, `-t` -- сортировка по времени последней модификации файла и `-i` -- выводит список файлов с их inode (см. [Пример 12-3](#)).

Пример 12-1. Создание оглавления диска для записи CDR, с помощью команды ls

```
#!/bin/bash
# burn-cd.sh
# Сценарий, автоматизирующий процесс прожигания CDR.

SPEED=2           # Если ваше "железо" поддерживает более высокую скорость записи
-- можете увеличить этот параметр
IMAGEFILE=cddata.iso
CONTENTSFIL=contents
DEFAULTDIR=/opt  # В этом каталоге находятся файлы, которые будут записаны на
CD.

# Каталог должен существовать.

# Используется пакет "cdrrecord" от Joerg Schilling.
```

```

# (http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html)

# Если этот сценарий предполагается запускать с правами обычного пользователя,
#+ то необходимо установить флаг suid на cdrecord
#+ (chmod u+s /usr/bin/cdrecord, эта команда должна быть выполнена root-ом).

if [ -z "$1" ]
then
    IMAGE_DIRECTORY=$DEFAULTDIR
    # Каталог по-умолчанию, если иной каталог не задан из командной строки.
else
    IMAGE_DIRECTORY=$1
fi

# Создать файл "table of contents".
ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFIL
# Ключ "l" -- "расширенный" формат вывода списка файлов.
# Ключ "R" -- рекурсивный обход дерева каталогов.
# Ключ "F" -- добавляет дополнительные метки к именам файлов (к именам каталогов
добавляет окончательный символ /).
echo "Создано оглавление."

# Создать iso-образ.
mkisofs -r -o $IMAGFILE $IMAGE_DIRECTORY
echo "Создан iso-образ файловой системы ISO9660 ($IMAGFILE)."

```

cat, tac

cat -- это акроним от *concatenate*, выводит содержимое списка файлов на stdout. Для объединения файлов в один файл может использоваться в комбинации с операциями перенаправления (> или >>).

```
cat filename cat file.1 file.2 file.3 > file.123
```

Ключ **-n**, команды **cat**, вставляет порядковые номера строк в выходном файле. Ключ **-b** -- нумерует только не пустые строки. Ключ **-v** выводит непечатаемые символы в нотации с символом **^**. Ключ **-s** заменяет несколько пустых строк, идущих подряд, одной пустой строкой.

см. также [Пример 12-21](#) and [Пример 12-17](#).

tac -- выводит содержимое файлов в обратном порядке, от последней строки к первой.

rev

выводит все строки файла задом наперед на stdout. Это не то же самое, что **tac**. Команда **rev** сохраняет порядок следования строк, но переворачивает каждую строку задом наперед.

```
bash$ cat file1.txt
Это строка 1.
Это строка 2.
```


```
bash$ tac file1.txt
Это строка 2.
```


Это строка 1.

```
bash$ rev file1.txt
.1 акортс отЭ
.2 акортс отЭ
```


cp

Команда копирования файлов. `cp file1 file2` скопирует `file1` в `file2`, перезаписав `file2` если он уже существовал (см. [Пример 12-5](#)).

 С флагами `-a` и `-r`, или `-R` выполняет копирование дерева каталогов.

mv

Команда *перемещения* файла. Эквивалентна комбинации команд `cp` и `rm`. Может использоваться для перемещения большого количества файлов или для переименования каталогов. Примеры использования команды `mv` вы найдете в [Пример 9-17](#) и [Пример A-3](#).

 При использовании в неинтерактивных сценариях, команде `mv` следует передавать ключ `-f`, чтобы подавить запрос подтверждения на перемещение.

Если в качестве каталога назначения указан существующий каталог, то перемещаемый каталог становится подкаталогом каталога 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

Удаляет (remove) файл(ы). Ключ `-f` позволяет удалять даже файлы ТОЛЬКО-ДЛЯ-ЧТЕНИЯ и подавляет запрос подтверждения на удаление.

 С ключом `-r`, удаляет все файлы в подкаталогах.

rmdir

Удаляет каталог. Удаляемый каталог не должен содержать файлов, включая "скрытые файлы", [\[28\]](#) иначе каталог не будет удален.

mkdir

Создает новый каталог. `mkdir -p project/programs/December` создает каталог с заданным именем в требуемом каталоге. Ключ `-p` позволяет создавать промежуточные родительские каталоги.

chmod

Изменяет атрибуты существующего файла (см. [Пример 11-10](#)).

```
chmod +x filename
# Делает файл "filename" доступным для исполнения всем пользователям.
```

```
chmod u+s filename
# Устанавливается бит "suid" для "filename".
# В результате, любой пользователь сможет запустить "filename" с привилегиями
владельца файла.
# (Это не относится к файлам-сценариям на языке командной оболочки.)
```

```
chmod 644 filename
# Выдает право на запись/чтение владельцу файла "filename", и право на чтение
# всем остальным
# (восьмеричное число).
```

```
chmod 1777 directory-name
# Выдает право на чтение, запись и исполнение файлов в каталоге,
# дополнительно устанавливает "sticky bit".
# Это означает, что удалять файлы в этом каталоге могут только владельцы файлов,
# владелец каталога и, само собой разумеется, root.
```

chattr

Изменяет атрибуты файла. Эта команда подобна команде **chmod**, за исключением синтаксиса вызова, и работает исключительно в файловой системе *ext2*.

ln

Создает ссылку на существующий файл. Чаще всего используется с ключом `-s`, что означает символическую, или "мягкую" (*symbolic* или *"soft"*) ссылку. Позволяет задавать несколько имен одному и тому же файлу и превосходная альтернатива "псевдонимам" (алиасам) (см. [Пример 4-6](#)).

```
ln -s oldfile newfile
```

 создает ссылку, с именем `newfile`, на существующий файл `oldfile`, .

man, info

Команды доступа к справочным и информационным страницам по системным командам и установленным программам и утилитам. Как правило, страницы *info* содержат более подробную информацию, чем *man*.

12.2. Более сложные команды

Команды для более опытных пользователей

find

-exec *COMMAND* \;

Для каждого найденного файла, соответствующего заданному шаблону поиска, выполняет команду *COMMAND*. Командная строка должна завершаться последовательностью символов \; (здесь символ ";" экранирован обратным слэшем, чтобы информировать командную оболочку о том, что символ ";" должен быть передан команде **find** как обычный символ). Если *COMMAND* содержит {}, то **find** подставляет полное имя найденного файла вместо "{}".

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
/home/bozo/misc/irmeyc.txt
/home/bozo/test-scripts/1.txt
```

```
find /home/bozo/projects -mtime 1
# Найти все файлы в каталоге /home/bozo/projects и вложенных подкаталогах,
#+ которые изменялись в течение последних суток.
#
# mtime = время последнего изменения файла
# ctime = время последнего изменения атрибутов файла (через 'chmod' или как-то
иначе)
# atime = время последнего обращения к файлу
```

```
DIR=/home/bozo/junk_files
find "$DIR" -type f -atime +5 -exec rm {} \;
# Удалить все файлы в каталоге "/home/bozo/junk_files"
#+ к которым не было обращений в течение последних 5 дней.
#
# "-type filetype", где
# f = обычный файл
# d = каталог, и т.п.
# (Полный список ключей вы найдете в 'man find'.)
```

```
find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*' {} \;
```

```
# Поиск всех IP-адресов (xxx.xxx.xxx.xxx) в файлах каталога /etc.
# Однако эта команда выводит не только IP-адреса, как этого избежать?
```

```
# Примерно так:
```

```
find /etc -type f -exec cat '{}' \; | tr -c '[:digit:]' '\n' \
| grep '^^[^.]^*\.[^.]^*\.[^.]^*\.[^.]^*$'
# [:digit:] -- один из символьных классов
# введен в стандарт POSIX 1003.2.
```

```
# Спасибо S.C.
```



Не следует путать опцию -exec команды **find** с внутренней командой Bash -- [exec](#).

Пример 12-2. Badname, удаление файлов в текущем каталоге, имена которых содержат недопустимые символы и пробелы.

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

```

# Удаление файлов в текущем каталоге, чьи имена содержат недопустимые символы.

for filename in *
do
badname=`echo "$filename" | sed -n /[\+\{\;\\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
# Недопустимые символы в именах файлов:      + { ; " \ = ? ~ ( ) < > & * | $
rm $badname 2>/dev/null      # Сообщения об ошибках "выстреливаются" в никуда.
done

# Теперь "позаботимся" о файлах, чьи имена содержат пробельные символы.
find . -name "* *" -exec rm -f {} \;
# На место "{}", find подставит полное имя файла.
# Символ '\' указывает на то, что ';' интерпретируется как обычный символ, а не
как конец команды.

exit 0

#-----
# Строки, приведенные ниже, не будут выполнены, т.к. выше стоит команда "exit".

# Альтернативный вариант сценария:
find . -name '*[+{;"\\=\?~()<>&*|$ ]*' -exec rm -f '{}' \;
exit 0
# (Спасибо S.C.)

```

Пример 12-3. Удаление файла по его номеру *inode*

```

#!/bin/bash
# idelete.sh: Удаление файла по номеру inode.

# Этот прием используется в тех случаях, когда имя файла начинается с
недопустимого символа,
#+ например, ? или -.

ARGCOUNT=1                                # Имя файла должно быть передано в сценарий.
E_WRONGARGS=70
E_FILE_NOT_EXIST=71
E_CHANGED_MIND=72

if [ $# -ne "$ARGCOUNT" ]
then
    echo "Порядок использования: `basename $0` filename"
    exit $E_WRONGARGS
fi

if [ ! -e "$1" ]
then
    echo "Файл \"$1\" не найден."
    exit $E_FILE_NOT_EXIST
fi

inum=`ls -i | grep "$1" | awk '{print $1}'`
# inum = номер inode (index node) файла
# Каждый файл имеет свой inode, где хранится информация о физическом
расположении файла.

echo; echo -n "Вы совершенно уверены в том, что желаете удалить \"$1\" (y/n)? "
# Ключ '-v' в команде 'rm' тоже заставит команду вывести подобный запрос.
read answer
case "$answer" in
[nN]) echo "Передумали?"
    exit $E_CHANGED_MIND
;;
*)    echo "Удаление файла \"$1\".:";
esac

find . -inum $inum -exec rm {} \;
echo "Файл \"$1\" удален!"

```

```
exit 0
```

Дополнительные примеры по использованию команды **find** вы найдете в [Пример 12-22](#), [Пример 3-4](#) и [Пример 10-9](#). В страницах справочного руководства (`man find`) вы найдете более подробную информацию об этой достаточно сложной и мощной команде.

xargs

Команда передачи аргументов указанной команде. Она разбивает поток аргументов на отдельные составляющие и поочередно передает их заданной команде для обработки. Эта команда может рассматриваться как мощная замена обратным одиночным кавычкам. Зачастую, когда команды, заключенные в обратные одиночные кавычки, завершаются с ошибкой `too many arguments` (слишком много аргументов), использование **xargs** позволяет обойти это ограничение. Обычно, **xargs** считывает список аргументов со стандартного устройства ввода `stdin` или из канала (конвейера), но может считывать информацию и из файла.

Если команда не задана, то по-умолчанию выполняется [echo](#). При передаче аргументов по конвейеру, **xargs** допускает наличие пробельных символов и символов перевода строки, которые затем автоматически отбрасываются.

```
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
29 23:58 file2
```

`ls | xargs -p -l gzip` -- упакует с помощью [gzip](#) все файлы в текущем каталоге, выводя запрос на подтверждение для каждого файла.

- ❗ **xargs** имеет очень любопытный ключ `-n NN`, который ограничивает количество передаваемых аргументов за один "присест" числом *NN*.

```
ls | xargs -n 8 echo
```

 -- выведет список файлов текущего каталога в 8 колонок.

- ❗ Еще одна полезная опция `-- -0`, в комбинации с **find -print0** или **grep -lZ** позволяет обрабатывать аргументы, содержащие пробелы и кавычки.

```
find / -type f -print0 | xargs -0 grep -liwZ GUI |
xargs -0 rm -f
```

```
grep -rliwZ GUI / | xargs -0 rm -f
```

Обе вышеприведенные команды удалят все файлы, содержащие в своем имени комбинацию символов "GUI". (Спасибо S.C.)

Пример 12-4. Использование команды **xargs** для мониторинга системного журнала

```
#!/bin/bash

# Создание временного файла мониторинга в текущем каталоге,
# куда переписываются несколько последних строк из /var/log/messages.

# Обратите внимание: если сценарий запускается обычным пользователем,
# то файл /var/log/messages должен быть доступен на чтение этому пользователю.
# #root chmod 644 /var/log/messages

LINES=5

( date; uname -a ) >>logfile
# Время и информация о системе
echo -----
>>logfile
tail -$LINES /var/log/messages | xargs | fmt -s >>logfile
echo >>logfile
echo >>logfile

exit 0

# Упражнение:
# -----
# Измените сценарий таким образом, чтобы он мог отслеживать изменения в
# /var/log/messages
# с интервалом в 20 минут.
# Подсказка: воспользуйтесь командой "watch".
```

Пример 12-5. corydir, копирование файлов из текущего каталога в другое место, с помощью xargs

```
#!/bin/bash

# Копирует все файлы из текущего каталога
# в каталог, указанный в командной строке.

if [ -z "$1" ] # Выход, если каталог назначения не задан.
then
    echo "Порядок использования: `basename $0` directory-to-copy-to"
    exit 65
fi

ls . | xargs -i -t cp ./{} $1
# Этот сценария является точным эквивалентом
# cp * $1
# если в именах файлов не содержатся пробельные символы.

exit 0
```

expr

Универсальный обработчик выражений: вычисляет заданное выражение (аргументы должны отделяться пробелами). Выражения могут быть арифметическими, логическими или строковыми.

expr 3 + 5

возвратит 8

expr 5 % 3

возвратит 2

expr 5 * 3

возвратит 15

В арифметических выражениях, оператор умножения обязательно должен экранироваться обратным слэшем.

```
y=`expr $y + 1`
```

Операция инкремента переменной, то же самое, что и `let y=y+1`, или `y=$((y+1))`. Пример [подстановки арифметических выражений](#).

```
z=`expr substr $string $position $length`
```

Извлекает подстроку длиной `$length` символов, начиная с позиции `$position`.

Пример 12-6. Пример работы с `expr`

```
#!/bin/bash

# Демонстрация некоторых приемов работы с командой '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 "(инкремент переменной)"

a=`expr 5 % 3`
# остаток от деления (деление по модулю)
echo
echo "5 mod 3 = $a"

echo
echo

# Логические операции
# -----

# Возвращает 1 если выражение истинно, 0 -- если ложно,
#+ в противоположность соглашениям, принятым в Bash.

echo "Логические операции"
echo

x=24
y=25
b=`expr $x = $y`          # Сравнение.
echo "b = $b"            # 0 ( $x -ne $y )
echo

a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, поэтому...'
echo "Если a > 10, то b = 0 (ложь)"
echo "b = $b"            # 0 ( 3 ! -gt 10 )
echo
```

```

b=`expr $a \< 10`
echo "Если a < 10, то b = 1 (истина)"
echo "b = $b"           # 1 ( 3 -lt 10 )
echo
# Обратите внимание на необходимость экранирования операторов.

b=`expr $a \<= 3`
echo "Если a <= 3, то b = 1 (истина)"
echo "b = $b"           # 1 ( 3 -le 3 )
# Существует еще оператор "\>=" (больше или равно).

echo
echo

# Операции сравнения
# -----

echo "Операции сравнения"
echo
a=zipper
echo "a is $a"
if [ `expr $a = snap` ]
then
    echo "a -- это не zipper"
fi

echo
echo

# Операции со строками
# -----

echo "Операции со строками"
echo

a=1234zipper43231
echo "Строка над которой производятся операции: \"$a\"."

# length: длина строки
b=`expr length $a`
echo "длина строки \"$a\" равна $b."

# index: позиция первого символа подстроки в строке
b=`expr index $a 23`
echo "Позиция первого символа \"2\" в строке \"$a\" : \"$b\"."

# substr: извлечение подстроки, начиная с заданной позиции, указанной длины
b=`expr substr $a 2 6`
echo "Подстрока в строке \"$a\", начиная с позиции 2,\
и длиной в 6 символов: \"$b\"."

# При выполнении поиска по шаблону, по-умолчанию поиск
#+ начинается с ***начала*** строки.
#
#      Использование регулярных выражений
b=`expr match "$a" '[0-9]*'` # Подсчет количества цифр.
echo "Количество цифр с начала строки \"$a\" : $b."
b=`expr match "$a" '\([0-9]*\)\'` # Обратите внимание на экранирование
круглых скобок
#      ==      ==
echo "Цифры, стоящие в начале строки \"$a\" : \"$b\"."

echo

```



```
exit 0
```

! Вместо оператора **match** можно использовать оператор **:**. Например, команда **b=`expr \$a : [0-9]*`** является точным эквивалентом для **b=`expr match \$a [0-9]*`** в примере, рассмотренном выше.

```
#!/bin/bash

echo
echo "Операции над строками с использованием конструкции \"expr \$string : \"
"
echo
"=====
echo

a=1234zipper5FLIPPER43231

echo "Строка, над которой выполняются операции: \"`expr \"$a\" : \"\(.*\)\`\`\"."
#      Экранирование круглых скобок в шаблоне                == ==

# Если скобки не экранировать...
#+ то 'expr' преобразует строковый операнд в целое число.

echo "Длина строки \"\$a\" равна `expr \"$a\" : \".*\`\`\"      # Длина строки
echo "Количество цифр с начала строки \"\$a\" равно `expr \"$a\" : \"[0-9]*\`\`\"

# ----- #

echo

echo "Цифры, стоящие в начале строки \"\$a\" : `expr \"$a\" : \"\([0-9]*\)\`\`\"
#
echo "Первые 7 символов в строке \"\$a\" : `expr \"$a\" : \"\(.*\)\`\`\"
#      =====
# Опять же, необходимо экранировать круглые скобки в шаблоне.
#
echo "Последние 7 символов в строке \"\$a\" : `expr \"$a\" : \".*\(.*\)\`\`\"
#      ===== оператор конца строки      ^^
# (фактически означает переход через любое количество символов, пока
#+ не будет найдена требуемая подстрока)

echo

exit 0
```

Этот пример демонстрирует необходимость *экранирования оператора группировки -- \ (... \)* в [регулярных выражениях](#), при поиске по шаблону командой **expr**.

[Perl](#), [sed](#) и [awk](#) имеют в своем распоряжении более мощный аппарат анализа строк. Коротенький скрипт на **sed** или **awk**, внутри сценария (см. [Section 33.2](#)) -- значительно более привлекательная альтернатива использованию **expr** при анализе строк.

Дополнительные примеры, по обработке строк, вы найдете в [Section 9.2](#).

12.3. Команды для работы с датой и временем

Время/дата и измерение интервалов времени

date

Команда **date** без параметров выводит дату и время на стандартное устройство вывода stdout. Она становится гораздо интереснее при использовании дополнительных ключей форматирования вывода.

Пример 12-7. Команда date

```
#!/bin/bash
# Примеры использования команды 'date'

echo "Количество дней, прошедших с начала года: `date +%j`."
# Символ '+' обязателен при использовании форматирующего аргумента
# %j, возвращающего количество дней, прошедших с начала года.

echo "Количество секунд, прошедших с 01/01/1970 : `date +%s`."
# %s количество секунд, прошедших с начала "эпохи UNIX",
#+ но насколько этот ключ полезен?

prefix=temp
suffix=`eval date +%s` # Ключ "+%s" характерен для GNU-версии 'date'.
filename=$prefix.$suffix
echo $filename
# Прекрасный способ получения "уникального" имени для временного файла,
#+ даже лучше, чем с использованием $$

# Дополнительную информацию вы найдете в 'man date'.

exit 0
```

Ключ -u дает UTC время (Universal Coordinated Time -- время по Гринвичу).

```
bash$ date
Fri Mar 29 21:07:39 MST 2002
```

```
bash$ date -u
Sat Mar 30 04:07:42 UTC 2002
```

zdump

Отображает время для указанной временной зоны.

```
bash$ zdump EST
EST Tue Sep 18 22:09:22 2001 EST
```

time

Выводит подробную статистику по исполнению некоторой команды.

time ls -l / даст нечто подобное:

```
0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (149major+27minor)pagefaults 0swaps
```

См. так же очень похожую команду [times](#), обсуждавшуюся в предыдущем разделе.



Начиная с [версии 2.0](#) Bash, команда **time** стала зарезервированным словом интерпретатора, с несколько измененным поведением в конвейере.

touch

Утилита устанавливает время последнего обращения/изменения файла в текущее системное время или в заданное время, но так же может использоваться для создания нового пустого файла. Команда `touch zzz` создаст новый пустой файл с именем `zzz`, если перед этим файл `zzz` отсутствовал. Кроме того, такие пустые файлы могут использоваться для индикации, например, времени последнего изменения в проекте.



Эквивалентом команды **touch** могут служить : `>> newfile` или `>> newfile` (для обычных файлов).

at

Команда **at** -- используется для запуска заданий в заданное время. В общих чертах она напоминает [cron](#), однако, **at** используется для однократного запуска набора команд.

`at 2pm January 15` -- попросит ввести набор команд, которые необходимо запустить в указанное время. Эти команды должны быть совместимыми со сценариями командной оболочки. Ввод завершается нажатием комбинации клавиш [Ctl-D](#).

Ключ - `f` или операция перенаправления ввода (`<`), заставляет **at** прочитать список команд из файла. Этот файл должен представлять из себя обычный сценарий, на языке командной оболочки и, само собой разумеется, такой сценарий должен быть неинтерактивным. Может использоваться совместно с командой [run-parts](#) для запуска различных наборов сценариев.

```
bash$ at 2:30 am Friday < at-jobs.list
job 2 at 2000-10-27 02:30
```

batch

Команда **batch**, управляющая запуском заданий, напоминает команду **at**, но запускает список команд только тогда, когда загруженность системы упадет ниже `.8`. Подобно команде **at**, с ключом - `f`, может считывать набор команд из файла.

cal

Выводит на `stdout` аккуратно отформатированный календарь на текущий месяц. Может выводить календарь за определенный год.

sleep

Приостанавливает исполнение сценария на заданное количество секунд, ничего не делая. Может использоваться для синхронизации процессов, запущенных в фоне,

проверяя наступление ожидаемого события так часто, как это необходимо. Например, [Пример 29-6](#).

```
sleep 3
# Пауза, длительностью в 3 секунды.
```



Команда **sleep** по-умолчанию принимает количество секунд, но ей можно передать и количество часов и минут и даже дней.

```
sleep 3 h
# Приостановка на 3 часа!
```



Для запуска команд через заданные интервалы времени лучше использовать [watch](#).

usleep

Microsleep (здесь символ "u" должен читаться как буква греческого алфавита -- "мю", или префикс микро). Это то же самое, что и **sleep**, только интервал времени задается в микросекундах. Может использоваться для очень тонкой синхронизации процессов.

```
usleep 30
# Приостановка на 30 микросекунд.
```

Эта команда является частью пакета *initscripts/rc-scripts* в дистрибутиве Red Hat.



Команда **usleep** не обеспечивает особую точность соблюдения интервалов, и поэтому она не подходит для применений, критичных ко времени.

hwclock, clock

Команда **hwclock** используется для получения доступа или коррекции аппаратных часов компьютера. С некоторыми ключами требует наличия привилегий **root**. Сенарий `/etc/rc.d/rc.sysinit` использует команду **hwclock** для установки системного времени во время загрузки.

Команда **clock** -- это синоним команды **hwclock**.

12.4. Команды обработки текста

sort

Сортирует содержимое файла, часто используется как промежуточный фильтр в конвейерах. Эта команда сортирует поток текста в порядке убывания или возрастания, в зависимости от заданных опций. Ключ `-m` используется для сортировки и объединения входных файлов. В *странице info* перечислено большое

количество возможных вариантов ключей. См. [Пример 10-9](#), [Пример 10-10](#) и [Пример А-9](#).

tsort

Топологическая сортировка, считывает пары строк, разделенных пробельными символами, и выполняет сортировку, в зависимости от заданного шаблона.

uniq

Удаляет повторяющиеся строки из отсортированного файла. Эту команду часто можно встретить в конвейере с командой [sort](#).

```
cat list-1 list-2 list-3 | sort | uniq > final.list
# Содержимое файлов,
# сортируется,
# затем удаляются повторяющиеся строки,
# и результат записывается в выходной файл.
```

Ключ `-c` выводит количество повторяющихся строк.

```
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 *статистику встречаемости* строк в файле INPUTFILE (ключ `-nr`, в команде `sort`, означает сортировку в порядке убывания). Этот шаблон может с успехом использоваться при анализе файлов системного журнала, словарей и везде, где необходимо проанализировать лексическую структуру документа.

Пример 12-8. Частота встречаемости отдельных слов

```
#!/bin/bash
# wf.sh: "Сырой" анализ частоты встречаемости слова в текстовом файле.
```

```
ARGS=1
E_BADARGS=65
E_NOFILE=66
```

```

if [ $# -ne "$ARGS" ] # Файл для анализа задан?
then
    echo "Порядок использования: `basename $0` filename"
    exit $E_BADARGS
fi

if [ ! -f "$1" ] # Проверка существования файла.
then
    echo "Файл \"$1\" не найден."
    exit $E_NOFILE
fi

```

```

#####
# main ()
sed -e 's/\.//g' -e 's/ /\
/g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
#
# Подсчет количества вхождений

# Точки и пробелы заменяются
#+ символами перевода строки,
#+ затем символы переводятся в нижний регистр
#+ и наконец подсчитывается количество вхождений,
#+ и выполняется сортировка по числу вхождений.
#####

# Упражнения:
# -----
# 1) Добавьте команду 'sed' для отсеечения других знаков пунктуации, например,
запятых.
# 2) Добавьте удаление лишних пробелов и других пробельных символов.
# 3) Добавьте дополнительную сортировку так, чтобы слова с одинаковой частотой
встречаемости
#+ сортировались бы в алфавитном порядке.

exit 0

```

```

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 встречается
6 строка
3 трижды
2 дважды
1 только
1 один
1 раз

```

expand, unexpand

Команда **expand** преобразует символы табуляции в пробелы. Часто используется в конвейерной обработке текста.

Команда **unexpand** преобразует пробелы в символы табуляции. Т.е. она является обратной по отношению к команде **expand**.

cut

Предназначена для извлечения отдельных полей из текстовых файлов. Напоминает команду `print $N` в [awk](#), но более ограничена в своих возможностях. В простейших случаях может быть неплохой заменой **awk** в сценариях. Особую значимость, для команды **cut**, представляют ключи `-d` (разделитель полей) и `-f` (номер(а) поля(ей)).

Использование команды **cut** для получения списка смонтированных файловых систем:

```
cat /etc/mtab | cut -d ' ' -f1,2
```

Использование команды **cut** для получения версии ОС и ядра:

```
uname -a | cut -d" " -f1,3,11,12
```

Использование команды **cut** для извлечения заголовков сообщений из электронных писем:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
MAKE MILLIONS WORKING AT HOMEЗ
Spam complaint
Re: Spam complaint
```

Использование команды **cut** при разборе текстового файла:

```
# Список пользователей в /etc/passwd.

FILENAME=/etc/passwd

for user in $(cut -d: -f1 $FILENAME)
do
    echo $user
done

# Спасибо Oleg Philon за этот пример.
```

```
cut -d ' ' -f2,3 filename эквивалентно awk -F'[ ]' '{ print $2, $3 }' filename
```

См. также [Пример 12-33](#).

paste

Используется для объединения нескольких файлов в один многоколоночный файл.

join

Может рассматриваться как команда, родственная команде **paste**. Эта мощная утилита позволяет объединять два файла по общему полю, что представляет собой упрощенную версию реляционной базы данных.

Команда **join** оперирует только двумя файлами и объединяет только те строки, которые имеют общее поле (обычно числовое), результат объединения выводится на `stdout`. Объединяемые файлы должны быть отсортированы по ключевому полю.

```
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
```



На выходе ключевое поле встречается только один раз.

head

Выводит начальные строки из файла на `stdout` (по-умолчанию -- 10 строк, но это число можно задать иным). Эта команда имеет ряд интересных ключей.

Пример 12-9. Какие из файлов являются сценариями?

```
#!/bin/bash
# script-detector.sh: Отыскивает файлы сценариев в каталоге.

TESTCHARS=2 # Проверяются первые два символа.
SHABANG='#!' # Сценарии как правило начинаются с "sha-bang."

for file in * # Обход всех файлов в каталоге.
do
  if [[ `head -c$TESTCHARS "$file"` = "$SHABANG" ]]
  # head -c2 #!
  # Ключ '-с' в команде "head" выводит заданное
  #+ количество символов, а не строк.
  then
    echo "Файл \"$file\" -- сценарий."
  else
    echo "Файл \"$file\" не является сценарием."
  fi
done
```


done

exit 0

Пример 12-10. Генератор 10-значных случайных чисел

```
#!/bin/bash
# rnd.sh: Генератор 10-значных случайных чисел

# Автор: Stephane Chazelas.

head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.*/p'

# ===== #

# Описание
# -----

# head:
# -c4 -- первые 4 байта.

# od:
# -N4 ограничивает вывод 4-мя байтами.
# -tu4 беззнаковый десятичный формат вывода.

# sed:
# -n, в комбинации с флагом "p", в команде "s",
# выводит только совпадающие с шаблоном строки.

# Автор сценария описывает действия 'sed' таким образом:

# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.*/p'
# -----> |

# Передает вывод в "sed" -----> |
# пусть это будет 00000000 1198195154\n

# sed начинает читать символы: 00000000 1198195154\n.
# Здесь он находит символ перевода строки,
# таким образом он получает строку (00000000 1198195154).
# Затем он просматривает <диапазон><действие>. Первый и единственный -- это

# диапазон действие
# 1 s/.*/p

# Номер строки попадает в заданный диапазон, так что теперь он приступает к
выполнению действия:
# пытается заменить наибольшую подстроку, заканчивающуюся пробелом
# ("00000000 ") "ничем" (//), и если замена произведена -- выводит результат
# ("p" -- это флаг команды "s", а не команда "p", которая имеет иное значение).

# теперь sed готов продолжить чтение входного потока. (Обратите внимание:
# если опустить ключ -n, то sed выведет строку еще раз)

# Теперь sed дочитывает остаток строки.
# Он готов приступить к анализу 2-й строки (которая отмечена '$'
# как последняя).
# Поскольку строка не попадает в заданный <диапазон>, на этом обработка
прекращается.

# Проще говоря, команда sed означает:
# "В первой строке удалить любые символы, вплоть до последнего встреченного
пробела,
# и затем вывести остаток."

# Сделать это можно более простым способом:
```

```
#          sed -e 's/. * //;q'

# Где, заданы два <диапазона><действия> (можно записать и по другому
#          sed -e 's/. * //' -e q):

#   диапазон                действие
#   ничего (для совпадающих строк)  s/. * //
#   ничего (для совпадающих строк)  q (quit)

# Здесь sed считывает только первую строку.
# Выполняет оба действия, и выводит строку перед завершением
# (действие "q"), поскольку ключ "-n" опущен.

# ===== #

# Простая альтернатива:
#          head -c4 /dev/urandom| od -An -tu4

exit 0
```

См. также [Пример 12-30](#).

tail

Выводит последние строки из файла на stdout (по-умолчанию -- 10 строк). Обычно используется для мониторинга системных журналов. Ключ -f, позволяет вести непрерывное наблюдение за добавляемыми строками в файл.

Пример 12-11. Мониторинг системного журнала с помощью tail

```
#!/bin/bash

filename=sys.log

cat /dev/null > $filename; echo "Создание / очистка временного файла."
# Если файл отсутствует, то он создается,
#+ и очищается, если существует.
# : > filename и > filename дают тот же эффект.

tail /var/log/messages > $filename
# Файл /var/log/messages должен быть доступен для чтения.

echo "В файл $filename записаны последние строки из /var/log/messages."

exit 0
```

См. также [Пример 12-4](#), [Пример 12-30](#) и [Пример 29-6](#).

grep

Многоцелевая поисковая утилита, использующая [регулярные выражения](#). Изначально это была команда в древнем строчном редакторе **ed**, **g / r e / p**, что означает -- *global - regular expression - print*.

```
grep pattern [file...]
```

Поиск участков текста в файле(ах), соответствующих шаблону *pattern*, где *pattern* может быть как обычной строкой, так и регулярным выражением.

```
bash$ grep '[rst]system.$' osinfo.txt
The GPL governs the distribution of the Linux operating system.
```

Если файл(ы) для поиска не задан, то команда **grep** работает как фильтр для устройства stdout, например в [конвейере](#).

```
bash$ ps ax | grep clock
765 tty1      S          0:00 xclock
 901 pts/1    S          0:00 grep clock
```

-i -- выполняется поиск без учета регистра символов.

-w -- поиск совпадений целого слова.

-l -- вывод только имен файлов, в которых найдены участки, совпадающие с заданным образцом/шаблоном, без вывода совпадающих строк.

-r -- (рекурсивный поиск) поиск выполняется в текущем каталоге и всех вложенных подкаталогах.

The -n option lists the matching lines, together with line numbers.

```
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.
```

-v (или --invert-match) -- выводит только строки, не содержащие совпадений.

```
grep pattern1 *.txt | grep -v pattern2
```

```
# Выводятся строки из "*.txt", совпадающие с "pattern1",
# но ***не*** совпадающие с "pattern2".
```

-c (--count) -- выводит количество совпадений без вывода самих совпадений.


```
grep -c txt *.sgml # (количество совпадений с "txt" в "*.sgml" файлах)
```

```
# grep -cz .
#           ^ точка
# означает подсчет (-c) непустых (". " -- содержащих хотя бы один символ)
# элементов,
# разделенных нулевыми байтами (-z)
#
printf 'a b\nc d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz . # 4
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, в качестве разделителя, принимается символ перевода строки (\n).
# Обратите внимание: ключ -z характерен для GNU-версии "grep".

# Спасибо S.C.
```

Если **grep** вызывается для поиска по группе файлов, то вывод будет содержать указание на имена файлов, в которых найдены 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.
```

 Для того, чтобы заставить **grep** выводить имя файла, когда поиск производится по одному-единственному файлу, достаточно указать устройство `/dev/null` в качестве второго файла.

```
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 было найдено, то **grep** возвращает [код завершения](#) `-- 0`, это может оказаться полезным при выполнении поиска в условных операторах (в таких случаях особый интерес может представлять ключ `-q`, который подавляет вывод).

```
SUCCESS=0                # если найдено совпадение
word=Linux
filename=data.file

grep -q "$word" "$filename" # "-q" -- подавляет вывод на stdout.

if [ $? -eq $SUCCESS ]
then
    echo "Образец $word найден в $filename"
else
    echo "Образец $word в файле $filename не найден"
fi
```

[Пример 29-6](#) -- пример поиска заданного образца в системном журнале, с помощью **grep**.

Пример 12-12. Сценарий-эмулятор "grep"

```
#!/bin/bash
# grp.sh: Очень "грубая" реализация 'grep'.

E_BADARGS=65

if [ -z "$1" ]      # Проверка наличия аргументов.
then
    echo "Порядок использования: `basename $0` pattern"
    exit $E_BADARGS
fi

echo

for file in *      # Обход всех файлов в $PWD.
do
```

```

output=$(sed -n /"$1"/p $file) # Подстановка команд.

if [ ! -z "$output" ]          # Что произойдет, если кавычки вокруг
"$output" убрать?
then
    echo -n "$file: "
    echo $output
fi                               # эквивалент: sed -ne "/$1/s|^|${file}: |p"

echo
done

echo

exit 0

# Упражнения:
# -----
# 1) Добавьте вывод символов перевода строки, если найдено более одного
совпадения в любом из файлов.
# 2) Добавьте обработку различных ключей.

```



egrep -- то же самое, что и **grep -E**. Эта команда использует несколько отличающийся, расширенный набор [регулярных выражений](#), что позволяет выполнять поиск более гибко.

fgrep -- то же самое, что и **grep -F**. Эта команда выполняет поиск строк символов (не регулярных выражений), что несколько увеличивает скорость поиска.

Утилита **agrep** имеет более широкие возможности поиска приблизительных совпадений. Образец поиска может отличаться от найденной строки на указанное число символов.



Для поиска по сжатым файлам следует использовать утилиты **zgrep**, **zegrep** или **zfgrep**. Они с успехом могут использоваться и для не сжатых файлов, но в этом случае они уступают в скорости обычным **grep**, **egrep** и **fgrep**. Они очень удобны при выполнении поиска по смешенному набору файлов -- когда одни файлы сжаты, а другие нет.

Для поиска по [bzip](#)-файлам используйте **bzgrep**.

look

Команда **look** очень похожа на **grep**, и предназначена для поиска по "словарям" -- отсортированным файлам. По-умолчанию, поиск выполняется в файле `/usr/dict/words`, но может быть указан и другой словарь.

Пример 12-13. Поиск слов в словаре

```

#!/bin/bash
# lookup: Выполняется поиск каждого слова из файла в словаре.

file=words.data # Файл с искомыми словами.

echo

while [ "$word" != end ] # Последнее слово в файле.
do
    read word           # Из файла, потому, что выполнено перенаправление в конце
цикла.
    look $word > /dev/null # Подавление вывода строк из словаря.

```

```

lookup=$?      # Код возврата команды 'look'.

if [ "$lookup" -eq 0 ]
then
    echo "Слово \"$word\" найдено."
else
    echo "Слово \"$word\" не найдено."
fi

done <"$file"  # Перенаправление ввода из файла $file, так что "чтение"
производится оттуда.

echo

exit 0

# -----
# Строки, расположенные ниже не будут исполнены, поскольку выше стоит команда
"exit".

# Stephane Chazelas предложил более короткий вариант:

while read word && [[ $word != end ]]
do if look "$word" > /dev/null
    then echo "Слово \"$word\" найдено."
    else echo "Слово \"$word\" не найдено."
    fi
done <"$file"

exit 0

```

sed, awk

Скриптовые языки, специально разработанные для анализа текстовых данных.

sed

Неинтерактивный "поточковый редактор". Широко используется в сценариях на языке командной оболочки.

awk

Утилита контекстного поиска и преобразования текста, замечательный инструмент для извлечения и/или обработки полей (колонок) в структурированных текстовых файлах. Синтаксис awk напоминает язык С.

wc

wc -- "word count", счетчик слов в файле или в потоке:

```

bash $ wc /usr/doc/sed-3.02/README
20      127      838 /usr/doc/sed-3.02/README
[20 строк  127 слов  838 символов]

```

wc -w подсчитывает только слова.

wc -l подсчитывает только строки.

wc -c подсчитывает только символы.

`wc -L` возвращает длину наибольшей строки.

Подсчет количества `.txt`-файлов в текущем каталоге с помощью `wc`:

```
$ ls *.txt | wc -l
# Эта команда будет работать, если ни в одном из имен файлов "*.txt" нет символа перевода строки.

# Альтернативный вариант:
#   find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#   (shopt -s nullglob; set -- *.txt; echo $#)

# Спасибо S.C.
```

Подсчет общего размера файлов, чьи имена начинаются с символов, в диапазоне `d - h`

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

От переводчика: в случае, если у вас локаль отлична от "C", то вышеприведенная команда может не дать результата, поскольку `wc` вернет не слово "total", в конце вывода, а "итого". Тогда можно попробовать несколько измененный вариант:

```
bash$ wc [d-h]* | grep итого | awk '{print $3}'
71832
```

Использование `wc` для подсчета количества вхождений слова "Linux" в основной исходный файл с текстом этого руководства.

```
bash$ grep Linux abs-book.sgm1 | wc -l
50
```

См. также [Пример 12-30](#) и [Пример 16-7](#).

Отдельные команды располагают функциональностью `wc` в виде своих ключей.

```
... | grep foo | wc -l
# Часто встречающаяся конструкция, которая может быть сокращена.

... | grep -c foo
# Ключ "-c" ("--count") команды grep.

# Спасибо S.C.
```

tr

Замена одних символов на другие.



В отдельных случаях [символы необходимо заключать в кавычки и/или квадратные скобки](#). Кавычки предотвращают интерпретацию специальных символов командной оболочкой. Квадратные скобки должны заключаться в кавычки.

Команда `tr "A-Z" "*" <filename` или `tr A-Z * <filename` заменяет все символы верхнего регистра в `filename` на звездочки (вывод производится на `stdout`). В некоторых системах этот вариант может оказаться неработоспособным, тогда попробуйте `tr A-Z ' [**] '`.

Ключ `-d` удаляет символы из заданного диапазона.

```
echo "abcdef"           # abcdef
echo "abcdef" | tr -d b-d # aef
```

```
tr -d 0-9 <filename
# Удалит все цифровые символы из файла "filename".
```

Ключ `--squeeze-repeats (-s)` удалит все повторяющиеся последовательности символов. Может использоваться для удаления лишних [пробельных символов](#).

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

Ключ `-c "complement"` *заменит* символы в соответствии с шаблоном. Этот ключ воздействует только на те символы, которые НЕ соответствуют заданному шаблону.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

Обратите внимание: команда `tr` корректно распознает [символьные классы POSIX](#). [\[29\]](#)

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

Пример 12-14. `tourper`: Преобразование символов в верхний регистр.

```
#!/bin/bash
# Преобразование символов в верхний регистр.

E_BADARGS=65

if [ -z "$1" ] # Стандартная проверка командной строки.
then
    echo "Порядок использования: `basename $0` filename"
    exit $E_BADARGS
fi

tr a-z A-Z <"$1"
```



```
# Тот же эффект можно получить при использовании символьных классов POSIX:
#   tr '[:lower:]' '[:upper:]' <"$1"
# Спасибо S.C.
```

```
exit 0
```

Пример 12-15. lowercase: Изменение имен всех файлов в текущем каталоге в нижний регистр.

```
#!/bin/bash
#
# Изменит все имена файлов в текущем каталоге в нижний регистр.
#

for filename in *           # Обход всех файлов в каталоге.
do
    fname=`basename $filename`
    n=`echo $fname | tr A-Z a-z` # Перевести символы в нижний регистр.
    if [ "$fname" != "$n" ]     # Переименовать только те файлы, имена которых
изменились.
    then
        mv $fname $n
    fi
done

exit 0
```

```
# Сроки приведенные ниже не будут исполняться, поскольку выше стоит команда
"exit".
```

```
#-----#
# Запустите эту часть сценария, удалив строки , стоящие выше.
```

```
# Сценарий, приведенный выше, не работает с именами файлов, содержащими пробелы
или символы перевода строки.
```

```
# В связи с этим, Stephane Chazelas предложил следующий вариант:
```

```
for filename in * # Нет необходимости использовать basename,
                  # поскольку "*" возвращает имена, не содержащие "/".
do n=`echo "$filename/" | tr '[:upper:]' '[:lower:]'`
#
#                               символьные классы POSIX.
#                               Завершающий слэш добавлен для того, чтобы символ перевода
строки
#                               не был удален при подстановке команды.
# Подстановка переменной:
n=${n%/} # Удаление завершающего слэша, добавленного выше.
[[ $filename == $n ]] || mv "$filename" "$n"
# Проверка -- действительно ли изменилось имя файла.
done

exit 0
```

Пример 12-16. du: Преобразование текстового файла из формата DOS в формат UNIX.

```
#!/bin/bash
# du.sh: Преобразование текстового файла из формата DOS в формат UNIX.

E_WRONGARGS=65

if [ -z "$1" ]
then
    echo "Порядок использования: `basename $0` filename-to-convert"
    exit $E_WRONGARGS
```

```

fi

NEWFILENAME=$1.unx

CR='\015' # Возврат каретки.
# Строки в текстовых файлах DOS завершаются комбинацией символов CR-LF.

tr -d $CR < $1 > $NEWFILENAME
# Удалить символы CR и записать в новый файл.

echo "Исходный текстовый файл: \"$1\"."
echo "Преобразованный файл: \"$NEWFILENAME\"."

exit 0

```

Пример 12-17. rot13: Сверхслабое шифрование по алгоритму rot13.

```

#!/bin/bash
# rot13.sh: Классический алгоритм шифрования rot13,
#           который способен "расколоть" даже 3-х летний ребенок.

# Порядок использования: ./rot13.sh filename
# или                      ./rot13.sh <filename
# или                      ./rot13.sh и ввести текст с клавиатуры (stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M' # "a" заменяется на "n", "b" на "o", и
т.д.
# Конструкция 'cat "$@"'
#+ позволяет вводить данные как со stdin, так и из файла.

exit 0

```

Пример 12-18. Более "сложный" шифр

```

#!/bin/bash
# crypto-quote.sh: Ограниченное шифрование

# Шифрование ограничивается простой заменой одних алфавитных символов другими.
# Результат очень похож на шифры-загадки

key=ETAOINSHRDLUBCFGJMQPVWZYXK
# Здесь, "key" -- ни что иное, как "перемешанный" алфавит.
# Изменение ключа "key" приведет к изменению шифра.

# Конструкция 'cat "$@"' позволяет вводить данные как со stdin, так и из файла.
# Если используется stdin, то ввод должен завершаться комбинацией Control-D.
# Иначе, в командной строке, сценарию должно быть передано имя файла.

cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
#           | в верхний регистр | шифрование
# Такой прием позволяет шифровать как символы в верхнем регистре, так и в
нижнем.
# Неалфавитные символы остаются без изменений.

# Попробуйте зашифровать какой либо текст, например
# "Nothing so needs reforming as other people's habits."
# --Mark Twain
#
# Результат будет:
# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
# --BEML PZERC

# Для дешифрации можно использовать следующую комбинацию:
# cat "$@" | tr "$key" "A-Z"

```

```
# Этот нехитрый шифр может быть "взломан" 12-ти летним ребенком
#+ с помощью карандаша и бумаги.
```

```
exit 0
```

Различные версии tr

Утилита **tr** имеет две, исторически сложившиеся, версии. BSD-версия не использует квадратные скобки (`tr a-z A-Z`), в то время как SysV-версия использует их (`tr '[a-z]' '[A-Z]'`). GNU-версия утилиты **tr** напоминает версию BSD, но диапазоны символов обязательно должны заключаться в квадратные скобки.

fold

Выравнивает текст по ширине, разрывая, если это необходимо, слова. Особый интерес представляет ключ `-s`, который производит перенос строк по пробелам, стараясь не разрывать слова. (см. [Пример 12-19](#) и [Пример A-2](#)).

fmt

Очень простая утилита форматирования текста, чаще всего используемая как фильтр в конвейерах для того, чтобы выполнить "перенос" длинных строк текста.

Пример 12-19. Отформатированный список файлов.

```
#!/bin/bash

WIDTH=40                # 40 символов в строке.

b=`ls /usr/local/bin`   # Получить список файлов...

echo $b | fmt -w $WIDTH

# То же самое можно выполнить командой
# echo $b | fold - -s -w $WIDTH

exit 0
```

См. также [Пример 12-4](#).



Очень мощной альтернативой утилите **fmt**, является утилита **par** (автор Kamil Toman), которую вы сможете найти на <http://www.cs.berkeley.edu/~amc/Par/>.

col

Эта утилита с обманчивым названием удаляет из входного потока символы обратной подачи бумаги (код ESC 7). Она так же пытается заменить пробелы на табуляции. Основная область применения утилиты **col** -- фильтрация вывода отдельных утилит обработки текста, таких как **groff** и **tbl**.

column

Форматирование по столбцам. Эта утилита преобразует текст, например какой либо список, в табличное, более "удобочитаемое", представление, вставляя символы табуляции по мере необходимости.

Пример 12-20. Пример форматирования списка файлов в каталоге

```
#!/bin/bash
# За основу сценария взят пример "man column".

(printf "PERMISSIONS LINKS OWNER GROUP SIZE DATE TIME PROG-NAME\n" \
; ls -l | sed 1d) | column -t

# Команда "sed 1d" удаляет первую строку, выводимую командой ls,
#+ (для локали "C" это строка: "total          N",
#+ где "N" -- общее количество файлов.

# Ключ -t, команды "column", означает "табличное" представление.

exit 0
```

colrm

Утилита удаления колонок. Удаляет колонки (столбцы) символов из файла и выводит результат на stdout. `colrm 2 4 <filename` -- удалит символы со 2-го по 4-й включительно, в каждой строке в файле `filename`.



Если файл содержит символы табуляции или непечатаемые символы, то результат может получиться самым неожиданным. В таких случаях, как правило, утилиту `colrm`, в конвейере, окружают командами [expand](#) и `unexpand`.

nl

Нумерует строки в файле. `nl filename` -- выведет файл `filename` на stdout, и в начале каждой строки вставит ее порядковый номер, счет начинается с первой непустой строки. Если файл не указывается, то принимается ввод со stdin.

Вывод команды `nl` очень напоминает `cat -n`, однако, по-умолчанию `nl` не нумерует пустые строки.

Пример 12-21. nl: Самонумерующийся сценарий.

```
#!/bin/bash

# Сценарий выводит себя сам на stdout дважды, нумеруя строки сценария.

# 'nl' вставит для этой строки номер 3, поскольку она не нумерует пустые строки.
# 'cat -n' вставит для этой строки номер 5.

nl `basename $0`

echo; echo # А теперь попробуем вывести текст сценария с помощью 'cat -n'

cat -n `basename $0`
# Различия состоят в том, что 'cat -n' нумерует все строки.
# Обратите внимание: 'nl -ba' -- сделает то же самое.

exit 0
```

pr

Подготовка файла к печати. Утилита производит разбивку файла на страницы, приводя его в вид пригодный для печати или для вывода на экран. Разнообразные ключи позволяют выполнять различные манипуляции над строками и колонками,

соединять строки, устанавливать поля, нумеровать строки, добавлять колонтитулы и многое, многое другое. Утилита **pr** соединяет в себе функциональность таких команд, как **nl**, **paste**, **fold**, **column** и **expand**.

```
pr -o 5 --width=65 fileZZZ | more
```

 -- выдаст хорошо оформленное и разбитое на страницы содержимое файла `fileZZZ`.

Хочу особо отметить ключ `-d`, который выводит строки с двойным интервалом (тот же эффект, что и **sed -G**).

gettext

GNU утилита, предназначена для нужд [локализации](#) и перевода сообщений программ, выводимых на экран, на язык пользователя. Не смотря на то, что это актуально, прежде всего, для программ на языке C, тем не менее **gettext** с успехом может использоваться в сценариях командной оболочки для тех же целей. См. *info page*.

iconv

Утилита преобразования текста из одной кодировки в другую. В основном используется для нужд локализации.

recode

Может рассматриваться как разновидность утилиты **iconv**, описанной выше. Универсальная утилита для преобразования текстовой информации в различные кодировки.

TeX, gs

TeX и **Postscript** -- языки разметки текста, используемые для подготовки текста к печати или выводу на экран.

TeX -- это сложная система подготовки к печати, разработанная Дональдом Кнудом (Donald Knuth). Эту утилиту удобнее использовать внутри сценария, чем в командной строке, поскольку в сценарии проще один раз записать все необходимые параметры, передаваемые утилите, для получения необходимого результата.

Ghostscript (**gs**) -- это GPL-версия интерпретатора Postscript.

groff, tbl, eqn

groff -- это еще один язык разметки текста и форматированного вывода. Является расширенной GNU-версией пакета **roff/troff** в UNIX-системах.

tbl -- утилита обработки таблиц, должна рассматриваться как составная часть **groff**, так как ее задачей является преобразование таблиц в команды **groff**.

eqn -- утилита преобразования математических выражений в команды **groff**.

lex, yacc

lex -- утилита лексического разбора текста. В Linux-системах заменена на свободно распространяемую утилиту **flex**.

yacc -- утилита для создания синтаксических анализаторов, на основе набора грамматик, задаваемых разработчиком. В Linux-системах, эта утилита заменена на свободно распространяемую утилиту **bison**.


12.5. Команды для работы с файлами и архивами

Архивация

tar


Стандартная, для UNIX, утилита архивирования. Первоначально -- это была программа *Tape ARchiving*, которая впоследствии переросла в универсальный пакет, который может работать с любыми типами устройств (см. [Пример 3-4](#)). В GNU-версию tar была добавлена возможность одновременно производить сжатие tar-архива, например команда **tar czvf archive_name.tar.gz *** создает tar-архив дерева подкаталогов и вызывает [gzip](#) для выполнения сжатия, исключение составляют [скрытые файлы](#) в текущем каталоге (**\$PWD**). [\[30\]](#)

Некоторые, часто используемые, ключи команды **tar**:

1. -c -- создать (create) новый архив
2. -x -- извлечь (extract) файлы из архива
3. --delete -- удалить (delete) файлы из архива
 Этот ключ игнорируется для накопителей на магнитной ленте.
4. -r -- добавить (append) файлы в существующий архив
5. -A -- добавить (append) tar-файлы в существующий архив
6. -t -- список файлов в архиве (содержимое архива)
7. -u -- обновить (update) архив
8. -d -- операция сравнения архива с заданной файловой системой
9. -z -- обработка архива с помощью [gzip](#)

(Сжатие или разжатие, в зависимости от комбинации сопутствующих ключей -c или -x)

10. -j -- обработка архива с помощью [bzip2](#)

 При восстановлении "битых" tar.gz архивов могут возникнуть определенные сложности, поэтому делайте несколько резервных копий.

shar

Утилита создания shell-архива. Архивируемые файлы объединяются в единый файл без выполнения сжатия, в результате получается архив -- по сути полноценный сценарий на языке командной оболочки, начинающийся со строки `#!/bin/sh`, который содержит полный набор команд, необходимый для разархивирования. Такого рода архивы до сих пор можно найти в некоторых телеконференциях в Internet, но в последнее время они активно вытесняются связкой **tar/gzip**. Для распаковки shar-архивов предназначена команда **unshar**.

ar

Утилита создания и обслуживания архивов, главным образом применяется к двоичным файлам библиотек.

rpm

Red Hat Package Manager, или **rpm** -- набор утилит, предназначенных для построения и обслуживания пакетов программного обеспечения как в исходном коде, так и в собранном (откомпилированном) виде. Среди всего прочего, включает в себя утилиты, производящие установку ПО, проверку зависимостей пакетов и проверку их целостности.

Самый простой вариант установки ПО из rpm -- выполнить команду **rpm -i package_name.rpm**.

- ⓘ Команда **rpm -qa** выдаст полный список всех установленных rpm-пакетов в данной системе. Команда **rpm -qa package_name** выведет только пакет(ы) с именем, содержащим комбинацию символов `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
ksymoos-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 утилита архивации и копирования (**copy input and output**). Используется все реже и реже, поскольку вытесняется более мощным архиватором

tar/gzip. Наиболее употребительна для таких операций, как перемещение дерева каталогов.

Пример 12-22. Пример перемещения дерева каталогов с помощью `cpio`

```
#!/bin/bash

# Копирование дерева каталогов с помощью cpio.

ARGS=2
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
    echo "Порядок использования: `basename $0` source destination"
    exit $E_BADARGS
fi

source=$1
destination=$2

find "$source" -depth | cpio -admv "$destination"
# Информацию по ключам утилиты cpio вы найдете в страницах руководства "man
cpio".

exit 0
```

rpm2cpio

Эта утилита конвертирует [rpm](#)-пакет в архив `cpio`.

Пример 12-23. Распаковка архива *rpm*

```
#!/bin/bash
# de-rpm.sh: Распаковка архива 'rpm'

: ${1?"Порядок использования: `basename $0` target-file"}
# Сценарию должно быть передано имя архива 'rpm'.

TEMPFILE=${$.cpio} # Временный файл с "уникальным" именем.
                    # $$ -- PID процесса сценария.

rpm2cpio < $1 > $TEMPFILE # Конверсия из rpm в cpio.
cpio --make-directories -F $TEMPFILE -i # Распаковка cpio-архива.
rm -f $TEMPFILE # Удаление cpio-архива.

exit 0

# Упражнение:
# Добавьте проверку на: 1) Существование "target-file"
#+ 2) Действительно ли "target-file" является rpm-архивом.
# Подсказка: используйте команду 'file'.
```


Сжатие

gzip

Стандартная GNU/UNIX утилита сжатия, заменившая более слабую, и к тому же проприетарную, утилиту **compress**. Соответствующая утилита декомпрессии (разжатия) -- **gunzip**, которая является эквивалентом команды **gzip -d**.

Для работы со сжатыми файлами в конвейере используется фильтр **zcat**, который выводит результат своей работы на `stdout`, допускает перенаправление вывода.

Фактически это та же команда **cat**, только приспособленная для работы со сжатыми файлами (включая файлы, сжатые утилитой **compress**). Эквивалент команды **zcat --gzip -dc**.

 В некоторых коммерческих версиях UNIX, команда **zcat** является синонимом команды **uncompress -c**, и не может работать с файлами, сжатыми с помощью *gzip*.

См. также [Пример 7-7](#).

bzip2

Альтернативная утилита сжатия, обычно дает более высокую степень сжатия (но при этом работает медленнее), чем **gzip**, особенно это проявляется на больших файлах. Соответствующая утилита декомпрессии -- **bunzip2**.

 В современные версии [tar](#) добавлена поддержка **bzip2**.


compress, uncompress

Устаревшие проприетарные утилиты для работы с архивами, входящие в состав некоторых коммерческих дистрибутивов UNIX. В последнее время вытесняются более мощной утилитой **gzip**. Linux-дистрибутивы, как правило, включают в свой состав эти утилиты для обратной совместимости, однако **gunzip** корректно разархивирует файлы, обработанные с помощью **compress**.

 Утилита **znew** предназначена для преобразования *compress*-архивов в *gzip*-архивы.

sq

Еще одна утилита-фильтр сжатия, которая обслуживает только отсортированные списки слов. Использует стандартный, для фильтров, синтаксис вызова -- **sq < input-file > output-file**. Быстрая, но не такая эффективная как [gzip](#). Соответствующая ей утилита декомпрессии называется **unsq**, синтаксис вызова аналогичен утилите **sq**.

 Вывод от **sq** может быть передан по конвейеру утилите **gzip**, для дальнейшего сжатия.

zip, unzip

Кроссплатформенная утилита архивирования и сжатия, совместимая, по формату архивного файла, с утилитой DOS -- *pkzip.exe*. "Zip"-архивы, по-моему, более приемлемый вариант для обмена данными через Internet, чем "tarballs" (тарболлы, или tar-архивы).

unarc, unarj, unrar

Этот набор утилит предназначен для распаковки архивов, созданных с помощью DOS архиваторов -- *arc.exe*, *arj.exe* и *rar.exe*.

Получение сведений о файлах

file

Утилита идентификации файлов. Команда `file file-name` верне тип файла `file-name`, например, `ascii text` или `data`. Для этого она анализирует сигнатуру, или [магическое число](#) и сопоставляет ее со списком известных сигнатур из `/usr/share/magic`, `/etc/magic` или `/usr/lib/magic` (в зависимости от дистрибутива Linux/UNIX).

-f -- ключ пакетного режима работы утилиты **file**, в этом случае утилита принимает список анализируемых имен файлов из заданного файла. Ключ -z используется для анализа файлов в архиве.

```
bash$ file test.tar.gz
test.tar.gz: gzip compressed data, deflated, last modified: Sun Sep 16 13:34:51
2001, os: Unix
```

```
bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated, last modified: Sun
Sep 16 13:34:51 2001, os: Unix)
```

Пример 12-24. Удаление комментариев из файла с текстом программы на языке C

```
#!/bin/bash
# strip-comment.sh: Удаление комментариев (/ * COMMENT */) из исходных текстов
программ на языке C.

E_NOARGS=65
E_ARGERROR=66
E_WRONG_FILE_TYPE=67

if [ $# -eq "$E_NOARGS" ]
then
    echo "Порядок использования: `basename $0` C-program-file" >&2 # Вывод
сообщения на stderr.
    exit $E_ARGERROR
fi

# Проверка типа файла.
type=`eval file $1 | awk '{ print $2, $3, $4, $5 }'`
# "file $1" -- выводит тип файла...
# затем awk удаляет первое поле -- имя файла...
# после этого результат записывается в переменную "type".
correct_type="ASCII C program text"

if [ "$type" != "$correct_type" ]
then
    echo
    echo "Этот сценарий работает только с исходными текстами программ на языке C."
    echo
    exit $E_WRONG_FILE_TYPE
fi

# Довольно замысловатый сценарий sed :
#-----
sed '
/^\//\*/d
/.*\//\*/d
' $1
#-----
# Если вы потратите несколько часов на изучение основ sed, то он станет немного
понятнее.
```

```

# Следовало бы добавить еще обработку
#+ комментариев, расположенных в одной строке с кодом.
# Оставляю это вам, в качестве упражнения.

# Кроме того, этот сценарий удалит все строки, которые содержат комбинации
символов "*" или "/*",
# не всегда желаемый результат.

exit 0

# -----
# Строки, расположенные ниже не будут исполнены из-за стоящей выше команды 'exit
0'.

# Stephane Chazelas предложил другой, альтернативный вариант:

usage() {
    echo "Порядок использования: `basename $0` C-program-file" >&2
    exit 1
}

WEIRD=`echo -n -e '\377'` # или WEIRD='\377'
[[ $# -eq 1 ]] || usage
case `file "$1"` in
    *"C program text"*) sed -e "s%/\*%${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
        | tr '\377\n' '\n\377' \
        | sed -ne 'p;n' \
        | tr -d '\n' | tr '\377' '\n';;
    *) usage;;
esac

# Этот вариант, все еще некорректно обрабатывает такие строки как:
# printf("/*");
# или
# /* /* ошибочный вложенный комментарий */
#
# Для обработки специальных случаев ("\", "\\") ... придется написать
синтаксический анализатор
# (может быть с помощью lex или yacc?).

exit 0

```

which

Команда **which command-xxx** вернет полный путь к "command-xxx". Очень полезна для того, чтобы узнать -- установлена ли та или иная утилита в системе.

```
$bash which rm
```

```
/usr/bin/rm
```

whereis

Очень похожа на **which**, упоминавшуюся выше. Команда **whereis command-xxx** вернет полный путь к "command-xxx", но кроме того, еще и путь к *manpage* -- файлу, странице справочника по заданной утилите.

```
$bash whereis rm
```

```
rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

Утилита **whatis filexxx** отыщет "filexxx" в своей базе данных. Может рассматриваться как упрощенный вариант команды **man**.

```
$bash whatis whatis
```

```
whatis (1) - search the whatis database for complete words
```

Пример 12-25. Исследование каталога /usr/X11R6/bin

```
#!/bin/bash

# Что находится в каталоге /usr/X11R6/bin?

DIRECTORY="/usr/X11R6/bin"
# Попробуйте также "/bin", "/usr/bin", "/usr/local/bin", и т.д.

for file in $DIRECTORY/*
do
  whatis `basename $file` # Вывод информации о файле.
done

exit 0
# Вывод этого сценария можно перенаправить в файл:
# ./what.sh >>whatis.db
# или включить страничный просмотр на экране,
# ./what.sh | less
```

См. также [Пример 10-3](#).

vdir

Вывод списка файлов в каталоге. Тот же эффект имеет команда [ls -l](#).

Это одна из утилит GNU *fileutils*.

```
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** определяет местонахождение файла, используя свою базу данных, создаваемую специально для этих целей. Команда **slocate** -- это защищенная версия **locate** (которая может оказаться простым псевдонимом команды **slocate**).

\$bash locate hickson

```
/usr/lib/xephem/catalogs/hickson.edb
```

readlink

Возвращает имя файла, на который указывает символическая ссылка.

```
bash$ readlink /usr/bin/awk
../../bin/gawk
```

strings

Команда **strings** используется для поиска печатаемых строк в двоичных файлах. Она выводит последовательности печатаемых символов, обнаруженных в заданном файле. Может использоваться для прикидочного анализа дампов-файлов (core dump) или для отыскания информации о типе файла, например для графических файлов неизвестного формата (например, `strings image-file | more` может вывести такую строку: JFIF, что говорит о том, что мы имеем дело с графическим файлом в формате *jpeg*). В сценариях, вероятнее всего, вам придется использовать эту команду в связке с [grep](#) или [sed](#). См. [Пример 10-7](#) и [Пример 10-9](#).

Пример 12-26. "Расширенная" команда *strings*

```
#!/bin/bash
# wstrings.sh: "word-strings" (расширенная команда "strings")
#
# Этот сценарий фильтрует вывод команды "strings" путем проверки на
соответствие
#+ выводимых слов по файлу словаря.
# Таким способом эффективно "отсекается" весь "мусор",
#+ и выводятся только распознанные слова.

# =====
#                               Стандартная проверка входных аргументов
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne $ARGS ]
then
    echo "Порядок использования: `basename $0` filename"
    exit $E_BADARGS
fi

if [ ! -f "$1" ]
then
    echo "Файл \"$1\" не найден."
    exit $E_NOFILE
fi

# =====

MINSTRLEN=3
WORDFILE=/usr/share/dict/linux.words

# Минимальная длина строки.
# Файл словаря.
# Можно указать иной
#+ файл словаря
#+ в формате -- "одно слово на строке".
```

```
wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`

# Трансляция вывода от 'strings' с помощью нескольких 'tr'.
# "tr A-Z a-z" -- перевод в нижний регистр.
# "tr '[:space:]'" -- конвертирует пробелы в символы Z.
# "tr -cs '[:alpha:]' Z" -- конвертирует неалфавитные символы в символы Z,
#+ и удаляет повторяющиеся символы Z.
# "tr -s '\173-\377' Z" -- Конвертирует все символы, с кодами выше 'z' в Z
#+ и удаляет повторяющиеся символы Z,
#+ эта команда удалит все символы, которые не были распознаны предыдущими
#+ командами трансляции (tr).
# Наконец, "tr Z ' '" -- преобразует все символы Z в пробелы,
#+ которые будут рассматриваться в качестве разделителя слов в цикле,
приведенном ниже.

# Обратите внимание на технику многоуровневой обработки с помощью 'tr',
#+ каждый раз эта команда вызывается с различным набором аргументов.

for word in $wlist
# Важно:
# переменная $wlist не должна заключаться
в кавычки.
# "$wlist" -- не работает.
# Почему?
do
    strlen=${#word}
    if [ "$strlen" -lt "$MINSTRLEN" ]
    # Дина строки.
    # Не рассматривать короткие строки.
    then
        continue
    fi

    grep -Fw $word "$WORDFILE"
    # Проверка слова по словарю.
done

exit 0
```


Сравнение

diff, patch

diff: очень гибкая утилита сравнения файлов. Она выполняет построчное сравнение файлов. В отдельных случаях, таких как поиск по словарю, может оказаться полезной фильтрация файлов с помощью [sort](#) и [uniq](#) перед тем как отдать поток данных через конвейер утилите **diff**. `diff file-1 file-2` -- выведет строки, имеющие отличия, указывая -- какому файлу, какая строка принадлежит.

С ключом `--side-by-side`, команда **diff** выведет сравниваемые файлы в две колонки, с указанием несопадающих строк. Ключи `-c` и `-u` так же служат для облегчения интерпретации результатов работы **diff**.

Существует ряд интерфейсных оболочек для утилиты **diff**, среди них можно назвать: **spiff**, **wdiff**, **xdiff** и **mgdiff**.

-  Команда **diff** возвращает код завершения 0, если сравниваемые файлы идентичны и 1, если они отличаются. Это позволяет использовать **diff** в условных операторах внутри сценариев на языке командной оболочки (см. ниже).

В общем случае, **diff** используется для генерации файла различий, который используется как аргумент команды **patch**. Ключ `-e` отвечает за вывод файла различий в формате, пригодном для использования с **ed** или **ex**.

patch: гибкая утилита для "наложения заплат". С помощью файла различий, сгенерированного утилитой **diff**, утилита **patch** может использоваться для обновления устаревших версий файлов. Это позволяет распространять относительно небольшие "diff"-файлы вместо целых пакетов. Распространение "заплат" к ядру стало наиболее предпочтительным методом распространения более новых версий ядра Linux.

```
patch -p1 <patch-file
# Применит все изменения из 'patch-file'
# к файлам, описанным там же.
# Так выполняется обновление пакетов до более высоких версий.
```

Наложение "заплат" на ядро:

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
# Обновление исходных текстов ядра с помощью 'patch'.
# Пример взят из файла "README",
# автор не известен (Alan Cox?).
```



Кроме того, утилита **diff** в состоянии выполнять рекурсивный обход каталогов.

```
bash$ diff -r ~/notes1 ~/notes2
Only in /home/bozo/notes1: file02
Only in /home/bozo/notes1: file03
Only in /home/bozo/notes2: file04
```



Утилита **zdiff** сравнивает сжатые, с помощью *gzip*, файлы.

diff3

Расширенная версия **diff**, которая сравнивает сразу 3 файла. В случае успеха возвращает 0, но, к сожалению, не дает никакой информации о результатах сравнения.

```
bash$ diff3 file-1 file-2 file-3
====
1:1c
  This is line 1 of "file-1".
2:1c
  This is line 1 of "file-2".
3:1c
  This is line 1 of "file-3"
```

sdiff

Сравнение и/или редактирование двух файлов перед объединением их в один файл. Это интерактивная утилита, по своей природе, и из-за этого она довольно редко используется в сценариях.

cmp

Утилита **cmp** -- это упрощенная версия **diff**. В то время, как **diff** выводит подробную информацию об имеющихся различиях, утилита **cmp** лишь показывает номер строки и позицию в строке, где было встречено различие.



Подобно команде **diff**, команда **cmp** возвращает код завершения 0, если файлы идентичны и 1, если они различны. Это позволяет использовать команду **cmp** в условных операторах.

Пример 12-27. Пример сравнения двух файлов с помощью cmp.

```
#!/bin/bash

ARGS=2 # Ожидаются два аргумента командной строки.
E_BADARGS=65
E_UNREADABLE=66

if [ $# -ne "$ARGS" ]
then
    echo "Порядок использования: `basename $0` file1 file2"
    exit $E_BADARGS
fi

if [[ ! -r "$1" || ! -r "$2" ]]
then
    echo "Оба файла должны существовать и должны быть доступны для чтения."
    exit $E_UNREADABLE
fi

cmp $1 $2 &> /dev/null # /dev/null -- "похоронит" вывод от команды "cmp".
# cmp -s $1 $2 даст тот же результат ("-s" -- флаг "тишины" для "cmp")
# Спасибо Anders Gustavsson за замечание.
#
# Также применимо к 'diff', т.е., diff $1 $2 &> /dev/null

if [ $? -eq 0 ] # Проверка кода возврата команды "cmp".
then
    echo "Файл \"$1\" идентичен файлу \"$2\"."
else
    echo "Файл \"$1\" отличается от файла \"$2\"."
fi

exit 0
```



Для работы с *gzip* файлами используется утилита **zcmp**.

comm

Универсальная утилита сравнения. Работает с отсортированными файлами.

comm -options first-file second-file

comm file-1 file-2 -- вывод в три колонки:

- колонка 1 = уникальные строки для file-1
- колонка 2 = уникальные строки для file-2

- колонка 3 = одинаковые строки.

Ключи, подавляющие вывод в одной или более колонках.

- -1 -- подавление вывода в колонку 1
- -2 -- подавление вывода в колонку 2
- -3 -- подавление вывода в колонку 3
- -12 -- подавление вывода в колонки 1 и 2, и т.д.

Утилиты

basename

Выводит только название файла, без каталога размещения. Конструкция `basename $0` -- позволяет сценарию узнать свое имя, то есть имя файла, который был запущен. Это имя может быть использовано для вывода сообщений, например:

```
echo "Порядок использования: `basename $0` arg1 arg2 ... argn"
```

dirname

Отсекает **basename** от полного имени файла и выводит только путь к файлу.



Утилитам **basename** и **dirname** может быть передана любая строка, в качестве аргумента. Этот аргумент необязательно должен быть именем существующего файла (см. [Пример А-8](#)).

Пример 12-28. Утилиты **basename** и **dirname**

```
#!/bin/bash

a=/home/bozo/daily-journal.txt

echo "Basename для /home/bozo/daily-journal.txt = `basename $a`"
echo "Dirname для /home/bozo/daily-journal.txt = `dirname $a`"
echo
echo "Мой домашний каталог `basename ~/`." # Можно указать просто ~.
echo "Каталог моего домашнего каталога `dirname ~/`." # Можно указать просто ~.

exit 0
```

split

Утилита разбивает файл на несколько частей. Обычно используется для разбиения больших файлов, чтобы их можно было записать на дискеты или передать по электронной почте по частям.

sum, cksum, md5sum

Эти утилиты предназначены для вычисления контрольных сумм. Контрольная сумма -- это некоторое число, вычисляемое исходя из содержимого файла, и служит для контроля целостности информации в файле. Сценарий может выполнять проверку контрольных сумм для того, чтобы убедиться, что файл не был изменен или поврежден. Для большей безопасности, рекомендуется использовать 128-битную сумму, генерируемую утилитой **md5sum** (**message digest checksum**).

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz
```

```
bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba /boot/vmlinuz
```

Обратите внимание: утилита **cksum** выводит контрольную сумму и размер файла в байтах.

Пример 12-29. Проверка целостности файла

```
#!/bin/bash
# file-integrity.sh: Проверка целостности файлов в заданном каталоге

E_DIR_NOMATCH=70
E_BAD_DBFILE=71

dbfile=File_record.md5
# Файл для хранения контрольных сумм.

set_up_database ()
{
    echo "$directory" > "$dbfile"
    # Записать название каталога в первую строку файла.
    md5sum "$directory"/* >> "$dbfile"
    # Записать контрольные суммы md5 и имена файлов.
}

check_database ()
{
    local n=0
    local filename
    local checksum

    # ----- #
    # Возможно эта проверка и не нужна,
    #+ но лучше перестраховаться сейчас, чем жалеть об этом потом.

    if [ ! -r "$dbfile" ]
    then
        echo "Не могу прочитать файл с контрольными суммами!"
        exit $E_BAD_DBFILE
    fi
    # ----- #

    while read record[n]
    do

        directory_checked="${record[0]}"
        if [ "$directory_checked" != "$directory" ]
        then
            echo "Имя каталога не совпадает с записанным в файле!"
            # Попытка использовать файл контрольных сумм для другого каталога.
            exit $E_DIR_NOMATCH
        fi
    done
}
```

```

fi

if [ "$n" -gt 0 ] # Не имя каталога.
then
  filename[n]=$ ( echo ${record[$n]} | awk '{ print $2 }' )
  # md5sum записывает в обратном порядке,
  #+ сначала контрольную сумму, затем имя файла.
  checksum[n]=$ ( md5sum "${filename[n]}" )

  if [ "${record[n]}" = "${checksum[n]}" ]
  then
    echo "Файл ${filename[n]} не был изменен."
  else
    echo "ОШИБКА КОНТРОЛЬНОЙ СУММЫ для файла ${filename[n]}!"
    # Файл был изменен со времени последней проверки.
  fi
fi

fi

let "n+=1"
done <"$dbfile" # Чтение контрольных сумм из файла.

}

# ===== #
# main ()

if [ -z "$1" ]
then
  directory="$PWD" # Если каталог не задан,
else
  directory="$1" #+ то используется текущий каталог.
fi

clear # Очистка экрана.

# ----- #
if [ ! -r "$dbfile" ] # Необходимо создать файл с контрольными суммами?
then
  echo "Создание файла с контрольными суммами, \"$directory\"/\"$dbfile\".";
echo
  set_up_database
  fi
# ----- #

check_database # Выполнить проверку.

echo

# Вывод этого сценария можно перенаправить в файл,
#+ это особенно полезно при проверке большого количества файлов.

# Более строгая проверка целостности файлов,
#+ может быть выполнена с помощью пакета "Tripwire",
#+ http://sourceforge.net/projects/tripwire/.

exit 0

```

Более творческий подход к использованию **md5sum** вы найдете в [Пример А-21](#).

shred

Надежное, с точки зрения безопасности, стирание файла, посредством предварительной, многократной записи в файл случайной информации, перед тем как удалить его. Эта команда имеет тот же эффект, что и [Пример 12-42](#), но делает это более изящным и безопасным способом.

Является составной частью пакета GNU *fileutils*.



Имеется ряд технологий, с помощью которых все-таки возможно восстановить файлы, удаленные утилитой **shred**.

Кодирование и шифрование

uencode

Эта утилита используется для кодирования двоичных файлов в символы ASCII, после такого кодирования файлы могут, с достаточной степенью безопасности, передаваться по сети, вкладываться в электронные письма и т.п..

udecode

Утилита декодирования файлов, прошедших обработку утилитой uencode.

Пример 12-30. Декодирование файлов

```
#!/bin/bash

lines=35          # 35 строк для заголовка (более чем достаточно).

for File in *     # Обход всех файлов в текущем каталоге...
do
  search1=`head -$lines $File | grep begin | wc -w`
  search2=`tail -$lines $File | grep end | wc -w`
  # Закодированные файлы начинаются со слова "begin",
  #+ и заканчиваются словом "end".
  if [ "$search1" -gt 0 ]
  then
    if [ "$search2" -gt 0 ]
    then
      echo "декодируется файл - $File -"
      udecode $File
    fi
  fi
done

# Обратите внимание: если передать сценарию самого себя, для декодирования,
#+ то это введет его в заблуждение
#+ поскольку в тексте сценария встречаются слова "begin" и "end".

exit 0
```



При декодировании и выводе длинных текстовых сообщений из новостных групп Usenet, очень нелишним будет передать текст, по конвейеру, команде [fold -s](#).

mimencode, mmencode

Утилиты **mimencode** и **mmencode** предназначены для обработки закодированных мультимедийных вложений в электронные письма. Хотя *почтовые программы* (такие как **pine** или **kmail**) имеют возможность автоматической обработки таких вложений, тем не менее эти утилиты позволяют обрабатывать вложения вручную, из командной строки или в пакетном режиме, из сценария на языке командной оболочки.

crypt

Одно время, это была стандартная, для UNIX, утилита шифрования файлов. [\[31\]](#) Политически мотивированные, правительственные постановления ряда стран, напрямую запрещают экспорт программного обеспечения для шифрования, что, в результате, привело практически к полному исчезновению **crypt** из большинства UNIX-систем (в том числе и Linux). К счастью, программистами было разработано множество вполне приличных альтернатив, и среди них [cruft](#) (см. [Пример А-5](#)).

Прочее

mktemp

Создает временный файл с "уникальным" именем.

```
PREFIX=filename
tempfile=`mktemp $PREFIX.XXXXXX`
#           ^^^^^^^^ Необходимо по меньшей мере 6 заполнителей
echo "имя временного файла = $tempfile"
# имя временного файла = filename.QA2ZpY
#           или нечто подобное...
```

make

Утилита для компиляции и сборки программ. Но может использоваться для выполнения любых других операций, основанных на анализе наличия изменений в исходных файлах.

Команда **make** использует в своей работе `Makefile`, который содержит перечень зависимостей и операций, которые необходимо выполнить для удовлетворения этих зависимостей.

install

Своего рода -- утилита копирования файлов, похожа на **cp**, но дополнительно позволяет изменять права доступа и атрибуты копируемых файлов. Напрямую эта команда практически не используется, чаще всего она встречается в `Makefile` (в разделе `make install` :). Она может использоваться в сценариях установки ПО.

dos2unix

Автор утилиты -- Benjamin Lin со-товарищи. Предназначена для преобразования текстовых файлов из формата DOS (в котором строки завершаются комбинацией символов CR-LF) в формат UNIX (в котором строки завершаются одним символом LF) и обратно.

ptx

Команда **ptx [targetfile]** выводит а упорядоченный предметный указатель для `targetfile`, который можно обработать, по мере необходимости, какой либо утилитой форматирования, в конвейере.

more, less

Команды страничного просмотра текстовых файлов или потоков на `stdout`. Могут использоваться в сценариях в качестве фильтров.

12.6. Команды для работы с сетью

Команды, описываемые в этом разделе, могут найти применение при исследовании и анализе процессов передачи данных по сети, а также могут использоваться в [борьбе со спамерами](#).

Информация и статистика

host

Возвращает информацию об узле Интернета, по заданному имени или IP адресу, выполняя поиск с помощью службы DNS.

```
bash$ host surfacemail.com
surfacemail.com. has address 202.92.42.236
```

ipcalc

Производит поиск IP адреса. С ключом **-h**, **ipcalc** выполняет поиск имени хоста в DNS, по заданному IP адресу.

```
bash$ ipcalc -h 202.92.42.236
HOSTNAME=surfacemail.com
```

nslookup

Выполняет "поиск имени узла" Интернета по заданному IP адресу. По сути, эквивалентна командам **ipcalc -h** и **dig -x**. Команда может исполняться как в интерактивном, так и в неинтерактивном режиме, т.е. в пределах сценария.

```
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

Подобно команде **nslookup**, выполняет "поиск имени узла" в Интернете.

Сравните вывод команды **dig -x** с выводом команд **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
```

traceroute

Утилита предназначена для исследования топологии сети посредством передачи ICMP пакетов удаленному узлу. Эта программа может работать в LAN, WAN и в Интернет. Удаленный узел может быть указан как по имени, так и по IP адресу. Вывод команды traceroute может быть передан по конвейеру утилитам [grep](#) или [sed](#), для дальнейшего анализа.

```
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

Выполняет передачу пакета "ICMP ECHO_REQUEST" другой системе в сети. Чаще всего служит в качестве инструмента диагностики соединений, должна использоваться с большой осторожностью.

В случае успеха, **ping** возвращает [код завершения](#) 0, поэтому команда ping может использоваться в условных операторах.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
Warning: time of day goes back, taking countermeasures.
 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
```

whois

Выполняет поиск в DNS (Domain Name System). Ключом -h можно указать какой из

whois серверов будет запрошен. См. [Пример 4-6](#).

finger

Возвращает информацию о пользователях в сети. По желанию, эта команда может выводить содержимое файлов `~/ .plan`, `~/ .project` и `~/ .forward`, указанного пользователя.

```
bash$ finger
Login  Name           Tty      Idle  Login Time   Office      Office Phone
bozo   Bozo Bozeman   tty1    8      Jun 25 16:59
bozo   Bozo Bozeman   tty0
bozo   Bozo Bozeman   tty1    Jun 25 17:07
```

```
bash$ finger bozo
Login: bozo                               Name: Bozo Bozeman
Directory: /home/bozo                     Shell: /bin/bash
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
No mail.
No Plan.
```

По соображениям безопасности, в большинстве сетей служба **finger**, и соответствующий демон, отключена. [\[32\]](#)

vrfy

Проверка адреса электронной почты.

Доступ к удаленным системам

sx, rx

Команды **sx** и **rx** служат для приема/передачи файлов на/из удаленный узел в сети, по протоколу *xmodem*. Входят в состав пакета **minicom**.

sz, rz

Команды **sz** и **rz** служат для приема/передачи файлов на/из удаленный узел в сети, по протоколу *zmodem*. Протокол *zmodem* имеет некоторые преимущества перед протоколом *xmodem*, в качестве такого преимущества можно назвать более высокую скорость передачи и возможность возобновления передачи, в случае ее разрыва. Входят в состав пакета **minicom**.

ftp

Под этим именем подразумевается утилита и протокол передачи файлов. Сеансы ftp могут устанавливаться из сценариев (см. [Пример 17-7](#), [Пример A-5](#) и [Пример A-14](#)).

uucp

UNIX to UNIX copy. Это коммуникационный пакет для передачи файлов между UNIX

серверами. Сценарий на языке командной оболочки -- один из самых эффективных способов автоматизации такого обмена.

Похоже, что с появлением Интернет и электронной почты, **uucp** постепенно уходит в небытие, однако, она с успехом может использоваться в изолированных, не имеющих выхода в Интернет, сетях.

cu

Call Up -- выполняет соединение с удаленной системой, как простой терминал. Эта команда является частью пакета **uucp** и, своего рода, упрощенным вариантом команды [telnet](#).

telnet

Утилита и протокол для подключения к удаленной системе.



Протокол **telnet** небезопасен по своей природе, поэтому следует воздерживаться от его использования.

wget

wget -- *неинтерактивная* утилита для скачивания файлов с Web или ftp сайтов.

```
wget -p http://www.xyz23.com/file01.html
wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -o $SAVEFILE
```

lynx

lynx -- Web браузер, внутри сценариев (с ключом `-dump`) может использоваться для скачивания файлов с Web или ftp сайтов, в неинтерактивном режиме.

```
lynx -dump http://www.xyz23.com/file01.html >$SAVEFILE
```

rlogin

Remote login -- инициирует сессию с удаленной системой. Эта команда небезопасна, вместо нее лучше использовать [ssh](#).

rsh

Remote shell -- исполняет команду на удаленной системе. Эта команда небезопасна, вместо нее лучше использовать [ssh](#).

rcp

Remote copy -- копирование файлов между двумя машинами через сеть. Подобно прочим *r** утилитах, команда **rcp** небезопасна и потому, использовать ее в сценариях нежелательно. В качестве замены можно порекомендовать **ssh** или **expect**.

ssh

Secure shell -- устанавливает сеанс связи и выполняет команды на удаленной системе. Выступает в качестве защищенной замены для **telnet**, **rlogin**, **rccp** и **rsh**. Использует идентификацию, аутентификацию и шифрование информации, передаваемой через сеть. Подробности вы найдете в *man ssh*.

Локальная сеть

write

Эта утилита позволяет передать текст сообщения на другой терминал (console или xterm). Разрешить или запретить доступ к терминалу можно с помощью команды [mesg](#).

Поскольку команда **write** работает в интерактивном режиме, то, как правило, она не употребляется в сценариях.

Mail

mail

Чтение или передача электронной почты.

Этот почтовый клиент командной строки с успехом может использоваться в сценариях.

Пример 12-31. Сценарий, отправляющий себя самого по электронной почте

```
#!/bin/sh
# self-mailer.sh: Сценарий отправляет себя самого по электронной почте

adr=${1:-`whoami`}      # Если пользователь не указан, то -- себе самому.
# Вызов 'self-mailer.sh wiseguy@superdupergenius.com'
#+ приведет к передаче электронного письма по указанному адресу.
# Вызов 'self-mailer.sh' (без аргументов) -- отправит письмо
#+ пользователю, запустившему сценарий, например, bozo@localhost.localdomain.
#
# Дополнительно о конструкции ${parameter:-default},
#+ см. раздел "Подстановка параметров"
#+ в главе "К вопросу о переменных".

# =====
# cat $0 | mail -s "Сценарий \"`basename $0`\" отправил себя сам." "$adr"
# =====

# -----
# Поздравляю!
# Этот сценарий запустила какая-то "редиска",
#+ и заставила отправить этот текст к Вам.
# Очевидно кто-то не знает
#+ куда девать свое время.
# -----

echo "`date`, сценарий \"`basename $0`\" отправлен \"$adr\"."

exit 0
```

mailto

Команда **mailto**, похожа на **mail**, она также отправляет сообщения по электронной почте. Однако, кроме этого, **mailto** позволяет отправлять MIME (multimedia) сообщения.

vacation

Эта утилита предназначена для автоматической передачи ответов на электронные письма, например для того, чтобы уведомить отправителя о том, что получатель временно отсутствует. Работает совместно с **sendmail** и не может использоваться для передачи сообщений через коммутируемые линии (по модему).

12.7. Команды управления терминалом

Команды, имеющие отношение к консоли или терминалу

tput

инициализация терминала или выполнение запроса к базе данных терминалов *terminfo*. С помощью **tput** можно выполнять различные операции. **tput clear** -- эквивалентно команде **clear**. **tput reset** -- эквивалентно команде **reset**. **tput sgr0** -- так же сбрасывает настройки терминал, но без очистки экрана.

```
bash$ tput longname
xterm terminal emulator (XFree86 4.0 Window System)
```

Команда **tput cup X Y** перемещает курсор в координаты (X,Y). Обычно этой команде предшествует **clear**, очищающая экран.

Обратите внимание: [stty](#) предлагает более широкий диапазон возможностей.

infocmp

Сравнение или печать информации о характеристиках терминалов, хранящейся в базе 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=``aaffggjjkkllmmnnooppqrrssttuuvvwxxyyz{|}|}~~,
      bel=^G, blink=\E[5m, bold=\E[1m,
      civis=\E[?25l,
      clear=\E[H\E[2J, cnorm=\E[?25h, cr=^M,
      ...
```

reset

Сбрасывает настройки терминала и очищает экран. Как и в случае команды **clear**, курсор и приглашение к вводу (prompt) выводятся в верхнем левом углу терминала.

clear

Команда **clear** просто очищает экран терминала или окно xterm. Курсор и приглашение к вводу (prompt) выводятся в верхнем левом углу терминала. Эта команда может запускаться как из командной строки, так и из сценария. См. [Пример 10-25](#).

script

Эта утилита позволяет сохранять в файле все символы, введенные пользователем с клавиатуры (вывод тоже). Получая, фактически, подробнейший синхронный протокол сессии.

12.8. Команды выполнения математических операций

factor

Разложение целого числа на простые множители.

```
bash$ factor 27417
27417: 3 13 19 37
```

bc

Bash не в состоянии выполнять действия над числами с плавающей запятой и не содержит многих важных математических функций. К счастью существует **bc**.

Универсальная, выполняющая вычисления с произвольной точностью, утилита **bc** обладает некоторыми возможностями, характерными для языков программирования.

Синтаксис **bc** немного напоминает язык C.

Поскольку это утилита UNIX, то она может достаточно широко использоваться в сценариях на языке командной оболочки, в том числе и в [конвейерной](#) обработке данных.

Ниже приводится простой шаблон работы с утилитой **bc** в сценарии. Здесь используется прием [подстановки команд](#).

```
variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

Пример 12-32. Ежемесячные выплаты по займу

```
#!/bin/bash
# monthlypmt.sh: Расчет ежемесячных выплат по займу.

# Это измененный вариант пакета "mcalc" (mortgage calculator),
#+ написанного Jeff Schmidt и Mendel Cooper (ваш покорный слуга).
# http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz [15k]
```

```

echo
echo "Введите сумму займа, процентную ставку и срок займа,"
echo "для расчета суммы ежемесячных выплат."

bottom=1.0

echo
echo -n "Сумма займа (без запятых -- с точностью до доллара) "
read principal
echo -n "Процентная ставка (процент) " # Если 12%, то нужно вводить "12", а не
".12".
read interest_r
echo -n "Срок займа (месяцев) "
read term

interest_r=$(echo "scale=9; $interest_r/100.0" | bc) # Здесь "scale" --
точность вычислений.

interest_rate=$(echo "scale=9; $interest_r/12 + 1.0" | bc)

top=$(echo "scale=9; $principal*$interest_rate^$term" | bc)

echo; echo "Прошу подождать. Вычисления потребуют некоторого времени."

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 Voivie предложил более эффективную реализацию
#+ цикла вычислений, который дает выигрыш по времени на 2/3.

# for ((x=1; x <= $months; x++))
# do
#     bottom=$(echo "scale=9; $bottom * $interest_rate + 1" | bc)
# done

# А затем нашел еще более эффективную альтернативу,
#+ которая выполняется в 20 раз быстрее !!!

# 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` # Внедрить цикл 'for' в конструкцию подстановки команд.

# =====

# let "payment = $top/$bottom"
payment=$(echo "scale=2; $top/$bottom" | bc)
# Два знака после запятой, чтобы показать доллары и центы.

echo
echo "ежемесячные выплаты = \$$payment" # Вывести знак "доллара" перед числом.
echo

```

```
exit 0
```

```
# Упражнения:
```

```
# 1) Добавьте возможность ввода суммы с точностью до цента.
```

```
# 2) Добавьте возможность ввода процентной ставки как в виде процентов, так и в виде десятичного числа -- доли целого.
```

```
# 3) Если вы действительно честолюбивы,
```

```
# добавьте в сценарий вывод полной таблицы помесечных выплат.
```

Пример 12-33. Перевод чисел из одной системы счисления в другую

```
:
#####
# Shellscript: base.sh - вывод чисел в разных системах счисления (Bourne Shell)
# Author      : 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 $
#####
# Description
#
# Changes
# 21-03-95 stv исправлена ошибка, возникающая при вводе числа 0xb (0.2)
#####

# ==> Используется в данном документе с разрешения автора.
# ==> Комментарии добавлены автором документа.

NOARGS=65
PN=`basename "$0" ` # Имя программы
VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2` # ==> VER=1.2

Usage () {
    echo "$PN - вывод чисел в различных системах счисления, $VER (stv '95)
Порядок использования: $PN [number ...]

Если число не задано, то производится ввод со stdin.
Число может быть:
    двоичное           должно начинаться с комбинации символов 0b (например
0b1100)
    восьмеричное      должно начинаться с 0 (например 014)
    шестнадцатичное  должно начинаться с комбинации символов 0x (например
0xc)
    десятичное        в любом другом случае (например 12)" >&2
    exit $NOARGS
} # ==> Функция вывода сообщения о порядке использования.

Msg () {
    for i # ==> [список] параметров опущен.
    do echo "$PN: $i" >&2
    done
}

Fatal () { Msg "$@"; exit 66; }

PrintBases () {
    # Определение системы счисления
    for i # ==> [список] параметров опущен...
    do # ==> поэтому работает с аргументами командной строки.
        case "$i" in
            0b*)          ibase=2;; # двоичная
            0x*|[a-f]*|[A-F]*) ibase=16;; # шестнадцатичная
            0*)           ibase=8;; # восьмеричная
            [1-9]*)       ibase=10;; # десятичная
            *)
                Msg "Ошибка в числе $i - число проигнорировано"
                continue;;
        esac
    done
}
```

```

# Удалить префикс и преобразовать шестнадцатиричные цифры в верхний
регистр (этого требует bc)
number=`echo "$i" | sed -e 's:^@[bVxX]::' | tr '[a-f]' '[A-F]'`
# ==> вместо "/", здесь используется символ ":" как разделитель для sed.

# Преобразование в десятичную систему счисления
dec=`echo "ibase=$ibase; $number" | bc` # ==> 'bc' используется как
калькулятор.
case "$dec" in
    [0-9]*) ;; # все в порядке
    *) continue;; # ошибка: игнорировать
esac

# Напечатать все преобразования в одну строку.
# ==> 'вложенный документ' -- список команд для '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 ]
do
    case "$1" in
        --) shift; break;;
        -h) Usage;; # ==> Вывод справочного сообщения.
        -*) Usage;;
        *) break;; # первое число
    esac # ==> Хорошо бы расширить анализ вводимых символов.
    shift
done

if [ $# -gt 0 ]
then
    PrintBases "$@"
else
    # чтение со stdin
    while read line
    do
        PrintBases $line
    done
fi

```

Один из вариантов вызова **bc** -- использование [вложенного документа](#), внедряемого в блок с [подстановкой команд](#). Это особенно актуально, когда сценарий должен передать **bc** значительный по объему список команд и аргументов.

```

variable=`bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
`

```

...ИЛИ...

```

variable=$(bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
)

```

Пример 12-34. Пример взаимодействия bc со "встроенным документом"

```
#!/bin/bash
# Комбинирование 'bc' с
# 'вложенным документом'.

var1=`bc << EOF
18.33 * 19.78
EOF
`
echo $var1          # 362.56

# запись $( ... ) тоже 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
)
# Возвращается значение синуса от 1.7 радиана.
# Ключом "-l" вызывается математическая библиотека 'bc'.
echo $var3          # .991664810

# Попробуем функции...
hyp=                # Объявление глобальной переменной.
hypotenuse ()      # Расчет гипотенузы прямоугольного треугольника.
{
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# К сожалению, функции Bash не могут возвращать числа с плавающей запятой.
}

hypotenuse 3.68 7.31
echo "гипотенуза = $hyp"    # 8.184039344

exit 0
```

Пример 12-35. Вычисление числа "пи"

```
#!/bin/bash
# cannon.sh: Аппроксимация числа "пи".

# Это очень простой вариант реализации метода "Monte Carlo",
#+ математическое моделирование событий реальной жизни,
#+ для эмуляции случайного события используются псевдослучайные числа.
```



```

# Допустим, что мы располагаем картой квадратного участка поверхности со
стороной квадрата 10000 единиц.
# На этом участке, в центре, находится совершенно круглое озеро,
#+ с диаметром в 10000 единиц.
# Т.е. озеро покрывает почти всю карту, кроме ее углов.
# (Фактически -- это квадрат со вписанным кругом.)
#
# Пусть по этому участку ведется стрельба железными ядрами из древней пушки
# Все ядра падают где-то в пределах данного участка,
#+ т.е. либо в озеро, либо на сушу, по углам участка.
# Поскольку озеро покрывает большую часть участка,
#+ то большинство ядер будет падать в воду.
# Незначительная часть ядер будет падать на твердую почву.
#
# Если произвести достаточно большое число неприцельных выстрелов по данному
участку,
#+ то отношение попаданий в воду к общему числу выстрелов будет примерно равно
#+ значению PI/4.
#
# По той простой причине, что стрельба фактически ведется только
#+ по правому верхнему квадранту карты.
# (Предыдущее описание было несколько упрощено.)
#
# Теоретически, чем больше будет произведено выстрелов, тем точнее будет
результат.
# Однако, сценарий на языке командной оболочки, в отличие от других языков
программирования,
#+ в которых доступны операции с плавающей запятой, имеет некоторые ограничения.
# К сожалению, это делает вычисления менее точными.

DIMENSION=10000 # Длина стороны квадратного участка поверхности.
                # Он же -- верхний предел для генератора случайных чисел.

MAXSHOTS=1000  # Количество выстрелов.
                # 10000 выстрелов (или больше) даст лучший результат,
                # но потребует
значительного количества времени.
PMULTIPLIER=4.0 # Масштабирующий коэффициент.

get_random ()
{
SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
RANDOM=$SEED # Из примера "seeding-random.sh"

let "rnum = $RANDOM % $DIMENSION" # Число не более чем 10000.
echo $rnum
}

distance=      # Объявление глобальной переменной.
hypotenuse () # Расчет гипотенузы прямоугольного треугольника.
{             # Из примера "alt-bc.sh".
distance=$(bc -l << EOF
scale = 0
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# Установка "scale" в ноль приводит к округлению результата "вниз",
#+ это и есть то самое ограничение, накладываемое командной оболочкой.
# Что, к сожалению, снижает точность аппроксимации.
}

# main() {

# Инициализация переменных.
shots=0
splashes=0

```

```

thuds=0
Pi=0

while [ "$shots" -lt "$MAXSHOTS" ]           # Главный цикл.
do

    xCoord=$(get_random)                       # Получить случайные координаты X
и Y.
    yCoord=$(get_random)
    hypotenuse $xCoord $yCoord                 # Гипотенуза = расстоянию.
    ((shots++))

    printf "#%4d    " $shots
    printf "Xc = %4d  " $xCoord
    printf "Yc = %4d  " $yCoord
    printf "Distance = %5d  " $distance        # Расстояние от
#+ центра озера,
#+ с координатами (0,0).

    if [ "$distance" -le "$DIMENSION" ]
    then
        echo -n "ШЛЕП!  "                     # попадание в озеро
        ((splashes++))
    else
        echo -n "БУХ!    "                     # попадание на твердую почву
        ((thuds++))
    fi

    Pi=$(echo "scale=9; $PMULTIPLIER*$splashes/$shots" | bc)
    # Умножение на коэффициент 4.0.
    echo -n "PI ~ $Pi"
    echo

done

echo
echo "После $shots выстрела, примерное значение числа \"пи\" равно $Pi."
# Имеет тенденцию к завышению...
# Вероятно из-за ошибок округления и несовершенства генератора случайных чисел.
echo

# }

exit 0

# Самое время задуматься над тем, является ли сценарий удобным средством
#+ для выполнения большого количества столь сложных вычислений.
#
# Тем не менее, этот пример может расцениваться как
# 1) Доказательство возможностей языка командной оболочки.
# 2) Прототип для "обкатки" алгоритма перед тем как перенести
#+ его на высокоуровневые языки программирования компилирующего типа.

```

dc

Утилита **dc** (**desk calculator**) -- это калькулятор, использующий "Обратную Польскую Нотацию", и ориентированный на работу со стеком.

Многие стараются избегать использования **dc**, из-за непривычной формы записи операндов и операций. Однако, **dc** имеет и своих сторонников.

Пример 12-36. Преобразование чисел из десятичной в шестнадцатеричную систему счисления

```

#!/bin/bash
# hexconvert.sh: Преобразование чисел из десятичной в шестнадцатеричную систему

```

счисления.

```
BASE=16      # Шестнадцатеричная.
```

```
if [ -z "$1" ]
then
    echo "Порядок использования: $0 number"
    exit $E_NOARGS
    # Необходим аргумент командной строки.
fi
# Упражнение: добавьте проверку корректности аргумента.
```

```
hexcvt ()
{
    if [ -z "$1" ]
    then
        echo 0
        return # "Return" 0, если функции не был передан аргумент.
    fi
```

```
    echo ""$1" "$BASE" o p" | dc
    #           "o" устанавливает основание системы счисления для вывода.
    #           "p" выводит число, находящееся на вершине стека.
    # См. 'man dc'.
    return
}
```

```
hexcvt "$1"
```

```
exit 0
```

Изучение страниц *info dc* позволит детальнее разобраться с утилитой. Однако, отряд "гуру", которые могут похвастать своим знанием этой мощной, но весьма запутанной утилиты, весьма немногочислен.

Пример 12-37. Разложение числа на простые множители

```
#!/bin/bash
# factr.sh: Разложение числа на простые множители
```

```
MIN=2      # Не работает с числами меньше 2.
E_NOARGS=65
E_TOOSMALL=66
```

```
if [ -z $1 ]
then
    echo "Порядок использования: $0 number"
    exit $E_NOARGS
fi
```

```
if [ "$1" -lt "$MIN" ]
then
    echo "Исходное число должно быть больше или равно $MIN."
    exit $E_TOOSMALL
fi
```

Упражнение: Добавьте проверку типа числа (не целые числа должны отвергаться).

```
echo "Простые множители для числа $1:"
#
```

```
-----
-
echo "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=1lrli2+dsi!>.]ds.xd1<2" |
dc
#
```

```
-----
-
```

```
# Автор вышеприведенной строки: Michel Charpentier <charpov@cs.unh.edu>.  
# Используется с его разрешения (спасибо).
```

```
exit 0
```

awk

Еще один способ выполнения математических операций, над числами с плавающей запятой, состоит в создании [сценария-обертки](#), использующего математические функции [awk](#).

Пример 12-38. Расчет гипотенузы прямоугольного треугольника

```
#!/bin/bash  
# hypotenuse.sh: Возвращает "гипотенузу" прямоугольного треугольника.  
# ( корень квадратный от суммы квадратов катетов)  
  
ARGS=2          # В сценарий необходимо передать два катета.  
E_BADARGS=65    # Ошибка в аргументах.  
  
if [ $# -ne "$ARGS" ] # Проверка количества аргументов.  
then  
    echo "Порядок использования: `basename $0` катет_1 катет_2"  
    exit $E_BADARGS  
fi  
  
AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '  
#          команды и параметры, передаваемые в awk  
  
echo -n "Гипотенуза прямоугольного треугольника, с катетами $1 и $2, = "  
echo $1 $2 | awk "$AWKSCRIPT"  
  
exit 0
```

12.9. Прочие команды

Команды, которые нельзя отнести ни к одной из вышеперечисленных категорий

jot, seq

Эти утилиты выводят последовательность целых чисел с шагом, заданным пользователем.

По-умолчанию, выводимые числа отделяются друг от друга символом перевода строки, однако, с помощью ключа `-s` может быть задан другой разделитель.

```
bash$ seq 5  
1  
2  
3  
4  
5
```

```
bash$ seq -s : 5  
1:2:3:4:5
```

Обе утилиты, и **jot**, и **seq**, очень удобно использовать для генерации списка аргументов в цикле [for](#).

Пример 12-39. Использование **seq** для генерации списка аргументов цикла **for**

```
#!/bin/bash
# Утилита "seq"

echo

for a in `seq 80` # или так:  for a in $( seq 80 )
# То же самое, что и  for a in 1 2 3 4 5 ... 80  (но как экономит время и
силы!).
# Можно использовать и 'jot' (если эта утилита имеется в системе).
do
    echo -n "$a "
done      # 1 2 3 4 5 ... 80
# Пример использования вывода команды для генерации
# [списка] аргументов цикла "for".

echo; echo

COUNT=80 # Да, 'seq' допускает указание переменных в качестве параметра.

for a in `seq $COUNT` # или так:  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 $END`
# Если "seq" передаются два аргумента, то первый означает начальное число
последовательности,
#+ второй -- последнее,
do
    echo -n "$a "
done      # 75 76 77 78 79 80

echo; echo

BEGIN=45
INTERVAL=5
END=80

for a in `seq $BEGIN $INTERVAL $END`
# Если "seq" передается три аргумента, то первый аргумент -- начальное число в
последовательности,
#+ второй -- шаг последовательности,
#+ и третий -- последнее число в последовательности.
do
    echo -n "$a "
done      # 45 50 55 60 65 70 75 80

echo; echo

exit 0
```

getopt

Команда **getopt** служит для разбора командной строки, выделяя из нее ключи --

символы, с предшествующим знаком [дефис](#). Этой утилите имеется, встроенный в Bash, аналог -- [getopts](#), более мощная и универсальная команда.

Пример 12-40. Использование `getopt` для разбора аргументов командной строки

```
#!/bin/bash
# ex33a.sh

# Попробуйте следующие варианты вызова этого сценария.
# sh ex33a -a
# sh ex33a -abc
# sh ex33a -a -b -c
# sh ex33a -d
# sh ex33a -dXYZ
# sh ex33a -d XYZ
# sh ex33a -abcd
# sh ex33a -abcdZ
# sh ex33a -z
# sh ex33a a
# Объясните полученные результаты.

E_OPTERR=65

if [ "$#" -eq 0 ]
then # Необходим по меньшей мере один аргумент.
    echo "Порядок использования: $0 -[options a,b,c]"
    exit $E_OPTERR
fi

set -- `getopt "abcd:" "$@"`
# Запись аргументов командной строки в позиционные параметры.
# Что произойдет, если вместо "$@" указать "$*"?

while [ ! -z "$1" ]
do
    case "$1" in
        -a) echo "Опция \"a\"";;
        -b) echo "Опция \"b\"";;
        -c) echo "Опция \"c\"";;
        -d) echo "Опция \"d\" $2";;
        *) break;;
    esac

    shift
done

# Вместо 'getopt' лучше использовать встроенную команду 'getopts',
# См. "ex33.sh".

exit 0
```

run-parts

Команда **run-parts** [\[33\]](#) запускает на исполнение все сценарии, в порядке возрастания имен файлов-сценариев, в заданном каталоге. Естественно, файлы сценариев должны иметь права на исполнение.

[Демон crond](#) вызывает **run-parts** для запуска сценариев из каталогов `/etc/cron.*`.

yes

По-умолчанию, команда **yes** выводит на `stdout` непрерывную последовательность символов `y`, разделенных символами перевода строки. Исполнение команды можно прервать комбинацией клавиш **control-c**. Команду **yes** можно заставить выводить

иную последовательность символов. Теперь самое время задаться вопросом о практической пользе этой команды. Основное применение этой команды состоит в том, что вывод от нее может быть передан, через конвейер, другой команде, ожидающей реакции пользователя. В результате получается, своего рода, слабенькая версия команды **expect**.

`yes | fsck /dev/hda1` запускает **fsck** в неинтерактивном режиме (будьте осторожны!).

`yes | rm -r dirname` имеет тот же эффект, что и `rm -rf dirname` (будьте осторожны!).



Внимание! Передача вывода команды **yes** по конвейеру потенциально опасным командам, таким как [fsck](#) или [fdisk](#) может дать нежелательные побочные эффекты.

banner

Печатает на `stdout` заданную строку символов (не более 10), рисуя каждый символ строки при помощи символа '#'. Вывод от команды может быть перенаправлен на принтер.

printenv

Выводит все [переменные окружения](#) текущего пользователя.

```
bash$ printenv | grep HOME
HOME=/home/bozo
```

lp

Команды **lp** и **lpr** отправляют файлы в очередь печати [\[34\]](#) для вывода на принтер. Названия этих команд произошли от "line printers".

```
bash$ lp file1.txt или bash lp <file1.txt
```

Очень часто используются в комбинации с командой форматированного вывода **pr**.

```
bash$ pr -options file1.txt | lp
```

Программы подготовки текста к печати, такие как **groff** и *Ghostscript*, так же могут напрямую взаимодействовать с **lp**.

```
bash$ groff -Tascii file.tr | lp
```

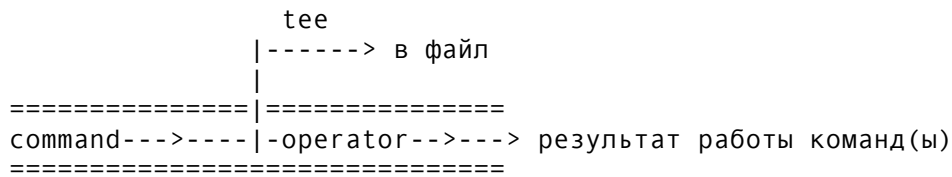
```
bash$ gs -options | lp file.ps
```

Команда **lpq** предназначена для просмотра очереди заданий печати, а **lprm** -- для удаления заданий из очереди.

tee

[UNIX заимствовал эту идею из водопроводного дела.]

Это оператор перенаправления, но с некоторыми особенностями. Подобно водопроводным трубам, "tee" позволяет "направить поток" данных *в несколько файлов* и на stdout одновременно, никак не влияя на сами данные. Эта команда может оказаться очень полезной при отладке.



```
cat listfile* | sort | tee check.file | uniq > result.file
```

(Здесь, в файл `check.file` будут записаны данные из всех "listfile*", в отсортированном виде до того, как повторяющиеся строки будут удалены командой [uniq](#).)

mkfifo

Эта, редко встречающаяся, команда создает *именованный канал* - очередь, через который производится обмен данными между процессами. [\[35\]](#) Как правило, один процесс записывает данные в очередь (FIFO), а другой читает данные из очереди. См. [Пример А-17](#).

pathchk

Производит проверку полного имени файла -- проверяет, доступны ли на чтение, каталоги в пути к файлу, и не превышает ли длина полного имени файла 255 символов. При несоблюдении одного из условий -- возвращает сообщение об ошибке.

К сожалению, **pathchk** не возвращает соответствующего кода ошибки, и потому, в общем-то, бесполезна в сценариях. Вместо нее лучше использовать [операторы проверки файлов](#).

dd

Эта немного непонятная и "страшная" команда ("data duplicator") изначально использовалась для переноса данных на магнитной ленте между микрокомпьютерами с ОС UNIX и майнфреймами IBM. Команда **dd** просто создает копию файла (или stdin/stdout), выполняя по пути некоторые преобразования. Один из вариантов: преобразование из ASCII в EBCDIC, [\[36\]](#) `dd --help` выведет список возможных вариантов преобразований и опций этой мощной утилиты.

```
# Изучаем 'dd'.
```

```
n=3
p=5
input_file=project.txt
output_file=log.txt
```

```
dd if=$input_file of=$output_file bs=1 skip=$((n-1)) count=$((p-n+1)) 2>
/dev/null
```

```
# Извлекает из $input_file символы с n-го по p-й.
```



```
echo -n "hello world" | dd cbs=1 conv=unblock 2> /dev/null
# Выведет "hello world" вертикально.
```

```
# Спасибо, S.C.
```

Для демонстрации возможностей **dd**, попробуем перехватить нажатия на клавиши.

Пример 12-41. Захват нажатых клавиш

```
#!/bin/bash
# Захват нажатых клавиш.

keypresses=4                # Количество фиксируемых нажатий.

old_tty_setting=$(stty -g)   # Сохранить настройки терминала.

echo "Нажмите $keypresses клавиши."
stty -icanon -echo          # Запретить канонический режим.
                             # Запретить эхо-вывод.
keys=$(dd bs=1 count=$keypresses 2> /dev/null)
# 'dd' использует stdin, если "if" не задан.

stty "$old_tty_setting"     # Восстановить настройки терминала.

echo "Вы нажали клавиши \"$keys\"."

# Спасибо S.C.
exit 0
```

Команда **dd** имеет возможность произвольного доступа к данным в потоке.

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc
# Здесь, опция "conv=notrunc" означает, что выходной файле будет усечен.

# Спасибо, S.C.
```

Команда **dd** может использоваться для создания образов дисков, считывая данные прямо с устройств, таких как дискеты, компакт диски, магнитные ленты ([Пример А-6](#)). Обычно она используется для создания загрузочных дискет.

```
dd if=kernel-image of=/dev/fd0H1440
```

Точно так же, **dd** может скопировать все содержимое дискеты, даже с неизвестной файловой системой, на жесткий диск в виде файла-образа.

```
dd if=/dev/fd0 of=/home/bozo/projects/floppy.img
```

Еще одно применение **dd** -- создание временного swar-файла ([Пример 28-2](#)) и гат-дисков ([Пример 28-3](#)). Она может создавать даже образы целых разделов жесткого диска, хотя и не рекомендуется делать это без особой на то необходимости.

Многие (которые, вероятно, не знают чем себя занять) постоянно придумывают все новые и новые области применения команды **dd**.

Пример 12-42. Надежное удаление файла

```

#!/bin/bash
# blotout.sh: Надежно удаляет файл.

# Этот сценарий записывает случайные данные в заданный файл,
#+ затем записывает туда нули и наконец удаляет файл.
# После такого удаления даже анализ дисковых секторов
#+ не даст ровным счетом ничего.

PASSES=7          # Количество проходов по файлу.
BLOCKSIZE=1      # операции ввода/вывода в/из /dev/urandom требуют указания
размера блока,
                #+ иначе вы не получите желаемого результата.
E_BADARGS=70
E_NOT_FOUND=71
E_CHANGED_MIND=72

if [ -z "$1" ]   # Имя файла не указано.
then
    echo "Порядок использования: `basename $0` filename"
    exit $E_BADARGS
fi

file=$1

if [ ! -e "$file" ]
then
    echo "Файл \"$file\" не найден."
    exit $E_NOT_FOUND
fi

echo; echo -n "Вы совершенно уверены в том, что желаете уничтожить \"$file\" (y/
n)? "
read answer
case "$answer" in
[nN]) echo "Передумали? Операция отменена."
        exit $E_CHANGED_MIND
        ;;
*)    echo "Уничтожается файл \"$file\".";;
esac

flength=$(ls -l "$file" | awk '{print $5}') # Поле с номером 5 -- это длина
файла.

pass_count=1

echo

while [ "$pass_count" -le "$PASSES" ]
do
    echo "Проход #$pass_count"
    sync          # Вытолкнуть буферы.
    dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
                # Заполнить файл случайными данными.
    sync          # Снова вытолкнуть буферы.
    dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
                # Заполнить файл нулями.
    sync          # Снова вытолкнуть буферы.
    let "pass_count += 1"
    echo
done

rm -f $file     # Наконец удалить изрядно "подпорченный" файл.
sync           # Вытолкнуть буферы в последний раз.

echo "Файл \"$file\" уничтожен."; echo

```

```

# Это довольно надежный, хотя и достаточно медленный способ уничтожения файлов.
#+ Более эффективно это делает команда "shred",
#+ входящая в состав пакета GNU "fileutils".

# Уничтоженный таким образом файл, не сможет быть восстановлен обычными
методами.
# Однако...
#+ эта метода вероятно НЕ сможет противостоять аналитическим службам
#+ из СООТВЕТСТВУЮЩИХ ОРГАНОВ

# Tom Vier разработал пакет "wipe", который более надежно стирает файлы
#+ чем этот простой сценарий.
# http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2

# Для более глубоко изучения проблемы надежного удаления файлов,
#+ рекомендую обратиться к cnfnmt Peter Gutmann,
#+ "Secure Deletion of Data From Magnetic and Solid-State Memory".
# http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html

exit 0

```

od

Команда **od** (*octal dump*) производит преобразование ввода (или файла) в один или несколько форматов, в соответствии с указанными опциями. При отсутствии опций используется восьмеричный формат (опция -o). Эта команда полезна при просмотре или обработке файлов с двоичными данными, например /dev/urandom. См. [Пример 9-26](#) и [Пример 12-10](#).

hexdump

Выводит дампы двоичных данных из файла в восьмеричном, шестнадцатиричном, десятичном виде или в виде ASCII. Эту команду, с массой оговорок, можно назвать эквивалентом команды **of od**.

objdump

Отображает содержимое исполняемого или объектного файла либо в шестнадцатиричной форме, либо в виде дизассемблерного листинга (с ключом -d).

```

bash$ objdump -d /bin/ls
/bin/ls:      file format elf32-i386

Disassembly of section .init:

080490bc <.init>:
 80490bc:      55                push   %ebp
 80490bd:      89 e5            mov   %esp,%ebp
  . . .

```

mcookie

Эта команда создает псевдослучайные шестнадцатиричные 128-битные числа, так называемые "magic cookie", обычно используется X-сервером в качестве "сигнатуры" авторизации. В сценариях может использоваться как малоэффективный генератор случайных чисел.

```
random000=`mcookie | sed -e '2p'`  
# 'sed' удаляет посторонние символы.
```

Конечно, для тех же целей, сценарий может использовать [md5](#).

```
# Сценарий вычисляет контрольную сумму для самого себя.  
random001=`md5sum $0 | awk '{print $1}'`  
# 'awk' удаляет имя файла.
```

С помощью **mcookie** можно создавать "уникальные" имена файлов.

Пример 12-43. Генератор имен файлов

```
#!/bin/bash  
# tempfile-name.sh: Генератор имен временных файлов  
  
BASE_STR=`mcookie` # 32-символьный (128 бит) magic cookie.  
POS=11 # Произвольная позиция в строке magic cookie.  
LEN=5 # $LEN последовательных символов.  
  
prefix=temp # В конце концов это временный ("temp") файл.  
  
suffix=${BASE_STR:POS:LEN}  
# Извлечь строку, длиной в 5 символов, начиная с позиции  
11.  
  
temp_filename=$prefix.$suffix  
# Сборка имени файла.  
  
echo "Имя временного файла = \"$temp_filename\""  
  
# sh tempfile-name.sh  
# Имя временного файла = temp.e19ea  
  
exit 0
```

units

Эта утилита производит преобразование величин из одних единиц измерения в другие. Как правило вызывается в интерактивном режиме, ниже приводится пример использования **units** в сценарии.

Пример 12-44. Преобразование метров в мили

```
#!/bin/bash  
# unit-conversion.sh  
  
convert_units () # Принимает в качестве входных параметров единицы измерения.  
{  
  cf=$(units "$1" "$2" | sed --silent -e '1p' | awk '{print $2}')  
  # Удаляет все кроме коэффициентов преобразования.  
  echo "$cf"  
}  
  
Unit1=miles  
Unit2=meters  
cfactor=`convert_units $Unit1 $Unit2`  
quantity=3.73
```

```

result=$(echo $quantity*$cfactor | bc)

echo "В $quantity милях $result метров."

# Что произойдет, если в функцию передать несовместимые единицы измерения,
#+ например "acres" (акры) and "miles" (мили)?

exit 0

```

m4

Не команда, а клад, **m4** -- это мощный фильтр обработки макроопределений, [\[37\]](#) фактически -- целый язык программирования. Изначально создававшаяся как препроцессор для *RatFor*, **m4** оказалась очень полезной и как самостоятельная утилита. Фактически, **m4** сочетает в себе функциональные возможности [eval](#), [tr](#), [awk](#), и дополнительно предоставляет обширные возможности по созданию новых макроопределений.

В апрельском выпуске, за 2002 год, журнала [Linux Journal](#) вы найдете замечательную статью, описывающую возможности утилиты **m4**.

Пример 12-45. Пример работы с m4

```

#!/bin/bash
# m4.sh: Демонстрация некоторых возможностей макропроцессора 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

# Арифметика
echo "incr(22)" | m4                   # 23
echo "eval(99 / 3)" | m4               # 33

exit 0

```

doexec

Команда **doexec** предоставляет возможность передачи произвольного списка аргументов внешней программе. В частности, передавая `argv [0]` (для сценариев соответствует специальной переменной `$0`), можно вызвать программу под другим именем, определяя тем самым, ее реакцию.

Например, Пусть в каталоге `/usr/local/bin` имеется программа с именем "aaa", которая при вызове **doexec /usr/local/bin/aaa list** выведет список всех файлов в текущем каталоге, имена которых начинаются с символа "a", а при вызове той же самой программы как **doexec /usr/local/bin/aaa delete**, она удалит эти файлы.



Естественно, реакция программы на свое собственное имя должна быть реализована в коде программы, для сценария на языке командной оболочки это может выглядеть примерно так:

```

case `basename $0` in
"name1" ) реакция на вызов под именем name1;;
"name2" ) реакция на вызов под именем name2;;
"name3" ) реакция на вызов под именем name3;;
*       ) действия по-умолчанию;;
esac

```

Глава 13. Команды системного администрирования

Примеры использования большинства этих команд вы найдете в сценариях начальной загрузки и остановки системы, в каталогах `/etc/rc.d`. Они, обычно, вызываются пользователем `root` и используются для администрирования системы или восстановления файловой системы. Эти команды должны использоваться с большой осторожностью, так как некоторые из них могут разрушить систему, при неправильном использовании.

Пользователи и группы

users

Выведет список всех зарегистрировавшихся пользователей. Она, до некоторой степени, является эквивалентом команды `who -q`.

groups

Выводит список групп, в состав которых входит текущий пользователь. Эта команда соответствует внутренней переменной `$GROUPS`, но выводит названия групп, а не их числовые идентификаторы.

```
bash$ groups
bozita cdrom cdwriter audio xgrp
```

```
bash$ echo $GROUPS
501
```

chown, chgrp

Команда `chown` изменяет владельца файла или файлов. Эта команда полезна в случаях, когда `root` хочет передать монопольное право на файл от одного пользователя другому. Обычный пользователь не в состоянии изменить владельца файла, за исключением своих собственных файлов.

```
root# chown bozo *.txt
```

Команда `chgrp` изменяет группу, которой принадлежит файл или файлы. Чтобы изменить группу, вы должны быть владельцем файла (при этом должны входить в состав указываемой группы) или привилегированным пользователем (`root`).

```
chgrp --recursive dunderheads *.data
# Группа "dunderheads" станет владельцем всех файлов "*.data"
#+ во всех подкаталогах текущей директории ($PWD) (благодаря ключу "--recursive").
```

useradd, userdel

Команда **useradd** добавляет учетную запись нового пользователя в систему и создает домашний каталог для данного пользователя. Противоположная, по смыслу, команда **userdel** удаляет учетную запись пользователя из системы. [\[38\]](#) и удалит соответствующие файлы.



Команда **adduser** является синонимом для **useradd** и, как правило, является обычной символической ссылкой на **useradd**.

id

Команда **id** выводит идентификатор пользователя (реальный и эффективный) и идентификаторы групп, в состав которых входит пользователь. По сути -- выводит содержимое переменных [\\$UID](#), [\\$EUID](#) и [\\$GROUPS](#).

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)

bash$ echo $UID
501
```

См. также [Пример 9-5](#).

who

Выводит список пользователей, работающих в настоящий момент в системе.

```
bash$ who
bozo  tty1      Apr 27 17:45
bozo  pts/0      Apr 27 17:46
bozo  pts/1      Apr 27 17:47
bozo  pts/2      Apr 27 17:49
```

С ключом **-m** -- выводит информацию только о текущем пользователе. Если число аргументов, передаваемых команде, равно двум, то это эквивалентно вызову **who -m**, например **who am i** или **who The Man**.

```
bash$ who -m
localhost.localdomain!bozo pts/2 Apr 27 17:49
```

whoami -- похожа на **who -m**, но выводит только имя пользователя.

```
bash$ whoami
bozo
```

w

Выводит информацию о системе, список пользователей, подключенных к системе и процессы, связанные с пользователями. Это расширенная версия команды **who**. Вывод от команды **w** может быть передан по конвейеру команде **grep**, с целью поиска требуемого пользователя и/или процесса.

```
bash$ w | grep startx
bozo  tty1      -
```

```
4:22pm  6:41    4.47s  0.45s  startx
```

logname

Выводит имя текущего пользователя (из файла `/var/run/utmp`). Это довольно близкий эквивалент команды [whoami](#).

```
bash$ logname
bozo
```

```
bash$ whoami
bozo
```

Однако...

```
bash$ su
Password: .....
```

```
bash# whoami
root
bash# logname
bozo
```

su

Команда предназначена для запуска программы или сценария от имени другого пользователя. **su rjones** -- запускает командную оболочку от имени пользователя *rjones*. Запуск команды **su** без параметров означает запуск командной оболочки от имени привилегированного пользователя *root*. См. [Пример А-17](#).

sudo

Исполняет заданную команду от имени пользователя *root* (или другого пользователя).

```
#!/bin/bash

# Доступ к "секретным" файлам.
sudo cp /root/secretfile /home/bozo/secret
```

Имена пользователей, которым разрешено использовать команду **sudo**, хранятся в файле `/etc/sudoers`.

passwd

Устанавливает или изменяет пароль пользователя.

Команда **passwd** может использоваться в сценариях, но это плохая практика.

```
#!/bin/bash
# set-new-password.sh: Плохая идея.
# Этот сценарий должен запускаться пользователем root,
#+ а еще лучше -- не запускать его вообще.

ROOT_UID=0      # $UID root = 0.
E_WRONG_USER=65 # He root?

if [ "$UID" -ne "$ROOT_UID" ]
```



```
then
  echo; echo "Только root может запускать этот сценарий."; echo
  exit $E_WRONG_USER
else
  echo; echo "Вам не следовало бы запускать этот сценарий."
fi
```

```
username=bozo
NEWPASSWORD=security_violation

echo "$NEWPASSWORD" | passwd --stdin "$username"
# Ключ '--stdin' указывает 'passwd'
#+ получить новый пароль со stdin (или из конвейера).

echo; echo "Пароль пользователя $username изменен!"

# Использование команды 'passwd' в сценариях -- опасно.

exit 0
```

ac

Выводит время работы пользователей, основываясь на записях в файле `/var/log/wtmp`. Это одна из утилит пакета GNU `acct`.

```
bash$ ac
      total          68.08
```

last

Выводит информацию о *последних* входах/выходах пользователей в систему, основываясь на записях в файле `/var/log/wtmp`. Эта команда может отображать информацию об удаленных (в смысле -- с удаленного терминала) соединениях.

newgrp

Позволяет сменить активную группу пользователя. Пользователь остается в системе и текущий каталог не изменяется, но права доступа к файлам вычисляются в соответствии с новыми реальным и эффективным идентификаторами группы. Эта команда используется довольно редко, так как пользователь, обычно, является членом нескольких групп.

Терминалы

tty

Выводит имя терминала текущего пользователя. Обратите внимание: каждое отдельное окно `xterm` считается отдельным терминалом.

```
bash$ tty
/dev/pts/1
```

stty

Выводит и/или изменяет настройки терминала. Эта сложная команда используется в сценариях для управления поведением терминала.

Пример 13-1. Установка символа "забоя"

```
#!/bin/bash
# erase.sh: Использование команды "stty" для смены клавиши "забоя" при чтении
ввода.

echo -n "Как Вас зовут? "
read name # Попробуйте стереть последние символы при вводе.
# Все работает.

echo "Вас зовут $name."

stty erase '#' # Теперь, чтобы стереть символ нужно использовать
клавишу "#".
echo -n "Как Вас зовут? "
read name # Попробуйте стереть последние символы при вводе
с помощью "#".
echo "Вас зовут $name."

exit 0
```

Пример 13-2. невидимый пароль: Отключение эхо-вывода на терминал

```
#!/bin/bash

echo
echo -n "Введите пароль "
read passwd
echo "Вы ввели пароль: $passwd"
echo -n "Если кто-нибудь в это время заглядывал Вам через плечо, "
echo "то теперь он знает Ваш пароль."

echo && echo # Две пустых строки через "and list".

stty -echo # Отключить эхо-вывод.

echo -n "Введите пароль еще раз "
read passwd
echo
echo "Вы ввели пароль: $passwd"
echo

stty echo # Восстановить эхо-вывод.

exit 0
```

Перехват нажатия на клавиши с помощью **stty**.

Пример 13-3.

```
#!/bin/bash
# keypress.sh: Определение нажатых клавиш.

echo

old_tty_settings=$(stty -g) # Сохранить прежние настройки.
stty -icanon
Keypress=$(head -c1) # или $(dd bs=1 count=1 2> /dev/null)
# для других, не GNU, систем

echo
echo "Была нажата клавиша \"\"$Keypress\"\"."
echo

stty "$old_tty_settings" # Восстановить прежние настройки.

# Спасибо, Stephane Chazelas.
```

```
exit 0
```

См. также [Пример 9-3](#).

терминалы и их режимы работы

Как правило, терминалы работают в *каноническом* режиме. Когда пользователь нажимает какую-либо клавишу, то соответствующий ей символ не сразу передается программе, исполняемой в окне терминала. Этот символ поступает сначала в локальный буфер терминала. Когда пользователь нажимает клавишу **ENTER**, то тогда все содержимое буфера передается программе.

```
bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = <undef>; eol2 =
<undef>;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
...
isig icanon ixexten echo echoe echok -echonl -noflsh -xcase -tostop -echopr
```

В каноническом режиме можно использовать символы редактирования во время ввода.

```
bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>
<ctl-D>
bash$ cat filexxx
hello world
bash$ bash$ wc -c < file
13
```

Процесс в терминале получит только 13 символов (12 алфавитных символов и символ перевода строки), хотя пользователь нажал 26 клавиш.

В неканоническом ("сыром") режиме, каждая нажатая клавиша (включая специальные символы редактирования, такие как **ctl-H**) сразу же передается исполняющемуся в терминале процессу.

Под управлением Bash, базовый терминальный редактор заменяется более сложным терминальным редактором Bash. Например, если вы нажмете комбинацию клавиш **ctl-A** в командной строке Bash, то вы не увидите символов **^A**, которые выводит терминал, вместо этого Bash получит символ **\1**, проанализирует его и переместит курсор в начало строки.

Stephane Chazelas

tset

Выводит или изменяет настройки терминала. Это более слабая версия **stty**.

```
bash$ tset -r
Terminal type is xterm-xfree86.
Kill is control-U (^U).
Interrupt is control-C (^C).
```

setserial

Настройка параметров последовательного порта. Эта команда должна запускаться пользователем, обладающим привилегиями `root`. Эту команду можно встретить в сценариях настройки системы.


```
# Взято из /etc/pcmcia/serial :  
  
IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`  
setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty,agetty

Программа **getty** или **agetty** запускается процессом `init` и обслуживает процедуру входа пользователя в систему. Эти команды не используются в сценариях.

mesg

Разрешает или запрещает доступ к терминалу текущего пользователя командой [write](#).


-  Наверное это очень неприятно, когда, во время работы над текстовым файлом, в окне терминала, прямо среди текста, вдруг появляется предложение заказать пиццу. Поэтому, при работе в многопользовательской системе, вам наверняка захочется отключить доступ к своему терминалу.

wall

Имя этой команды -- аббревиатура от "[write](#) all", т.е., передать сообщение всем пользователям на все терминалы в сети. Это, в первую очередь, инструмент администратора, который можно использовать, например, для оповещения всех пользователей о предстоящей, в ближайшее время, перезагрузке системы (см. [Пример 17-2](#)).

```
bash$ wall System going down for maintenance in 5 minutes!  
Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...
```

```
System going down for maintenance in 5 minutes!
```

-  Если доступ к терминалу был закрыт командой **mesg**, то сообщение на этом терминале выводиться не будет.

dmesg

Выводит все сообщения, выдаваемые системой во время загрузки на `stdout`. Очень полезная утилита для отладочных целей. Вывод **dmesg** может анализироваться с помощью [grep](#), [sed](#) или [awk](#) внутри сценария.

```
bash$ dmesg | grep hda  
Kernel command line: ro root=/dev/hda2
```

```
hda: IBM-DLGA-23080, ATA DISK drive
hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
hda: hda1 hda2 hda3 < hda5 hda6 hda7 > hda4
```

Информационные и статистические утилиты

uname

Выводит на `stdout` имя системы. С ключом `-a`, выводит подробную информацию, содержащую имя системы, имя узла (то есть имя, под которым система известна в сети), версию операционной системы, наименование модификации операционной системы, аппаратную архитектуру (см. [Пример 12-4](#)).

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43 EST 2000 i686
unknown
```

```
bash$ uname -s
Linux
```

arch

Выводит тип аппаратной платформы компьютеров. Эквивалентна команде `uname -m`. См. [Пример 10-26](#).

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

Выводит информацию, о ранее выполненных командах, из файла `/var/account/pacct`. Дополнительно могут указываться команда и пользователь. Это одна из утилит пакета GNU `acct`.

lastlog

Выводит список всех пользователей, с указанием времени последнего входа в систему. Данные берутся из файла `/var/log/lastlog`.

```
bash$ lastlog
root          tty1          Fri Dec  7 18:43:21 -0700 2001
bin           **Never logged in**
daemon       **Never logged in**
...
bozo         tty1          Sat Dec  8 21:14:29 -0700 2001

bash$ lastlog | grep root
root          tty1          Fri Dec  7 18:43:21 -0700 2001
```



Исполнение этой команды будет завершаться неудачей, если

пользователь, вызвавший утилиту, не имеет прав на чтение файла /var/log/lastlog.

lsdf

Выводит детальный список открытых, в настоящий момент времени, файлов в виде таблицы. В таблице указаны -- владелец файла, размер файла, тип файла, процесс, открывший файл, и многое другое. Само собой разумеется, что вывод команды **lsdf** может быть обработан, в конвейере, с помощью утилит [grep](#) и/или [awk](#).

```
bash$ lsdf
COMMAND  PID    USER  FD  TYPE  DEVICE  SIZE  NODE NAME
init      1     root  mem  REG   3,5    30748 30303 /sbin/init
init      1     root  mem  REG   3,5    73120 8069  /lib/ld-2.1.3.so
init      1     root  mem  REG   3,5    931668 8075
/lib/libc-2.1.3.so
cardmgr   213   root  mem  REG   3,5    36956 30357 /sbin/cardmgr
...
```

strace

Диагностическая и отладочная утилита, предназначенная для трассировки системных вызовов и сигналов. В простейшем случае, запускается как: **strace COMMAND**.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
uname({sys="Linux", node="bozo.localdomain", ...}) = 0
brk(0) = 0x804f5e4
...
```

Эквивалентна команде **truss**.

nmap

Сканер сетевых портов. Эта утилита сканирует сервер в поисках открытых портов и сервисов. Это очень важный инструмент, используемый для поиска уязвимостей при настройке системы.

```
#!/bin/bash

SERVER=$HOST # localhost.localdomain (127.0.0.1).
PORT_NUMBER=25 # порт службы SMTP.

nmap $SERVER | grep -w "$PORT_NUMBER" # Проверить -- открыт ли данный порт?
# grep -w -- поиск только целых слов,
#+ так, например, порт 1025 будет пропущен.

exit 0

# 25/tcp open smtp
```

free

Показывает информацию об использовании памяти, в табличной форме. Вывод команды может быть проанализирован с помощью [grep](#), [awk](#) или **Perl**. Команда **procinfo** тоже выводит эту информацию, среди всего прочего.

```
bash$ free
              total        used        free      shared    buffers     cached
Mem:          30504        28624        1880       15820       1608       16376
-/+ buffers/cache:    10640        19864
Swap:         68540         3128        65412
```

Показать размер неиспользуемой памяти RAM:

```
bash$ free | grep Mem | awk '{ print $4 }'
1880
```

procinfo

Извлекает и выводит информацию из [файловой системы /proc](#).

```
bash$ procinfo | grep Bootup
Bootup: Wed Mar 21 15:15:50 2001      Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev

Список аппаратных устройств в системе.

```
bash$ lsdev
Device          DMA   IRQ  I/O Ports
-----
cascade         4     2
dma
dma1             0080-008f
dma2             0000-001f
                 00c0-00df
fpu              00f0-00ff
ide0             14   01f0-01f7 03f6-03f6
...
```

du

Выводит сведения о занимаемом дисковом пространстве в каталоге и вложенных подкаталогах. Если каталог не указан, то по-умолчанию выводятся сведения о текущем каталоге.

```
bash$ du -ach
1.0k    ./wi.sh
1.0k    ./tst.sh
1.0k    ./random.file
6.0k    .
6.0k    total
```

df

Выводит в табличной форме сведения о смонтированных файловых системах.

```
bash$ df
Filesystem      1k-blocks      Used Available Use% Mounted on
/dev/hda5        273262        92607   166547   36% /
/dev/hda8        222525        123951   87085    59% /home
/dev/hda7        1408796       1075744  261488   80% /usr
```

stat

Дает подробную информацию о заданном файле (каталоге или файле устройства) или наборе файлов.

```
bash$ stat test.cru
File: "test.cru"
  Size: 49970      Allocated Blocks: 100      Filetype: Regular File
  Mode: (0664/-rw-rw-r--)      Uid: ( 501/ bozo)  Gid: ( 501/ bozo)
Device: 3,8      Inode: 18185      Links: 1
Access: Sat Jun  2 16:40:24 2001
Modify: Sat Jun  2 16:40:24 2001
Change: Sat Jun  2 16:40:24 2001
```

Если заданный файл отсутствует, то **stat** вернет сообщение об ошибке.

```
bash$ stat nonexistent-file
nonexistent-file: No such file or directory
```

vmstat

Выводит информацию о виртуальной памяти.

```
bash$ vmstat
procs
 r  b  w  swpd  free  buff  cache  si  so  bi  bo  in  cs  us  sy  id
 0  0  0    0 11040 2636 38952  0  0 33   7 271  88  8  3 89
```

netstat

Показывает сведения о сетевой подсистеме, такие как: таблицы маршрутизации и активные соединения. Эта утилита получает сведения из /proc/net ([Глава 27](#)). См. [Пример 27-2](#).

netstat -r -- эквивалентна команде [route](#).

uptime

Показывает количество времени, прошедшего с момента последней перезагрузки системы.

```
bash$ uptime
10:28pm up 1:57, 3 users, load average: 0.17, 0.34, 0.27
```

hostname

Выводит имя узла (сетевое имя системы). С помощью этой команды устанавливается сетевое имя системы в сценарии /etc/rc.d/rc.sysinit. Эквивалентна команде **uname -n** и внутренней переменной [\\$HOSTNAME](#).

```
bash$ hostname
```



```
localhost.localdomain
```

```
bash$ echo $HOSTNAME
localhost.localdomain
```

hostid

Выводит 32-битный шестнадцатиричный идентификатор системы.

```
bash$ hostid
7f0100
```



Эта команда генерирует "уникальный" числовой идентификатор системы. Некоторые программные продукты используют этот идентификатор в процедуре регистрации. К сожалению, при генерации идентификатора, **hostid** использует только IP адрес системы, переводя его в шестнадцатиричное представление и переставляя местами пары байт.

Обычно, IP адрес системы можно найти в файле `/etc/hosts`.

```
bash$ cat /etc/hosts
127.0.0.1          localhost.localdomain localhost
```

Переставив местами байты, попарно, в начальном адресе `127.0.0.1`, мы получим `0.127.1.0`, в шестнадцатиричном представлении это будет `007f0100`, что в точности совпадает с приведенным выше результатом выполнения **hostid**. Наверняка можно найти несколько миллионов компьютеров с таким же "уникальным" идентификатором.

sar

Команда **sar** (system activity report) выводит очень подробную статистику о функционировании операционной системы. Эту команду можно найти в отдельных коммерческих дистрибутивах UNIX-систем. Она, как правило, не входит в базовый комплект пакетов Linux-систем. Она входит в состав пакета [sysstat utilities](#), автор: [Sebastien Godard](#).

```
bash$ sar
Linux 2.4.7-10 (localhost.localdomain)      12/31/2001

10:30:01 AM      CPU      %user      %nice      %system      %idle
10:40:00 AM      all       1.39       0.00       0.77       97.84
10:50:00 AM      all      76.83       0.00       1.45       21.72
11:00:00 AM      all       1.32       0.00       0.69       97.99
11:10:00 AM      all       1.17       0.00       0.30       98.53
11:20:00 AM      all       0.51       0.00       0.30       99.19
06:30:00 PM      all     100.00       0.00     100.01        0.00
Average:         all       1.39       0.00       0.66       97.95
```

readelf

Показывает сведения о заданном бинарном файле формата *elf*. Входит в состав пакета *binutils*.

```
bash$ readelf -h /bin/bash
```

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00
  Class:                   ELF32
  Data:                     2's complement, little endian
  Version:                  1 (current)
  OS/ABI:                   UNIX - System V
  ABI Version:              0
  Type:                     EXEC (Executable file)
  . . .
```

size

Команда **size** `[/path/to/binary]` выведет информацию о размерах различных сегментов в исполняемых или библиотечных файлах. В основном используется программистами.

```
bash$ size /bin/bash
  text  data  bss   dec   hex filename
495971 22496 17392 535859 82d33 /bin/bash
```

Системный журнал

logger

Добавляет в системный журнал (`/var/log/messages`) сообщение от пользователя. Для добавления сообщения пользователь не должен обладать привилегиями суперпользователя.

```
logger Experiencing instability in network connection at 23:10, 05/21.
# Теперь попробуйте дать команду 'tail /var/log/messages'.
```

Встраивая вызов **logger** в сценарии, вы получаете возможность заносить отладочную информацию в системный журнал `/var/log/messages`.

```
logger -t $0 -i Logging at line "$LINENO".
# Ключ "-t" задает тэг записи в журнале.
# Ключ "-i" -- записывает ID процесса.

# tail /var/log/message
# ...
# Jul  7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.
```

logrotate

Эта утилита производит манипуляции над системным журналом: ротация, сжатие, удаление и/или отправляет его по электронной почте, по мере необходимости. Как правило, утилита **logrotate** вызывается демоном [crond](#) ежедневно.

Добавляя соответствующие строки в `/etc/logrotate.conf`, можно заставить **logrotate** обрабатывать не только системный журнал, но и ваш личный.

Управление заданиями

ps

Process Statistics: Список исполняющихся в данный момент процессов. Обычно вызывается с ключами `ax`, вывод команды может быть обработан командами [grep](#) или [sed](#), с целью поиска требуемого процесса (см. [Пример 11-10](#) и [Пример 27-1](#)).

```
bash$ ps ax | grep sendmail
295 ?      S        0:00 sendmail: accepting connections on port 25
```

pstree

Список исполняющихся процессов в виде "дерева". С ключом `-p` -- вместе с именами процессов отображает их PID.

top

Выводит список наиболее активных процессов. С ключом `-b` -- отображение ведется в обычном текстовом режиме, что дает возможность анализа вывода от команды внутри сценария.

```
bash$ top -b
 8:30pm up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
 45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
Mem:      78396K av,   65468K used,   12928K free,        0K shrd,    2352K buff
Swap:    157208K av,        0K used,  157208K free          37244K cached

  PID USER      PRI  NI  SIZE  RSS  SHARE STAT %CPU %MEM    TIME COMMAND
  848 bozo      17   0   996   996    800 R    5.6  1.2    0:00 top
    1 root       8   0   512   512    444 S    0.0  0.6    0:04 init
    2 root       9   0     0     0     0  SW   0.0  0.0    0:00 keventd
  ...
```

nice

Запускает фоновый процесс с заданным приоритетом. Приоритеты могут задаваться числом из диапазона от 19 (низший приоритет) до -20 (высший приоритет). Но только *root* может указать значение приоритета меньше нуля (отрицательные значения). См. так же команды **renice**, **snice** и **skill**.

nohup

Запуск команд в режиме игнорирования сигналов прерывания и завершения, что предотвращает завершение работы команды даже если пользователь, запустивший ее, вышел из системы. Если после команды не указан символ `&`, то она будет исполняться как процесс "переднего плана". Если вы собираетесь использовать **nohup** в сценариях, то вам потребуется использовать его в связке с командой [wait](#), чтобы не породить процесс "зомби".

pidof

Возвращает идентификатор процесса (*pid*) по его имени. Поскольку многие команды управления процессами, такие как [kill](#) и **renice**, требуют указать *pid* процесса, а не его имя, то **pidof** может сослужить неплохую службу при идентификации процесса по его имени. Эта команда может рассматриваться как приблизительный эквивалент внутренней переменной [\\$PPID](#).

```
bash$ pidof xclock
880
```

Пример 13-4. Использование команды pidof при остановке процесса

```
#!/bin/bash
# kill-process.sh

NOPROCESS=2

process=xxxuyzzz # Несуществующий процесс.
# Только в демонстрационных целях...
# ... чтобы не уничтожить этим сценарием какой-нибудь процесс.
#
# Если с помощью этого сценария вы задумаете разрывать связь с Internet, то
#   process=pppd

t=`pidof $process`      # Поиск pid (process id) процесса $process.
# pid требует команда 'kill' (невозможно остановить процесс, указав его имя).

if [ -z "$t" ]          # Если процесс с таким именем не найден, то 'pidof'
вернет null.
then
    echo "Процесс $process не найден."
    exit $NOPROCESS
fi

kill $t                 # В некоторых случаях может потребоваться 'kill -9'.

# Здесь нужно проверить -- был ли уничтожен процесс.
# Возможно так: " t=`pidof $process` ".

# Этот сценарий мог бы быть заменен командой
#   kill $(pidof -x process_name)
# но это было бы не так поучительно.

exit 0
```

fuser

Возвращает идентификаторы процессов, использующих указанный файл(ы) или каталог. С ключом -k, завершает найденные процессы. Может с успехом использоваться для защиты системы, особенно в сценариях разграничения доступа к системным службам.

crond

Планировщик заданий. С его помощью выполняются такие задачи, как очистка и удаление устаревших файлов системных журналов, обновление базы данных slocate. Это суперпользовательская версия команды [at](#) (хотя любой пользователь может создать собственную таблицу crontab). Эта утилита запускается как фоновый процесс-[daemon](#) и выполняет задания, находящиеся в файле /etc/crontab.

Команды управления процессами и загрузкой

init

init -- [предок \(родитель\)](#) всех процессов в системе. Вызывается на последнем этапе

загрузки системы и определяет уровень загрузки (runlevel) из файла /etc/inittab.

telinit

Символическая ссылка на **init** -- инструмент для смены уровня загрузки (runlevel), как правило используется при обслуживании системы или восстановлении файловой системы. Может быть вызвана только суперпользователем. Эта команда может быть очень опасна, при неумелом обращении -- прежде чем использовать ее, убедитесь в том, что вы совершенно точно понимаете что делаете!

runlevel

Выводит предыдущий и текущий уровни загрузки (runlevel). Уровень загрузки может иметь одно из 6 значений: 0 -- остановка системы, 1 -- однопользовательский режим, 2 или 3 -- многопользовательский режим, 5 -- многопользовательский режим и запуск X Window, 6 -- перезагрузка. Уровни загрузки определяются из файла /var/run/utmp.

halt, shutdown, reboot

Набор команд для остановки системы, обычно перед выключением питания.

Команды для работы с сетью

ifconfig

Утилита конфигурирования и запуска сетевых интерфейсов. Чаще всего используется в сценариях начальной загрузки системы, для настройки и запуска сетевых интерфейсов или для их остановки перед остановкой или перезагрузкой.

```
# Фрагменты кода из /etc/rc.d/init.d/network
# ...

# Проверка сетевой подсистемы.
[ ${NETWORKING} = "no" ] && exit 0

[ -x /sbin/ifconfig ] || exit 0

# ...

for i in $interfaces ; do
    if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
        action "Останавливается $i: " ./ifdown $i boot
    fi
# Ключ "-q", характерный для GNU-версии "grep", означает "quiet" ("молча"), т.е.
подавляет вывод.
# Поэтому нет необходимости переадресовывать вывод на /dev/null.

# ...

echo "В настоящее время активны устройства:"
echo ` /sbin/ifconfig | grep ^[a-z] | awk '{print $1}' `
#           ^^^^^^ скобки необходимы для предотвращения
подстановки имен файлов (globbing).
# Следующий код делает то же самое.
# echo $( /sbin/ifconfig | awk '/^[a-z]/ { print $1 } )'
# echo $( /sbin/ifconfig | sed -e 's/ .*//')
# Спасибо S.C. за комментарии.
```

См. также [Пример 29-6](#).

route

Выводит сведения о таблице маршрутизации ядра или вносит туда изменения.

```
bash$ route
Destination      Gateway           Genmask           Flags   MSS Window  irtt Iface
pm3-67.bozosisp *                  255.255.255.255 UH       40  0         0 ppp0
127.0.0.0        *                  255.0.0.0        U        40  0         0 lo
default          pm3-67.bozosisp  0.0.0.0          UG       40  0         0 ppp0
```

chkconfig

Проверка сетевой конфигурации. Обслуживает список, запускаемых на этапе загрузки, сетевых сервисов, список сервисов хранится в каталогах `/etc/rc?.d` (строго говоря, `chkconfig` работает не только с сетевыми сервисами, а с сервисами вообще, не зависимо от того сетевые это службы или нет. прим. перев.).

Изначально эта утилита была перенесена в Red Hat Linux из ОС IRIX, **chkconfig** входит в состав далеко не всех дистрибутивов Linux.

```
bash$ chkconfig --list
atd          0:off  1:off  2:off  3:on   4:on   5:on   6:off
rwhod        0:off  1:off  2:off  3:off  4:off  5:off  6:off
...
```

tcpdump

"Сниффер" ("sniffer") сетевых пакетов. Инструмент для перехвата и анализа сетевого трафика по определенным критериям.

Дамп трафика ip-пакетов между двумя узлами сети -- *bozoville* и *caduceus*:

```
bash$ tcpdump ip host bozoville and caduceus
```

Конечно же, вывод команды **tcpdump** может быть проанализирован с помощью [команд обработки текста](#), обсуждавшихся выше.

Команды для работы с файловыми системами

mount

Выполняет монтирование файловой системы, обычно на устройстве со сменными носителями, такими как дискеты или CDRом. Файл `/etc/fstab` содержит перечень доступных для монтирования файловых систем, разделов и устройств, включая опции монтирования, благодаря этому файлу, монтирование может производиться автоматически или вручеую. Файл `/etc/mtab` содержит список смонтированных файловых систем и разделов (включая виртуальные, такие как `/proc`).

mount -a -- монтирует все (all) файловые системы и разделы, перечисленные в

/etc/fstab, за исключением тех, которые имеют флаг noauto. Эту команду можно встретить в сценариях начальной загрузки системы из /etc/rc.d (rc.sysinit или нечто похожее).

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Монтирование CDROM-a
mount /mnt/cdrom
# Более короткий и удобный вариант, если точка монтирования /mnt/cdrom описана в
/etc/fstab
```

Эта команда может даже смонтировать обычный файл как блочное устройство. Достигается это за счет связывания файла с [loopback-устройством](#). Эту возможность можно использовать для проверки ISO9660 образа компакт-диска перед его записью на болванку. [\[39\]](#)

Пример 13-5. Проверка образа CD

```
# С правами root...

mkdir /mnt/cdtest # Подготовка точки монтирования.

mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest # Монтирование образа
диска.
#          ключ "-o loop" эквивалентен "losetup /dev/loop0"
cd /mnt/cdtest # Теперь проверим образ диска.
ls -alR       # Вывод списка файлов
```

umount

Отмонтирует смонтированную файловую систему. Перед тем как физически вынуть компакт-диск или дискету из устройства, это устройство должно быть отмонтировано командой **umount**, иначе файловая система может оказаться поврежденной (особенно это относится к накопителям на гибких магнитных дисках, прим. перев.).

```
umount /mnt/cdrom
# Теперь вы можете извлечь диск из привода.
```



Утилита **automount**, если она установлена, может выполнять автоматическое монтирование/размонтирование устройств со сменными носителями, такие как дискеты и компакт-диски. На ноутбуках со сменными устройствами FDD и CDROM, такой подход может привести к возникновению определенных проблем.

sync

Принудительный сброс содержимого буферов на жесткий диск (синхронизация содержимого буферов ввода-вывода и устройства-носителя). Несмотря на то, что нет такой уж острой необходимости в этой утилите, тем не менее **sync** придает уверенности системным администраторам или пользователям в том, что их данные будут сохранены на жестком диске, и не будут потеряны в случае какого-либо сбоя. В былые дни, команда `sync ; sync` (дважды -- для абсолютной уверенности) была упреждающей мерой перед перезагрузкой системы.

Иногда возникает необходимость принудительной синхронизации буферов ввода-вывода с содержимым на магнитном носителе, как, например, при надежном

удалении файла (см. [Пример 12-42](#)) или когда наблюдаются скачки напряжения в сети электроснабжения.

losetup

Устанавливает и конфигурирует [loopback-устройства](#).

Пример 13-6. Создание файловой системы в обычном файле

```
SIZE=1048576 # 1 Мб

head -c $SIZE < /dev/zero > file # Создается файл нужного размера.
losetup /dev/loop0 file          # Файл назначается как loopback-устройство.
mke2fs /dev/loop0                # Создание файловой системы.
mount -o loop /dev/loop0 /mnt    # Монтирование только что созданной файловой
системы.

# Спасибо S.C.
```

mkswap

Создание swap-раздела или swap-файла. Созданный swap-раздел (файл) нужно затем подключить командой **swapon**.

swapon, swaponoff

Разрешает/запрещает использование swap-раздела (файла). Эта команда обычно используется во время загрузки системы или во время остановки.

mke2fs

Создает файловую систему ext2. Должна вызываться с правами суперпользователя.

Пример 13-7. Добавление нового жесткого диска

```
#!/bin/bash

# Добавление в систему второго жесткого диска.
# Программное конфигурирование. Предполагается, что устройство уже подключено к
аппаратуре компьютера.
# Взято из статьи автора документа.
# "Linux Gazette", выпуск #38, http://www.linuxgazette.com.

ROOT_UID=0      # Этот сценарий должен запускать только root.
E_NOTROOT=67    # Код ошибки, если сценарий запущен простым пользователем.

if [ "$UID" -ne "$ROOT_UID" ]
then
    echo "Для запуска этого сценария вы должны обладать правами root."
    exit $E_NOTROOT
fi

# Будьте крайне осторожны!
# Если что-то пойдет не так, то вы можете потерять текущую файловую систему.

NEWDISK=/dev/hdb      # Предполагается, что /dev/hdb -- это новое устройство.
Проверьте!
MOUNTPOINT=/mnt/newdisk # Или выберите иное устройство для монтирования.

fdisk $NEWDISK
mke2fs -cv $NEWDISK1  # Проверка на "плохие" блоки (bad blocks) и подробный
```




```
Вывод.  
# Обратите внимание: /dev/hdb1, *не* то же самое, что /dev/hdb!  
mkdir $MOUNTPOINT  
chmod 777 $MOUNTPOINT # Сделать новое устройство доступным для всех  
пользователей.  
  
# Теперь проаерим...  
# mount -t ext2 /dev/hdb1 /mnt/newdisk  
# Попробуйте создать каталог.  
# Если получилось -- отмонтируйте устройство и продолжим.  
  
# Последний штрих:  
# Добавьте следующую строку в /etc/fstab.  
# /dev/hdb1 /mnt/newdisk ext2 defaults 1 1  
  
exit 0
```

См. также [Пример 13-6](#) и [Пример 28-3](#).

tune2fs

Настройка отдельных параметров файловой системы ext2, например счетчик максимального количества монтирований без проверки. Должна вызываться с привилегиями пользователя root.

 Очень опасная утилита. Вы можете использовать ее только на свой страх и риск, поскольку, по неосторожности, вы запросто можете разрушить файловую систему.

dumpe2fs

Выводит на stdout очень подробную информацию о файловой системе. Должна вызываться с привилегиями пользователя root.


```
root# dumpe2fs /dev/hda7 | grep 'ount count'  
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09  
Mount count: 6  
Maximum mount count: 20
```

hdparm

Выводит или изменяет параметры настройки жесткого диска. Должна вызываться с привилегиями пользователя root. Потенциально опасна при неправильном использовании.

fdisk

Создание или изменение таблицы разделов на устройствах хранения информации, обычно -- жестких дисках. Должна вызываться с привилегиями пользователя root.

 Пользуйтесь этой утилитой с особой осторожностью, т.к. при неправильном использовании можно легко разрушить существующую файловую систему.

fsck, e2fsck, debugfs

Набор команд для проверки, восстановления и отладки файловой системы.

fsck: интерфейсная утилита для проверки файловых систем в UNIX (может вызывать другие утилиты проверки).

e2fsck: проверка файловой системы ext2.

debugfs: отладчик файловой системы ext2. Одно из применений этой универсальной (и опасной) утилиты -- это восстановление удаленных файлов. Только для опытных пользователей!



Все эти утилиты должны вызываться с привилегиями пользователя root. При неправильном использовании, любая из них может разрушить файловую систему.

badblocks

Выполняет поиск плохих блоков (физические повреждения носителей) на устройствах хранения информации. Эта команда может использоваться для поиска плохих блоков при форматировании вновь устанавливаемых жестких дисков или для проверки устройств резервного копирования. [40] Например, **badblocks /dev/fd0**, проверит дискету на наличие поврежденных блоков.

Утилита **badblocks** может быть вызвана в деструктивном (проверка осуществляется путем записи некоторого шаблона в каждый блок, а затем производится попытка чтения этого блока) или в недеструктивном (неразрушающем -- только чтение) режиме.

mkbootdisk

Создание загрузочной дискеты, которая может быть использована для загрузки системы, если, например, была повреждена MBR (master boot record -- главная загрузочная запись). Команда **mkbootdisk** -- это сценарий на языке командной оболочки Bash, автор: Erik Troan, располагается в каталоге /sbin.

chroot

CHange ROOT -- смена корневого каталога. Обычно, команды и утилиты ориентируются в файловой системе посредством переменной **\$PATH**, относительно корневого каталога /. Команда **chroot** изменяет корневой каталог по-умолчанию на другой (рабочий каталог также изменяется). Эта утилита, как правило, используется с целью защиты системы, например, с ее помощью можно ограничить доступ к разделам файловой системы для пользователей, подключающихся к системе с помощью **telnet** (это называется -- "поместить пользователя в chroot окружение"). Обратите внимание: после выполнения команды **chroot** изменяется путь к исполняемому файлам системы.

Команда **chroot /opt** приведет к тому, что все обращения к каталогу /usr/bin будут переводиться на каталог /opt/usr/bin. Аналогично, **chroot /aaa/bbb /bin/ls** будет пытаться вызвать команду **ls** из каталога /aaa/bbb/bin, при этом, корневым каталогом для ls станет каталог /aaa/bbb. Поместив строчку **alias XX 'chroot /aaa/bbb ls'** в пользовательский ~/.bashrc, можно эффективно ограничить доступ команде "XX", запускаемой пользователем, к разделам файловой системы.



При изменении корневого каталога, вам наверняка потребуется скопировать системные утилиты и разделяемые библиотеки в новый корневой каталог, поскольку после смены корневого каталога, директории

с системными утилитами могут оказаться за пределами нового корневого каталога.

lockfile

Эта утилита входит в состав пакета **procmail** (www.procmail.org). Она создает *lock file*, файл-семафор (или, если угодно, файл блокировки), который управляет доступом к заданному файлу, устройству или ресурсу. Lock file служит признаком того, что данный файл, устройство или ресурс "занят" некоторым процессом, и ограничивает (или вообще запрещает) доступ к ресурсу другим процессам.

Файлы блокировок широко применяются для защиты системных почтовых каталогов от одновременной записи несколькими пользователями, для индикации занятости порта модема, и т.п. Сценарии могут использовать файлы блокировок для того, чтобы выяснить -- запущен ли тот или иной процесс. Обратите внимание: если в сценарии будет предпринята попытка создать файл блокировки, когда он уже существует, то такой сценарий скорее всего зависнет.

Как правило, файлы блокировки создаются в каталоге `/var/lock`. Проверка наличия файла блокировки может быть проверена примерно таким образом:.

```
appname=xyzip
# Приложение "xyzip" создает файл блокировки "/var/lock/xyzip.lock".

if [ -e "/var/lock/$appname.lock" ]
then
    ...
```

mknod

Создает специальный файл для блочного или символьного устройства (может потребоваться при установке новых устройств в компьютер).

tmpwatch

Автоматически удаляет файлы, к которым не было обращений в течение заданного периода времени. Обычно вызывается демоном [cron](#) для удаления устаревших файлов системного журнала.

MAKEDEV

Утилита предназначена для создания файлов-устройств. Должна запускаться с привилегиями пользователя `root`, в каталоге `/dev`.

```
root# ./MAKEDEV
```

Это своего рода расширенная версия утилиты **mknod**.

Команды резервного копирования

dump, restore

Команда **dump** создает резервные копии целых файловых систем, обычно используется в крупных системах и сетях. [\[41\]](#) Она считывает дисковые разделы и сохраняет их в файле, в двоичном формате. Созданные таким образом файлы,

могут быть сохранены на каком-либо носителе -- на жестком диске или магнитной ленте. Команда **restore** -- "разворачивает" файлы, созданные утилитой **dump**.

fdformat

Выполняет низкоуровневое форматирование дискет.

Команды управления системными ресурсами

ulimit

Устанавливает *верхний предел* для системных ресурсов. Как правило вызывается с ключом **-f**, что означает наложение ограничений на размер файлов (**ulimit -f 1000** ограничит размер вновь создаваемых файлов одним мегабайтом). Ключ **-c** ограничивает размер файлов **coredump** (**ulimit -c 0** запретит создание **coredump**-файлов). Обычно, все ограничения прописываются в файле `/etc/profile` и/или `~/.bash_profile` (см. [Глава 26](#)).



Грамотное использование **ulimit** поможет избежать нападений, целью которых является исчерпание системных ресурсов, известных под названием *fork bomb*.

```
#!/bin/bash

while 1      # Бесконечный цикл.
do
    $0 &    # Этот сценарий вызывает сам себя . . .
            #+ порождая дочерние процессы бесконечное число раз . . .
            #+ точнее -- до тех пор, пока не иссякнут системные
ресурсы.
done        # Это печально известный сценарий "sorcerer's apprentice".

exit 0      # Сценарий никогда не завершит свою работу.
```

Команда **ulimit -Hu XX** (где **XX** -- это верхний предел количества процессов, которые может запустить пользователь одновременно) в `/etc/profile` вызовет аварийное завершение этого сценария, когда количество процессов превысит установленный предел.

umask

Установка маски режима создания файлов. Накладывает ограничения на атрибуты по-умчанию для создаваемых файлов. Маска представляет собой восьмеричное значение и определяет запрещенные атрибуты файла. Например, **umask 022** удаляет права на запись для группы и прочих пользователей (у файлов, создававшихся с режимом **777**, он оказывается равным **755**; а режим **666** преобразуется в **644**, т.е. $777 \text{ NAND } 022 = 755$, $666 \text{ NAND } 022 = 644$). [\[42\]](#) Конечно же, впоследствии, пользователь может откорректировать права доступа к своему файлу с помощью команды **chmod**. Как правило, значение **umask** устанавливается в файле `/etc/profile` и/или `~/.bash_profile` (см. [Глава 26](#)).

rdev

Выводит или изменяет корневое устройство, размер RAM-диска или видео режим. Функциональные возможности утилиты **rdev** вообще повторяются загрузчиком **lilo**, но **rdev** по-прежнему остается востребованной, например, при установке

электронного диска (RAM-диск). Это еще одна потенциально опасная, при неумелом использовании, утилита.

Команды для работы с модулями ядра

lsmod

Выводит список загруженных модулей.

```
bash$ lsmod
Module                Size  Used by
autofs                 9456    2 (autoclean)
opl3                  11376    0
serial_cs             5456    0 (unused)
sb                    34752    0
uart401               6384    0 [sb]
sound                 58368    0 [opl3 sb uart401]
soundlow              464     0 [sound]
soundcore             2800    6 [sb sound]
ds                    6448    2 [serial_cs]
i82365                22928    2
pcmcia_core           45984    0 [serial_cs ds i82365]
```



Команда **cat /proc/modules** выведет на экран эту же информацию.

insmod

Принудительная загрузка модуля ядра (старайтесь вместо **insmod** использовать команду **modprobe**). Должна вызываться с привилегиями пользователя root.

rmmod

Выгружает модуль ядра. Должна вызываться с привилегиями пользователя root.

modprobe

Загрузчик модулей, который обычно вызывается из сценариев начальной загрузки системы. Должна вызываться с привилегиями пользователя root.


depmod

Создает файл зависимостей между модулями, обычно вызывается из сценариев начальной загрузки системы.


Прочие команды

env

Запускает указанную программу или сценарий с модифицированными [переменными окружения](#) (не изменяя среду системы в целом, изменения касаются только окружения запускаемой программы/сценария). Посредством [varname=xxx], устанавливает значение переменной окружения varname, которая будет доступна из запускаемой программы/сценария. Без параметров -- просто выводит список переменных окружения с их значениями.

-  В Bash, и других производных от Bourne shell, имеется возможность установки переменных окружения и запуска программы (или сценария) одной командной строкой.

```
var1=value1 var2=value2 commandXXX
# $var1 и $var2 -- будут определены только в окружении для
'commandXXX'.
```

-  В первой строке сценария ("sha-bang") можно указать команду **env**, если путь к командному интерпретатору не известен.

```
#!/usr/bin/env perl

print "Этот сценарий, на языке программирования Perl, будет
запущен,\n";
print "даже если я не знаю где в системе находится Perl.\n";

# Прекрасно подходит для написания кросс-платформенных сценариев,
# когда Perl может находиться совсем не там, где вы ожидаете.
# Спасибо S.C.
```

ldd

Выводит список разделяемых библиотек, необходимых для исполняемого файла.

```
bash$ ldd /bin/ls
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

watch

Периодически запускает указанную программу с заданным интервалом времени.

По-умолчанию интервал между запусками принимается равным 2 секундам, но может быть изменен ключом `-n`.

```
watch -n 5 tail /var/log/messages
# Выводит последние 10 строк из системного журнала, /var/log/messages, каждые
пять секунд.
```

strip

Удаляет отладочную информацию из исполняемого файла. Это значительно уменьшает размер исполняемого файла, но при этом делает отладку программы невозможной.

Эту команду часто можно встретить в [Makefile](#)-ах, и редко -- в сценариях на языке командной оболочки.

nm

Выводит список символов (используемых в целях отладки), содержащихся в откомпилированном двоичном файле.

rdist

Позволяет на заданных машинах хранить идентичные копии файлов. По умолчанию, rdist просматривает только те файлы, версия которых на удаленных машинах более старая, чем на локальной машине. Это делается сравнением последнего времени модификации и размера файла на локальной машине и на удаленных.

А теперь, используя полученные нами знания, попробуем разобраться с одним из системных сценариев. Один из самых коротких и простых -- это **killall**, который вызывается для остановки процессов при перезагрузке или выключении компьютера.

Пример 13-8. Сценарий killall, из каталога /etc/rc.d/init.d

```
#!/bin/sh

# --> Комментарии, начинающиеся с "# -->", добавлены автором документа.

# --> Этот сценарий является частью пакета 'rc'-сценариев
# --> Автор: Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>

# --> Этот сценарий характерен для дистрибутива Red Hat
# --> (в других дистрибутивах может отсутствовать).

# Остановить все ненужные сервисы которые еще работают (собственно,
# их уже не должно быть, это лишь формальная проверка, на всякий случай)

for i in /var/lock/subsys/*; do
    # --> Стандартный заголовок цикла for/in, но, поскольку "do"
    # --> находится в той же самой строке, что и for,
    # --> необходимо разделить их символом ";".
    # Проверяется наличие сценария.
    [ ! -f $i ] && continue
    # --> Очень интересное использование "И-списка", эквивалентно:
    # --> if [ ! -f "$i" ]; then continue

    # Получить имя подсистемы.
    subsys=${i#/var/lock/subsys/}
    # --> В данном случае совпадает с именем файла.
    # --> Это точный эквивалент subsys=`basename $i`.

    # --> Таким образом получается имя файла блокировки (если он присутствует,
    # --> то это означает, что процесс запущен).
    # --> См. описание команды "lockfile" выше.

    # Остановить службу.
    if [ -f /etc/rc.d/init.d/$subsys.init ]; then
        /etc/rc.d/init.d/$subsys.init stop
    else
        /etc/rc.d/init.d/$subsys stop
    # --> Останавливает задачу или демона
    # --> посредством встроенной команды 'stop'.
    fi
done
```

Вобщем все довольно понятно. Кроме хитрого манипулирования с переменными, при определении имени подсистемы (службы), здесь нет ничего нового.

Упражнение 1. Просмотрите сценарий **halt** в каталоге /etc/rc.d/init.d. Он по размеру немного больше, чем **killall**, но придерживается той же концепции. Создайте копию этого сценария в своем домашнем каталоге и поэкспериментируйте с ним (*НЕ* запускайте его с привилегиями суперпользователя). Попробуйте запустить его с ключами `-vn (sh -vn scriptname)`. Добавьте свои комментарии. Замените действующие команды на "echo".

Упражнение 2. Просмотрите другие, более сложные сценарии из /etc/rc.d/init.d.

Попробуйте разобраться в их работе. Проверьте их работу, следуя рекомендациям, приведенным выше. За дополнительной информацией вы можете обратиться к документу `sysvinitfiles` в каталоге `/usr/share/doc/initscripts-?.??`, который входит в пакет документации к "initscripts".

Глава 14. Подстановка команд

Подстановка команд -- это подстановка результатов выполнения команды [\[43\]](#) или даже серии команд; буквально, эта операция позволяет вызвать команду в другом окружении.

Классический пример подстановки команд -- использование обратных одиночных кавычек (``...``). Команды внутри этих кавычек представляют собой текст командной строки.

```
script_name=`basename $0`  
echo "Имя этого файла-сценария: $script_name."
```

Вывод от команд может использоваться: как аргумент другой команды, для установки значения переменной и даже для генерации списка аргументов цикла [for](#).

```
rm `cat filename` # здесь "filename" содержит список удаляемых файлов.  
#  
# S. C. предупреждает, что в данном случае может возникнуть ошибка "arg list too  
# long".  
# Такой вариант будет лучше: xargs rm -- < filename  
# ( -- подходит для случая, когда "filename" начинается с символа "-" )  
  
textfile_listing=`ls *.txt`  
# Переменная содержит имена всех файлов *.txt в текущем каталоге.  
echo $textfile_listing  
  
textfile_listing2=$(ls *.txt) # Альтернативный вариант.  
echo $textfile_listing2  
# Результат будет тем же самым.  
  
# Проблема записи списка файлов в строковую переменную состоит в том,  
# что символы перевода строки заменяются на пробел.  
#  
# Как вариант решения проблемы -- записывать список файлов в массив.  
# shopt -s nullglob # При несоответствии, имя файла игнорируется.  
# textfile_listing=( *.txt )  
#  
# Спасибо S.C.
```



Подставляемая команда может получиться разбитой на отдельные слова.

```
COMMAND `echo a b` # 2 аргумента: a и b  
COMMAND "`echo a b`" # 1 аргумент: "a b"  
COMMAND `echo` # без аргументов  
COMMAND "`echo`" # один пустой аргумент
```


Спасибо S.C.

Даже когда не происходит разбиения на слова, операция подстановки команд может удалить символы перевода строки.

```
# cd "`pwd`" # Должна выполняться всегда.  
# Однако...
```

```
mkdir 'dir with trailing newline  
,
```

```
cd 'dir with trailing newline  
,
```

```
cd "`pwd`" # Ошибка:  
# bash: cd: /tmp/dir with trailing newline: No such file or directory
```

```
cd "$PWD" # Выполняется без ошибки.
```

```
old_tty_setting=$(stty -g) # Сохранить настройки терминала.  
echo "Нажмите клавишу "  
stty -icanon -echo # Запретить "канонический" режим терминала.  
# Также запрещает эхо-вывод.
```

```
key=$(dd bs=1 count=1 2> /dev/null) # Поймать нажатие на клавишу.
```

```
stty "$old_tty_setting" # Восстановить настройки терминала.
```

```
echo "Количество нажатых клавиш = ${#key}." # ${#variable} = количество символов в переменной  
$variable
```

```
#
```

```
# Нажмите любую клавишу, кроме RETURN, на экране появится "Количество нажатых клавиш
```

```
# Нажмите RETURN, и получите: "Количество нажатых клавиш = 0."
```

```
# Символ перевода строки будет "съеден" операцией подстановки команды.
```

Спасибо S.C.



При выводе значений переменных, полученных в результате подстановки команд, командой **echo**, без кавычек, символы перевода строки будут удалены. Это может оказаться неприятным сюрпризом.

```
dir_listing=`ls -l`  
echo $dir_listing # без кавычек
```

```
# Вы наверно ожидали увидеть удобочитаемый список каталогов.
```

```
# Однако, вы получите:
```

```
# 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
```

```
# Символы перевода строки были заменены пробелами.
```

```
echo "$dir_listing" # в кавычках  
# -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 записывать в переменные содержимое целых файлов, с помощью [перенаправления](#) или команды [cat](#).

```

variable1=`<file1`      # Записать в переменную "variable1" содержимое файла
"file1".
variable2=`cat file2`   # Записать в переменную "variable2" содержимое файла "file2".

# Замечание 1:
# Удаляются символы перевода строки.
#
# Замечание 2:
# В переменные можно записать даже управляющие символы.

# Выдержки из системного файла /etc/rc.d/rc.sysinit
#+ (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`"
...
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 используйте переменные для хранения содержимого текстовых файлов *большого* объема, без веских на то оснований. Не записывайте в переменные содержимое *бинарных* файлов, даже шутки ради.

Пример 14-1. Глупая выходка

```

#!/bin/bash
# stupid-script-tricks.sh: Люди! Будьте благоразумны!
# Из "Глупые выходки", том I.

dangerous_variable=`cat /boot/vmlinuz` # Сжатое ядро Linux.

echo "длина строки \$dangerous_variable = ${#dangerous_variable}"
# длина строки $dangerous_variable = 794151
# ('wc -c /boot/vmlinuz' даст другой результат.)

# echo "$dangerous_variable"
# Даже не пробуйте раскомментировать эту строку! Это приведет к зависанию
сценария.

# Автор этого документа не знает, где можно было бы использовать

```

#+ запись содержимого двоичных файлов в переменные.

```
exit 0
```

Обратите внимание: в данной ситуации не возникает ошибки *переполнения буфера*. Этот пример показывает превосходство защищенности интерпретирующих языков, таких как Bash, от ошибок программиста, над компилируемыми языками программирования.

Подстановка команд, позволяет записать в переменную результаты выполнения [цикла](#). Ключевым моментом здесь является команда [echo](#), в теле цикла.

Пример 14-2. Запись результатов выполнения цикла в переменную

```
#!/bin/bash
# csubloop.sh: Запись результатов выполнения цикла в переменную

variable1=`for i in 1 2 3 4 5
do
  echo -n "$i"                # Здесь 'echo' -- это ключевой момент
done`

echo "variable1 = $variable1" # variable1 = 12345

i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"                # Опять же, команда 'echo' просто необходима.
  let "i += 1"                # Увеличение на 1.
done`

echo "variable2 = $variable2" # variable2 = 0123456789

exit 0
```

Подстановка команд позволяет существенно расширить набор инструментальных средств, которыми располагает Bash. Суть состоит в том, чтобы написать программу или сценарий, которая выводит результаты своей работы на stdout (как это делает подавляющее большинство утилит в UNIX) и записать вывод от программы в переменную.

```
#include <stdio.h>

/* Программа на C "Hello, world." */

int main()
{
  printf( "Hello, world." );
  return (0);
}

bash$ gcc -o hello hello.c

#!/bin/bash
# hello.sh

greeting=`./hello`
```

```
echo $greeting
```

```
bash$ sh hello.sh  
Hello, world.
```



Альтернативой обратным одиночным кавычкам, используемым для подстановки команд, можно считать такую форму записи: **\$(COMMAND)**.

```
output=$(sed -n /"$1"/p $file) # К примеру из "grp.sh".
```

```
# Запись в переменную содержимого текстового файла.
```

```
File_contents1=$(cat $file1)
```

```
File_contents2=$(<$file2) # Bash допускает и такую запись.
```

Примеры подстановки команд в сценариях:

1. [Пример 10-7](#)
 2. [Пример 10-26](#)
 3. [Пример 9-26](#)
 4. [Пример 12-2](#)
 5. [Пример 12-15](#)
 6. [Пример 12-12](#)
 7. [Пример 12-39](#)
 8. [Пример 10-13](#)
 9. [Пример 10-10](#)
 10. [Пример 12-24](#)
 11. [Пример 16-7](#)
 12. [Пример А-19](#)
 13. [Пример 27-1](#)
 14. [Пример 12-32](#)
 15. [Пример 12-33](#)
 16. [Пример 12-34](#)
-

Глава 15. Арифметические подстановки

Арифметические подстановки -- это мощный инструмент, предназначенный для выполнения арифметических операций в сценариях. Перевод строки в числовое выражение производится с помощью [обратных одиночных кавычек](#), [двойных круглых скобок](#) или предложения [let](#).

Вариации

Арифметические подстановки в обратных одиночных кавычках (часто используются совместно с командой [expr](#))

```
z=`expr $z + 3`          # Команда 'expr' вычисляет значение выражения.
```

Арифметические подстановки в двойных круглых скобках, и предложение **let**

В арифметических подстановках, обратные одиночные кавычки могут быть заменены на двойные круглые скобки `$ (. . .)` или очень удобной конструкцией, с применением предложения **let**.

```
z=$((z+3))
# $((EXPRESSION)) -- это подстановка арифметического выражения. # Не путайте с
#+ подстановкой команд.
```

```
let z=z+3
let "z += 3" # Кавычки позволяют вставлять пробелы и специальные операторы.
# Оператор 'let' вычисляет арифметическое выражение,
#+ это не подстановка арифметического выражения.
```

Все вышеприведенные примеры эквивалентны. Вы можете использовать любую из этих форм записи "по своему вкусу".

Примеры арифметических подстановок в сценариях:

1. [Пример 12-6](#)
2. [Пример 10-14](#)
3. [Пример 25-1](#)
4. [Пример 25-6](#)
5. [Пример A-19](#)

Глава 16. Перенаправление ввода/вывода

В системе по-умолчанию всегда открыты три "файла" -- `stdin` (клавиатура), `stdout` (экран) и `stderr` (вывод сообщений об ошибках на экран). Эти, и любые другие открытые

файлы, могут быть перенаправлены. В данном случае, термин "перенаправление" означает получить вывод из файла, команды, программы, сценария или даже отдельного блока в сценарии (см. [Пример 3-1](#) и [Пример 3-2](#)) и передать его на вход в другой файл, команду, программу или сценарий.

С каждым открытым файлом связан дескриптор файла. [44] Дескрипторы файлов `stdin`, `stdout` и `stderr` -- 0, 1 и 2, соответственно. При открытии дополнительных файлов, дескрипторы с 3 по 9 остаются незанятыми. Иногда дополнительные дескрипторы могут сослужить неплохую службу, временно сохраняя в себе ссылку на `stdin`, `stdout` или `stderr`. [45] Это упрощает возврат дескрипторов в нормальное состояние после сложных манипуляций с перенаправлением и перестановками (см. [Пример 16-1](#)).

```
COMMAND_OUTPUT >
# Перенаправление stdout (вывода) в файл.
# Если файл отсутствовал, то он создается, иначе -- перезаписывается.

ls -lR > dir-tree.list
# Создает файл, содержащий список дерева каталогов.

: > filename
# Операция > усекает файл "filename" до нулевой длины.
# Если до выполнения операции файла не существовало,
# то создается новый файл с нулевой длиной (тот же эффект дает команда
'touch').
# Символ : выступает здесь в роли местозаполнителя, не выводя ничего.

> filename
# Операция > усекает файл "filename" до нулевой длины.
# Если до выполнения операции файла не существовало,
# то создается новый файл с нулевой длиной (тот же эффект дает команда
'touch').
# (тот же результат, что и выше -- ": >", но этот вариант неработоспособен
# в некоторых командных оболочках.)

COMMAND_OUTPUT >>
# Перенаправление stdout (вывода) в файл.
# Создает новый файл, если он отсутствовал, иначе -- дописывает в конец файла.

# Однострочные команды перенаправления
# (затрагивают только ту строку, в которой они встречаются):
# -----

1>filename
# Перенаправление вывода (stdout) в файл "filename".
1>>filename
# Перенаправление вывода (stdout) в файл "filename", файл открывается в режиме
добавления.
2>filename
# Перенаправление stderr в файл "filename".
2>>filename
# Перенаправление stderr в файл "filename", файл открывается в режиме
добавления.
&>filename
# Перенаправление stdout и stderr в файл "filename".

#=====
# Перенаправление stdout, только для одной строки.
LOGFILE=script.log

echo "Эта строка будет записана в файл \"\$LOGFILE\"." 1>$LOGFILE
echo "Эта строка будет добавлена в конец файла \"\$LOGFILE\"." 1>>$LOGFILE
echo "Эта строка тоже будет добавлена в конец файла \"\$LOGFILE\"." 1>>$LOGFILE
echo "Эта строка будет выведена на экран и не попадет в файл \"\$LOGFILE\"."
# После каждой строки, сделанное перенаправление автоматически "сбрасывается".
```

```

# Перенаправление stderr, только для одной строки.
ERRORFILE=script.errors

bad_command1 2>$ERRORFILE      # Сообщение об ошибке запишется в $ERRORFILE.
bad_command2 2>>$ERRORFILE    # Сообщение об ошибке добавится в конец
$ERRORFILE.
bad_command3                   # Сообщение об ошибке будет выведено на
stderr,
                                #+ и не попадет в $ERRORFILE.
# После каждой строки, сделанное перенаправление также автоматически
"сбрасывается".
#=====

2>&1
# Перенаправляется stderr на stdout.
# Сообщения об ошибках передаются туда же, куда и стандартный вывод.

i>&j
# Перенаправляется файл с дескриптором i в j.
# Вывод в файл с дескриптором i передается в файл с дескриптором j.

>&j
# Перенаправляется файл с дескриптором 1 (stdout) в файл с дескриптором j.
# Вывод на stdout передается в файл с дескриптором j.

0< FILENAME
< FILENAME
# Ввод из файла.
# Парная команде ">", часто встречается в комбинации с ней.
#
# grep search-word <filename

[j]<>filename
# Файл "filename" открывается на чтение и запись, и связывается с дескриптором
"j".
# Если "filename" отсутствует, то он создается.
# Если дескриптор "j" не указан, то, по-умолчанию, берется дескриптор 0, stdin.
#
# Как одно из применений этого -- запись в конкретную позицию в файле.
echo 1234567890 > File      # Записать строку в файл "File".
hex 3<> File              # Открыть "File" и связать с дескриптором 3.
read -n 4 <&3             # Прочитать 4 символа.
echo -n . >&3              # Записать символ точки.
hex 3>&-                  # Закрыть дескриптор 3.
cat File                  # ==> 1234.67890
# Произвольный доступ, да и только!

|
# Конвейер (канал).
# Универсальное средство для объединения команд в одну цепочку.
# Похоже на ">", но на самом деле -- более обширная.
# Используется для объединения команд, сценариев, файлов и программ в одну
цепочку (конвейер).
cat *.txt | sort | uniq > result-file
# Содержимое всех файлов .txt сортируется, удаляются повторяющиеся строки,
# результат сохраняется в файле "result-file".

```

Операции перенаправления и/или конвейеры могут комбинироваться в одной командной строке.

```
command < input-file > output-file
```

```
command1 | command2 | command3 > output-file
```

См. [Пример 12-23](#) и [Пример A-17](#).

Допускается перенаправление нескольких потоков в один файл.

```
ls -yz >> command.log 2>&1
# Сообщение о неверной опции "yz" в команде "ls" будет записано в файл "command.log".
# Поскольку stderr перенаправлен в файл.
```

Заккрытие дескрипторов файлов

`n<&-`

Закрывает дескриптор входного файла *n*.

`0<&-`, `<&-`

Закрывает `stdin`.

`n>&-`

Закрывает дескриптор выходного файла *n*.

`1>&-`, `>&-`

Закрывает `stdout`.

Дочерние процессы наследуют дескрипторы открытых файлов. По этой причине и работают конвейеры. Чтобы предотвратить наследование дескрипторов -- закройте их перед запуском дочернего процесса.

```
# В конвейер передается только stderr.
```

```
exec 3>&1 # Сохранить текущее "состояние" stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&- # Закрывает дескр. 3 для 'grep' (но не для
'ls').
#          ^^^^  ^^^^
exec 3>&- # Теперь закрыть его для оставшейся части
сценария.
```

```
# Спасибо S.C.
```

Дополнительные сведения о перенаправлении ввода/вывода вы найдете в [Приложение D](#).

16.1. С помощью команды `exec`

Команда `exec <filename` перенаправляет ввод со `stdin` на файл. С этого момента весь ввод, вместо `stdin` (обычно это клавиатура), будет производиться из этого файла. Это дает возможность читать содержимое файла, строку за строкой, и анализировать каждую

введенную строку с помощью [sed](#) и/или [awk](#).

Пример 16-1. Перенаправление stdin с помощью exec

```
#!/bin/bash
# Перенаправление stdin с помощью 'exec'.

exec 6<&0          # Связать дескр. #6 со стандартным вводом (stdin).
                  # Сохраняя stdin.

exec < data-file  # stdin заменяется файлом "data-file"

read a1           # Читается первая строка из "data-file".
read a2           # Читается вторая строка из "data-file."

echo
echo "Следующие строки были прочитаны из файла."
echo "-----"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
# Восстанавливается stdin из дескр. #6, где он был предварительно сохранен,
#+ и дескр. #6 закрывается ( 6<&- ) освобождая его для других процессов.
#
# <&6 6<&-      дает тот же результат.

echo -n "Введите строку "
read b1 # Теперь функция "read", как и следовало ожидать, принимает данные с
обычного stdin.
echo "Строка, принятая со stdin."
echo "-----"
echo "b1 = $b1"

echo

exit 0
```

Аналогично, конструкция **exec >filename** перенаправляет вывод на stdout в заданный файл. После этого, весь вывод от команд, который обычно направляется на stdout, теперь выводится в этот файл.

Пример 16-2. Перенаправление stdout с помощью exec

```
#!/bin/bash
# reassign-stdout.sh

LOGFILE=logfile.txt

exec 6>&1          # Связать дескр. #6 со stdout.
                  # Сохраняя stdout.

exec > $LOGFILE  # stdout замещается файлом "logfile.txt".

# ----- #
# Весь вывод от команд, в данном блоке, записывается 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>&-      # Восстановить stdout и закрыть дескр. #6.

echo
echo "== stdout восстановлено в значение по-умолчанию =="
echo
ls -al
echo

exit 0

```

Пример 16-3. Одновременное перенаправление устройств, stdin и stdout, с помощью команды exec

```

#!/bin/bash
# upperconv.sh
# Преобразование символов во входном файле в верхний регистр.

E_FILE_ACCESS=70
E_WRONG_ARGS=71

if [ ! -r "$1" ]      # Файл доступен для чтения?
then
    echo "Невозможно прочитать из заданного файла!"
    echo "Порядок использования: $0 input-file output-file"
    exit $E_FILE_ACCESS
fi
    # В случае, если входной файл ($1) не задан
    #+ код завершения будет этим же.

if [ -z "$2" ]
then
    echo "Необходимо задать выходной файл."
    echo "Порядок использования: $0 input-file output-file"
    exit $E_WRONG_ARGS
fi

exec 4<&0
exec < $1          # Назначить ввод из входного файла.

exec 7>&1
exec > $2          # Назначить вывод в выходной файл.
                  # Предполагается, что выходной файл доступен для записи
                  # (добавить проверку?).

# -----
#   cat - | tr a-z A-Z  # Перевод в верхний регистр
#   ^^^^^             # Чтение со stdin.
#   ^^^^^^^^^^^      # Запись в stdout.
# Однако, и stdin и stdout были перенаправлены.
# -----

exec 1>&7 7>&-      # Восстановить stdout.
exec 0<&4 4<&-      # Восстановить stdin.

# После восстановления, следующая строка выводится на stdout, чего и следовало
# ожидать.
echo "Символы из `"$1`" преобразованы в верхний регистр, результат записан в `"$2`"."

exit 0

```

16.2. Перенаправление для блоков кода

Блоки кода, такие как циклы [while](#), [until](#) и [for](#), условный оператор [if/then](#), так же могут смешиваться с перенаправлением `stdin`. Даже функции могут использовать эту форму перенаправления (см. [Пример 22-7](#)). Оператор перенаправления `<`, в таких случаях, ставится в конце блока.

Пример 16-4. Перенаправление в цикл *while*

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data      # По-умолчанию, если имя файла не задано.
else
  Filename=$1
fi
# Конструкцию проверки выше, можно заменить следующей строкой (подстановка
# параметров):
#+ Filename=${1:-names.data}

count=0

echo

while [ "$name" != Smith ] # Почему переменная $name взята в кавычки?
do
  read name                # Чтение из $Filename, не со stdin.
  echo $name
  let "count += 1"
done <"$Filename"         # Перенаправление на ввод из файла $Filename.
#   ^^^^^^^^^^^^^^^

echo; echo "Имен прочитано: $count"; echo

# Обратите внимание: в некоторых старых командных интерпретаторах,
#+ перенаправление в циклы приводит к запуску цикла в субоболочке (subshell).
# Таким образом, переменная $count, по окончании цикла, будет содержать 0,
# значение, записанное в нее до входа в цикл.
# Bash и ksh стремятся избежать запуска субоболочки (subshell), если это возможно,
#+ так что этот сценарий, в этих оболочках, работает корректно.
#
# Спасибо Heiner Steven за это примечание.

exit 0
```

Пример 16-5. Альтернативная форма перенаправления в цикле *while*

```
#!/bin/bash

# Это альтернативный вариант предыдущего сценария.

# Предложил: by Heiner Steven
#+ для случаев, когда циклы с перенаправлением
#+ запускаются в субоболочке, из-за чего переменные, устанавливаемые в цикле,
#+ не сохраняют свои значения по завершении цикла.

if [ -z "$1" ]
then
  Filename=names.data      # По-умолчанию, если имя файла не задано.
else
  Filename=$1
```

```

fi

exec 3<&0          # Сохранить stdin в дескр. 3.
exec 0<"$Filename" # Перенаправить stdin.

count=0
echo

while [ "$name" != Smith ]
do
    read name          # Прочитать с перенаправленного stdin ($Filename).
    echo $name
    let "count += 1"
done <"$Filename"    # Цикл читает из файла $Filename.
#   ^^^^^^^^^^^^^^^

exec 0<&3          # Восстановить stdin.
exec 3<&-         # Закрыть временный дескриптор 3.

echo; echo "Имен прочитано: $count"; echo

exit 0

```

Пример 16-6. Перенаправление в цикл *until*

```

#!/bin/bash
# То же самое, что и в предыдущем примере, только для цикла "until".

if [ -z "$1" ]
then
    Filename=names.data          # По-умолчанию, если файл не задан.
else
    Filename=$1
fi

# while [ "$name" != Smith ]
until [ "$name" = Smith ]      # Проверка != изменена на =.
do
    read name                  # Чтение из $Filename, не со stdin.
    echo $name
done <"$Filename"             # Перенаправление на ввод из файла $Filename.
#   ^^^^^^^^^^^^^^^

# Результаты получаются теми же, что и в случае с циклом "while", в предыдущем
# примере.

exit 0

```

Пример 16-7. Перенаправление в цикл *for*

```

#!/bin/bash

if [ -z "$1" ]
then
    Filename=names.data          # По-умолчанию, если файл не задан.
else
    Filename=$1
fi

line_count=`wc $Filename | awk '{ print $1 }'`
#           Число строк в файле.
#
# Слишком запутано, тем не менее показывает
#+ возможность перенаправления stdin внутри цикла "for"...
#+ если вы достаточно умны.

```



```
echo $name
fi <"$Filename"
# ^^^^^^^^^^^^^^^
```

Читает только первую строку из файла.

```
exit 0
```

Пример 16-10. Файл с именами "names.data", для примеров выше

```
Aristotle
Belisarius
Capablanca
Euler
Goethe
Hamurabi
Jonah
Laplace
Maroczy
Purcell
Schmidt
Simmelweiss
Smith
Turing
Venn
Wilson
Znosko-Borowski
```

```
# Это файл с именами для примеров
#+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

Перенаправление stdout для блока кода, может использоваться для сохранения результатов работы этого блока в файл. См. [Пример 3-2](#).

[Встроенный документ](#) -- это особая форма перенаправления для блоков кода.

16.3. Область применения

Как один из вариантов грамотного применения перенаправления ввода/вывода, можно назвать разбор и "сшивание" вывода от команд (см. [Пример 11-6](#)). Это позволяет создавать файлы отчетов и журналов регистрации событий.

Пример 16-11. Регистрация событий

```
#!/bin/bash
# logevents.sh, автор: Stephane Chazelas.

# Регистрация событий в файле.
# Сценарий должен запускаться с привилегиями root (что бы иметь право на запись в /var/log).

ROOT_UID=0      # Привилегии root имеет только пользователь с $UID = 0.
E_NOTROOT=67    # Код завершения, если не root.

if [ "$UID" -ne "$ROOT_UID" ]
then
    echo "Сценарий должен запускаться с привилегиями root."
    exit $E_NOTROOT
fi
```

```

FD_DEBUG1=3
FD_DEBUG2=4
FD_DEBUG3=5

# Раскомментируйте одну из двух строк, ниже, для активизации сценария.
# LOG_EVENTS=1
# LOG_VARS=1

log() # Запись даты и времени в файл.
{
echo "$(date)  $" >&7      # Добавляет в конец файла.
                        # 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      # Подавить вывод.
fi

FD_LOGEVENTS=7
if [[ $LOG_EVENTS ]]
then
  # then exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
  # Строка, выше, не работает в Bash, версии 2.04.
  exec 7>> /var/log/event.log      # Добавление в конец "event.log".
  log                             # Записать дату и время.
else exec 7> /dev/null          # Подавить вывод.
fi

echo "DEBUG3: beginning" >&${FD_DEBUG3}

ls -l >&5 2>&4                  # command1 >&5 2>&4

echo "Done"                    # command2

echo "sending mail" >&${FD_LOGEVENTS} # Написать "sending mail" в дескр. #7.

exit 0

```

Глава 17. Встроенные документы

Встроенный документ (here document) является специальной формой [перенаправления ввода/вывода](#), которая позволяет передать список команд интерактивной программе или команде, например [ftp](#), [telnet](#) или [ex](#). Конец встроенного документа выделяется "строкой-ограничителем", которая задается с помощью специальной последовательности символов <<. Эта последовательность -- есть перенаправление вывода из файла в программу, напоминает конструкцию `interactive-program < command-file`, где `command-file` содержит строки:

```
command #1
```

```
command #2
...
```

Сценарий, использующий "встроенный документ" для тех же целей, может выглядеть примерно так:

```
#!/bin/bash
interactive-program <<LimitString
command #1
command #2
...
LimitString
```

В качестве строки-ограничителя должна выбираться такая последовательность символов, которая не будет встречаться в теле "встроенного документа".

Обратите внимание: использование *встроенных документов* может иногда с успехом применяться и при работе с неинтерактивными командами и утилитами.

Пример 17-1. dummyfile: Создание 2-х строчного файла-заготовки

```
#!/bin/bash

# Неинтерактивное редактирование файла с помощью 'vi'.
# Эмуляция 'sed'.

E_BADARGS=65

if [ -z "$1" ]
then
    echo "Порядок использования: `basename $0` filename"
    exit $E_BADARGS
fi

TARGETFILE=$1

# Вставить 2 строки в файл и сохранить.
#-----Начало встроенного документа-----#
vi $TARGETFILE <<x23LimitStringx23
i
Это строка 1.
Это строка 2.
^[
ZZ
x23LimitStringx23
#-----Конец встроенного документа-----#

# Обратите внимание: ^[, выше -- это escape-символ
#+ Control-V <Esc>.

# Bram Moolenaar указывает, что этот скрипт может не работать с 'vim',
#+ из-за возможных проблем взаимодействия с терминалом.

exit 0
```

Этот сценарий, с тем же эффектом, мог бы быть реализован, основываясь не на **vi**, а на **ex**. Встроенные документы, содержащие команды для **ex**, стали настолько обычным делом, что их уже смело можно вынести в отдельную категорию -- *ex-сценарии*.

Пример 17-2. broadcast: Передача сообщения всем, работающим в системе,

ПОЛЬЗОВАТЕЛЯМ

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
Пошлите, по электронной почте, ваш заказ на пиццу, системному администратору.
(Добавьте дополнительный доллар, если вы желаете положить на пиццу анчоусы или
грибы.)
# Внимание: строки комментария тоже будут переданы команде 'wall' как часть текста.
zzz23EndOfMessagezzz23

# Возможно, более эффективно это может быть сделано так:
#     wall <message-file
# Однако, встроенный документ помогает сэкономить ваши силы и время.

exit 0
```

Пример 17-3. Вывод многострочных сообщений с помощью cat

```
#!/bin/bash

# Команда 'echo' прекрасно справляется с выводом однострочных сообщений,
# но иногда необходимо вывести несколько строк.
# Команда 'cat' и встроенный документ помогут вам в этом.

cat <<End-of-message
-----
Это первая строка сообщения.
Это вторая строка сообщения.
Это третья строка сообщения.
Это четвертая строка сообщения.
Это последняя строка сообщения.
-----
End-of-message

exit 0

#-----
# Команда "exit 0", выше, не позволяет исполнить нижележащие строки.

# S.C. отмечает, что следующий код работает точно так же.
echo "-----"
Это первая строка сообщения.
Это вторая строка сообщения.
Это третья строка сообщения.
Это четвертая строка сообщения.
Это последняя строка сообщения.
-----"
# Однако, в этом случае, двойные кавычки в теле сообщения, должны экранироваться.
```

Если строка-ограничитель встроенного документа начинается с символа - (<<- `LimitString`), то это приводит к подавлению вывода символов табуляции (но не пробелов). Это может оказаться полезным при форматировании текста сценария для большей удобочитаемости.

Пример 17-4. Вывод многострочных сообщений с подавлением символов табуляции

```
#!/bin/bash
# То же, что и предыдущий сценарий, но...

# Символ "-", начинающий строку-ограничитель встроенного документа: <<-
# подавляет вывод символов табуляции, которые могут встречаться в теле документа,
# но не пробелов.
```

```

cat <<-ENDOFMESSAGE
    Это первая строка сообщения.
    Это вторая строка сообщения.
    Это третья строка сообщения.
    Это четвертая строка сообщения.
    Это последняя строка сообщения.
ENDOFMESSAGE
# Текст, выводимый сценарием, будет смещен влево.
# Ведущие символы табуляции не будут выводиться.

# Вышеприведенные 5 строк текста "сообщения" начинаются с табуляции, а не с пробелов.

exit 0

```

Встроенные документы поддерживают подстановку команд и параметров. Что позволяет передавать различные параметры в тело встроенного документа.

Пример 17-5. Встроенные документы и подстановка параметров

```

#!/bin/bash
# Вывод встроенного документа командой 'cat', с использованием подстановки
# параметров.

# Попробуйте запустить сценарий без аргументов, ./scriptname
# Попробуйте запустить сценарий с одним аргументом, ./scriptname Mortimer
# Попробуйте запустить сценарий с одним аргументом, из двух слов, в кавычках,
# ./scriptname "Mortimer Jones"

CMDLINEPARAM=1      # Минимальное число аргументов командной строки.

if [ $# -ge $CMDLINEPARAM ]
then
    NAME=$1          # Если аргументов больше одного,
                    # то рассматривается только первый.
else
    NAME="John Doe" # По-умолчанию, если сценарий запущен без аргументов.
fi

RESPONDENT="автора этого сценария"

cat <<Endofmessage

Привет, $NAME!
Примите поздравления от $RESPONDENT.

# Этот комментарий тоже выводится (почему?).

Endofmessage

# Обратите внимание на то, что пустые строки тоже выводятся.

exit 0

```

Закрывая строку-ограничитель в кавычки или экранируя ее, можно запретить подстановку параметров в теле встроенного документа.

Пример 17-6. Отключение подстановки параметров

```

#!/bin/bash
# Вывод встроенного документа командой 'cat', с запретом подстановки параметров.

NAME="John Doe"
RESPONDENT="автора этого сценария"

```

```
cat <<'Endofmessage'
```

```
Привет, $NAME.  
Примите поздравления от $RESPONDENT.
```

```
Endofmessage
```

```
# Подстановка параметров не производится, если строка ограничитель  
# заключена в кавычки или экранирована.  
# Тот же эффект дают:  
# cat <<"Endofmessage"  
# cat <<\Endofmessage
```

```
exit 0
```

Еще один пример сценария, содержащего встроенный документ и подстановку параметров в его теле.

Пример 17-7. Передача пары файлов во входящий каталог на "Sunsite"

```
#!/bin/bash  
# upload.sh  
  
# Передача пары файлов (Filename.lsm, Filename.tar.gz)  
# на Sunsite (ibiblio.org).  
  
E_ARGERROR=65  
  
if [ -z "$1" ]  
then  
    echo "Порядок использования: `basename $0` filename"  
    exit $E_ARGERROR  
fi  
  
Filename=`basename $1`          # Отсечь имя файла от пути к нему.  
  
Server="ibiblio.org"  
Directory="/incoming/Linux"  
# Вообще, эти строки должны бы не "зашиваться" жестко в сценарий,  
# а приниматься в виде аргумента из командной строки.  
  
Password="your.e-mail.address"  # Измените на свой.  
  
ftp -n $Server <<End-Of-Session  
# Ключ -n запрещает автоматическую регистрацию (auto-login)  
  
user anonymous "$Password"  
binary  
bell          # "Звякнуть" после передачи каждого файла  
cd $Directory  
put "$Filename.lsm"  
put "$Filename.tar.gz"  
bye  
End-Of-Session  
  
exit 0
```

Встроенные документы могут передаваться на вход функции, находящейся в том же сценарии.

Пример 17-8. Встроенные документы и функции

```
#!/bin/bash  
# here-function.sh
```

```

GetPersonalData ()
{
    read firstname
    read lastname
    read address
    read city
    read state
    read zipcode
} # Это немного напоминает интерактивную функцию, но...

```

```

# Передать ввод в функцию.
GetPersonalData <<RECORD001
Bozo
Bozeman
2726 Nondescript Dr.
Baltimore
MD
21226
RECORD001

```

```

echo
echo "$firstname $lastname"
echo "$address"
echo "$city, $state $zipcode"
echo

exit 0

```

Встроенный документ можно передать "пустой команде" :. Такая конструкция, фактически, создает "анонимный" встроенный документ.

Пример 17-9. "Анонимный" Встроенный Документ


```

#!/bin/bash

: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?} # Если одна из переменных не определена, то выводится
сообщение об ошибке.
TESTVARIABLES

exit 0

```

 Подобную технику можно использовать для создания "блочных комментариев".

Пример 17-10. Блочный комментарий

```

#!/bin/bash
# commentblock.sh

: << COMMENTBLOCK
echo "Эта строка не будет выведена."
Эта строка комментария не начинается с символа "#".
Это еще одна строка комментария, которая начинается не с символа "#".

&*@!!+=
Эта строка не вызовет ошибки,
поскольку Bash проигнорирует ее.
COMMENTBLOCK

echo "Код завершения \"COMMENTBLOCK\" = $?." # 0
# Показывает, что ошибок не возникало.


```

Такая методика создания блочных комментариев

```
#+ может использоваться для комментирования блоков кода во время отладки.  
# Это экономит силы и время, т.к. не нужно вставлять символ "#" в начале каждой  
строки,  
#+ а затем удалять их.
```

```
: << DEBUGXXX  
for file in *  
do  
  cat "$file"  
done  
DEBUGXXX
```

```
exit 0
```

 Еще одно остроумное применение встроенных документов -- встроенная справка к сценарию.

Пример 17-11. Встроенная справка к сценарию

```
#!/bin/bash  
# self-document.sh: сценарий со встроенной справкой  
# Модификация сценария "colm.sh".  
  
DOC_REQUEST=70  
  
if [ "$1" = "-h" -o "$1" = "--help" ]      # Request help.  
then  
  echo; echo "Порядок использования: $0 [directory-name]"; echo  
  sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATION/p' "$0" |  
  sed -e '/DOCUMENTATIONXX/d'; exit $DOC_REQUEST; fi
```

```
: << DOCUMENTATIONXX
```

Сценарий выводит сведения о заданном каталоге в виде таблицы.


Сценарию необходимо передать имя каталога. Если каталог не
указан или он недоступен для чтения, то выводятся сведения
о текущем каталоге.

```
DOCUMENTATIONXX
```


```
if [ -z "$1" -o ! -r "$1" ]  
then  
  directory=.  
else  
  directory="$1"  
fi
```

```
echo "Сведения о каталоге \"$directory\":"; echo  
(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \  
; ls -l "$directory" | sed 1d) | column -t
```

```
exit 0
```

 Для встроенных документов, во время исполнения, создаются временные файлы, но эти файлы удаляются после открытия и недоступны для других процессов.

```
bash$ bash -c 'ls -lsof -a -p $$ -d0' << EOF  
> EOF  
ls -lsof 1213 bozo 0r REG 3,5 0 30386 /tmp/t1213-0-sh (deleted)
```

 Некоторые утилиты не могут работать внутри *встроенных документов*.

Если какая-либо задача не может быть решена с помощью "встроенного документа", то

вам следует попробовать язык сценариев **expect**, который приспособлен для передачи параметров на вход интерактивных программ.

Часть 4. Материал повышенной сложности

Итак, мы вплотную подошли к изучению очень сложных и необычных аспектов написания сценариев. В этой части мы попытаемся "сбросить покров тайны" и *заглянуть за пределы известного нам мира* (представьте себе путешествие по территории, не отмеченной на карте).

Содержание

18. [Регулярные выражения](#)
 - 18.1. [Краткое введение в регулярные выражения](#)
 - 18.2. [Globbing -- Подстановка имен файлов](#)
 19. [Подоболочки, или Subshells](#)
 20. [Ограниченный режим командной оболочки](#)
 21. [Подстановка процессов](#)
 22. [Функции](#)
 - 22.1. [Сложные функции и сложности с функциями](#)
 - 22.2. [Локальные переменные](#)
 - 22.2.1. [Локальные переменные делают возможной рекурсию.](#)
 23. [Псевдонимы](#)
 24. [Списки команд](#)
 25. [Массивы](#)
 26. [Файлы](#)
 27. [/dev и /proc](#)
 - 27.1. [/dev](#)
 - 27.2. [/proc](#)
 28. [/dev/zero и /dev/null](#)
 29. [Отладка сценариев](#)
 30. [Необязательные параметры \(ключи\)](#)
 31. [Широко распространенные ошибки](#)
 32. [Стиль программирования](#)
 - 32.1. [Неофициальные рекомендации по оформлению сценариев](#)
 33. [Разное](#)
 - 33.1. [Интерактивный и неинтерактивный режим работы](#)
 - 33.2. [Сценарии-обертки](#)
 - 33.3. [Операции сравнения: Альтернативные решения](#)
 - 33.4. [Рекурсия](#)
 - 33.5. ["Цветные" сценарии](#)
 - 33.6. [Оптимизация](#)
 - 33.7. [Разные советы](#)
 - 33.8. [Проблемы безопасности](#)
 - 33.9. [Проблемы переносимости](#)
 - 33.10. [Сценарии командной оболочки под Windows](#)
 34. [Bash, версия 2](#)
-

Глава 18. Регулярные выражения

Для того, чтобы полностью реализовать потенциал командной оболочки, вам придется овладеть Регулярными Выражениями. Многие команды и утилиты, обычно используемые

в сценариях, такие как [grep](#), [expr](#), [sed](#) и [awk](#), используют Регулярные Выражения.

18.1. Краткое введение в регулярные выражения

Выражение -- это строка символов. Символы, которые имеют особое назначение, называются *метасимволами*. Так, например, кавычки могут выделять прямую речь, т.е. быть *метасимволами* для строки, заключенной в эти кавычки. Регулярные выражения -- это набор символов и/или метасимволов, которые наделены особыми свойствами. [\[46\]](#)

Основное назначение регулярных выражений -- это поиск текста по шаблону и работа со строками.

- Звездочка -- * -- означает любое количество символов в строке, предшествующих "звездочке", *в том числе и нулевое число символов*.

Выражение "1133*" -- означает *11 + один или более символов "3" + любые другие символы*: 113, 1133, 113312, и так далее.

- Точка -- . -- означает не менее одного любого символа, за исключением символа перевода строки (\n). [\[47\]](#)

Выражение "13." будет означать *13 + по меньшей мере один любой символ (включая пробел)*: 1133, 11333, но не 13 (отсутствуют дополнительные символы).

- Символ -- ^ -- означает начало строки, но иногда, в зависимости от контекста, означает отрицание в регулярных выражениях.
- Знак доллара -- \$ -- в конце регулярного выражения соответствует концу строки.

Выражение "^\$" соответствует пустой строке.



Символы ^ и \$ иногда еще называют *якорями*, поскольку они означают, или закрепляют, позицию в регулярных выражениях.

- Квадратные скобки -- [...] -- предназначены для задания подмножества символов. Квадратные скобки, внутри регулярного выражения, считаются одним символом, который может принимать значения, перечисленные внутри этих скобок..

Выражение "[xyz]" -- соответствует одному из символов *x*, *y* или *z*.

Выражение "[c-n]" соответствует одному из символов в диапазоне от *c* до *n*, включительно.

Выражение "[B-Pk-y]" соответствует одному из символов в диапазоне от *B* до *P* или в диапазоне от *k* до *y*, включительно.

Выражение "[a-z0-9]" соответствует одному из символов латиницы в нижнем регистре или цифре.

Выражение "[^b-d]" соответствует любому символу, кроме символов из диапазона от *b* до *d*, включительно. В данном случае, метасимвол ^ означает отрицание.

Объединяя квадратные скобки в одну последовательность, можно задать шаблон искомого слова. Так, выражение "[Yy][Ee][Ss]" соответствует словам *yes*, *Yes*, *YES*, *yEs* и так далее. Выражение "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9]" определяет шаблон для поиска любого номера карточки социального страхования (для США).

- Обратный слэш -- \ -- служит для [экранирования](#) специальных символов, это означает, что экранированные символы должны интерпретироваться буквально, т.е. как простые символы.

Комбинация "\\$" указывает на то, что символ "\$" трактуется как обычный символ, а не как признак конца строки в регулярных выражениях. Аналогично, комбинация "\" соответствует простому символу "\".

- [Экранированные](#) "угловые скобки" -- \<...\> -- отмечают границы слова.

Угловые скобки должны экранироваться, иначе они будут интерпретироваться как простые символы.

Выражение "\<the\>" соответствует слову "the", и не соответствует словам "them", "there", "other" и т.п.

```
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.
```

- **Дополнительные метасимволы.** Используемые при работе с [egrep](#), [awk](#) и [Perl](#)
- Знак вопроса -- ? -- означает, что предыдущий символ или регулярное выражение встречается 0 или 1 раз. В основном используется для поиска одиночных символов.
- Знак "плюс" -- + -- указывает на то, что предыдущий символ или выражение встречается 1 или более раз. Играет ту же роль, что и символ "звездочка" (*), за исключением случая нулевого количества вхождений.

```
# GNU версии sed и awk допускают использование "+",
# но его необходимо экранировать.
```

```
echo a111b | sed -ne '/a1\b/p'
echo a111b | grep 'a1\b'
echo a111b | gawk '/a1+b/'
# Все три варианта эквивалентны.
```

```
# Спасибо S.C.
```


- [Экранированные](#) "фигурные скобки" -- `\{ \}` -- задают число вхождений предыдущего выражения.

Экранирование фигурных скобок -- обязательное условие, иначе они будут интерпретироваться как простые символы. Такой порядок использования, технически, не является частью основного набора правил построения регулярных выражений.

Выражение `"[0-9]{5}"` -- в точности соответствует подстроке из пяти десятичных цифр (символов из диапазона от 0 до 9, включительно).

- 👉 В "классической" (не совместимой с POSIX) версии [awk](#), фигурные скобки не могут быть использованы. Однако, в **gawk** предусмотрен ключ `--re-interval`, который позволяет использовать (неэкранированные) фигурные скобки.

```
bash$ echo 2222 | gawk --re-interval '/2{3}/'  
2222
```

Язык программирования **Perl** и некоторые версии **egrep** не требуют экранирования фигурных скобок.

- Круглые скобки `-- ()` -- предназначены для выделения групп регулярных выражений. Они полезны при использовании с оператором `"|"` и при [извлечении подстроки](#) с помощью команды [expr](#).
- Вертикальная черта `-- |` -- выполняет роль логического оператора "ИЛИ" в регулярных выражениях и служит для задания набора альтернатив.

```
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 экранированные версии регулярных выражений, описанных выше.


- **Классы символов POSIX.** `[: class :]`

Это альтернативный способ указания диапазона символов.

- Класс `[: alnum :]` -- соответствует алфавитным символам и цифрам. Эквивалентно выражению `[A - Z a - z 0 - 9]`.
- Класс `[: alpha :]` -- соответствует символам алфавита. Эквивалентно выражению `[A - Z a - z]`.
- Класс `[: blank :]` -- соответствует символу пробела или символу табуляции.
- Класс `[: cntrl :]` -- соответствует управляющим символам (control characters).
- Класс `[: digit :]` -- соответствует набору десятичных цифр. Эквивалентно

выражению [0-9].

- Класс [:graph:] (печатаемые и псевдографические символы) -- соответствует набору символов из диапазона ASCII 33 - 126. Это то же самое, что и класс [:print:], за исключением символа пробела.
- Класс [:lower:] -- соответствует набору алфавитных символов в нижнем регистре. Эквивалентно выражению [a-z].
- Класс [:print:] (печатаемые символы) -- соответствует набору символов из диапазона ASCII 32 - 126. По своему составу этот класс идентичен классу [:graph:], описанному выше, за исключением того, что в этом классе дополнительно присутствует символ пробела.
- Класс [:space:] -- соответствует пробельным символам (пробел и горизонтальная табуляция).
- Класс [:upper:] -- соответствует набору символов алфавита в верхнем регистре. Эквивалентно выражению [A-Z].
- Класс [:xdigit:] -- соответствует набору шестнадцатеричных цифр. Эквивалентно выражению [0-9A-Fa-f].

 Вообще, символьные классы POSIX требуют заключения в кавычки или [двойные квадратные скобки](#) ([[]]).

```
bash$ grep [[:digit:]] test.file
abc=723
```

Эти символьные классы могут использоваться, с некоторыми ограничениями, даже в операциях [подстановки имен файлов \(globbing\)](#).

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r-- 1 bozo bozo 0 Aug 21 14:47 a33b
```

Примеры использования символьных классов в сценариях вы найдете в [Пример 12-14](#) и [Пример 12-15](#).

[Sed](#), [awk](#) и [Perl](#), используемые в сценариях в качестве фильтров, могут принимать регулярные выражения в качестве входных аргументов. См. [Пример A-13](#) и [Пример A-19](#).

Книга "Sed & Awk" (авторы Dougherty и Robbins) дает полное и ясное представление о регулярных выражениях (см. раздел [Литература](#)).

18.2. Globbing -- Подстановка имен файлов

Bash, сам по себе, не распознает регулярные выражения. Но в сценариях можно использовать команды и утилиты, такие как [sed](#) и [awk](#), которые прекрасно справляются с

обработкой регулярных выражений.

Фактически, Bash может выполнять подстановку имен файлов, этот процесс называется "globbing", но при этом *не* используется стандартный набор регулярных выражений. Вместо этого, при выполнении подстановки имен файлов, производится распознавание и интерпретация шаблонных символов. В число интерпретируемых шаблонов входят символы * и ?, списки символов в квадратных скобках и некоторые специальные символы (например ^, используемый для выполнения операции отрицания). Применение шаблонных символов имеет ряд важных ограничений. Например, если имена файлов начинаются с точки (например так: .bashrc), то они не будут соответствовать шаблону, содержащему символ *. [48] Аналогично, символ ? в операции подстановки имен файлов имеет иной смысл, нежели в регулярных выражениях.

```
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$ echo *
a.1 b.1 c.1 t2.sh test1.txt
```

```
bash$ echo t*
t2.sh test1.txt
```

Даже команда [echo](#) может интерпретировать шаблонные символы в именах файлов.

См. также [Пример 10-4](#).

Глава 19. Подболочки, или Subshells

Запуск сценария приводит к запуску дочернего командного интерпретатора. Который

выполняет интерпретацию и исполнение списка команд, содержащихся в файле сценария, точно так же, как если бы они были введены из командной строки. Любой сценарий запускается как дочерний процесс [родительской](#) командной оболочки, той самой, которая выводит перед вами строку приглашения к вводу на консоли или в окне xterm.

Сценарий может, так же, запустить другой дочерний процесс, в своей подоболочке. Это позволяет сценариям распараллелить процесс обработки данных по нескольким задачам, исполняемым одновременно.

Список команд в круглых скобках

(command1; command2; command3; ...)

Список команд, в круглых скобках, исполняется в подоболочке.



Значения переменных, определенных в дочерней оболочке, *не* могут быть переданы родительской оболочке. Они недоступны [родительскому процессу](#). Фактически, они ведут себя как [локальные переменные](#).

Пример 19-1. Область видимости переменных

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

echo

outer_variable=Outer

(
  inner_variable=Inner
  echo "Дочерний процесс, \"inner_variable\" = $inner_variable"
  echo "Дочерний процесс, \"outer\" = $outer_variable"
)

echo

if [ -z "$inner_variable" ]
then
  echo "Переменная inner_variable не определена в родительской оболочке"
else
  echo "Переменная inner_variable определена в родительской оболочке"
fi

echo "Родительский процесс, \"inner_variable\" = $inner_variable"
# Переменная $inner_variable не будет определена
# потому, что переменные, определенные в дочернем процессе,
# ведут себя как "локальные переменные".

echo

exit 0
```

См. также [Пример 31-1](#).

+

Смена текущего каталога в дочернем процессе (подоболочке) не влечет за собой смену текущего каталога в родительской оболочке.

Пример 19-2. Личные настройки пользователей

```
#!/bin/bash
# allprofs.sh: вывод личных настроек (profiles) всех пользователей

# Автор: Heiner Steven
# С некоторыми изменениями, внесенными автором документа.

FILE=.bashrc # Файл настроек пользователя,
              #+ в оригинальном сценарии называется ".profile".

for home in `awk -F: '{print $6}' /etc/passwd`
do
    [ -d "$home" ] || continue # Перейти к следующей итерации, если нет домашнего
    # каталога.
    [ -r "$home" ] || continue # Перейти к следующей итерации, если не доступен для
    # чтения.
    (cd $home; [ -e $FILE ] && less $FILE)
done

# По завершении сценария -- нет необходимости выполнять команду 'cd', чтобы
# вернуться в первоначальный каталог,
#+ поскольку 'cd $home' выполняется в подоболочке.

exit 0
```

Подоболочка может использоваться для задания "специфического окружения" для группы команд.

```
COMMAND1
COMMAND2
COMMAND3
(
    IFS=:
    PATH=/bin
    unset TERINFO
    set -C
    shift 5
    COMMAND4
    COMMAND5
    exit 3 # Выход только из подоболочки.
)
# Изменение переменных окружения не коснется родительской оболочки.
COMMAND6
COMMAND7
```

Как вариант использования подоболочки -- проверка переменных.

```
if (set -u; : $variable) 2> /dev/null
then
    echo "Переменная определена."
fi

# Можно сделать то же самое по другому: [[ ${variable-x} != x || ${variable-y} !=
y ]]
# или [[ ${variable-x} != x$variable ]]
# или [[ ${variable+x} = x ]]
```

Еще одно применение -- проверка файлов блокировки:

```
if (set -C; : > lock_file) 2> /dev/null
then
    echo "Этот сценарий уже запущен другим пользователем."
    exit 65
fi

# Спасибо S.C.
```

Процессы в подболочках могут выполняться параллельно. Это позволяет разбить сложную задачу на несколько простых подзадач, выполняющих параллельную обработку информации.

Пример 19-3. Запуск нескольких процессов в подболочках

```
(cat list1 list2 list3 | sort | uniq > list123) &
(cat list4 list5 list6 | sort | uniq > list456) &
# Слияние и сортировка двух списков производится одновременно.
# Запуск в фоне гарантирует параллельное исполнение.
#
# Тот же эффект дает
#   cat list1 list2 list3 | sort | uniq > list123 &
#   cat list4 list5 list6 | sort | uniq > list456 &

wait # Ожидание завершения работы подболочек.

diff list123 list456
```

Перенаправление ввода/вывода в/из подболочки производится оператором построения конвейера "|", например, `ls -al | (command)`.



Блок команд, заключенный в *фигурные скобки* не приводит к запуску дочерней подболочки.

```
{ command1; command2; command3; ... }
```

Глава 20. Ограниченный режим командной оболочки

Команды, запрещенные в ограниченном режиме командной оболочки

Запуск сценария или его части в *ограниченном* режиме, приводит к наложению ограничений на использование некоторых команд. Эта мера предназначена для ограничения привилегий пользователя, запустившего сценарий, и минимизации возможного ущерба системе, который может нанести сценарий.

В ограниченном режиме запрещена команда `cd --` смена текущего каталога.

Запрещено изменять [переменные окружения](#) `$PATH`, `$SHELL`, `$BASH_ENV` и `$ENV`.

Запрещен доступ к переменной `$SHELLOPTS`.

Запрещено перенаправление вывода.

Запрещен вызов утилит, в названии которых присутствует хотя бы один символ "слэш" (/).

Запрещен вызов команды `exec` для запуска другого процесса.

Запрещен ряд других команд, которые могут использовать сценарий для выполнения непредусмотренных действий.

Запрещен выход из ограниченного режима.

Пример 20-1. Запуск сценария в ограниченном режиме

```
#!/bin/bash
# Если sha-bang задать в таком виде: "#!/bin/bash -r"
# то это приведет к включению ограниченного режима с момента запуска скрипта.

echo

echo "Смена каталога."
cd /usr/local
echo "Текущий каталог: `pwd`"
echo "Переход в домашний каталог."
cd
echo "Текущий каталог: `pwd`"
echo

# До сих пор сценарий исполнялся в обычном, неограниченном режиме.

set -r
# set --restricted имеет тот же эффект.
echo "==> Переход в ограниченный режим. <=="

echo
echo

echo "Попытка сменить текущий каталог в ограниченном режиме."
cd ..
echo "Текущий каталог остался прежним: `pwd`"

echo
echo

echo "\$SHELL = $SHELL"
echo "Попытка смены командного интерпретатора в ограниченном режиме."
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo

echo "Попытка перенаправления вывода в ограниченном режиме."
ls -l /usr/bin > bin.files
ls -l bin.files # Попробуем найти файл, который пытались создать.

echo

exit 0
```

Глава 21. Подстановка процессов

Подстановка процессов -- это аналог [подстановки команд](#). Операция подстановки команд записывает в переменную результат выполнения некоторой команды, например, **dir_contents=`ls -al`** или **xref=\$(grep word datafile)**. Операция подстановки процессов передает вывод одного процесса на ввод другого (другими словами, передает результат выполнения одной команды -- другой).

Шаблон подстановки команды

Внутри круглых скобок

>(command)

<(command)

Таким образом иницируется подстановка процессов. Здесь, для передачи результата работы процесса в круглых скобках, используются файлы `/dev/fd/<n>`. [\[49\]](#)



Между круглой скобкой и символом "<" или ">", не должно быть пробелов, в противном случае это вызовет сообщение об ошибке.

```
bash$ echo >(true)
/dev/fd/63
```

```
bash$ echo <(true)
/dev/fd/63
```

Bash создает канал с двумя [файловыми дескрипторами](#), `--fIn` и `fOut--`. `stdin` команды `true` присоединяется к `fOut` (`dup2(fOut, 0)`), затем Bash передает `/dev/fd/fIn` в качестве аргумента команде `echo`. В системах, где отсутствуют файлы `/dev/fd/<n>`, Bash может использовать временные файлы. (Спасибо S.C.)

```
cat <(ls -l)
# То же самое, что и      ls -l | cat
```

```
sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
# Список файлов в трех основных каталогах 'bin', отсортированный по именам файлов.
# Обратите внимание: на вход 'sort' поданы три самостоятельные команды.
```

```
diff <(command1) <(command2)    # Выдаст различия в выводе команд.
```

```
tar cf >(bzip2 -c > file.tar.bz2) $directory_name
# Вызовет "tar cf /dev/fd/?? $directory_name" и затем "bzip2 -c > file.tar.bz2".
#
# Из-за особенностей, присущих некоторым системам, связанным с /dev/fd/<n>,
# канал между командами не обязательно должен быть именованным.
#
# Это можно сделать и так.
#
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 S.C.
```

Ниже приводится еще один очень интересный пример использования подстановки процессов.

```
# Фрагмент сценария из дистрибутива SuSE:
```

```
while read des what mask iface; do
# Некоторые команды ...
```



```

done < <(route -n)

# Чтобы проверить это, попробуем вставить команду, выполняющую какие либо действия.
while read des what mask iface; do
    echo $des $what $mask $iface
done < <(route -n)

# Вывод на экран:
# 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 указывает S.C. -- более простой для понимания эквивалент:
route -n |
    while read des what mask iface; do # Переменные берут значения с устройства
        echo $des $what $mask $iface # вывода конвейера (канала).
    done # На экран выводится то же самое, что и выше.
        # Однако, Ulrich Gayer отметил, что ...
        #+ этот вариант запускает цикл while в подоболочке,
        #+ и поэтому переменные не видны за пределами цикла, после закрытия канала.

```

Глава 22. Функции

Подобно "настоящим" языкам программирования, Bash тоже имеет функции, хотя и в несколько ограниченном варианте. Функция -- это подпрограмма, [блок кода](#) который реализует набор операций, своего рода "черный ящик", предназначенный для выполнения конкретной задачи. Функции могут использоваться везде, где имеются участки повторяющегося кода.

```

function function_name {
    command...
}

```

или

```

function_name () {
    command...
}

```

Вторая форма записи ближе к сердцу С-программистам (она же более переносимая).

Как и в языке С, скобка, открывающая тело функции, может помещаться на следующей строке.

```

function_name ()
{
    command...
}

```

Вызов функции осуществляется простым указанием ее имени в тексте сценария.

Пример 22-1. Простая функция

```
#!/bin/bash

funky ()
{
    echo "Это обычная функция."
} # Функция должна быть объявлена раньше, чем ее можно будет использовать.

# Вызов функции.

funky

exit 0
```

Функция должна быть объявлена раньше, чем ее можно будет использовать. К сожалению, в Bash нет возможности "опережающего объявления" функции, как например в C.

```
f1
# Эта строка вызовет сообщение об ошибке, поскольку функция "f1" еще не определена.

declare -f f1      # Это не поможет.
f1                # По прежнему -- сообщение об ошибке.

# Однако...

f1 ()
{
    echo "Вызов функции \"f2\" из функции \"f1\"."
    f2
}

f2 ()
{
    echo "Функция \"f2\"."
}

f1 # Функция "f2", фактически, не вызывается выше этой строки,
#+ хотя ссылка на нее встречается выше, до ее объявления.
# Это допускается.

# Спасибо S.C.
```

Допускается даже создание вложенных функций, хотя пользы от этого немного.

```
f1 ()
{
    f2 () # вложенная
    {
        echo "Функция \"f2\", вложенная в \"f1\"."
    }
}

f2 # Вызывает сообщение об ошибке.
# Даже "declare -f f2" не поможет.

echo
```

```
f1 # Ничего не происходит, простой вызов "f1", не означает автоматический вызов "f2".
f2 # Теперь все нормально, вызов "f2" не приводит к появлению ошибки,
    #+ поскольку функция "f2" была определена в процессе вызова "f1".

    # Спасибо S.C.
```

Объявление функции может размещаться в самых неожиданных местах.

```
ls -l | foo() { echo "foo"; } # Допустимо, но бесполезно.
```

```
if [ "$USER" = bozo ]
then
    bozo_greet () # Объявление функции размещено в условном операторе.
    {
        echo "Привет, Bozo!"
    }
fi
```

```
bozo_greet # Работает только у пользователя bozo, другие получают сообщение об ошибке.
```

```
# Нечто подобное можно использовать с определенной пользой для себя.
NO_EXIT=1 # Will enable function definition below.
```

```
[[ $NO_EXIT -eq 1 ]] && exit() { true; } # Определение функции в
последовательности "И-список".
# Если $NO_EXIT равна 1, то объявляется "exit ()".
# Тем самым, функция "exit" подменяет встроенную команду "exit".
```

```
exit # Вызывается функция "exit ()", а не встроенная команда "exit".
```

```
# Спасибо S.C.
```

22.1. Сложные функции и сложности с функциями

Функции могут принимать входные аргументы и возвращать [код завершения](#).

```
function_name $arg1 $arg2
```

Доступ к входным аргументам, в функциях, производится посредством [позиционных параметров](#), т.е. \$1, \$2 и так далее.

Пример 22-2. Функция с аргументами

```
#!/bin/bash
# Функции и аргументы

DEFAULT=default # Значение аргумента по-умолчанию.

func2 () {
    if [ -z "$1" ] # Длина аргумента #1 равна нулю?
    then
```

```

    echo "-Аргумент #1 имеет нулевую длину.-" # Или аргумент не был передан
функции.
else
    echo "-Аргумент #1: \"$1\".-"
fi

variable=${1-$DEFAULT} # Что делает
echo "variable = $variable" #+ показанная подстановка параметра?
# -----
# Она различает отсутствующий аргумент
#+ от "пустого" аргумента.

if [ "$2" ]
then
    echo "-Аргумент #2: \"$2\".-"
fi

return 0
}

echo

echo "Вызов функции без аргументов."
func2
echo

echo "Вызов функции с \"пустым\" аргументом."
func2 ""
echo

echo "Вызов функции с неинициализированным аргументом."
func2 "$uninitialized_param"
echo


echo "Вызов функции с одним аргументом."
func2 first
echo


echo "Вызов функции с двумя аргументами."
func2 first second
echo

echo "Вызов функции с аргументами \"\" \"second\"."
func2 "" second # Первый параметр "пустой"
echo # и второй параметр -- ASCII-строка.

exit 0

```

 Команда [shift](#) вполне применима и к аргументам функций (см. [Пример 33-10](#)).

 В отличие от других языков программирования, в сценариях на языке командной оболочке, в функции передаются аргументы по значению. [\[50\]](#) Если имена переменных (которые фактически являются указателями) передаются функции в виде аргументов, то они интерпретируются как обычные строки символов и не могут быть разыменованы. *Функции интерпретируют свои аргументы буквально.*

Exit и Return

код завершения

Функции возвращают значение в виде *кода завершения*. Код завершения может быть задан явно, с помощью команды **return**, в противном случае будет возвращен код завершения последней команды в функции (0 -- в случае успеха, иначе -- ненулевой код ошибки). Код завершения в сценарии может быть получен через

переменную `$?`.

return

Завершает выполнение функции. Команда **return** [51] может иметь необязательный аргумент типа *integer*, который возвращается в вызывающий сценарий как "код завершения" функции, это значение так же записывается в переменную `$?`.

Пример 22-3. Наибольшее из двух чисел

```
#!/bin/bash
# max.sh: Наибольшее из двух целых чисел.

E_PARAM_ERR=-198      # Если функции передано меньше двух параметров.
EQUAL=-199            # Возвращаемое значение, если числа равны.

max2 ()               # Возвращает наибольшее из двух чисел.
{                     # Внимание: сравниваемые числа должны быть меньше 257.
  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 $E_PARAM_ERR ]
then
  echo "Функции должно быть передано два аргумента."
elif [ "$return_val" -eq $EQUAL ]
then
  echo "Числа равны."
else
  echo "Наибольшее из двух чисел: $return_val."
fi

exit 0

# Упражнение:
# -----
# Сделайте этот сценарий интерактивным,
#+ т.е. заставьте сценарий запрашивать числа для сравнения у пользователя (два
числа).
```

 Для случаев, когда функция должна возвращать строку или массив, используйте специальные переменные.

```
count_lines_in_etc_passwd()
{
  [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
  # Если файл /etc/passwd доступен на чтение, то в переменную REPLY заносится число
  строк.
```

```

    # Возвращаются как количество строк, так и код завершения.
}

if count_lines_in_etc_passwd
then
    echo "В файле /etc/passwd найдено $REPLY строк."
else
    echo "Невозможно подсчитать число строк в файле /etc/passwd."
fi

# Спасибо S.C.

```

Пример 22-4. Преобразование чисел в римскую форму записи

```

#!/bin/bash

# Преобразование чисел из арабской формы записи в римскую
# Диапазон: 0 - 200

# Расширение диапазона представляемых чисел и улучшение сценария
# оставляю вам, в качестве упражнения.

# Порядок использования: roman number-to-convert

LIMIT=200
E_ARG_ERR=65
E_OUT_OF_RANGE=66

if [ -z "$1" ]
then
    echo "Порядок использования: `basename $0` number-to-convert"
    exit $E_ARG_ERR
fi

num=$1
if [ "$num" -gt $LIMIT ]
then
    echo "Выход за границы диапазона!"
    exit $E_OUT_OF_RANGE
fi

to_roman () # Функция должна быть объявлена до того как она будет вызвана.
{
    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
    # Упражнение:
    # -----
    # Объясните -- как работает функция.
    # Подсказка: деление последовательным вычитанием.
}

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

echo

exit 0

```

См. также [Пример 10-28](#).



Наибольшее положительное целое число, которое может вернуть функция -- 255. Конечно, очень тесно связана с понятием [код завершения](#), что объясняет это специфическое ограничение. К счастью существуют [различные способы](#) преодоления этого ограничения.

Пример 22-5. Проверка возможности возврата функциями больших значений

```

#!/bin/bash
# return-test.sh

# Наибольшее целое число, которое может вернуть функция, не может превышать 256.

return_test ()          # Просто возвращает то, что ей передали.
{
    return $1
}

return_test 27          # o.k.
echo $?                # Возвращено число 27.

return_test 255        # o.k.
echo $?                # Возвращено число 255.

return_test 257        # Ошибка!
echo $?                # Возвращено число 1.

return_test -151896    # Как бы то ни было, но для больших отрицательных чисел п
echo $?                # Возвращено число -151896.

exit 0

```

Как видно из примера, функции могут возвращать большие отрицательные значения (ввиду -- большие по своему абсолютному значению, прим. перев.). Используя эту особенность можно обыграть возможность получения от функций большие положительные значения.

Еще один способ -- использовать глобальные переменные для хранения "возвращаемых"

```

Return_Val= # Глобальная переменная, которая хранит значение, возвращаемое функцией

alt_return_test ()
{
    fvar=$1
    Return_Val=$fvar
    return # Возвратить 0 (успешное завершение).
}

alt_return_test 1

```

```

echo $? # 0
echo "Функция вернула число $Return_Val" # 1

alt_return_test 255
echo "Функция вернула число $Return_Val" # 255

alt_return_test 257
echo "Функция вернула число $Return_Val" # 257

alt_return_test 25701
echo "Функция вернула число $Return_Val" #25701

```

Пример 22-6. Сравнение двух больших целых чисел

```

#!/bin/bash
# max2.sh: Наибольшее из двух БОЛЬШИХ целых чисел.

# Это модификация предыдущего примера "max.sh",
# которая позволяет выполнять сравнение больших целых чисел.

EQUAL=0 # Если числа равны.
MAXRETVAl=255 # Максимально возможное положительное число, которое может вернуть
функция.
E_PARAM_ERR=-99999 # Код ошибки в параметрах.
E_NPARAM_ERR=99999 # "Нормализованный" код ошибки в параметрах.

max2 () # Возвращает наибольшее из двух больших целых чисел.
{
if [ -z "$2" ]
then
return $E_PARAM_ERR
fi

if [ "$1" -eq "$2" ]
then
return $EQUAL
else
if [ "$1" -gt "$2" ]
then
retval=$1
else
retval=$2
fi
fi

# ----- #
# Следующие строки позволяют "обойти" ограничение
if [ "$retval" -gt "$MAXRETVAl" ] # Если больше предельного значения,
then # то
let "retval = (( 0 - $retval ))" # изменение знака числа.
# (( 0 - $VALUE )) изменяет знак числа.
fi
# Функции имеют возможность возвращать большие *отрицательные* числа.
# ----- #

return $retval
}

max2 33001 33997
return_val=$?

# ----- #
if [ "$return_val" -lt 0 ] # Если число отрицательное,
then # то
let "return_val = (( 0 - $return_val ))" # опять изменить его знак.
fi # "Абсолютное значение" переменной $

```



```

# ----- #

if [ "$return_val" -eq "$E_NPARAM_ERR" ]
then
    # Признак ошибки в параметрах, при выходе из функции так
    # знак.
    echo "Ошибка: Недостаточно аргументов."
elif [ "$return_val" -eq "$EQUAL" ]
then
    echo "Числа равны."
else
    echo "Наибольшее число: $return_val."
fi

exit 0

```

См. также [Пример А-8](#).

Упражнение : Используя только что полученные знания, добавьте в предыдущий [преобразование чисел в римскую форму записи](#), возможность обрабатывать большие

Перенаправление

Перенаправление ввода для функций

Функции -- суть есть [блок кода](#), а это означает, что устройство `stdin` для функций может быть переопределено (перенаправление `stdin`) (как в [Пример 3-1](#)).

Пример 22-7. Настоящее имя пользователя

```

#!/bin/bash

# По имени пользователя получить его "настоящее имя" из /etc/passwd.

ARGCOUNT=1 # Ожидается один аргумент.
E_WRONGARGS=65

file=/etc/passwd
pattern=$1

if [ $# -ne "$ARGCOUNT" ]
then
    echo "Порядок использования: `basename $0` USERNAME"
    exit $E_WRONGARGS
fi

file_excerpt () # Производит поиск в файле по заданному шаблону, выводит
требуемую часть строки.
{
while read line
do
    echo "$line" | grep $1 | awk -F":" '{ print $5 }' # Указывает awk использовать
":" как разделитель полей.
done
} <$file # Подменить stdin для функции.

file_excerpt $pattern

# Да, этот сценарий можно уменьшить до
# grep PATTERN /etc/passwd | awk -F":" '{ print $5 }'
# или
# awk -F: '/PATTERN/ {print $5}'
# или
# awk -F: '($1 == "username") { print $5 }'
# Однако, это было бы не так поучительно.

```

```
exit 0
```

Ниже приводится альтернативный, и возможно менее запутанный, способ перенаправления ввода для функций. Он заключается в использовании перенаправления ввода для блока кода, заключенного в фигурные скобки, в пределах функции.

```
# Вместо:
Function ()
{
    ...
} < file

# Попробуйте так:
Function ()
{
    {
        ...
    } < file
}

# Похожий вариант,
Function () # Тоже работает.
{
    {
        echo $*
    } | tr a b
}

Function () # Этот вариант не работает.
{
    echo $*
} | tr a b # Наличие вложенного блока кода -- обязательное условие.

# Спасибо S.C.
```

22.2. Локальные переменные

Что такое "локальная" переменная?

локальные переменные

Переменные, объявленные как *локальные*, имеют ограниченную область видимости, и доступны только в пределах [блока](#), в котором они были объявлены. Для функций это означает, что локальная переменная "видна" только в теле самой функции.

Пример 22-8. Область видимости локальных переменных

```
#!/bin/bash

func ()
{
    local loc_var=23          # Объявление локальной переменной.
    echo
    echo "\"loc_var\" в функции = $loc_var"
    global_var=999          # Эта переменная не была объявлена локальной.
```

```

    echo "\"global_var\" в функции = $global_var"
}

func

# Проверим, "видна" ли локальная переменная за пределами функции.

echo
echo "\"loc_var\" за пределами функции = $loc_var"
# "loc_var" за пределами функции =
# Итак, $loc_var не видна в глобальном
контексте.
echo "\"global_var\" за пределами функции = $global_var"
# "global_var" за пределами функции = 999
# $global_var имеет глобальную область
видимости.
echo

exit 0

```



Переменные, объявляемые в теле функции, считаются необъявленными до тех пор, пока функция не будет вызвана. Это касается *всех* переменных.

```

#!/bin/bash

func ()
{
global_var=37      # Эта переменная будет считаться необъявленной
                  #+ до тех пор, пока функция не будет вызвана.
}                  # КОНЕЦ ФУНКЦИИ

echo "global_var = $global_var" # global_var =
                          # Функция "func" еще не была вызвана,
                          #+ поэтому $global_var пока еще не "видна"
здесь.

func
echo "global_var = $global_var" # global_var = 37
                          # Переменная была инициализирована в функции.

```

22.2.1. Локальные переменные делают возможной рекурсию.

Хотя локальные переменные и допускают рекурсию, [\[52\]](#) но она сопряжена с большими накладными расходами и не рекомендуется для использования в сценариях. [\[53\]](#)

Пример 22-9. Использование локальных переменных при рекурсии

```

#!/bin/bash

#                факториал
#                -----

# Действительно ли bash допускает рекурсию?
# Да! Но...
# Нужно быть действительно дубинноголовым, чтобы использовать ее в сценариях
# на языке командной оболочки.

MAX_ARG=5
E_WRONG_ARGS=65
E_RANGE_ERR=66

```

```

if [ -z "$1" ]
then
    echo "Порядок использования: `basename $0` число"
    exit $_WRONG_ARGS
fi

if [ "$1" -gt $MAX_ARG ]
then
    echo "Выход за верхний предел (максимально возможное число -- 5)."
    # Вернитесь к реальности.
    # Если вам захочется поднять верхнюю границу,
    # то перепишите эту программу на настоящем языке программирования.
    exit $_RANGE_ERR
fi

fact ()
{
    local number=$1
    # Переменная "number" должна быть объявлена как локальная,
    # иначе результат будет неверный.
    if [ "$number" -eq 0 ]
    then
        factorial=1      # Факториал числа 0 = 1.
    else
        let "decrnum = number - 1"
        fact $decrnum # Рекурсивный вызов функции.
        let "factorial = $number * $?"
    fi

    return $factorial
}

fact $1
echo "Факториал числа $1 = $?."

exit 0

```

Еще один пример использования рекурсии вы найдете в [Пример А-18](#). Не забывайте, что рекурсия весьма ресурсоемкое удовольствие, к тому же она выполняется слишком медленно, поэтому не следует использовать ее в сценариях.

Глава 23. Псевдонимы

Псевдонимы в Bash -- это ни что иное, как "горячие клавиши", средство, позволяющее избежать набора длинных строк в командной строке. Если, к примеру, в файл [~/ .bashrc](#) вставить строку **alias lm="ls -l | more"**, то потом вы сможете экономить свои силы и время, набирая команду **lm**, вместо более длинной **ls -l | more**. Установив **alias rm="rm -i"** (интерактивный режим удаления файлов), вы сможете избежать многих неприятностей, потому что сократится вероятность удаления важных файлов по неосторожности.

Псевдонимы в сценариях могут иметь весьма ограниченную область применения. Было бы здорово, если бы псевдонимы имели функциональность, присущую макроопределениям в языке C, но, к сожалению, Bash не может "разворачивать" аргументы в теле псевдонима. [\[54\]](#) Кроме того, попытка обратиться к псевдониму, созданному внутри "составных конструкций", таких как [if/then](#), циклы и функции, будет приводить к появлению ошибок. Практически всегда, действия, возлагаемые на псевдоним, более эффективно могут быть выполнены с помощью [функций](#).

Пример 23-1. Псевдонимы в сценарии

```
#!/bin/bash

shopt -s expand_aliases
# Эта опция должна быть включена, иначе сценарий не сможет "разворачивать"
псевдонимы.

alias ll="ls -l"
# В определении псевдонима можно использовать как одиночные ('), так и двойные (")
кавычки.

echo "Попытка обращения к псевдониму \"ll\": "
ll /usr/X11R6/bin/mk*    #* Работает.

echo

directory=/usr/X11R6/bin/
prefix=mk* # Определить -- не будет ли проблем с шаблонами.
echo "Переменные \"directory\" + \"prefix\" = $directory$prefix"
echo

alias lll="ls -l $directory$prefix"

echo "Попытка обращения к псевдониму \"lll\": "
lll # Список всех файлов в /usr/X11R6/bin, чьи имена начинаются с mk.
# Псевдонимы могут работать с шаблонами.

TRUE=1

echo

if [ TRUE ]
then
    alias rr="ls -l"
    echo "Попытка обращения к псевдониму \"rr\", созданному внутри if/then:"
    rr /usr/X11R6/bin/mk*    #* В результате -- сообщение об ошибке!
    # К псевдонимам, созданным внутри составных инструкций, нельзя обратиться.
    echo "Однако, ранее созданный псевдоним остается работоспособным:"
    ll /usr/X11R6/bin/mk*
fi

echo

count=0
while [ $count -lt 3 ]
do
    alias rrr="ls -l"
    echo "Попытка обращения к псевдониму \"rrr\", созданному внутри цикла \"while\": "
    rrr /usr/X11R6/bin/mk*    #* Так же возникает ошибка.
    # alias.sh: line 57: rrr: command not found

    let count+=1
done

echo; echo

alias xyz='cat $0' # Сценарий печатает себя самого.
                  # Обратите внимание на "строгие" кавычки.

xyz
# Похоже работает,
#+ хотя документация Bash утверждает, что такой псевдоним не должен работать.
#
# Steve Jacobson отметил, что
#+ параметр "$0" интерпретируется непосредственно, во время объявления псевдонима.

exit 0
```

Команда **unalias** удаляет псевдоним, объявленный ранее .

Пример 23-2. unalias: Объявление и удаление псевдонимов

```
#!/bin/bash

shopt -s expand_aliases # Разрешить "разворачивание" псевдонимов.

alias llm='ls -al | more'
llm

echo

unalias llm          # Удалить псевдоним.
llm
# Сообщение об ошибке, т.к. команда 'llm' больше не распознается.

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: command not found
```

Глава 24. Списки команд

Средством обработки последовательности из нескольких команд служат списки: "И-списки" и "ИЛИ-списки". Они эффективно могут заменить сложную последовательность вложенных **if/then** или даже **case**.

Объединение команд в цепочки

И-список

```
command-1 && command-2 && command-3 && ... command-n
```

Каждая последующая команда, в таком списке, выполняется только тогда, когда предыдущая команда вернула код завершения true (ноль). Если какая-либо из команд возвращает false (не ноль), то исполнение списка команд в этом месте завершается, т.е. следующие далее команды не выполняются.

Пример 24-1. Проверка аргументов командной строки с помощью "И-списка"

```
#!/bin/bash
# "И-список"

if [ ! -z "$1" ] && echo "Аргумент #1 = $1" && [ ! -z "$2" ] && echo "Аргумент #2 = $2"
then
    echo "Сценарию передано не менее 2 аргументов."
    # Все команды в цепочке возвращают true.
else
    echo "Сценарию передано менее 2 аргументов."
    # Одна из команд в списке вернула false.
fi
# Обратите внимание: "if [ ! -z $1 ]" тоже работает, но, казалось бы
```

```

эквивалентный вариант
# if [ -n $1 ] -- нет. Однако, если добавить кавычки
# if [ -n "$1" ] то все работает. Будьте внимательны!
# Проверяемые переменные лучше всегда заключать в кавычки.

# То же самое, только без списка команд.
if [ ! -z "$1" ]
then
    echo "Аргумент #1 = $1"
fi
if [ ! -z "$2" ]
then
    echo "Аргумент #2 = $2"
    echo "Сценарию передано не менее 2 аргументов."
else
    echo "Сценарию передано менее 2 аргументов."
fi
# Получилось менее элегантно и длиннее, чем с использованием "И-списка".

exit 0

```

Пример 24-2. Еще один пример проверки аргументов с помощью "И-списков"

```

#!/bin/bash

ARGS=1      # Ожидаемое число аргументов.
E_BADARGS=65 # Код завершения, если число аргументов меньше ожидаемого.

test $# -ne $ARGS && echo "Порядок использования: `basename $0` $ARGS
аргумент(а)(ов)" && exit $E_BADARGS
# Если проверка первого условия возвращает true (неверное число аргументов),
# то выполняется оставшая часть строки, и сценарий завершается.

# Строка ниже выполняется только тогда, когда проверка выше не проходит.
# обратите внимание на условие "-ne" -- "не равно" (прим. перев.)
echo "Сценарию передано корректное число аргументов."

exit 0

# Проверьте код завершения сценария командой "echo $?".

```

Конечно же, с помощью *И-списка* можно присваивать переменным значения по умолчанию.

```

arg1=$@      # В $arg1 записать аргументы командной строки.

[ -z "$arg1" ] && arg1=DEFAULT
                # Записать DEFAULT, если аргументы командной строки отсутствуют.

```

ИЛИ-список

```

command-1 || command-2 || command-3 || ... command-n

```

Каждая последующая команда, в таком списке, выполняется только тогда, когда предыдущая команда вернула код завершения false (не ноль). Если какая-либо из команд возвращает true (ноль), то исполнение списка команд в этом месте завершается, т.е. следующие далее команды не выполняются. Очевидно, что "ИЛИ-списки" имеют смысл обратный, по отношению к "И-спискам"

Пример 24-3. Комбинирование "ИЛИ-списков" и "И-списков"

```
#!/bin/bash

# delete.sh, утилита удаления файлов.
# Порядок использования: delete имя_файла

E_BADARGS=65

if [ -z "$1" ]
then
    echo "Порядок использования: `basename $0` имя_файла"
    exit $E_BADARGS # Если не задано имя файла.
else
    file=$1          # Запомнить имя файла.
fi

[ ! -f "$file" ] && echo "Файл \"$file\" не найден. \
Робкий отказ удаления несуществующего файла."
# И-СПИСОК, выдать сообщение об ошибке, если файл не существует.
# Обратите внимание: выводимое сообщение продолжается во второй строке,
# благодаря экранированию символа перевода строки.

[ ! -f "$file" ] || (rm -f $file; echo "Файл \"$file\" удален.")
# ИЛИ-СПИСОК, удаляет существующий файл.

# Обратите внимание на логические условия.
# И-СПИСОК обрабатывает по true, ИЛИ-СПИСОК -- по false.

exit 0
```

 Списки возвращают [код завершения](#) последней выполненной команды.

Комбинируя "И" и "ИЛИ" списки, легко "перемудрить" с логическими условиями, поэтому, в таких случаях может потребоваться детальная отладка.

```
false && true || echo false          # false

# Тот же результат дает
( false && true ) || echo false      # false
# Но не эта комбинация
false && ( true || echo false )     # (нет вывода на экран)

# Обратите внимание на группировку и порядок вычисления условий -- слева-направо,
#+ поскольку логические операции "&&" и "||" имеют равный приоритет.

# Если вы не уверены в своих действиях, то лучше избегать таких сложных конструкций.

# Спасибо S.C.
```

См. [Пример А-8](#) и [Пример 7-4](#), иллюстрирующие использование И / ИЛИ - списков для проверки переменных.

Глава 25. Массивы

Новейшие версии Bash поддерживают одномерные массивы. Инициализация элементов массива может быть произведена в виде: `variable [xx]`. Можно явно объявить массив в сценарии, с помощью директивы `declare -a variable`. Обращаться к отдельным элементам массива можно с помощью *фигурных скобок*, т.е.: `$`


```
{variable[xx]}.
```

Пример 25-1. Простой массив

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

```
area[11]=23  
area[13]=37  
area[51]=UF0s
```

```
# Массивы не требуют, чтобы последовательность элементов в массиве была непрерывной.
```

```
# Некоторые элементы массива могут оставаться неинициализированными.  
# "Дыркм" в массиве не являются ошибкой.
```

```
echo -n "area[11] = "  
echo ${area[11]} # необходимы {фигурные скобки}
```

```
echo -n "area[13] = "  
echo ${area[13]}
```

```
echo "содержимое area[51] = ${area[51]}."
```

```
# Обращение к неинициализированным элементам дает пустую строку.
```

```
echo -n "area[43] = "  
echo ${area[43]}  
echo "(элемент area[43] -- неинициализирован)"
```

```
echo
```

```
# Сумма двух элементов массива, записанная в третий элемент
```

```
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]}
```

```
# Эта попытка закончится неудачей, поскольку сложение целого числа со строкой не  
допускается.
```

```
echo; echo; echo
```

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

```
# Другой массив, "area2".
```

```
# И другой способ инициализации массива...
```

```
# array_name=( XXX YYY ZZZ ... )
```

```
area2=( ноль один два три четыре )
```

```
echo -n "area2[0] = "
```

```
echo ${area2[0]}
```

```
# Ага, индексация начинается с нуля (первый элемент массива имеет индекс [0], а не  
[1]).
```

```
echo -n "area2[1] = "
```

```
echo ${area2[1]} # [1] -- второй элемент массива.
```

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

```
echo; echo; echo
```

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

```
# Еще один массив, "area3".
```

```
# И еще один способ инициализации...
```

```
# array_name=( [xx]=XXX [yy]=YYY ... )

area3=( [17]=семнадцать [21]=двадцать_один )

echo -n "area3[17] = "
echo ${area3[17]}

echo -n "area3[21] = "
echo ${area3[21]}
# -----

exit 0
```



Bash позволяет оперировать переменными, как массивами, даже если они не были явно объявлены таковыми.

```
string=abcABC123ABCabc
echo ${string[@]}           # abcABC123ABCabc
echo ${string[*]}          # abcABC123ABCabc
echo ${string[0]}          # abcABC123ABCabc
echo ${string[1]}          # Ничего не выводится!
                           # Почему?
echo $#string[@]           # 1
                           # Количество элементов в массиве.
```

Спасибо Michael Zick за этот пример.

Эти примеры еще раз подтверждают [отсутствие контроля типов в Bash](#).

Пример 25-2. Форматирование стихотворения

```
#!/bin/bash
# поем.sh

# Строки из стихотворения (одна строфа).
Line[1]="Мой дядя самых честных правил,"
Line[2]="Когда не в шутку занемог;"
Line[3]="Он уважать себя заставил,"
Line[4]="И лучше выдумать не мог."
Line[5]="Его пример другим наука..."

# Атрибуты.
Attrib[1]=" А.С. Пушкин"
Attrib[2]="\"Евгений Онегин\""

for index in 1 2 3 4 5      # Пять строк.
do
    printf "      %s\n" "${Line[index]}"
done

for index in 1 2           # Две строки дополнительных атрибутов.
do
    printf "      %s\n" "${Attrib[index]}"
done

exit 0
```

При работе с отдельными элементами массива можно использовать специфический синтаксис, даже стандартные команды и операторы Bash адаптированы для работы с массивами.

```
array=( ноль один два три четыре пять )

echo ${array[0]}           # ноль
echo ${array:0}           # ноль
                           # Подстановка параметра -- первого элемента.
echo ${array:1}           # оль
```

```

# Подстановка параметра -- первого элемента,
#+ начиная с позиции #1 (со 2-го символа).

echo ${#array}      # 4
                   # Длина первого элемента массива.

array2=( [0]="первый элемент" [1]="второй элемент" [3]="четвертый элемент" )

echo ${array2[0]}   # первый элемент
echo ${array2[1]}   # второй элемент
echo ${array2[2]}   #
                   # Элемент неинициализирован, поэтому на экран ничего не
выводится.
echo ${array2[3]}   # четвертый элемент

```

При работе с массивами, некоторые [встроенные команды](#) Bash имеют несколько иной смысл. Например, [unset](#) -- удаляет отдельные элементы массива, или даже массив целиком.

Пример 25-3. Некоторые специфичные особенности массивов

```

#!/bin/bash

declare -a colors
# Допускается объявление массива без указания его размера.

echo "Введите ваши любимые цвета (разделяя их пробелами)."

```

```

echo ${colors[@]}          # ${colors[*]} дает тот же результат.

echo

# Команда "unset" удаляет элементы из массива, или даже массив целиком.
unset colors[1]           # Удаление 2-го элемента массива.
                        # Тот же эффект дает команда colors[1]=
echo ${colors[@]}        # Список всех элементов массива -- 2-й элемент
                        # отсутствует.

unset colors              # Удаление всего массива.
                        # Тот же эффект имеют команды unset colors[*]
                        #+ и unset colors[@].

echo; echo -n "Массив цветов опустошен."
echo ${colors[@]}        # Список элементов массива пуст.

exit 0

```

Как видно из предыдущего примера, обращение к **`${array_name[@]}`** или **`${array_name[*]}`** относится ко *всем* элементам массива. Чтобы получить количество элементов массива, можно обратиться к **`${#array_name[@]}`** или к **`${#array_name[*]}`**. **`${#array_name}`** -- это длина (количество символов) первого элемента массива, т.е. **`${array_name[0]}`**.

Пример 25-4. Пустые массивы и пустые элементы

```

#!/bin/bash
# empty-array.sh

# Выражаю свою благодарность Stephane Chazelas за этот пример,
#+ и Michael Zick за его доработку.

# Пустой массив -- это не то же самое, что массив с пустыми элементами.

array0=( первый второй третий )
array1=( ' ' ) # "array1" имеет один пустой элемент.
array2=( )     # Массив "array2" не имеет ни одного элемента, т.е. пуст.

echo
ListArray()
{
echo
echo "Элементы массива array0:  ${array0[@]}"
echo "Элементы массива array1:  ${array1[@]}"
echo "Элементы массива array2:  ${array2[@]}"
echo
echo "Длина первого элемента массива array0 = ${#array0}"
echo "Длина первого элемента массива array1 = ${#array1}"
echo "Длина первого элемента массива array2 = ${#array2}"
echo
echo "Число элементов в массиве array0 = ${#array0[*]}" # 3
echo "Число элементов в массиве array1 = ${#array1[*]}" # 1 (сюрприз!)
echo "Число элементов в массиве array2 = ${#array2[*]}" # 0
}

# =====

ListArray

# Попробуем добавить новые элементы в массивы

# Добавление новых элементов в массивы.
array0=( "${array0[@]}" "новый1" )
array1=( "${array1[@]}" "новый1" )
array2=( "${array2[@]}" "новый1" )

```

ListArray

```
# или
array0[${#array0[*]}]="новый2"
array1[${#array1[*]}]="новый2"
array2[${#array2[*]}]="новый2"
```

ListArray

```
# Теперь представим каждый массив как 'стек' ('stack')
# Команды выше, можно считать командами 'push' -- добавление нового значения на
# вершину стека
# 'Глубина' стека:
height=${#array2[@]}
echo
echo "Глубина стека array2 = $height"
```

```
# Команда 'pop' -- выталкивание элемента стека, находящегося на вершине:
unset array2[${#array2[@]}-1] # Индексация массивов начинается с нуля
height=${#array2[@]}
echo
echo "POP"
echo "Глубина стека array2, после выталкивания = $height"
```

ListArray

```
# Вывести только 2-й и 3-й элементы массива array0
from=1          # Индексация массивов начинается с нуля
to=2            #
declare -a array3=( ${array0[@]:1:2} )
echo
echo "Элементы массива array3:  ${array3[@]}"
```

```
# Замена элементов по шаблону
declare -a array4=( ${array0[@]/второй/2-й} )
echo
echo "Элементы массива array4:  ${array4[@]}"
```

```
# Замена строк по шаблону
declare -a array5=( ${array0[@]//новый?/старый} )
echo
echo "Элементы массива array5:  ${array5[@]}"
```

```
# Надо лишь привыкнуть к такой записи...
declare -a array6=( ${array0[@]#*новый} )
echo # Это может вас несколько удивить
echo "Элементы массива array6:  ${array6[@]}"
```

```
declare -a array7=( ${array0[@]#новый1} )
echo # Теперь это вас уже не должно удивлять
echo "Элементы массива array7:  ${array7[@]}"
```

```
# Выглядит очень похоже на предыдущий вариант...
declare -a array8=( ${array0[@]/новый1/} )
echo
echo "Элементы массива array8:  ${array8[@]}"
```

```
# Итак, что вы можете сказать обо всем этом?
```

```
# Строковые операции выполняются последовательно, над каждым элементом
#+ в массиве var[@].
# Таким образом, BASH поддерживает векторные операции
# Если в результате операции получается пустая строка, то
#+ элемент массива "исчезает".
```

```
# Вопрос: это относится к строкам в "строгих" или "мягких" кавычках?
```

```
zap='новый*'
declare -a array9=( ${array0[@]/$zap/} )
```

```

echo
echo "Элементы массива array9:  ${array9[@]}"

# "...А с платформы говорят: "Это город Ленинград!"..."
declare -a array10=( ${array0[@]#$zap} )
echo
echo "Элементы массива array10:  ${array10[@]}"

# Сравните массивы array7 и array10
# Сравните массивы array8 и array9

# Ответ: в "мягких" кавычках.

exit 0

```

Разница между **`${array_name[@]}`** и **`${array_name[*]}`** такая же, как между **`$@`** и **`$*`**. Эти свойства массивов широко применяются на практике.


```

# Копирование массивов.
array2=( "${array1[@]}" )
# или
array2="${array1[@]}"

# Добавить элемент.
array=( "${array[@]}" "новый элемент" )
# или
array[${#array[*]}]="новый элемент"

# Спасибо S.C.

```

 **Операция [подстановки команд](#) -- `array=(element1 element2 ... elementN)`, позволяет загружать содержимое текстовых файлов в массивы.**

```

#!/bin/bash

filename=sample_file

#           cat sample_file
#
#           1 a b c
#           2 d e fg

declare -a array1

array1=( `cat "$filename" | tr '\n' ' '` ) # Загрузка содержимого файла
#                                           # $filename в массив array1.

#           Вывод на stdout.
#           с заменой символов перевода строки на пробелы.

echo ${array1[@]}           # список элементов массива.
#                           1 a b c 2 d e fg
#
# Каждое "слово", в текстовом файле, отделяемое от других пробелами
#+ заносится в отдельный элемент массива.

element_count=${#array1[*]}
echo $element_count        # 8

```

Пример 25-5. Копирование и конкатенация массивов

```

#!/bin/bash
# CopyArray.sh

```

```

#
# Автор: Michael Zick.
# Используется с его разрешения.

# "Принять из массива с заданным именем записать в массив с заданным именем"
#+ или "собственный Оператор Присваивания".

```

```

CpArray_Mac() {

```

```

# Оператор Присваивания

```

```

    echo -n 'eval '
    echo -n "$2" # Имя массива-результата
    echo -n '=( ${'
    echo -n "$1" # Имя исходного массива
    echo -n '[@] } )'

```

```

# Все это могло бы быть объединено в одну команду.
# Это лишь вопрос стиля.
}

```

```

declare -f CopyArray # "Указатель" на функцию
CopyArray=CpArray_Mac # Оператор Присваивания

```

```

Hype()
{

```

```

# Исходный массив с именем в $1.
# (Слить с массивом, содержащим "-- Настоящий Рок-н-Ролл".)
# Вернуть результат в массиве с именем $2.

```

```

    local -a TMP
    local -a hype=( -- Настоящий Рок-н-Ролл )

    $($CopyArray $1 TMP)
    TMP=( ${TMP[@]} ${hype[@]} )
    $($CopyArray TMP $2)
}

```

```

declare -a before=( Advanced Bash Scripting )
declare -a after

```

```

echo "Массив before = ${before[@]}"

```

```

Hype before after

```

```

echo "Массив after = ${after[@]}"

```

```

# Еще?

```

```

echo "Что такое ${after[@]:4:2}?"

```

```

declare -a modest=( ${after[@]:2:1} ${after[@]:3:3} )
# ----- выделение подстроки -----

```

```

echo "Массив Modest = ${modest[@]}"

```

```

# А что в массиве 'before' ?

```

```

echo "Массив Before = ${before[@]}"

```

```

exit 0

```

```

--

```

Массивы допускают перенос хорошо известных алгоритмов в сценарии на языке командной оболочки. Хорошо ли это -- решать вам.

Пример 25-6. Старая, добрая: "Пузырьковая" сортировка

```
#!/bin/bash
# bubble.sh: "Пузырьковая" сортировка.

# На каждом проходе по сортируемому массиву,
#+ сравниваются два смежных элемента, и, если необходимо, они меняются местами.
# В конце первого прохода, самый "тяжелый" элемент "опускается" в конец массива.
# В конце второго прохода, следующий по "тяжести" элемент занимает второе место
снизу.
# И так далее.
# Каждый последующий проход требует на одно сравнение меньше предыдущего.
# Поэтому вы должны заметить ускорение работы сценария на последних проходах.

exchange()
{
    # Поменять местами два элемента массива.
    local temp=${Countries[$1]} # Временная переменная
    Countries[$1]=${Countries[$2]}
    Countries[$2]=$temp

    return
}

declare -a Countries # Объявление массива,
                    #+ необязательно, поскольку он явно инициализируется ниже.

# Допустимо ли выполнять инициализацию массива в несколько строках?
# ДА!

Countries=(Нидерланды Украина Заир Турция Россия Йемен Сирия \
Бразилия Аргентина Никарагуа Япония Мексика Венесуэла Греция Англия \
Израиль Перу Канада Оман Дания Уэльс Франция Кения \
Занаду Катар Лихтенштейн Венгрия)

# "Занаду" -- это мифическое государство, где, согласно Coleridge,
#+ Kubla Khan построил величественный дворец.

clear # Очистка экрана.

echo "0: ${Countries[*]}" # Список элементов несортированного массива.

number_of_elements=${#Countries[@]}
let "comparisons = $number_of_elements - 1"

count=1 # Номер прохода.

while [ "$comparisons" -gt 0 ] # Начало внешнего цикла
do

    index=0 # Сбросить индекс перед началом каждого прохода.

    while [ "$index" -lt "$comparisons" ] # Начало внутреннего цикла
    do
        if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`] } ]
        # Если элементы стоят не по порядку...
        # Оператор \> выполняет сравнение ASCII-строк
        #+ внутри одиночных квадратных скобок.

        # if [ [ ${Countries[$index]} > ${Countries[`expr $index + 1`] } ] ]
        #+ дает тот же результат.
        then
            exchange $index `expr $index + 1` # Поменять местами.
        fi
        let "index += 1"
    done # Конец внутреннего цикла
```



```

let "comparisons -= 1" # Поскольку самый "тяжелый" элемент уже "опустился" на дно,
                      #+ то на каждом последующем проходе нужно выполнять на одно
сравнение меньше.

echo
echo "$count: ${Countries[@]}" # Вывести содержимое массива после каждого прохода.
echo
let "count += 1"              # Увеличить счетчик проходов.

done                          # Конец внешнего цикла

exit 0

--

```

Можно ли вложить один массив в другой?

```

#!/bin/bash
# Вложенный массив.

# Автор: Michael Zick.

AnArray=( $(ls --inode --ignore-backups --almost-all \
            --directory --full-time --color=none --time=status \
            --sort=time -l ${PWD} ) ) # Команды и опции.

# Пробелы важны . . .

SubArray=( ${AnArray[@]:11:1} ${AnArray[@]:6:5} )
# Массив имеет два элемента, каждый из которых, в свою очередь, является массивом.

echo "Текущий каталог и дата последнего изменения:"
echo "${SubArray[@]}"

exit 0

--

```

Вложенные массивы, в комбинации с [косвенными ссылками](#), предоставляют в распоряжение программиста ряд замечательных возможностей

Пример 25-7. Вложенные массивы и косвенные ссылки

```

#!/bin/bash
# embedded-arrays.sh
# Вложенные массивы и косвенные ссылки.

# Автор: Dennis Leeuw.
# Используется с его разрешения.
# Дополнен автором документа.

ARRAY1=(
    VAR1_1=value11
    VAR1_2=value12
    VAR1_3=value13
)

ARRAY2=(
    VARIABLE="test"
    STRING="VAR1=value1 VAR2=value2 VAR3=value3"
    ARRAY21=${ARRAY1[*]}
) # Вложение массива ARRAY1 в массив ARRAY2.

```

```

function print () {
    OLD_IFS="$IFS"
    IFS=$'\n'          # Вывод каждого элемента массива
                      #+ в отдельной строке.
    TEST1="ARRAY2[*]"
    local ${!TEST1} # Посмотрите, что произойдет, если убрать эту строку.
    # Косвенная ссылка.
    # Позволяет получить доступ к компонентам $TEST1
    #+ в этой функции.

    # Посмотрим, что получилось.
    echo
    echo "\$TEST1 = $TEST1"          # Просто имя переменной.
    echo; echo
    echo "${!TEST1} = ${!TEST1}" # Вывод на экран содержимого переменной.
                                # Это то, что дает
                                #+ косвенная ссылка.

    echo
    echo "-----"; echo
    echo

    # Вывод переменной
    echo "Переменная VARIABLE: $VARIABLE"

    # Вывод элементов строки
    IFS="$OLD_IFS"
    TEST2="STRING[*]"
    local ${!TEST2}          # Косвенная ссылка (то же, что и выше).
    echo "Элемент VAR2: $VAR2 из строки STRING"

    # Вывод элемента массива
    TEST2="ARRAY21[*]"
    local ${!TEST2}          # Косвенная ссылка.
    echo "Элемент VAR1_1: $VAR1_1 из массива ARRAY21"
}

print
echo

exit 0

--

```

С помощью массивов, на языке командной оболочки, вполне возможно реализовать алгоритм *Решето Эратосфена*. Конечно же -- это очень ресурсоемкая задача. В виде сценария она будет работать мучительно долго, так что лучше всего реализовать ее на каком либо другом, компилирующем, языке программирования, таком как С.

Пример 25-8. Пример реализации алгоритма *Решето Эратосфена*

```

#!/bin/bash
# sieve.sh

# Решето Эратосфена
# Очень старый алгоритм поиска простых чисел.

# Этот сценарий выполняется во много раз медленнее
# чем аналогичная программа на С.

LOWER_LIMIT=1          # Начиная с 1.
UPPER_LIMIT=1000       # До 1000.
# (Вы можете установить верхний предел и выше... если вам есть чем себя занять.)

PRIME=1

```

```

NON_PRIME=0

declare -a Primes
# Primes[] -- массив.

initialize ()
{
# Инициализация массива.

i=$LOWER_LIMIT
until [ "$i" -gt "$UPPER_LIMIT" ]
do
    Primes[i]=$PRIME
    let "i += 1"
done
# Все числа в заданном диапазоне считать простыми,
# пока не доказано обратное.
}

print_primes ()
{
# Вывод индексов элементов массива Primes[], которые признаны простыми.

i=$LOWER_LIMIT

until [ "$i" -gt "$UPPER_LIMIT" ]
do

    if [ "${Primes[i]}" -eq "$PRIME" ]
    then
        printf "%8d" $i
        # 8 пробелов перед числом придают удобочитаемый табличный вывод на экран.
    fi

    let "i += 1"
done

}

sift () # Отсеивание составных чисел.
{

let i=$LOWER_LIMIT+1
# Нам известно, что 1 -- это простое число, поэтому начнем с 2.

until [ "$i" -gt "$UPPER_LIMIT" ]
do

if [ "${Primes[i]}" -eq "$PRIME" ]
# Не следует проверять вторично числа, которые уже признаны составными.
then

    t=$i

    while [ "$t" -le "$UPPER_LIMIT" ]
    do
        let "t += $i "
        Primes[t]=$NON_PRIME
        # Все числа, которые делятся на $t без остатка, пометить как составные.
    done

fi

    let "i += 1"
done
}

```

```

}

# Вызов функций.
initialize
sift
print_primes
# Это называется структурным программированием.

echo

exit 0

# ----- #
# Код, приведенный ниже, не исполняется из-за команды exit, стоящей выше.

# Улучшенная версия, предложенная Stephane Chazelas,
# работает несколько быстрее.

# Должен вызываться с аргументом командной строки, определяющем верхний предел.

UPPER_LIMIT=$1          # Из командной строки.
let SPLIT=UPPER_LIMIT/2 # Рассматривать делители только до середины
диапазона.

Primes=( ' $(seq $UPPER_LIMIT) )

i=1
until (( ( i += 1 ) > SPLIT )) # Числа из верхней половины диапазона могут не
рассматриваться.
do
  if [[ -n $Primes[i] ]]
  then
    t=$i
    until (( ( t += i ) > UPPER_LIMIT ))
    do
      Primes[t]=
    done
  fi
done
echo ${Primes[*]}

exit 0

```

Сравните этот сценарий с генератором простых чисел, не использующим массивов, [Пример А-18](#).

--

Массивы позволяют эмулировать некоторые структуры данных, поддержка которых в Bash не предусмотрена.

Пример 25-9. Эмуляция структуры "СТЕК" ("первый вошел -- последний вышел")

```

#!/bin/bash
# stack.sh: Эмуляция структуры "СТЕК" ("первый вошел -- последний вышел")

# Подобно стеку процессора, этот "стек" сохраняет и возвращает данные по принципу
#+ "первый вошел -- последний вышел".

BP=100          # Базовый указатель на массив-стек.
                # Дно стека -- 100-й элемент.

SP=$BP         # Указатель вершины стека.
                # Изначально -- стек пуст.

```

```

Data=                # Содержимое вершины стека.
                    # Следует использовать дополнительную переменную,
                    #+ из-за ограничений на диапазон возвращаемых функциями значений.

declare -a stack

push()               # Поместить элемент на вершину стека.
{
if [ -z "$1" ]      # А вообще, есть что помещать на стек?
then
    return
fi

let "SP -= 1"       # Переместить указатель стека.
stack[$SP]=$1

return
}

pop()                # Снять элемент с вершины стека.
{
Data=               # Очистить переменную.

if [ "$SP" -eq "$BP" ] # Стек пуст?
then
    return
fi                  # Это предохраняет от выхода SP за границу стека -- 100,

Data=${stack[$SP]}
let "SP += 1"       # Переместить указатель стека.
return
}

status_report()     # Вывод вспомогательной информации.
{
echo "-----"
echo "ОТЧЕТ"
echo "Указатель стека SP = $SP"
echo "Со стека был снят элемент \"'$Data'\""
echo "-----"
echo
}

# =====
# А теперь позабавимся.

echo

# Попробуем вытолкнуть что-нибудь из пустого стека.
pop
status_report

echo

push garbage
pop
status_report      # Втолкнуть garbage, вытолкнуть garbage.

value1=23; push $value1
value2=skidoo; push $value2
value3=FINAL; push $value3

pop                # FINAL
status_report
pop                # skidoo
status_report

```

```

pop                # 23
status_report     # Первый вошел -- последний вышел!

# Обратите внимание как изменяется указатель стека на каждом вызове функций push и
pop.

echo
# =====

# Упражнения:
# -----

# 1) Измените функцию "push()" таким образом,
# + чтобы она позволяла помещать на стек несколько значений за один вызов.

# 2) Измените функцию "pop()" таким образом,
# + чтобы она позволяла снимать со стека несколько значений за один вызов.

# 3) Попробуйте написать простейший калькулятор, выполняющий 4 арифметических
действия?
# + используя этот пример.

exit 0

--

```

Иногда, манипуляции с "индексами" массивов могут потребовать введения переменных для хранения промежуточных результатов. В таких случаях вам предоставляется лишний повод подумать о реализации проекта на более мощном языке программирования, например Perl или C.

Пример 25-10. Исследование математических последовательностей

```

#!/bin/bash

# Пресловутая "Q-последовательность" Дугласа Хольфштадтера (*Douglas Hofstadter):

# Q(1) = Q(2) = 1
# Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), для n>2

# Это "хаотическая" последовательность целых чисел с непредсказуемым поведением.
# Первые 20 членов последовательности:
# 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12

# См. книгу Дугласа Хольфштадтера, "Goedel, Escher, Bach: An Eternal Golden Braid",
# p. 137, ff.

LIMIT=100      # Найти первые 100 членов последовательности
LINEWIDTH=20   # Число членов последовательности, выводимых на экран в одной строке

Q[1]=1         # Первые два члена последовательности равны 1.
Q[2]=1

echo
echo "Q-последовательность [первые $LIMIT членов]:"
echo -n "${Q[1]} "      # Вывести первые два члена последовательности.
echo -n "${Q[2]} "

for ((n=3; n <= $LIMIT; n++)) # C-подобное оформление цикла.
do # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]] для n>2
# Это выражение необходимо разбить на отдельные действия,
# поскольку Bash не очень хорошо поддерживает сложные арифметические действия над
элементами массивов.

    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 ] # Если выведено очередные 20 членов в строке.
then # то
    echo # перейти на новую строку.
fi

done

echo

exit 0

# Этот сценарий реализует итеративный алгоритм поиска членов Q-последовательности.
# Рекурсивную реализацию, как более интуитивно понятную, оставляю вам, в качестве
упражнения.
# Внимание: рекурсивный поиск членов последовательности будет занимать *очень*
продолжительное время.

--

```

Bash поддерживает только одномерные массивы, но, путем небольших ухищрений, можно эмулировать многомерные массивы.

Пример 25-11. Эмуляция массива с двумя измерениями

```

#!/bin/bash
# Эмуляция двумерного массива.

# Второе измерение представлено как последовательность строк.

Rows=5
Columns=5

declare -a alpha      # char alpha [Rows] [Columns];
                    # Необязательное объявление массива.

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
        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

    # Более простой вариант
    # 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 при таком объявлении второе измерение массива завуалировано.
}

print_alpha ()

```

```

{
local row=0
local index

echo

while [ "$row" -lt "$Rows" ] # Вывод содержимого массива построчно
do

    local column=0

    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 эквивалент:
# echo ${alpha[*]} | xargs -n $Columns

echo
}

filter () # Отфильтровывание отрицательных индексов.
{
echo -n " "

if [[ "$1" -ge 0 && "$1" -lt "$Rows" && "$2" -ge 0 && "$2" -lt "$Columns" ]]
then
    let "index = $1 * $Rows + $2"
    echo -n "${alpha[index]}" # alpha[$row][$column]
fi

}

rotate () # Поворот массива на 45 градусов
{
local row
local column

for (( row = Rows; row > -Rows; row-- )) # В обратном порядке.
do

    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 # Отфильтровать отрицательный индекс.
    done

    echo; echo

done

```



```

# Поворот массива выполнен на основе примеров (стр. 143-146)
# из книги "Advanced C Programming on the IBM PC", автор Herbert Mayer
# (см. библиографию).

}

#-----#
load_alpha      # Инициализация массива.
print_alpha     # Вывод на экран.
rotate          # Повернуть на 45 градусов против часовой стрелки.
#-----#

# Упражнения:
# -----
# 1) Сделайте инициализацию и вывод массива на экран
#   + более простым и элегантным способом.
#
# 2) Объясните принцип работы функции rotate().

exit 0

```

По существу, двумерный массив эквивалентен одномерному, с тем лишь различием, что для индексации отдельных элементов используются два индекса -- "строка" и "столбец".

Более сложный пример эмуляции двумерного массива вы найдете в [Пример А-11](#).

Глава 26. Файлы

сценарии начальной загрузки

Эти файлы содержат объявления псевдонимов и [переменных окружения](#), которые становятся доступны Bash после загрузки и инициализации системы.

/etc/profile

Настройки системы по-умолчанию, главным образом настраивается окружение командной оболочки (все Bourne-подобные оболочки, не только Bash [\[55\]](#))

/etc/bashrc

функции и [псевдонимы](#) Bash

\$HOME/.bash_profile

пользовательские настройки окружения Bash, находится в домашнем каталоге у каждого пользователя (локальная копия файла /etc/profile)

\$HOME/.bashrc

пользовательский файл инициализации Bash, находится в домашнем каталоге у каждого пользователя (локальная копия файла /etc/bashrc). См. [Приложение G](#) пример файла .bashrc.

Сценарий выхода из системы (logout)

```
$HOME/.bash_logout
```

Этот сценарий обрабатывает, когда пользователь выходит из системы.

Глава 27. /dev и /proc

Как правило, Linux или UNIX система имеет два каталога специального назначения: /dev и /proc.

27.1. /dev

Каталог /dev содержит файлы физических *устройств*, которые могут входить в состав аппаратного обеспечения компьютера. [56] Каждому из разделов не жестком диске соответствует свой файл-устройство в каталоге /dev, информация о которых может быть получена простой 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
```

Кроме того, каталог /dev содержит *loopback*-устройства ("петлевые" устройства), например /dev/loop0. С помощью такого устройства можно представить обычный файл как блочное устройство ввода/вывода. [57] Это позволяет монтировать целые файловые системы, находящиеся в отдельных больших файлах. См. [Пример 13-6](#) и [Пример 13-5](#).

Отдельные псевдоустройства в /dev имеют особое назначение, к таким устройствам можно отнести [/dev/null](#), [/dev/zero](#) и [/dev/urandom](#).

27.2. /proc

Фактически, каталог /proc -- это виртуальная файловая система. Файлы, в каталоге /proc, содержат информацию о процессах, о состоянии и конфигурации ядра и системы.

```
bash$ cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 tty
 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
```

Сценарии командной оболочки могут извлекать необходимую информацию из соответствующих файлов в каталоге /proc. [\[58\]](#)

```
bash$ cat /proc/filesystems | grep iso9660
iso9660
```

```
kernel_version=$( awk '{ print $3 }' /proc/version )
```

```
CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )
```

```
if [ $CPU = Pentium ]
then
```

```

    выполнить_ряд_специфичных_команд
    ...
else
    выполнить_ряд_других_специфичных_команд
    ...
fi

```

В каталоге /proc вы наверняка заметите большое количество подкаталогов, с не совсем обычными именами, состоящими только из цифр. Каждый из них соответствует исполняющемуся процессу, а имя каталога -- это [ID \(идентификатор\) процесса](#). Внутри каждого такого подкаталога находится ряд файлов, в которых содержится полезная информация о соответствующих процессах. Файлы stat и status хранят статистику работы процесса, cmdline -- команда, которой был запущен процесс, exe -- символическая ссылка на исполняемый файл программы. Здесь же вы найдете ряд других файлов, но, с точки зрения написания сценариев, они не так интересны, как эти четыре.

Пример 27-1. Поиск файла программы по идентификатору процесса

```

#!/bin/bash
# pid-identifier.sh: Возвращает полный путь к исполняемому файлу программы по
идентификатору процесса (pid).

ARGNO=1 # Число, ожидаемых из командной строки, аргументов.
E_WRONGARGS=65
E_BADPID=66
E_NOSUCHPROCESS=67
E_NOPERMISSION=68
PROCFILE=exe

if [ $# -ne $ARGNO ]
then
    echo "Порядок использования: `basename $0` PID-процесса" >&2 # Сообщение об ошибке
на >stderr.
    exit $E_WRONGARGS
fi

ps ax

pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
# Проверка наличия процесса с заданным pid в списке, выданном командой "ps", поле
#1.
# Затем следует убедиться, что этот процесс не был запущен этим сценарием ('ps').
# Это делает последний "grep $1".
if [ -z "$pidno" ] # Если после фильтрации получается пустая строка,
then # то это означает, что в системе нет процесса с заданным pid.
    echo "Нет такого процесса."
    exit $E_NOSUCHPROCESS
fi

# Альтернативный вариант:
# if ! ps $1 > /dev/null 2>&1
# then # в системе нет процесса с заданным pid.
#     echo "Нет такого процесса."
#     exit $E_NOSUCHPROCESS
# fi

if [ ! -r "/proc/$1/$PROCFILE" ] # Проверить право на чтение.
then
    echo "Процесс $1 найден, однако..."
    echo "у вас нет права на чтение файла /proc/$1/$PROCFILE."
    exit $E_NOPERMISSION # Обычный пользователь не имеет прав
# на доступ к некоторым файлам в каталоге /proc.

```

```

fi

# Последние две проверки могут быть заменены на:
#   if ! kill -0 $1 > /dev/null 2>&1 # '0' -- это не сигнал, но
#                                   # команда все равно проверит наличие
#                                   # процесса-получателя.
#   then echo "Процесс с данным PID не найден, либо вы не являетесь его владельцем"
>&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 исполняемый файл работающей программы.

if [ -e "$exe_file" ] # Если файл /proc/pid-number/exe существует...
then # то существует и соответствующий процесс.
  echo "Исполняемый файл процесса #$1: $exe_file."
else
  echo "Нет такого процесса."
fi

# В большинстве случаев, этот, довольно сложный сценарий, может быть заменен командой
# ps ax | grep $1 | awk '{ print $5 }'
# В большинстве, но не всегда...
# поскольку пятое поле листинга, выдаваемого командой 'ps', это argv[0] процесса,
# а не путь к исполняемому файлу.
#
# Однако, оба следующих варианта должны работать безотказно.
#   find /proc/$1/exe -printf '%l\n'
#   lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'

# Автор последнего комментария: Stephane Chazelas.

exit 0

```

Пример 27-2. Проверка состояния соединения

```

#!/bin/bash

PROCNAME=pppd # демон ppp
PROCFILENAME=status # Что смотреть.
NOTCONNECTED=65
INTERVAL=2 # Период проверки -- раз в 2 секунды.

pidno=$( ps ax | grep -v "ps ax" | grep -v grep | grep $PROCNAME | awk '{ print $1 }' )
)
# Найти идентификатор процесса 'pppd', 'ppp daemon'.
# По пути убрать из листинга записи о процессах, порожденных сценарием.
#
# Однако, как отмечает Oleg Philon,
#+ Эта последовательность команд может быть заменена командой "pidof".
# pidno=$( pidof $PROCNAME )
#
# Мораль:
#+ Когда последовательность команд становится слишком сложной,
#+ это повод к тому, чтобы поискать более короткий вариант.

if [ -z "$pidno" ] # Если получилась пустая строка, значит процесс не запущен.
then
  echo "Соединение не установлено."
  exit $NOTCONNECTED
else

```

```

    echo "Соединение установлено."; echo
fi

while [ true ]          # Бесконечный цикл.
do

    if [ ! -e "/proc/$pidno/$PROCFILENAME" ]
    # Пока работает процесс, файл "status" существует.
    then
        echo "Соединение разорвано."
        exit $NOTCONNECTED
    fi

    netstat -s | grep "packets received" # Получить некоторые сведения о соединении.
    netstat -s | grep "packets delivered"

    sleep $INTERVAL
    echo; echo

done

exit 0

# Как обычно, этот сценарий может быть остановлен комбинацией клавиш Control-C.

#   Упражнение:
#   -----
#   Добавьте возможность завершения работы сценария, по нажатию на клавишу "q".
#   Это сделает скрипт более дружелюбным к пользователю.

```



Будьте предельно осторожны при работе с файловой системой /proc, так как попытка записи в некоторые файлы может повредить файловую систему или привести к краху системы.

Глава 28. /dev/zero и /dev/null

/dev/null

Псевдоустройство /dev/null -- это, своего рода, "черная дыра" в системе. Это, пожалуй, самый близкий смысловой эквивалент. Все, что записывается в этот файл, "исчезает" навсегда. Попытки записи или чтения из этого файла не дают, ровным счетом, никакого результата. Тем не менее, псевдоустройство /dev/null вполне может пригодиться.

Подавление вывода на stdout.

```

cat $filename >/dev/null
# Содержимое файла $filename не появится на stdout.

```

Подавление вывода на stderr (from [Пример 12-2](#)).

```

rm $badname 2>/dev/null
# Сообщение об ошибке "уйдет в никуда".

```

Подавление вывода, как на `stdout`, так и на `stderr`.

```
cat $filename 2>/dev/null >/dev/null
# Если "$filename" не будет найден, то вы не увидите сообщения об ошибке.
# Если "$filename" существует, то вы не увидите его содержимое.
# Таким образом, вышеприведенная команда ничего не выводит на экран.
#
# Такая методика бывает полезной, когда необходимо лишь проверить код
завершения команды
#+ и нежелательно выводить результат работы команды на экран.
#
# cat $filename &>/dev/null
#     дает тот же результат, автор примечания Baris Cicek.
```

Удаление содержимого файла, сохраняя, при этом, сам файл, со всеми его правами доступа (очистка файла) (из [Пример 2-1](#) и [Пример 2-2](#)):

```
cat /dev/null > /var/log/messages
# : > /var/log/messages     дает тот же эффект, но не порождает дочерний процесс.

cat /dev/null > /var/log/wtmp
```

Автоматическая очистка содержимого системного журнала (logfile) (особенно хороша для борьбы с надоедливыми рекламными идентификационными файлами ("cookies")):

Пример 28-1. Удаление cookie-файлов

```
if [ -f ~/.netscape/cookies ] # Удалить, если имеются.
then
    rm -f ~/.netscape/cookies
fi

ln -s /dev/null ~/.netscape/cookies
# Теперь, все cookie-файлы, вместо того, чтобы сохраняться на диске, будут
"вылетать в трубу".
```

`/dev/zero`

Подобно псевдоустройству `/dev/null`, `/dev/zero` так же является псевдоустройством, с той лишь разницей, что содержит нули. Информация, выводимая в этот файл, так же бесследно исчезает. Чтение нулей из этого файла может вызвать некоторые затруднения, однако это можно сделать, к примеру, с помощью команды [od](#) или шестнадцатиричного редактора. В основном, `/dev/zero` используется для создания заготовки файла с заданой длиной.

Пример 28-2. Создание файла подкачки (swapfile), с помощью `/dev/zero`

```
#!/bin/bash

# Создание файла подкачки.
# Этот сценарий должен запускаться с правами root.

ROOT_UID=0          # Для root -- $UID 0.
E_WRONG_USER=65     # Не root?

FILE=/swap
BLOCKSIZE=1024
```

```

MINBLOCKS=40
SUCCESS=0

if [ "$UID" -ne "$ROOT_UID" ]
then
    echo; echo "Этот сценарий должен запускаться с правами root."; echo
    exit $E_WRONG_USER
fi

blocks=${1:-$MINBLOCKS}          # По-умолчанию -- 40 блоков,
                                #+ если размер не задан из командной строки.
# Ниже приводится эквивалентный набор команд.
# -----
# if [ -n "$1" ]
# then
#     blocks=$1
# else
#     blocks=$MINBLOCKS
# fi
# -----

if [ "$blocks" -lt $MINBLOCKS ]
then
    blocks=$MINBLOCKS           # Должно быть как минимум 40 блоков.
fi

echo "Создание файла подкачки размером $blocks блоков (KB)."
```

dd if=/dev/zero of=\$FILE bs=\$BLOCKSIZE count=\$blocks # "Забить" нулями.

```

mkswap $FILE $blocks           # Назначить как файл подкачки.
swapon $FILE                   # Активировать.

echo "Файл подкачки создан и активирован."

exit $SUCCESS
```

Еще одна область применения /dev/zero -- "очистка" специального файла заданного размера, например файлов, монтируемых как [loopback-устройства](#) (см. [Пример 13-6](#)) или для безопасного удаления файла (см. [Пример 12-42](#)).

Пример 28-3. Создание электронного диска

```

#!/bin/bash
# ramdisk.sh

# "электронный диск" -- это область в ОЗУ компьютера
#+ с которой система взаимодействует как с файловой системой.
# Основное преимущество -- очень высокая скорость чтения/записи.
# Недостатки -- энергозависимость, уменьшение объема ОЗУ, доступного системе,
# относительно небольшой размер.
#
# Чем хорош электронный диск?
# При хранении наборов данных, таких как таблиц баз данных или словарей, на
электронном диске
#+ вы получаете высокую скорость работы с этими наборами, поскольку время
доступа к ОЗУ
# неизмеримо меньше времени доступа к жесткому диску.

E_NON_ROOT_USER=70             # Сценарий должен запускаться с правами root.
ROOTUSER_NAME=root

MOUNTPT=/mnt/ramdisk
SIZE=2000                      # 2K блоков (измените, если это необходимо)
BLOCKSIZE=1024                # размер блока -- 1K (1024 байт)
```



```

DEVICE=/dev/ram0                # Первое устройство ram

username=`id -nu`
if [ "$username" != "$ROOTUSER_NAME" ]
then
    echo "Сценарий должен запускаться с правами root."
    exit $E_NON_ROOT_USER
fi

if [ ! -d "$MOUNTPT" ]          # Проверка наличия точки монтирования,
then                             #+ благодаря этой проверке, при повторных
запусках сценария
    mkdir $MOUNTPT              #+ ошибки возникать не будет.
fi

dd if=/dev/zero of=$DEVICE count=$SIZE bs=$BLOCKSIZE # Очистить электронный
диск.
mke2fs $DEVICE                  # Создать файловую систему ext2.
mount $DEVICE $MOUNTPT          # Смонтировать.
chmod 777 $MOUNTPT              # Сделать электронный диск доступным для обычных
пользователей.
                                # Но при этом, только root сможет его
отмонтировать.

echo "Электронный диск \"$MOUNTPT\" готов к работе."
# Теперь электронный диск доступен для любого пользователя в системе.

# Внимание! Электронный диск -- это энергозависимое устройство! Все данные,
хранящиеся на нем,
#+ будут утеряны при остановке или перезагрузке системы.
# Если эти данные представляют для вас интерес, то сохраняйте их копии в
обычном каталоге.

# После перезагрузки, чтобы вновь создать электронный диск, запустите этот
сценарий.
# Простое монтирование /mnt/ramdisk, без выполнения подготовительных действий,
не будет работать.

exit 0

```

Глава 29. Отладка сценариев

Командная оболочка Bash не имеет своего отладчика, и не имеет даже каких либо отладочных команд или конструкций. [\[59\]](#) Синтаксические ошибки или опечатки часто вызывают сообщения об ошибках, которые которые практически никак не помогают при отладке.

Пример 29-1. Сценарий, содержащий ошибку

```

#!/bin/bash
# ex74.sh

# Этот сценарий содержит ошибку.

a=37

if [ $a -gt 27 ]
then
    echo $a
fi

exit 0

```

В результате исполнения этого сценария вы получите такое сообщение:

```
./ex74.sh: [37: command not found
```

Что в этом сценарии может быть неправильно (подсказка: после ключевого слова **if**)?

Пример 29-2. Пропущено ключевое слово

```
#!/bin/bash
# missing-keyword.sh:
# Какое сообщение об ошибке будет выведено, при попытке запустить этот сценарий?

for a in 1 2 3
do
    echo "$a"
# done      # Необходимое ключевое слово 'done' закомментировано.

exit 0
```

На экране появится сообщение:

```
missing-keyword.sh: line 11: syntax error: unexpected end of file
```

Обратите внимание, сообщение об ошибке будет содержать номер не той строки, в которой возникла ошибка, а той, в которой Bash точно установил наличие ошибочной ситуации.

Сообщения об ошибках могут вообще не содержать номера строки, при исполнении которой эта ошибка появилась.

А что делать, если сценарий работает, но не так как ожидалось? Вот пример весьма распространенной логической ошибки.

Пример 29-3. test24

```
#!/bin/bash

# Ожидается, что этот сценарий будет удалять в текущем каталоге
#+ все файлы, имена которых содержат пробелы.
# Но он не работает. Почему?

badname=`ls | grep ' '`

# echo "$badname"

rm "$badname"

exit 0
```

Попробуйте найти ошибку, раскомментарив строку `echo "$badname"`. Инструкция `echo` очень полезна при отладке сценариев, она позволяет узнать -- действительно ли вы получаете то, что ожидали получить.

В данном конкретном случае, команда `rm "$badname"` не дает желаемого результата потому, что переменная `$badname` взята в кавычки. В результате, `rm` получает единственный аргумент (т.е. команда будет считать, что получила имя одного файла). Частично эта проблема может быть решена за счет удаления кавычек вокруг `$badname` и установки переменной `$IFS` так, чтобы она содержала только символ перевода строки, `IFS=$'\n'`. Однако, существует более простой способ выполнить эту задачу.

```
# Правильный способ удаления файлов, в чьих именах содержатся пробелы.  
rm *\ *  
rm *" "*  
rm *' '*  
# Спасибо S.C.
```

В общих чертах, ошибочными можно считать такие сценарии, которые

1. "сыплют" сообщениями о "синтаксических ошибках" или
2. запускаются, но работают не так как ожидалось (логические ошибки).
3. запускаются, делают то, что требуется, но имеют побочные эффекты (логическая бомба).

Инструменты, которые могут помочь при отладке неработающих сценариев

1. команда `echo`, в критических точках сценария, поможет отследить состояние переменных и отобразить ход исполнения.
2. команда-фильтр `tee`, которая поможет проверить процессы и потоки данных в критических местах.
3. ключи `-n` `-v` `-x`

`sh -n scriptname` -- проверит наличие синтаксических ошибок, не запуская сам сценарий. Того же эффекта можно добиться, вставив в сценарий команду `set -n` или `set -o noexec`. Обратите внимание, некоторые из синтаксических ошибок не могут быть выявлены таким способом.

`sh -v scriptname` -- выводит каждую команду прежде, чем она будет выполнена. Того же эффекта можно добиться, вставив в сценарий команду `set -v` или `set -o verbose`.

Ключи `-n` и `-v` могут употребляться совместно: `sh -nv scriptname`.

`sh -x scriptname` -- выводит, в краткой форме, результат исполнения каждой команды. Того же эффекта можно добиться, вставив в сценарий команду `set -x` или `set -o xtrace`.

Вставив в сценарий `set -u` или `set -o nounset`, вы будете получать сообщение об ошибке `unbound variable` всякий раз, когда будет производиться попытка обращения к необъявленной переменной.

4. Функция `"assert"`, предназначенная для проверки переменных или условий, в критических точках сценария. (Эта идея заимствована из языка программирования C.)

Пример 29-4. Проверка условия с помощью функции `"assert"`

```
#!/bin/bash  
# assert.sh  
  
assert ()  
# Если условие ложно,
```

```

{                                     #+ выход из сценария с сообщением об ошибке.
E_PARAM_ERR=98
E_ASSERT_FAILED=99

if [ -z "$2" ]                       # Недостаточное количество входных параметров.
then
    return $E_PARAM_ERR
fi

lineno=$2

if [ ! $1 ]
then
    echo "Утверждение ложно: \"$1\""
    echo "Файл: \"$0\"", строка: $lineno"
    exit $E_ASSERT_FAILED
# else
#     return
#     и продолжить исполнение сценария.
fi
}

a=5
b=4
condition="$a -lt $b"                # Сообщение об ошибке и завершение сценария.
                                     # Попробуйте поменять условие "condition"
                                     #+ на чтонибудь другое и
                                     #+ посмотреть -- что получится.

assert "$condition" $LINENO
# Сценарий продолжит работу только в том случае, если утверждение истинно.

# Прочие команды.
# ...
echo "Эта строка появится на экране только если утверждение истинно."
# ...
# Прочие команды.
# ...

exit 0

```


5. Ловушка на выхто в этом сценарии может быть неправильно (подсказка: после ключевого словоде.

Команда **exit**, в сценарии, порождает сигнал 0, по которому процесс завершает работу, т.е. -- сам сценарий. [\[60\]](#) Часто бывает полезным по выходу из сценария выдать "распечатку" переменных.

Установка ловушек на сигналы

trap

Определяет действие при получении сигнала; так же полезна при отладке.

-  Сигнал (*signal*) -- это просто сообщение, передается процессу либо ядром, либо другим процессом, чтобы побудить процесс выполнить какие либо действия (обычно -- завершить работу). Например, нажатие на **Control-C**, вызывает передачу сигнала SIGINT, исполняющейся программе.

```
trap '' 2
# Игнорировать прерывание 2 (Control-C), действие по сигналу не указано.

trap 'echo "Control-C disabled."' 2
# Сообщение при нажатии на Control-C.
```

Пример 29-5. Ловушка на выходе

```
#!/bin/bash

trap 'echo Список переменных --- a = $a b = $b' EXIT
# EXIT -- это название сигнала, генерируемого при выходе из сценария.

a=39

b=36

exit 0
# Примечательно, что если закомментировать команду 'exit',
# то это никак не скажется на работе сценария,
# поскольку "выход" из сценария происходит в любом случае.
```

Пример 29-6. Удаление временного файла при нажатии на Control-C

```
#!/bin/bash
# logon.sh: Сценарий, написанный "на скорую руку", контролирует вход в режим on-line.

TRUE=1
LOGFILE=/var/log/messages
# Обратите внимание: $LOGFILE должен быть доступен на чтение (chmod 644
/var/log/messages).
TEMPFILE=temp.$$
# "Уникальное" имя для временного файла, где расширение в имени -- это pid процесса-
сценария.
KEYWORD=address
# При входе, в файл /var/log/messages,
# добавляется строка "remote IP address xxx.xxx.xxx.xxx"
ONLINE=22
USER_INTERRUPT=13
CHECK_LINES=100
# Количество проверяемых строк.

trap 'rm -f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
# Удалить временный файл, когда сценарий завершает работу по control-c.

echo

while [ $TRUE ] #Бесконечный цикл.
do
  tail -$CHECK_LINES $LOGFILE> $TEMPFILE
  # Последние 100 строк из системного журнала переписать во временный файл.
  # Совершенно необходимо, т.к. новейшие версии ядер генерируют много сообщений при
входе.
  search=`grep $KEYWORD $TEMPFILE`
  # Проверить наличие фразы "address",
  # свидетельствующей об успешном входе.

  if [ ! -z "$search" ] # Кавычки необходимы, т.к. переменная может содержать
пробелы.
  then
    echo "On-line"
    rm -f $TEMPFILE # Удалить временный файл.
    exit $ONLINE
  else
```

```

    echo -n "."          # ключ -n подавляет вывод символа перевода строки,
                        # так вы получите непрерывную строку точек.
fi

sleep 1
done

# Обратите внимание: если изменить содержимое переменной KEYWORD
# на "Exit", то сценарий может использоваться для контроля
# неожиданного выхода (logout).

exit 0

# Nick Drage предложил альтернативный метод:

while true
do ifconfig ppp0 | grep UP 1> /dev/null && echo "соединение установлено" && exit 0
  echo -n "."          # Печать последовательности точек (.....), пока соединение не будет
  установлено.
  sleep 2
done

# Проблема: Нажатия Control-C может оказаться недостаточным, чтобы завершить этот
# процесс.
#           (Точки продолжают выводиться на экран.)
# Упражнение: Исправьте этот недостаток.

# Stephane Chazelas предложил еще одну альтернативу:

CHECK_INTERVAL=1

while ! tail -1 "$LOGFILE" | grep -q "$KEYWORD"
do echo -n .
  sleep $CHECK_INTERVAL
done
echo "On-line"

# Упражнение: Найдите сильные и слабые стороны
#           каждого из этих подходов.

```



Аргумент **DEBUG**, команды **trap**, заставляет сценарий выполнять указанное действие после выполнения каждой команды. Это можно использовать для трассировки переменных.

Пример 29-7. Трассировка переменной

```

#!/bin/bash

trap 'echo "VARIABLE-TRACE> $LINENO: \$variable = \"\$variable\""' DEBUG
# Выводить значение переменной после исполнения каждой команды.

variable=29

echo "Переменная \"\$variable\" инициализирована числом $variable."

let "variable *= 3"
echo "Значение переменной \"\$variable\" увеличено в 3 раза."

# Конструкция "trap 'commands' DEBUG" может оказаться очень полезной
# при отладке больших и сложных скриптов,
# когда размещение множества инструкций "echo $variable"
# может потребовать достаточно большого времени.

# Спасибо Stephane Chazelas.

```

```
exit 0
```



Конструкция `trap '' SIGNAL` (две одиночных кавычки) -- запрещает `SIGNAL` для оставшейся части сценария. Конструкция `trap SIGNAL` -- восстанавливает действие сигнала `SIGNAL`. Эти конструкции могут использоваться для защиты критических участков сценария от нежелательного прерывания.

```
trap '' 2 # Сигнал 2 (Control-C) -- запрещен.  
command  
command  
command  
trap 2 # Разрешение реакции на Control-C
```

Глава 30. Необязательные параметры (ключи)

Необязательные параметры -- это дополнительные ключи (опции), которые оказывают влияние на поведение сценария и/или командной оболочки.

Команда [set](#) позволяет задавать дополнительные опции прямо внутри сценария. В том месте сценария, где необходимо, чтобы та или иная опция вступила в силу, вставьте такую конструкцию **set -o option-name**, или в более короткой форме -- **set -option-abbrev**. Эти две формы записи совершенно идентичны по своему действию.

```
#!/bin/bash  
  
set -o verbose  
# Вывод команд перед их исполнением.
```

```
#!/bin/bash  
  
set -v  
# Имеет тот же эффект, что и выше.
```



Для того, чтобы отключить действие той или иной опции, следует вставить конструкцию **set +o option-name**, или **set +option-abbrev**.

```
#!/bin/bash  
  
set -o verbose  
# Вывод команд перед их исполнением.  
command  
...  
command
```

```

set +o verbose
# Запретить вывод команд перед их исполнением.
command
# команда не выводится.

set -v
# Вывод команд перед их исполнением.
command
...
command

set +v
# Запретить вывод команд перед их исполнением.
command

exit 0

```

Как вариант установки опций, можно предложить указывать их в заголовке сценария (в строке `sha-bang`) `-- #!`.

```

#!/bin/bash -x
#
# Далее следует текст сценария.

```

Так же можно указывать дополнительные ключи в командной строке, при запуске сценария. Некоторые из опций работают только если они заданы из командной строки, например `-i` -- ключ интерактивного режима работы скрипта.

`bash -v script-name`

`bash -o verbose script-name`

Ниже приводится список некоторых полезных опций, которые могут быть указаны как в полной форме так и в сокращенной.

Таблица 30-1. Ключи Bash

Краткое имя	Полное имя	Описание
-C	noclobber	Предотвращает перезапись файла в операциях перенаправления вывода (не распространяется на конвейеры (каналы) <code>-- > </code>)
-D	(нет)	Выводит список строк в двойных кавычках, которым предшествует символ \$, сам сценарий не исполняется
-a	allexport	Экспорт всех, определенных в сценарии, переменных
-b	notify	Выводит уведомление по завершении фоновой задачи (job) (довольно редко используется в сценариях)
-c ...	(нет)	Читает команды из ...
-f	noglob	Подстановка имен файлов (globbing) запрещена
-i	interactive	Сценарий запускается в <i>интерактивном</i> режиме
-p	privileged	Сценарий запускается как "suid" (осторожно!)

Краткое имя	Полное имя	Описание
-r	restricted	Сценарий запускается в <i>ограниченном</i> режиме (см. Глава 20).
-u	nounset	При попытке обращения к неопределенным переменным, выдает сообщение об ошибке и прерывает работу сценария
-v	verbose	Выводит на stdout каждую команду прежде, чем она будет исполнена
-x	xtrace	Подобна -v, но выполняет подстановку команд
-e	errexit	Прерывает работу сценария при появлении первой же ошибки (когда команда возвращает ненулевой код завершения)
-n	noexec	Читает команды из сценария, но не исполняет их (проверка синтаксиса)
-s	stdin	Читает команды с устройства stdin
-t	(нет)	Выход после исполнения первой команды
-	(нет)	Конец списка ключей (опций), последующие аргументы будут восприниматься как позиционные параметры .
--	(нет)	Эквивалент предыдущей опции (-).

Глава 31. Широко распространенные ошибки

Turandot: Gli enigmi sono tre, la morte una!

Caleph: No, no! Gli enigmi sono tre, una la vita!

Puccini

Использование зарезервированных слов и служебных символов в качестве имен переменных.

```
case=value0      # Может вызвать проблемы.
23skidoo=value1 # Тоже самое.
# Имена переменных, начинающиеся с цифр, зарезервированы командной оболочкой.
# Если имя переменной начинается с символа подчеркивания: _23skidoo=value1, то это не
# считается ошибкой.

# Однако... если имя переменной состоит из единственного символа подчеркивания, то
# это ошибка.
_=25
echo $_          # $_ -- это внутренняя переменная.

xyz(!*=value2   # Вызывает серьезные проблемы.
```

Использование дефиса, и других зарезервированных символов, в именах переменных.

```
var-1=23
# Вместо такой записи используйте 'var_1'.
```

Использование одинаковых имен для переменных и функций. Это делает сценарий трудным для понимания.

```
do_something ()
{
    echo "Эта функция должна что-нибудь сделать с \"\$1\"."
}

do_something=do_something

do_something do_something

# Все это будет работать правильно, но слишком уж запутанно.
```

Использование лишних [пробелов](#). В отличие от других языков программирования, Bash весьма привередлив по отношению к пробелам.

```
var1 = 23 # Правильный вариант: 'var1=23'.
# В вышеприведенной строке Bash будет трактовать "var1" как имя команды
# с аргументами "=" и "23".

let c = $a - $b # Правильный вариант: 'let c=$a-$b' или 'let "c = $a - $b"'

if [ $a -le 5] # Правильный вариант: if [ $a -le 5 ]
# if [ "$a" -le 5 ] еще лучше.
# [[ $a -le 5 ]] тоже верно.
```

Ошибочным является предположение о том, что неинициализированные переменные содержат "ноль". Неинициализированные переменные содержат "пустое" (null) значение, а *не* ноль.

```
#!/bin/bash

echo "uninitialized_var = $uninitialized_var"
# uninitialized_var =
```

Часто программисты путают операторы сравнения = и `-eq`. Запомните, оператор = используется для сравнения строковых переменных, а `-eq` -- для сравнения целых чисел.

```
if [ "$a" = 273 ] # Как вы полагаете? $a -- это целое число или строка?
if [ "$a" -eq 273 ] # Если $a -- целое число.

# Иногда, такого рода ошибка никак себя не проявляет.
# Однако...
```

```
a=273.0 # Не целое число.

if [ "$a" = 273 ]
then
    echo "Равны."
else
    echo "Не равны."
fi # Не равны.

# тоже самое и для a=" 273" и a="0273".
```

Подобные проблемы возникают при использовании "-eq" со строковыми значениями.

```
if [ "$a" -eq 273.0 ]
then
```

```
    echo "a = $a"
fi # Исполнение сценария прерывается по ошибке.
# test.sh: [: 273.0: integer expression expected
```

Ошибки при сравнении [целых чисел](#) и [строковых значений](#).

```
#!/bin/bash
# bad-op.sh

number=1

while [ "$number" < 5 ]      # Неверно! должно быть   while [ "number" -lt 5 ]
do
    echo -n "$number "
    let "number += 1"
done

# Этот сценарий генерирует сообщение об ошибке:
# bad-op.sh: 5: No such file or directory
```

Иногда, в операциях проверки, с использованием квадратных скобок ([]), переменные необходимо брать в двойные кавычки. См. [Пример 7-6](#), [Пример 16-4](#) и [Пример 9-6](#).

Иногда сценарий не в состоянии выполнить команду из-за нехватки прав доступа. Если пользователь не сможет запустить команду из командной строки, то эта команда не сможет быть запущена и из сценария. Попробуйте изменить атрибуты команды, возможно вам придется установить бит `suid`.

Использование символа `-` в качестве оператора перенаправления (каковым он не является) может приводить к неожиданным результатам.

```
command1 2> - | command2 # Попытка передать сообщения об ошибках команде command1
через конвейер...
# ...не будет работать.

command1 2>& - | command2 # Так же бессмысленно.

Спасибо S.C.
```

Использование функциональных особенностей Bash [версии 2](#) или выше, может привести к аварийному завершению сценария, работающему под управлением Bash версии 1.XX.

```
#!/bin/bash

minimum_version=2
# Поскольку Chet Ramey постоянно развивает Bash,
# вам может потребоваться указать другую минимально допустимую версию
$minimum_version=2.XX.
E_BAD_VERSION=80

if [ "$BASH_VERSION" \< " $minimum_version" ]
then
    echo "Этот сценарий должен исполняться под управлением Bash, версии $minimum или
выше."
    echo "Настоятельно рекомендуется обновиться."
    exit $E_BAD_VERSION
fi
```

...

Использование специфических особенностей Bash может приводить к аварийному завершению сценария в Bourne shell (`#!/bin/sh`). Как правило, в Linux дистрибутивах, **sh** является псевдонимом **bash**, но это не всегда верно для UNIX-систем вообще.

Сценарий, в котором строки отделяются друг от друга в стиле MS-DOS (`\r\n`), будет завершаться аварийно, поскольку комбинация `#!/bin/bash\r\n` считается недопустимой. Исправить эту ошибку можно простым удалением символа `\r` из сценария.

```
#!/bin/bash

echo "Начало"

unix2dos $0      # Сценарий переводит символы перевода строки в формат DOS.
chmod 755 $0    # Восстановление прав на запуск.
                # Команда 'unix2dos' удалит право на запуск из атрибутов файла.

./$0            # Попытка запустить себя самого.
                # Но это не сработает из-за того, что теперь строки отделяются
                # друг от друга в стиле DOS.

echo "Конец"

exit 0
```

Сценарий, начинающийся с `#!/bin/sh`, не может работать в режиме полной совместимости с Bash. Некоторые из специфических функций, присущих Bash, могут оказаться запрещенными к использованию. Сценарий, который требует полного доступа ко всем расширениям, имеющимся в Bash, должен начинаться строкой `#!/bin/bash`.

Сценарий не может **экспортировать** переменные [родительскому процессу](#) - оболочке. Здесь как в природе, потомок может унаследовать черты родителя, но не наоборот.

```
WHATEVER=/home/bozo
export WHATEVER
exit 0
```

```
bash$ echo $WHATEVER
```

```
bash$
```

Будьте уверены -- при выходе в командную строку переменная `$WHATEVER` останется неинициализированной.

Использование в подоболочке переменных с теми же именами, что и в родительской оболочке может не давать ожидаемого результата.

Пример 31-1. Западня в подоболочке

```
#!/bin/bash
# Западня в подоболочке.

outer_variable=внешняя_переменная
echo
echo "outer_variable = $outer_variable"
echo
```

```
(
# Запуск в подболочке

echo "внутри подболочки outer_variable = $outer_variable"
inner_variable=внутренняя_переменная # Инициализировать
echo "внутри подболочки inner_variable = $inner_variable"
outer_variable=внутренняя_переменная # Как думаете? Изменит внешнюю переменную?
echo "внутри подболочки outer_variable = $outer_variable"

# Выход из подболочки
)

echo
echo "за пределами подболочки inner_variable = $inner_variable" # Ничего не
выводится.
echo "за пределами подболочки outer_variable = $outer_variable" #
внешняя_переменная.
echo

exit 0
```

Передача вывода от **echo** по [конвейеру](#) команде [read](#) может давать неожиданные результаты. В этом сценарии, команда **read** действует так, как будто бы она была запущена в подболочке. Вместо нее лучше использовать команду [set](#) (см. [Пример 11-14](#)).

Пример 31-2. Передача вывода от команды echo команде read, по конвейеру

```
#!/bin/bash
# badread.sh:
# Попытка использования 'echo' и 'read'
#+ для записи значений в переменные.

a=aaa
b=bbb
c=ccc

echo "один два три" | read a b c
# Попытка записать значения в переменные a, b и c.

echo
echo "a = $a" # a = aaa
echo "b = $b" # b = bbb
echo "c = $c" # c = ccc
# Присваивания не произошло.

# -----

# Альтернативный вариант.

var=`echo "один два три"`
set -- $var
a=$1; b=$2; c=$3

echo "-----"
echo "a = $a" # a = один
echo "b = $b" # b = два
echo "c = $c" # c = три
# На этот раз все в порядке.

# -----

# Обратите внимание: в подболочке 'read', для первого варианта, переменные
присваиваются нормально.
# Но только в подболочке.
```

```

a=aaa          # Все сначала.
b=bbb
c=ccc

echo; echo
echo "один два три" | ( read a b c;
echo "Внутри подболочки: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
# a = один
# b = два
# c = три
echo "-----"
echo "Снаружи: "
echo "a = $a" # a = aaa
echo "b = $b" # b = bbb
echo "c = $c" # c = ccc
echo

exit 0

```

Огромный риск, для безопасности системы, представляет использование в скриптах команд, с установленным битом "suid". [\[61\]](#)

Использование сценариев в качестве CGI-приложений может приводить к серьезным проблемам из-за отсутствия контроля типов переменных. Более того, они легко могут быть заменены взломщиком на его собственные сценарии.

Bash не совсем корректно обрабатывает строки, содержащие [двойной слэш \(//\)](#).

Сценарии на языке Bash, созданные для Linux или BSD систем, могут потребовать доработки, перед тем как они смогут быть запущены в коммерческой версии UNIX. Такие сценарии, как правило, используют GNU-версии команд и утилит, которые имеют лучшую функциональность, нежели их аналоги в UNIX. Это особенно справедливо для таких утилит обработки текста, как [tr](#).

Danger is near thee --

Beware, beware, beware, beware.

Many brave hearts are asleep in the deep.

So beware --

Beware.

A.J. Lamb and H.W. Petrie

Глава 32. Стиль программирования

Возьмите в привычку структурный и систематический подход к программированию на языке командной оболочки. Даже для сценариев "выходного дня" и "писанных на коленке", не поленитесь, найдите время для того, чтобы разложить свои мысли по полочкам и продумать структуру будущего скрипта прежде чем приниматься за кодирование.

Ниже приводится несколько рекомендаций по оформлению сценариев, однако их не следует рассматривать как *Официальное Руководство*.

32.1. Неофициальные рекомендации по оформлению сценариев

- Комментируйте свой код. Это сделает ваши сценарии понятнее для других, и более простыми, в обслуживании, для вас.

```
PASS="$PASS${MATRIX:$((($RANDOM%${#MATRIX})):1}"
# Эта строка имела некоторый смысл в момент написания,
# но через год-другой будет очень тяжело вспомнить -- что она делает.
# (Из сценария "pw.sh", автор: Antek Sawicki)
```

Добавляйте заголовочные комментарии в начале сценария и перед функциями.

```
#!/bin/bash

#*****#
#           xyz.sh           #
#           автор: Bozo Bozeman   #
#           Июль 05, 2001         #
#                               #
#           Удаление файлов проекта. #
#*****#

BADDIR=65                # Нет такого каталога.
projectdir=/home/bozo/projects # Каталог проекта.

# ----- #
# cleanup_pfiles () #
# Удаляет все файлы в заданном каталоге. #
# Параметры: $target_directory #
# Возвращаемое значение: 0 -- в случае успеха, #
# $BADDIR -- в случае ошибки. #
# ----- #
cleanup_pfiles ()
{
    if [ ! -d "$1" ] # Проверка существования заданного каталога.
    then
        echo "$1 -- не является каталогом."
        return $BADDIR
    fi

    rm -f "$1"/*
    return 0 # Успешное завершение функции.
}

cleanup_pfiles $projectdir

exit 0
```

Не забывайте начинать ваш сценарий с sha-bang -- `#!/bin/bash`.

- Заменяйте повторяющиеся значения константами. Это сделает ваш сценарий более простым для понимания и позволит вносить изменения, не опасаясь за его работоспособность.

```
if [ -f /var/log/messages ]
then
    ...
fi
# Представьте себе, что через пару лет
# вы захотите изменить /var/log/messages на /var/log/syslog.
# Тогда вам придется отыскать все строки,
```

```
# содержащие /var/log/messages, и заменить их на /var/log/syslog.
# И проверить несколько раз -- не пропустили ли что-нибудь.

# Использование "констант" дает лучший способ:
LOGFILE=/var/log/messages # Если и придется изменить, то только в этой строке.
if [ -f "$LOGFILE" ]
then
    ...
fi
```

- В качестве имен переменных и функций выбирайте осмысленные названия.

```
fl=`ls -al $dirname` # Не очень удачное имя переменной.
file_listing=`ls -al $dirname` # Уже лучше.

MAXVAL=10 # Пишите имена констант в верхнем регистре.
while [ "$index" -le "$MAXVAL" ]
...

E_NOTFOUND=75 # Имена кодов ошибок -- в верхнем
регистре, # к тому же, их желательно дополнять
префиксом "E_".
if [ ! -e "$filename" ]
then
    echo "Файл $filename не найден."
    exit $E_NOTFOUND
fi

MAIL_DIRECTORY=/var/spool/mail/bozo # Имена переменных окружения
# так же желательно записывать символами
# в верхнем регистре.

export MAIL_DIRECTORY

GetAnswer () # Смешивание символов верхнего и нижнего
регистров # удобно использовать для имен функций.

{
    prompt=$1
    echo -n $prompt
    read answer
    return $answer
}

GetAnswer "Ваше любимое число? "
favorite_number=$?
echo $favorite_number

_underscore=23 # Допустимо, но не рекомендуется.
# Желательно, чтобы пользовательские переменные не начинались с символа
подчеркивания.
# Так обычно начинаются системные переменные.
```

- Используйте смысловые имена для [кодов завершения](#).

```
E_WRONG_ARGS=65
...
...
```



```
exit $E_WRONG_ARGS
```

См. так же [Приложение С](#).

- Разделяйте большие сложные сценарии на серию более коротких и простых модулей. Пользуйтесь функциями. См. [Пример 34-4](#).
- Не пользуйтесь сложными конструкциями, если их можно заменить простыми.

```
COMMAND
if [ $? -eq 0 ]
...
# Избыточно и неинтуитивно.

if COMMAND
...
# Более понятно и коротко.
```

... читая исходные тексты сценариев на Bourne shell (/bin/sh). Я был потрясен тем, насколько непонятно и загадочно могут выглядеть очень простые алгоритмы из-за неправильного оформления кода. Я не раз спрашивал себя: "Неужели кто-то может гордиться таким кодом?"

Landon Noll

Глава 33. Разное

Практически никто не знает грамматики Bourne shell-а. Даже изучение исходных текстов не дает ее полного понимания.

Tom Duff

33.1. Интерактивный и неинтерактивный режим работы

В *интерактивном* режиме, оболочка читает команды, вводимые пользователем, с устройства `tty`. Кроме того, такая оболочка считывает конфигурационные файлы на запуске, выводит строку приглашения к вводу (`prompt`), и, по-умолчанию, разрешает управление заданиями. Пользователь имеет возможность *взаимодействия* с оболочкой.

Сценарий всегда запускается в неинтерактивном режиме. Но, не смотря на это, он сохраняет доступ к своему `tty`. И даже может эмулировать интерактивный режим работы.

```
#!/bin/bash
MY_PROMPT='$ '
while :
do
    echo -n "$MY_PROMPT"
```

```

read line
eval "$line"
done

exit 0

# Этот сценарий, как иллюстрация к вышесказанному, предоставлен
# Stephane Chazelas (спасибо).

```

Будем считать *интерактивным* такой сценарий, который может принимать ввод от пользователя, обычно с помощью команды `read` (см. [Пример 11-2](#)). В "реальной жизни" все намного сложнее. Пока же, будем придерживаться предположения о том, что интерактивный сценарий ограничен рамками `tty`, с которого сценарий был запущен пользователем, т.е консоль или окно `xterm`.

Сценарии начальной инициализации системы не являются интерактивными, поскольку они не предполагают вмешательства человека в процессе своей работы. Большая часть сценариев, выполняющих администрирование и обслуживание системы -- так же работают в неинтерактивном режиме. Многие задачи автоматизации труда администратора очень трудно представить себе без неинтерактивных сценариев.

Неинтерактивные сценарии прекрасно могут работать в фоне, в то время, как интерактивные -- подвисают, останавливаясь на операциях, ожидающих ввода пользователя. Сложности, возникающие с запуском интерактивных сценариев в фоновом режиме, могут быть преодолены с помощью `expect`-сценария или [встроенного документа](#). В простейших случаях, можно организовать перенаправление ввода из файла в команду `read (read variable <file)`. Эти приемы позволяют создавать сценарии, которые смогут работать как в интерактивном, так и в неинтерактивном режимах.

Если внутри сценария необходимо проверить режим работы -- интерактивный или неинтерактивный, это можно сделать проверкой переменной окружения `$PS1`.

```

if [ -z $PS1 ] # интерактивный режим?
then
  # неинтерактивный
  ...
else
  # интерактивный
  ...
fi

```

Еще один способ -- проверка установки флага "i" в переменной `$-`.

```

case $- in
*i*) # интерактивный режим
;;
*) # неинтерактивный режим
;;
# (Из "UNIX F.A.Q.," 1993)

```



Сценарий может принудительно запускаться в интерактивном режиме, для этого необходимо указать ключ `-i` в строке-заголовке `#!/bin/bash -i`. Однако вы должны помнить о том, что в таких случаях сценарий может выдавать сообщения об ошибках даже тогда, когда ошибок, по сути, нет.

33.2. Сценарии-обертки

"Обертки" -- это сценарии, которые содержат один или несколько вызовов системных команд или утилит, с длинным списком параметров. Такой прием освобождает пользователя от необходимости вводить вручную сложные и длинные команды из командной строки. Он особенно полезен при работе с [sed](#) и [awk](#).

Сценарии **sed** или **awk**, как правило вызываются в форме: `sed -e 'commands'` или `awk 'commands'`. "Заворачивая" такие вызовы в сценарий на языке командной оболочки, мы делаем их использование более простым для конечного пользователя. Кроме того, этот прием позволяет комбинировать вызовы **sed** и **awk**, например в [конвейере](#), позволяя передавать данные с выхода одной утилиты на вход другой.

Пример 33-1. сценарий-обертка

```
#!/bin/bash

# Этот простой сценарий удаляет пустые строки из текстового файла.
# Проверка входных аргументов не производится.
#
# Однако вы можете дополнить сценарий такой проверкой,
# добавив нечто подобное:
# if [ -z "$1" ]
# then
#   echo "Порядок использования: `basename $0` текстовый_файл"
#   exit 65
# fi

# Для выполнения этих же действий,
# из командной строки можно набрать
#   sed -e '/^$/d' filename

sed -e '/^$/d' "$1"
# '-e' -- означает команду "editing" (правка), за которой следуют необязательные
# параметры.
# '^' -- с начала строки, '$' -- до ее конца.
# Что соответствует строкам, которые не содержат символов между началом и концом
# строки,
#+ т.е. -- пустым строкам.
# 'd' -- команда "delete" (удалить).

# Использование кавычек дает возможность
#+ обрабатывать файлы, чьи имена содержат пробелы.

exit 0
```

Пример 33-2. Более сложный пример сценария-обертки

```
#!/bin/bash

# "subst", Сценарий замены по шаблону
# т.е., "subst Smith Jones letter.txt".

ARGS=3
E_BADARGS=65 # Неверное число аргументов.

if [ $# -ne "$ARGS" ]
# Проверка числа аргументов.
then
  echo "Порядок использования: `basename $0` old-pattern new-pattern filename"
  exit $E_BADARGS
fi
```

```

old_pattern=$1
new_pattern=$2

if [ -f "$3" ]
then
    file_name=$3
else
    echo "Файл \"$3\" не найден."
    exit $E_BADARGS
fi

# Здесь, собственно, выполняется сама работа по поиску и замене.
sed -e "s/$old_pattern/$new_pattern/g" $file_name
# 's' -- команда "substitute" (замены),
# a /pattern/ -- задает шаблон искомого текста.
# "g" -- флаг "global" (всеобщий), означает "выполнить подстановку для *каждого*"
# обнаруженного $old_pattern во всех строках, а не только в первой строке.

exit 0    # При успешном завершении сценария -- вернуть 0.

```

Пример 33-3. Сценарий-обертка вокруг сценария awk

```

#!/bin/bash

# Суммирует числа в заданном столбце из заданного файла.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # Проверка числа аргументов.
then
    echo "Порядок использования: `basename $0` имя_файла номер_столбца"
    exit $E_WRONGARGS
fi

filename=$1
column_number=$2

# Здесь используется прием передачи переменных
# из командной оболочки в сценарий awk .

# Многострочный сценарий awk должен записываться в виде:  awk ' ..... '

# Начало awk-сценария.
# -----
awk '

{ total += "${column_number}"
}
END {
    print total
}

' "$filename"
# -----
# Конец awk-сценария.

# С точки зрения безопасности, передача shell-переменных
# во встроенный awk-скрипт, потенциально опасна,
# поэтому, Stephane Chazelas предлагает следующую альтернативу:
# -----
# awk -v column_number="$column_number" '
# { total += $column_number
# }
# END {
#     print total
# }' "$filename"

```

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

```
exit 0
```

Для сценариев, которые должны строиться по принципу швейцарского армейского ножа -- "все в одном", можно порекомендовать Perl. Perl совмещает в себе мощь и гибкость **sed**, **awk** и языка программирования **C**. Он поддерживает модульность и объектно-ориентированный стиль программирования. Короткие сценарии Perl могут легко встраиваться в сценарии командной оболочки, и даже полностью заменить из (хотя автор весьма скептически относится к последнему утверждению).

Пример 33-4. Сценарий на языке Perl, встроенный в Bash-скрипт

```
#!/bin/bash

# Это команды shell, предшествующий сценарию на Perl.
echo "Эта строка выводится средствами Bash, перед выполнением встроенного Perl-
скрипта, в \"\$0\"."
echo
"=====
=====

perl -e 'print "Эта строка выводится средствами Perl.\n";'
# Подобно sed, Perl тоже использует ключ "-e".

echo "=====

exit 0
```

Допускается даже комбинирование сценариев на Bash и на Perl, в пределах одного файла. В зависимости от того, какая часть сценария должна исполняться, сценарий вызывается с указанием требуемого интерпретатора.

Пример 33-5. Комбинирование сценария Bash и Perl в одном файле

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

echo "Вас приветствует часть сценария, написанная на Bash."
# Далее могут следовать другие команды Bash.

exit 0
# Конец сценария на Bash.

# =====

#!/usr/bin/perl
# Эта часть сценария должна вызываться с ключом -x.

print "Вас приветствует часть сценария, написанная на Perl.\n";
# Далее могут следовать другие команды Perl.

# Конец сценария на Perl.

bash$ bash bashandperl.sh
Вас приветствует часть сценария, написанная на Bash.

bash$ perl -x bashandperl.sh
Вас приветствует часть сценария, написанная на Perl.
```

33.3. Операции сравнения: Альтернативные решения

Операции сравнения, выполняемые с помощью конструкции `[[]]`, могут оказаться предпочтительнее, чем `[]`. Аналогично, при сравнении чисел, в более выгодном свете представляется конструкция `(())`.

```
a=8

# Все, приведенные ниже, операции сравнения -- эквивалентны.
test "$a" -lt 16 && echo "да, $a < 16" # "И-список"
/bin/test "$a" -lt 16 && echo "да, $a < 16"
[ "$a" -lt 16 ] && echo "да, $a < 16"
[[ $a -lt 16 ]] && echo "да, $a < 16" # Внутри [[ ]] и (( )) переменные
(( a < 16 )) && echo "да, $a < 16" # не обязательно брать в кавычки.

city="New York"
# Опять же, все, приведенные ниже, операции -- эквивалентны.
test "$city" \< Paris && echo "Да, Paris больше, чем $city" # В смысле ASCII-строк.
/bin/test "$city" \< Paris && echo "Да, Paris больше, чем $city"
[ "$city" \< Paris ] && echo "Да, Paris больше, чем $city"
[[ $city < Paris ]] && echo "Да, Paris больше, чем $city" # Кавычки вокруг $city
не обязательны.

# Спасибо S.C.
```

33.4. Рекурсия

Может ли сценарий [рекурсивно](#) вызывать себя самого? Да, может!

Пример 33-6. Сценарий (бесполезный), который вызывает себя сам

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

# Может ли сценарий вызвать себя сам?
# Да, но есть ли в этом смысл?

RANGE=10
MAXVAL=9

i=$RANDOM
let "i %= $RANGE" # Генерация псевдослучайного числа в диапазоне 0 .. $MAXVAL.

if [ "$i" -lt "$MAXVAL" ]
then
    echo "i = $i"
    ./$0 # Сценарий запускает новый экземпляр себя самого.
fi # если число $i больше или равно $MAXVAL.

# Если конструкцию "if/then" заменить на цикл "while", то это вызовет определенные
проблемы.
# Объясните -- почему?.

exit 0
```

Пример 33-7. Сценарий имеющий практическую ценность), который вызывает себя

сам

```
#!/bin/bash
# pb.sh: телефонная книга

# Автор: Rick Voivie
# используется с его разрешения.
# Дополнен автором документа.

MINARGS=1      # Сценарию должен быть передан, по меньшей мере, один аргумент.
DATAFILE=./phonebook
PROGRAMME=$0
E_NOARGS=70    # Ошибка, нет аргументов.

if [ $# -lt $MINARGS ]; then
    echo "Порядок использования: \"$PROGRAMME\" data"
    exit $E_NOARGS
fi

if [ $# -eq $MINARGS ]; then
    grep $1 "$DATAFILE"
else
    ( shift; "$PROGRAMME" $* ) | grep $1
    # Рекурсивный вызов.
fi

exit 0          # Сценарий завершает свою работу здесь.
                # Далее следует пример файла телефонной книги
                #+ в котором не используются символы комментария.
```

```
# -----
# Пример файла телефонной книги

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

# Если сценарию передаются несколько аргументов,
#+ то выводятся только те строки, которые содержат их все.
```



Слишком глубокая рекурсия может привести к исчерпанию пространства, выделенного под стек, и "вываливанию" сценария по "segfault".

33.5. "Цветные" сценарии

Для установки атрибутов отображения информации на экране, таких как: жирный текст, цвет символов, цвет фона и т.п., с давних пор используются ANSI [\[62\]](#) escape-последовательности. Эти последовательности широко используются в [пакетных файлах DOS](#), эти же последовательности используются и в сценариях Bash.

Пример 33-8. "Цветная" адресная книга

```

#!/bin/bash
# ex30a.sh: Версия сценария ex30.sh, с добавлением цвета .
#          Грубый пример базы данных

clear                # Очистка экрана

echo -n "            "
echo -e '\E[37;44m'\033[1mСписок\033[0m"
                                # Белый текст на синем фоне
echo; echo
echo -e "\033[1mВыберите интересующую Вас персону:\033[0m"
                                # Жирный шрифт
tput sgr0
echo "(Введите только первую букву имени.)"
echo
echo -en '\E[47;34m'\033[1mE\033[0m"   # Синий
tput sgr0                        # сброс цвета
echo "vans, Roland"              # "[E]vans, Roland"
echo -en '\E[47;35m'\033[1mJ\033[0m"   # Пурпурный
tput sgr0
echo "ones, 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: переменная взята в кавычки.
"E" | "e" )
    # Пользователь может ввести как заглавную, так и строчную букву.
    echo
    echo "Roland Evans"
    echo "4321 Floppy Dr."
    echo "Hardscrabble, CO 80753"
    echo "(303) 734-9874"
    echo "(303) 734-9892 fax"
    echo "revans@zzy.net"
    echo "Старый друг и партнер по бизнесу"
    ;;
"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 "Подружка"
    echo "День рождения: 11 февраля"
    ;;
# Информация о Smith и Zane будет добавлена позднее.
* )
    # Выбор по-умолчанию.
    # "Пустой" ввод тоже обрабатывается здесь.
    echo
    echo "Нет данных."
    ;;
esac

```



```
tput sgr0 # Сброс цвета
echo
exit 0
```

Самая простая и, на мой взгляд, самая полезная `escape`-последовательность -- это "жирный текст", `\033[1m ... \033[0m`. Здесь, комбинация `\033` представляет `escape`-символ, комбинация `"[1"` -- включает вывод жирным текстом, а `"[0"` -- выключает. Символ `"m"` -- завершает каждую из `escape`-последовательностей.

```
bash$ echo -e "\033[1mЭто жирный текст.\033[0m"
```

Простая `escape`-последовательность, которая управляет атрибутом подчеркивания (в `rxvt` и `aterm`).

```
bash$ echo -e "\033[4mЭто подчеркнутый текст.\033[0m"
```



Ключ `-e`, в команде `echo`, разрешает интерпретацию `escape`-последовательностей.

Другие `escape`-последовательности, изменяющие атрибуты цвета:

```
bash$ echo -e '\E[34;47mЭтот текст выводится синим цветом.'; tput sgr0
```

```
bash$ echo -e '\E[33;44m"желтый текст на синем фоне"; tput sgr0
```

Команда `tput sgr0` возвращает настройки терминала в первоначальное состояние.

Вывод цветного текста осуществляется по следующему шаблону:

```
echo -e '\E[COLOR1;COLOR2mКакой либо текст.'
```

Где `"\E["` -- начало `escape`-последовательности. Числа `"COLOR1"` и `"COLOR2"`, разделенные точкой с запятой, задают цвет символов и цвет фона, в соответствии с таблицей цветов, приведенной ниже. (Порядок указания цвета текста и фона не имеет значения, поскольку диапазоны числовых значений цвета для текста и фона не пересекаются). Символ `"m"` -- должен завершать `escape`-последовательность.

Обратите внимание: [одиночные кавычки](#) окружают все, что следует за `echo -e`.

Числовые значения цвета, приведенные ниже, справедливы для `rxvt`. Для других эмуляторов они могут несколько отличаться.

Таблица 33-1. Числовые значения цвета в `escape`-последовательностях

Цвет	Текст	Фон
черный	30	40

Цвет	Текст	Фон
красный	31	41
зеленый	32	42
желтый	33	43
синий	34	44
пурпурный	35	45
зеленовато-голубой	36	46
белый	37	47

Пример 33-9. Вывод цветного текста

```
#!/bin/bash
# color-echo.sh: Вывод цветных сообщений.

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'

cecho ()
# Color-echo.
# Аргумент $1 = текст сообщения
# Аргумент $2 = цвет
{
    local default_msg="Нет сообщений."
    # Не обязательно должна быть локальной.

    message=${1:-$default_msg} # Текст сообщения по-умолчанию.
    color=${2:-$black}         # Цвет по-умолчанию черный.

    echo -e "$color"
    echo "$message"
    tput sgr0                  # Восстановление первоначальных настроек терминала.
    return
}

# Попробум что-нибудь вывести.
# -----
cecho "Синий текст..." $blue
cecho "Пурпурный текст." $magenta
cecho "Позеленевший от зависти." $green
cecho "Похоже на красный?" $red
cecho "Циан, более известный как цвет морской волны." $cyan
cecho "Цвет не задан (по-умолчанию черный)."
# Аргумент $color отсутствует.
cecho "\"Пустой\" цвет (по-умолчанию черный)." ""
# Передан "пустой" аргумент цвета.
cecho
# Ни сообщение ни цвет не переданы.
cecho "" ""
# Функции переданы "пустые" аргументы $message и $color.
# -----

echo

exit 0

# Упражнения:
# -----
```

- # 1) Добавьте в функцию 'сecho ()' возможность вывода "жирного текста".
- # 2) Добавьте возможность управления цветом фона.



Однако, как обычно, в бочке меда есть ложка дегтя. *Escape-последовательности ANSI совершенно не переносимы*. Вывод в одном эмуляторе терминала (или в консоли) может разительно отличаться от вывода в другом эмуляторе. "Расцвеченные" сценарии, дающие изумительно красивый вывод текста на одном терминале, могут давать совершенно нечитаемый текст на другом. Это ставит под сомнение практическую ценность "расцвечивания" вывода в сценариях, низводя ее до уровня никчемной "игрушки".

Moshe Jacobson разработал утилиту **color** (<http://runslinux.net/projects/color>), которая значительно упрощает работу с ANSI escape-последовательностями, заменяя, только что обсуждавшиеся, неуклюжие конструкции, логичным и понятным синтаксисом.

33.6. Оптимизация

По большей части, сценарии на языке командной оболочки, используются для быстрого решения несложных задач. Поэтому оптимизация сценариев, по скорости исполнения, не является насущной проблемой. Тем не менее, представьте себе ситуацию, когда сценарий, выполняющий довольно важную работу, в принципе справляется со своей задачей, но делает это очень медленно. Написание же аналогичной программы на языке компилирующего типа -- неприемлемо. Самое простое решение -- переписать самые медленные участки кода сценария. Возможно ли применить принципы оптимизации к сценарию на практике?

Для начала проверьте все циклы в сценарии. Основная масса времени уходит на работу в циклах. Если это возможно, вынесите все ресурсоемкие операции за пределы циклов.

Старайтесь использовать [встроенные](#) команды. Они исполняются значительно быстрее и, как правило, не запускают подоболочку при вызове.

Избегайте использования избыточных команд, особенно это относится к [конвейерам](#).

```
cat "$file" | grep "$word"
```

```
grep "$word" "$file"
```

Эти команды дают один и тот же результат,
#+ но вторая работает быстрее, поскольку запускает на один подпроцесс меньше.

Не следует злоупотреблять командой [cat](#).

Для профилирования сценариев, можно воспользоваться командами [time](#) и [times](#). Не следует пренебрегать возможностью переписать особенно критичные участки кода на языке C или даже на ассемблере.

Попробуйте минимизировать количество операций с файлами. Bash не "страдает" излишней эффективностью при работе с файлами, попробуйте применить специализированные средства для работы с файлами в сценариях, такие как [awk](#) или [Perl](#).

Записывайте сценарии в структурированной форме, это облегчит их последующую реорганизацию и оптимизацию. Помните, что значительная часть методов оптимизации

кода, существующих в языках высокого уровня, вполне применима и к сценариям, однако есть и такие, которые не могут применяться. Основным критерий здесь -- это здравый смысл.

Прекрасный пример того, как оптимизация может сократить время работы сценария, вы найдете в [Пример 12-32](#).

33.7. Разные советы

- Для ведения учета использования сценария пользователями, добавьте следующие строки в сценарий. Они запишут в файл отчета название сценария и время запуска.

```
# Добавление (>>) учетной записи, об использовании сценария, в файл отчета.
```

```
date>> $SAVE_FILE      # Дата и время.
echo $0>> $SAVE_FILE   # Название сценария.
echo>> $SAVE_FILE      # Пустая строка -- как разделитель записей.
```

```
# Не забудьте определить переменную окружения SAVE_FILE в ~/.bashrc
# (чтонибудь, типа: ~/.scripts-run)
```

- Оператор >> производит добавление строки в конец файла. А как быть, если надо добавить строку в начало существующего файла?

```
file=data.txt
title="***Это титульная строка в текстовом файле***"

echo $title | cat - $file >$file.new
# "cat -" объединяет stdout с содержимым $file.
# В результате получится
#+ новый файл $file.new, в начало которого добавлена строка $title.
```

Само собой разумеется, то же самое можно сделать с помощью [sed](#).

- Сценарий командной оболочки может использоваться как команда внутри другого сценария командной оболочки, *Tcl*, или *wish* сценария или, даже в [Makefile](#). Он может быть вызван как внешняя команда из программы на языке C, с помощью функции *system()*, т.е. *system("script_name");*.
- Собирайте свои библиотеки часто используемых функций и определений. Эти "библиотеки" могут быть "подключены" к сценариям, с помощью команды [точка](#) (*.*) или [source](#).

```
# Сценарий-библиотека
# -----
```

```
# Обратите внимание:
# Здесь нет sha-bang ("#!").
# И нет "живого кода".
```

```
# Определения переменных
```

```
ROOT_UID=0          # UID root-a, 0.
```

```

E_NOTROOT=101          # Ошибка -- "обычный пользователь".
MAXRETVAl=255         # Максимальное значение, которое могут возвращать
функции.
SUCCESS=0
FAILURE=-1

# Функции

Usage ()              # Сообщение "Порядок использования:".
{
  if [ -z "$1" ]      # Нет аргументов.
  then
    msg=filename
  else
    msg=$@
  fi

  echo "Порядок использования: `basename $0` "$msg""
}

Check_if_root ()     # Проверка прав пользователя.
{
  # из примера "ex39.sh".
  if [ "$UID" -ne "$ROOT_UID" ]
  then
    echo "Этот сценарий должен запускаться с привилегиями root."
    exit $E_NOTROOT
  fi
}

CreateTempfileName () # Создание "уникального" имени для временного файла.
{
  # Из примера "ex51.sh".
  prefix=temp
  suffix=`eval date +%s`
  Tempfilename=$prefix.$suffix
}

isalpha2 ()          # Проверка, состоит ли строка только из алфавитных
символов.
{
  # Из примера "isalpha.sh".
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
    *[!a-zA-Z]*|") return $FAILURE;;
    *) return $SUCCESS;;
  esac
  # Спасибо S.C.
}

abs ()               # Абсолютное значение.
{
  # Внимание: Максимально возможное возвращаемое
  значение
  # не может превышать 255.

  E_ARGERR=-999999

  if [ -z "$1" ]     # Проверка наличия входного аргумента.
  then
    return $E_ARGERR # Код ошибки, обычно возвращаемый в таких
случаях.
  fi

  if [ "$1" -ge 0 ]  # Если не отрицательное,
  then
    #
    absval=$1       # оставить как есть.
  else
    # Иначе,

```

```

    let "absval = (( 0 - $1 ))" # изменить знак.
fi

return $absval
}

tolower ()          # Преобразование строк символов в нижний регистр
{
    if [ -z "$1" ]   # Если нет входного аргумента,
    then             #+ выдать сообщение об ошибке
        echo "(null)"
        return       #+ и выйти из функции.
    fi

    echo "$@" | tr A-Z a-z
    # Преобразовать все входные аргументы ($@).

    return

# Для записи результата работы функции в переменную, используйте операцию
# подстановки команды.
# Например:
#   oldvar="A seT of miXed-caSe LEtTerS"
#   newvar=`tolower "$oldvar"`
#   echo "$newvar"    # a set of mixed-case letters
#
# Упражнение: Добавьте в эту библиотеку функцию перевода символов в верхний
# регистр.
#           toupper() [это довольно просто].
}

```

- Для повышения ясности комментариев, выделяйте их особым образом.

```

## Внимание!
rm -rf *.zzy    ## Комбинация ключей "-rf", в команде "rm", чрезвычайно опасна,
                ##+ особенно при удалении по шаблону.

#+ Продолжение комментария на новой строке.
# Это первая строка комментария
#+ это вторая строка комментария,
#+ это последняя строка комментария.

#* Обратите внимание.

#o Элемент списка.

#> Альтернативный вариант.
while [ "$var1" != "end" ]    #> while test "$var1" != "end"

```

- Для создания блочных комментариев, можно использовать конструкцию [if-test](#).

```

#!/bin/bash

COMMENT_BLOCK=
# Если попробовать инициализировать эту переменную чемнибудь,
#+ то вы получите неожиданный результат.

if [ $COMMENT_BLOCK ]; then

Блок комментария --
=====

```

```
Это строка комментария.  
Это другая строка комментария.  
Это еще одна строка комментария.  
=====
```

```
echo "Эта строка не выводится."
```

Этот блок комментария не вызывает сообщения об ошибке! Круто!

```
fi
```

```
echo "Эта строка будет выведена на stdout."
```

```
exit 0
```

Сравните этот вариант создания блочных комментариев со [встроенным документом, использующимся для создания блочных комментариев](#).

- С помощью служебной переменной `$?`, можно проверить -- является ли входной аргумент целым числом.

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

```
SUCCESS=0  
E_BADINPUT=65
```

```
test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null  
# Проверка: "равно нулю или не равно нулю".  
# 2>/dev/null подавление вывода сообщений об ошибках.
```

```
if [ $? -ne "$SUCCESS" ]  
then  
    echo "Порядок использования: `basename $0` целое_число"  
    exit $E_BADINPUT  
fi
```

```
let "sum = $1 + 25"           # Будет выдавать ошибку, если $1 не является  
целым числом.  
echo "Sum = $sum"
```

```
# Любая переменная может быть проверена таким образом, а не только входные  
аргументы.
```

```
exit 0
```

- Диапазон, возвращаемых функциями значений, 0 - 255 -- серьезное ограничение. Иногда может оказаться весьма проблематичным использование глобальных переменных, для передачи результата из функции. В таких случаях можно порекомендовать передачу результатов работы функции через запись в `stdout`.

Пример 33-10. Необычный способ передачи возвращаемого значения

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

```
# multiplication.sh
```

```
multiply ()                 # Функция выполняет перемножение всех переданных  
аргументов.  
{
```

```
    local product=1
```

```
    until [ -z "$1" ]      # Пока не дошли до последнего аргумента...
```

```

do
    let "product *= $1"
    shift
done

echo $product          # Значение не будет выведено на экран,
                        #+ поскольку оно будет записано в переменную.

mult1=15383; mult2=25211
val1=`multiply $mult1 $mult2`
echo "$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

```

Такой прием срабатывает и для строковых значений. Таким образом, функция может "возвращать" и нечисловой результат.

```

capitalize_ichar ()    # Первый символ всех строковых аргументов
{
    #+ переводится в верхний регистр.

    string0="$@"      # Принять все аргументы.

    firstchar=${string0:0:1} # Первый символ.
    string1=${string0:1}    # Остаток строки.

    FirstChar=`echo "$firstchar" | tr a-z A-Z`
                    # Преобразовать в верхний регистр.

    echo "$FirstChar$string1" # Выдать на stdout.
}

newstring=`capitalize_ichar "each sentence should start with a capital
letter."`
echo "$newstring"      # Each sentence should start with a capital letter.

```

Используя этот прием, функция может "возвращать" даже несколько значений.

Пример 33-11. Необычный способ получения нескольких возвращаемых значений

```

#!/bin/bash
# sum-product.sh
# Функция может "возвращать" несколько значений.

sum_and_product () # Вычисляет сумму и произведение аргументов.
{
    echo $(( $1 + $2 )) $(( $1 * $2 ))
# Вывод на stdout двух значений, разделенных пробелом.
}

echo
echo "Первое число: "

```



```

read first

echo
echo "Второе число: "
read second
echo

retval=`sum_and_product $first $second`      # Получить результат.
sum=`echo "$retval" | awk '{print $1}'`      # Первое значение (поле).
product=`echo "$retval" | awk '{print $2}'`  # Второе значение (поле).

echo "$first + $second = $sum"
echo "$first * $second = $product"
echo

exit 0

```

- Следующая хитрость -- передача [массива](#) в [функцию](#), и "возврат" массива из функции.

Передача массива в функцию выполняется посредством записи элементов массива, разделенных пробелами, в переменную, с помощью операции [подстановки команды](#). Получить массив обратно можно, следуя вышеописанной стратегии, через вывод на stdout, а затем, с помощью все той же операции подстановки команды и оператора (...) -- записать в массив.

Пример 33-12. Передача массива в функцию и возврат массива из функции

```

#!/bin/bash
# array-function.sh: Передача массива в функцию и...
#                      "возврат" массива из функции

Pass_Array ()
{
    local passed_array # Локальная переменная.
    passed_array=( `echo "$1"` )
    echo "${passed_array[@]}"
    # Список всех элементов в новом массиве,
    #+ объявленном и инициализированном в функции.
}

original_array=( element1 element2 element3 element4 element5 )

echo
echo "original_array = ${original_array[@]}"
#                      Список всех элементов исходного массива.

# Так можно отдать массив в функцию.
# *****
argument=`echo ${original_array[@]}`
# *****
# Поместив все элементы массива в переменную,
#+ разделяя их пробелами.
#
# Обратите внимание: метод прямой передачи массива в функцию не работает.

# Так можно получить массив из функции.
# *****
returned_array=( `Pass_Array "$argument"` )
# *****
# Записать результат в переменную-массив.

```

```

echo "returned_array = ${returned_array[@]}"

echo "======"

# А теперь попробуйте получить доступ к локальному массиву
#+ за пределами функции.
Pass_Array "$argument"

# Функция выведет массив, но...
#+ доступ к локальному массиву, за пределами функции, окажется невозможен.
echo "Результирующий массив (внутри функции) = ${passed_array[@]}"
# "ПУСТОЕ" ЗНАЧЕНИЕ, поскольку это локальная переменная.

echo

exit 0

```

Более сложный пример передачи массивов в функции, вы найдете в [Пример A-11](#).

- Использование конструкций с двойными круглыми скобками позволяет применять C-подобный синтаксис операций присвоения и инкремента переменных, а также оформления циклов [for](#) и [while](#). См. [Пример 10-12](#) и [Пример 10-17](#).
- Иногда очень удобно "пропускать" данные через один и тот же фильтр, но с разными параметрами, используя конвейерную обработку. Особенно это относится к [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-\177' Z | tr Z ' '`

```

Пример 33-13. Игры с анаграммами

```

#!/bin/bash
# agram.sh: Игры с анаграммами.

# Поиск анаграмм...
LETTERSET=etaoinshrdlu

agram "$LETTERSET" | # Найти все анаграммы в наборе символов...
grep '.....' |      # состоящие, как минимум из 7 символов,
grep '^is' |        # начинающиеся с 'is'
grep -v 's$' |      # исключая множественное число
grep -v 'ed$'       # и глаголы в прошедшем времени

# Здесь используется утилита "agram"
#+ которая входит в состав пакета "yawl" , разработанного автором.
# http://ibiblio.org/pub/Linux/libs/yawl-0.2.tar.gz

exit 0                # Конец.

bash$ sh agram.sh
islander
isolate
isolead
isotheral

```

См. также [Пример 27-2](#), [Пример 12-18](#) и [Пример A-10](#).

- Для создания блочных комментариев можно использовать "[анонимные встроенные документы](#)". См. [Пример 17-10](#).

- Попытка вызова утилиты из сценария на машине, где эта утилита отсутствует, потенциально опасна. Для обхода подобных проблем можно воспользоваться утилитой [whatis](#).

```


CMD=command1           # Основной вариант.
PlanB=command2         # Запасной вариант.

command_test=$(whatis "$CMD" | grep 'nothing appropriate')
# Если 'command1' не найдена в системе, то 'whatis' вернет
#+ "command1: nothing appropriate."
#==> От переводчика: Будьте внимательны! Если у вас локализованная версия
whatis
#==> то вывод от нее может отличаться от используемого здесь ('nothing
appropriate')

if [[ -z "$command_test" ]] # Проверка наличия утилиты в системе.
then
    $CMD option1 option2    # Запуск команды с параметрами.
else
    $PlanB                  # Иначе,
                           #+ запустить command2 (запасной вариант).
fi

```

- Команда [run-parts](#) удобна для запуска нескольких сценариев, особенно в комбинации с [cron](#) или [at](#).
- Было бы неплохо снабдить сценарий графическим интерфейсом X-Window. Для этого можно порекомендовать пакеты *Xscript*, *Xmenu* и *widtools*. Правда, первые два, кажется больше не поддерживаются разработчиками. Зато *widtools* можно получить [здесь](#).

 Пакет *widtools* (widget tools) требует наличия библиотеки *XForms*. Кроме того, необходимо слегка подправить [Makefile](#), чтобы этот пакет можно было собрать на типичной Linux-системе. Но хуже всего то, что три из шести виджетов не работают :-(((segfault).

Для постороения приложений с графическим интерфейсом, можно попробовать *Tk*, или *wish* (надстройка над *Tcl*), *PerlTk* (Perl с поддержкой Tk), *tksh* (ksh с поддержкой Tk), *XForms4Perl* (Perl с поддержкой XForms), *Gtk-Perl* (Perl с поддержкой Gtk) или *PyQt* (Python с поддержкой Qt).

33.8. Проблемы безопасности

Уместным будет лишний раз предупредить о соблюдении мер предосторожности при работе с незнакомыми сценариями. Сценарий может содержать *червя*, *трояна* или даже *вирус*. Если вы получили сценарий не из источника, которому доверяете, то никогда не запускайте его с привилегиями root и не позволяйте вставлять его в список сценариев начальной инициализации системы в `/etc/rc.d`, пока не убедитесь в том, что он безвреден для системы.

Исследователи из Bell Labs и других организаций, включая M. Douglas McIlroy, Tom Duff, и Fred Cohen исследовали вопрос о возможности создания вирусов на языке сценариев командной оболочки, и пришли к выводу, что это делается очень легко и доступно даже для новичков. [\[63\]](#)

Это еще одна из причин, по которым следует изучать язык командной оболочки. Способность читать и понимать сценарии поможет вам предотвратить возможность взлома и/или разрушения вашей системы.

33.9. Проблемы переносимости

Эта книга делает упор на создании сценариев для командной оболочки Bash, для операционной системы GNU/Linux. Тем не менее, многие рекомендации, приводимые здесь, могут быть вполне применимы и для других командных оболочек, таких как **sh** и **ksh**.

Многие версии командных оболочек стремятся следовать стандарту POSIX 1003.2. Вызывая Bash с ключом `--posix`, или вставляя **set -o posix** в начало сценария, вы можете заставить Bash очень близко следовать этому стандарту. Но, даже без этого ключа, большинство сценариев, написанных для Bash, будут работать под управлением **ksh**, и наоборот, т.к. Chet Ramey перенес многие особенности, присущие **ksh**, в последние версии Bash.

В коммерческих версиях UNIX, сценарии, использующие GNU-версии стандартных утилит и команд, могут оказаться неработоспособными. Однако, с течением времени, таких проблем остается все меньше и меньше, поскольку утилиты GNU, в большинстве своем, заместили свои проприетарные аналоги в UNIX. После того, как [Caldera дала разрешение на публикацию исходного кода](#) некоторых версий оригинальных утилит UNIX, этот процесс значительно ускорился.

Bash имеет некоторые особенности, недоступные в традиционном Bourne shell. Среди них:

- Некоторые дополнительные [ключи вызова](#)
- [Подстановка команд](#), с использованием нотации `$()`
- Некоторые [операции над строками](#)
- [Подстановка процессов](#)
- [встроенные команды](#) Bash

Более подробный список характерных особенностей Bash, вы найдете в [Bash F.A.Q.](#)

33.10. Сценарии командной оболочки под Windows

Даже те пользователи, которые работают в *другой*, не UNIX-подобной операционной системе, смогут запускать сценарии командной оболочки, а потому -- найти для себя много полезного в этой книге. Пакеты [Cygwin](#) от Cygnus, и [MKS utilities](#) от Mortice Kern Associates, позволяют дополнить Windows возможностями командной оболочки.

Глава 34. Bash, версия 2

Текущая версия *Bash*, та, которая скорее всего установлена в вашей системе, фактически -- 2.XX.Y.

```
bash$ echo $BASH_VERSION
2.05.8(1)-release
```

В этой версии классического языка сценариев Bash были добавлены переменные-массивы, [\[64\]](#) расширение строк и подстановка параметров, улучшен метод косвенных ссылок на переменные.

Пример 34-1. Расширение строк

```
#!/bin/bash

# "Расширение" строк (String expansion).
# Введено в Bash, начиная с версии 2.

# Строки вида '$xxx'
# могут содержать дополнительные экранированные символы.

echo '$Звонок звенит 3 раза \a \a \a'
echo '$Три перевода формата \f \f \f'
echo '$10 новых строк \n\n\n\n\n\n\n\n\n\n\n'

exit 0
```

Пример 34-2. Косвенные ссылки на переменные -- новый метод

```
#!/bin/bash

# Косвенные ссылки на переменные.

a=letter_of_alphabet
letter_of_alphabet=z

echo "a = $a"           # Прямая ссылка.

echo "Now a = ${!a}"    # Косвенная ссылка.
# Форма записи ${!variable} намного удобнее старой "eval var1=\$$var2"

echo

t=table_cell_3
table_cell_3=24
echo "t = ${!t}"        # t = 24
table_cell_3=387
echo "Значение переменной t изменилось на ${!t}"    # 387

# Теперь их можно использовать для ссылок на элементы массива,
# или для эмуляции многомерных массивов.
# Было бы здорово, если бы косвенные ссылки допускали индексацию.

exit 0
```

Пример 34-3. Простая база данных, с применением косвенных ссылок

```
#!/bin/bash
# resistor-inventory.sh
# Простая база данных, с применением косвенных ссылок.
```

```

# ===== #
# Данные

B1723_value=470                # сопротивление (Ом)
B1723_powerdissip=.25         # рассеиваемая мощность (Вт)
B1723_colorcode="желтый-фиолетовый-коричневый" # цветовая маркировка
B1723_loc=173                 # где
B1723_inventory=78           # количество (шт)

B1724_value=1000
B1724_powerdissip=.25
B1724_colorcode="коричневый-черный-красный"
B1724_loc=24N
B1724_inventory=243

B1725_value=10000
B1725_powerdissip=.25
B1725_colorcode="коричневый-черный-оранжевый"
B1725_loc=24N
B1725_inventory=89

# ===== #

echo

PS3='Введите номер: '

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 по каталогу $catalog_number:"
    echo "Имеется в наличии ${!Inv} шт. [${!Val} Ом / ${!Pdissip} Вт]."
    echo "Находятся в лотке # ${!Loc}."
    echo "Цветовая маркировка: \"${!Ccode}\"."

    break
done

echo; echo

# Упражнение:
# -----
# Переделайте этот сценарий так, чтобы он использовал массивы вместо косвенных
# ссылок.
# Какой из вариантов более простой и интуитивный?

# Примечание:
# -----
# Язык командной оболочки не очень удобен для написания приложений,
#+ работающих с базами данных.
# Для этой цели лучше использовать языки программирования, имеющие
#+ развитые средства для работы со структурами данных,
#+ такие как C++ или Java (может быть Perl).

exit 0

```

Пример 34-4. Массивы и другие хитрости для раздачи колоды карт в четыре руки

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

```

# На старых системах может потребоваться вставить #!/bin/bash2.

# Карты:
# раздача в четыре руки.

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
# Проще и понятнее было бы, имей мы дело
# с одним 3-мерным массивом.
# Будем надеяться, что в будущем, поддержка многомерных массивов будет введена в
# Bash.

initialize_Deck ()
{
i=$LOWER_LIMIT
until [ "$i" -gt $UPPER_LIMIT ]
do
    Deck[i]=$UNPICKED    # Пометить все карты в колоде "Deck", как "невыданная".
    let "i += 1"
done
echo
}

initialize_Suits ()
{
Suits[0]=T # Трефы
Suits[1]=Б # Бубны
Suits[2]=Ч # Червы
Suits[3]=П # Пики
}

initialize_Cards ()
{
Cards=(2 3 4 5 6 7 8 9 10 В Д К Т)
# Альтернативный способ инициализации массива.
}

pick_a_card ()
{
card_number=$RANDOM
let "card_number %= $CARDS"
if [ "${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 по столбцам.
}

seed_random () # Переустановка генератора случайных чисел.
{
seed=`eval date +%s`
let "seed %= 32766"
RANDOM=$seed
}

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
# Возврат к индексации с 1 (временно).
let "u %= $CARDS_IN_SUIT"
if [ "$u" -eq 0 ] # вложенный if/then.
then
echo
echo
fi
# Смена руки.

let "cards_picked += 1"
fi
done

echo

return 0
}

# Структурное программирование:
# вся логика приложения построена на вызове функций.

#=====
seed_random
initialize_Deck
initialize_Suits
initialize_Cards
deal_cards

exit 0
#=====

# Упражнение 1:
# Добавьте комментарии, чтобы до конца задокументировать этот сценарий.

# Упражнение 2:
# Исправьте сценарий так, чтобы карты в каждой руке выводились отсортированными по масти.
# Вы можете добавить и другие улучшения.

```


Глава 35. Замечания и дополнения

35.1. От автора

Как я пришел к мысли о написании этой книги? Это необычная история. Случилось это лет несколько тому назад. Мне потребовалось изучить язык командной оболочки -- а что может быть лучше, как не чтение хорошей книги!? Я надеялся купить учебник и справочник, которые охватывали бы в полной мере данную тематику. Я искал книгу, которая возьмет трудные понятия, вывернет их наизнанку и подробно разжует на хорошо откомментированных примерах. В общем, я искал очень хорошую книгу. К сожалению, в природе таковой не существовало, поэтому я счел необходимым написать ее.

Это напоминает мне сказку о сумасшедшем профессоре. Помешанный, до безумия, при виде книги, любой книги -- в библиотеке, в книжном магазине -- не важно где, им овладевала уверенность в том, что и он мог бы написать эту книгу, причем сделать это гораздо лучше. Он стремительно мчался домой и садился за создание своей собственной книги с тем же названием. Когда он умер, в его доме нашли несколько тысяч, написанных им книг, этого количества хватило бы, чтобы посрамить самого Айзека Азимова. Книги, может быть и не были так хороши -- кто знает, но разве это имеет какое-то значение? Вот -- человек, жил своими грезами, пусть одержимый и движимый ими, но я не могу удержаться от восхищения старым чудаком...

35.2. Об авторе

Автор не стремится ни к званиям, ни к наградам, им движет неодолимое желание писать. [\[65\]](#) Эта книга -- своего рода отдых от основной работы, [HOW-2 Meet Women: The Shy Man's Guide to Relationships](#) (Руководство Застенчивого Мужчины о том Как Познакомиться С Женщиной) . Он также написал [Software-Building HOWTO](#).

Пользуется Linux с 1995 года (Slackware 2.2, kernel 1.2.1). Выпустил несколько программ, среди которых [cruft](#) -- утилита шифрования, заменявшая стандартную UNIX-овую [crypt](#), [mcalc](#) -- финансовый калькулятор, для выполнения расчетов по займам, [judge](#) и [yawl](#) -- пакет игр со словами. Программировать начинал с языка FORTRAN IV на CDC 3800, но не испытывает ностальгии по тем дням.

Живет в глухой, заброшенной деревушке со своей женой и собакой.

35.3. Инструменты, использовавшиеся при создании книги

35.3.1. Аппаратура

IBM Thinkpad, model 760XL laptop (P166, 104 Mb RAM) под управлением Red Hat 7.1/7.3.

Несомненно, это довольно медлительный агрегат, но он имеет отличную клавиатуру, и это много лучше, чем пара карандашей и письменный стол.

35.3.2. Программное обеспечение

- i. Мощный текстовый редактор [vim](#) (автор: Bram Moolenaar) .
 - ii. [OpenJade](#) -- инструмент, выполняющий, на основе DSSSL, верификацию и преобразование SGML-документов в другие форматы.
 - iii. [Таблицы стилей DSSSL от Norman Walsh](#).
 - iv. *DocBook, The Definitive Guide* (Norman Walsh, Leonard Mueller O'Reilly, ISBN 1-56592-580-7). Полное руководство по созданию документов в формате Docbook SGML.
-

35.4. Благодарности

Без участия сообщества этот проект был бы невозможен. Автор признает, что без посторонней помощи, написание этой книги стало бы невыполнимой задачей и благодарит всех, кто оказал посильную помощь.

[Philippe Martin](#) -- перевел этот документ в формат DocBook/SGML. Работает в маленькой французской компании, в качестве разработчика программного обеспечения. В свободное от работы время -- любит работать над документацией или программным обеспечением для GNU/Linux, читать книги, слушать музыку и веселиться с друзьями. Вы можете столкнуться с ним, где-нибудь во Франции, в провинции Басков, или написать ему письмо на feloy@free.fr.

Philippe Martin также отметил, что возможно использование позиционных параметров за \$9, при использовании {фигурных скобок}, см. [Пример 4-5](#).

[Stephane Chazelas](#) -- выполнил титаническую работу по корректировке, дополнению и написанию примеров сценариев. Фактически, он взвалил на свои плечи обязанности **редактора** этого документа. Огромное спасибо!

Особенно я хотел бы поблагодарить *Patrick Callahan*, *Mike Novak* и *Pal Domokos* за исправление ошибок и неточностей, за разъяснения и дополнения. Их живое обсуждение проблем, связанных с созданием сценариев на языке командной оболочки вдохновило меня на попытку сделать этот документ более удобочитаемым.

Я благодарен Jim Van Zandt за выявленные им ошибки и упущения, в версии 0.2 этого документа, и за поучительный пример сценария.

Большое спасибо [Jordi Sanfeliu](#) за то, что он дал возможность использовать его прекрасный сценарий в этой книге ([Пример A-19](#)).

Выражаю свою благодарность [Michel Charpentier](#) за разрешение использовать его [dc](#) сценарий разложения на простые множители ([Пример 12-37](#)).

Спасибо [Noah Friedman](#), предоставившему право использовать его сценарий ([Пример A-20](#)).

[Emmanuel Rouat](#) предложил несколько изменений и дополнений в разделах, посвященных [подстановке команд](#) и [псевдонимам](#). Он так же предоставил замечательный пример файла `.bashrc` ([Приложение G](#)).

[Heiner Steven](#) любезно разрешил опубликовать его сценарий [Пример 12-33](#). Он сделал множество исправлений и внес большое количество предложений. Особое спасибо!

Rick Boivie предоставил отличный сценарий, демонстрирующий рекурсию, `pb.sh` ([Пример 33-7](#)) и внес предложения по повышению производительности сценария `monthlypmt.sh` ([Пример 12-32](#)).

Florian Wisser оказывал содействие при написании разделов, посвященных строкам (см. [Пример 7-6](#)).

Oleg Philon передал свои предложения относительно команд [cut](#) и [pidof](#).

Michael Zick расширил пример с [пустыми массивами](#), введя туда демонстрацию необычных свойств массивов. Он также предоставил ряд других примеров.

Marc-Jano Knorr выполнил исправления в разделе, посвященном пакетным файлам DOS.

Hyun Jin Cha, в процессе работы над корейским переводом, обнаружил несколько опечаток в документе. Спасибо ему за это!

Andreas Abraham передал большое число типографских ошибок и внес ряд исправлений. Особое спасибо!

Кроме того, я хотел бы выразить свою признательность Gabor Kiss, Leopold Toetsch, Peter Tillier, Marcus Berglof, Tony Richardson, Nick Drage, 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", Emilio Conti, Dennis Leeuw, Dan Jacobson и David Lawyer (автор 4-х HOWTO).

Мои благодарности [Chet Ramey](#) и Brian Fox за создание **Bash** -- этого элегантного и мощного инструмента!

Особое спасибо добровольцам из [Linux Documentation Project](#). Проект LDP сделал возможным публикацию этой книги в своем архиве.

Больше всего я хотел бы выразить свою благодарность моей супруге, Anita, за ее эмоциональную поддержку.

Литература

Edited by Peter Denning, *Computers Under Attack: Intruders, Worms, and Viruses*, ACM Press, 1990, 0-201-53067-8.

Содержит несколько статей о вирусах, написанных на языке командной оболочки.

*

Dale Dougherty and Arnold Robbins, *Sed and Awk*, 2nd edition, O'Reilly and Associates, 1997, 1-156592-225-5.

Чтобы раскрыть всю мощь командной оболочки, вам наверняка потребуется знакомство с **sed** и **awk**. Это обычный учебник. Здесь вы найдете превосходное введение в "регулярные выражения". Обязательно прочитайте эту книгу.

*

Aeleen Frisch, *Essential System Administration*, 3rd edition, O'Reilly and Associates, 2002, 0-596-00343-9.

Это замечательное руководство для системных администраторов. Может служить неплохим введением в программирование сценариев. Содержит подробные пояснения к сценариям загрузки и инициализации системы.

*

Stephen Kochan and Patrick Woods, *Unix Shell Programming*, Hayden, 1990, 067248448X.

Стандартный справочник, хотя немного устаревший.

*

Neil Matthew and Richard Stones, *Beginning Linux Programming*, Wrox Press, 1996, 1874416680.

Дает хороший, глубокий охват различных языков программирования, доступных в Linux, включая довольно сильную главу по программированию в командной оболочке.

*

Herbert Mayer, *Advanced C Programming on the IBM PC*, Windcrest Books, 1989, 0830693637.

Замечательная книга по алгоритмам и практическому программированию.

*

David Medinets, *Unix Shell Programming Tools*, McGraw-Hill, 1999, 0070397333.

Отличная книга по программированию в командной оболочке, с примерами, и 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 отважная попытка создать учебник для начинающих, но он получился несколько несовершенным, к тому же не изобилует примерами сценариев.

*

Anatole Olczak, *Bourne Shell Quick Reference Guide*, ASP, Inc., 1991, 093573922X.

Очень удобный карманный справочник, несмотря на недостатки, при охвате специфичных свойств Bash.

*

Jerry Peek, Tim O'Reilly, and Mike Loukides, *Unix Power Tools*, 2nd edition, O'Reilly and Associates, Random House, 1997, 1-56592-260-3.

Содержит ряд очень информативных разделов, посвященных программированию в командной оболочке, но не может рассматриваться как учебное пособие.

*

Clifford Pickover, *Computers, Pattern, Chaos, and Beauty*, St. Martin's Press, 1990, 0-312-04123-3.

Сокровищница идей и рецептов по машинным вычислениям.

*

George Polya, *How To Solve It*, Princeton University Press, 1973, 0-691-02356-5.

Классический учебник по методам решения задач.

*

Arnold Robbins, *Bash Reference Card*, SSC, 1998, 1-58731-010-5.

Замечательный карманный справочник по Bash. Стоит всего \$4.95, но также доступен для свободного скачивания [on-line](#) в формате PDF.

*

Arnold Robbins, *Effective Awk Programming*, Free Software Foundation / O'Reilly and Associates, 2000, 1-882114-26-4.

Самое лучшее учебное руководство и справочник по **awk**. Свободная электронная версия книги включена в состав документации к **awk**. Печатное издание последней версии доступно на сайте O'Reilly and Associates.

Эта книга служила источником вдохновения для автора этой книги.

*

Bill Rosenblatt, *Learning the Korn Shell*, O'Reilly and Associates, 1993, 1-56592-054-6.

Эта, хорошо написанная книга, содержит массу указаний по созданию сценариев командной оболочки.

*

Paul Sheer, *LINUX: Rute User's Tutorial and Exposition*, 1st edition, , 2002, 0-13-033351-4.

Очень хорошее введение в системное администрирование Linux.

Эта книга доступна в [on-line](#).

*

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.

Один из лучших справочников по командам Linux, имеет раздел, посвященный Bash.

*

The UNIX CD Bookshelf, 3rd edition, O'Reilly and Associates, 2003, 0-596-00392-7.

Сборник из 7-ми книг по UNIX на CD ROM. В состав сборника входят такие книги, как *UNIX Power Tools*, *Sed and Awk* и *Learning the Korn Shell*. Полный набор необходимых справочных и учебных материалов, который вам только может понадобиться. Стоит примерно \$130.

*

Книги издательства O'Reilly, посвященные Perl.

Ben Okornik опубликовал серию отличных статей *introductory Bash scripting* в выпусках 53, 54, 55, 57 и 59 на сайте [Linux Gazette](#) , и статью "The Deep, Dark Secrets of Bash" в выпуске 56.

Chet Ramey *bash - The GNU Shell* -- серия статей в 3 и 4 выпусках [Linux Journal](#), Июль-Август 1994.

Mike G [Bash-Programming-Intro HOWTO](#).

Richard [UNIX Scripting Universe](#).

Chet Ramey [Bash F.A.Q.](#)

Ed Schaefer [Shell Corner](#) на [Unix Review](#).

Примеры сценариев: [Lucc's Shell Scripts](#) .

Примеры сценариев: [SHELLdorado](#) .

Примеры сценариев: [Noah Friedman's script site](#).

Steve Parker [Shell Programming Stuff](#).

Примеры сценариев: [SourceForge Snippet Library - shell scripts](#).

Giles Orr [Bash-Prompt HOWTO](#).

Замечательное руководство по регулярным выражениям, **sed** и **awk** [The UNIX Grymoire](#).

Eric Pement [sed resources page](#).

[The GNU gawk reference manual](#) (**gawk** -- GNU-версия **awk** для ОС Linux и BSD).

Trent Fisher [groff tutorial](#).

Mark Komarinski [Printing-Usage HOWTO](#).

Хороший материал по [перенаправлению ввода/вывода глава 10](#) на сайте [University of Alberta](#).

[Rick Hohensee osimpa](#) -- ассемблер для процессора i386, написан полностью на Bash.

Rocky Bernstein ведет разработку "полнофункционального" [отладчика](#) для Bash.

Отличное руководство "Bash Reference Manual", авторы Chet Ramey и Brian Fox, распространяется в составе пакета "bash-2-doc" (доступен как rpm). В этом пакете вы найдете особенно поучительные примеры.

Группа новостей [comp.os.unix.shell](#).

Страницы руководства man по **bash** и **bash2**, **date**, **expect**, **expr**, **find**, **grep**, **gzip**, **ln**, **patch**, **tar**, **tr**, **bc**, **xargs**. Странички info по **bash**, **dd**, **m4**, **gawk** и **sed**.

Приложение А. Дополнительные примеры сценариев

В этом приложении собраны сценарии, которые не попали в основной текст документа. Однако, они определенно стоят того, что бы вы потратили время на их изучение.

Пример А-1. manview: Просмотр страниц руководств man

```
#!/bin/bash
# manview.sh: Просмотр страниц руководств man в форматированном виде.

# Полезен писателям страниц руководств, позволяет просмотреть страницы в исходном
# коде
#+ как они будут выглядеть в конечном виде.

E_WRONGARGS=65

if [ -z "$1" ]
then
    echo "Порядок использования: `basename $0` имя_файла"
    exit $E_WRONGARGS
fi

groff -Tascii -man $1 | less

# Если страница руководства включает в себя таблицы и/или выражения,
# то этот сценарий "стошнит".
# Для таких случаев можно использовать следующую строку.
#
#   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
#
```

```
# Спасибо S.C.
```

```
exit 0
```

Пример А-2. mailformat: Форматирование электронных писем

```
#!/bin/bash
# mail-format.sh: Форматирование электронных писем.

# Удаляет символы "^", табуляции и ограничивает чрезмерно длинные строки.

# =====
#                               Стандартная проверка аргументов
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne $ARGS ] # Проверка числа аргументов
then
    echo "Порядок использования: `basename $0` имя_файла"
    exit $E_BADARGS
fi

if [ -f "$1" ]      # Проверка наличия файла.
then
    file_name=$1
else
    echo "Файл \"$1\" не найден."
    exit $E_NOFILE
fi

# =====

MAXWIDTH=70        # Максимальная длина строки.

# Удаление символов "^" начиная с первого символа строки,
#+ и ограничить длину строки 70-ю символами.
sed '
s/^>//
s/^ *>//
s/^ *//
s/          *//
' $1 | fold -s --width=$MAXWIDTH
# ключ -s команды "fold" разрывает, если это возможно, строку по
пробельному символу.

# Этот сценарий был написан после прочтения статьи, в котором расхваливалась
#+ утилита под Windows, размером в 164К, с подобной функциональностью.
#
# Хороший набор утилит для обработки текста и эффективный
#+ скриптовый язык -- это все, что необходимо, чтобы составить серьезную конкуренцию
#+ чрезмерно "раздутым" программам.

exit 0
```

Пример А-3. ren: Очень простая утилита для переименования файлов

Этот сценарий является модификацией [Пример 12-15](#).

```
#!/bin/bash
#
# Очень простая утилита для переименования файлов
#
# Утилита "ren", автор Vladimir Lanin (lanin@csd2.nyu.edu),
#+ выполняет эти же действия много лучше.

ARGS=2
```



```

E_BADARGS=65
ONE=1                                # Единственное или множественное число (см. ниже).

if [ $# -ne "$ARGS" ]
then
    echo "Порядок использования: `basename $0` старый_шаблон новый_шаблон"
    # Например: "rn gif jpg", поменяет расширения всех файлов в текущем каталоге с gif
на jpg.
    exit $E_BADARGS
fi

number=0                               # Количество переименованных файлов.

for filename in *$1*                   # Проход по списку файлов в текущем каталоге.
do
    if [ -f "$filename" ]
    then
        fname=`basename $filename`     # Удалить путь к файлу из имени.
        n=`echo $fname | sed -e "s/$1/$2/"` # Поменять старое имя на новое.
        mv $fname $n                   # Переименовать.
        let "number += 1"
    fi
done

if [ "$number" -eq "$ONE" ]           # Соблюдение правил грамматики.
then
    echo "$number файл переименован."
else
    echo "Переименовано файлов: $number."
fi

exit 0

```

```

# Упражнения:
# -----
# С какими типами файлов этот сценарий не будет работать?
# Как это исправить?
#
# Переделайте сценарий таким образом, чтобы он мог обрабатывать все файлы в
каталоге,
#+ в именах которых содержатся пробелы, заменяя пробелы символом подчеркивания.

```

Пример А-4. blank-rename: переименование файлов, чьи имена содержат пробелы

Это даже более простая версия предыдущего примера.

```

#!/bin/bash
# blank-rename.sh
#
# Заменяет пробелы символом подчеркивания в именах файлов в текущем каталоге.

ONE=1                                # единственное или множественное число (см. ниже).
number=0                               # Количество переименованных файлов.
FOUND=0                                 # Код завершения в случае успеха.

for filename in *                       # Перебор всех файлов в текущем каталоге.
do
    echo "$filename" | grep -q " "      # Проверить -- содержит ли имя файла
    if [ $? -eq $FOUND ]               #+ пробелы.
    then
        fname=$filename                # Удалить путь из имени файла.
        n=`echo $fname | sed -e "s/ /_/g"` # Заменить пробелы символом
подчеркивания.
        mv "$fname" "$n"                # Переименование.
        let "number += 1"
    fi
done

```

```

done

if [ "$number" -eq "$ONE" ]
then
  echo "$number файл переименован."
else
  echo "Переименовано файлов: $number"
fi

exit 0

```

Пример А-5. encryptedpw: Передача файла на ftp-сервер, с использованием пароля

```

#!/bin/bash

# Модификация примера "ex72.sh", добавлено шифрование пароля.

# Обратите внимание: этот вариант все еще нельзя считать безопасным,
#+ поскольку в сеть пароль уходит в незашифрованном виде.
# Используйте "ssh", если вас это беспокоит.

E_BADARGS=65

if [ -z "$1" ]
then
  echo "Порядок использования: `basename $0` имя_файла"
  exit $E_BADARGS
fi

Username=bozo          # Измените на свой.
pword=/home/bozo/secret/password_encrypted.file
# Файл, содержащий пароль в зашифрованном виде.

Filename=`basename $1` # Удалить путь из имени файла

Server="XXX"
Directory="YYY"        # Подставьте фактические имя сервера и каталога.

Password=`cruft <$pword` # Расшифровка.
# Используется авторская программа "cruft",
#+ основанная на алгоритме "onetime pad",
#+ ее можно скачать с :
#+ Primary-site: 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
# ключ -n, команды "ftp", запрещает автоматический вход.
# "bell" -- звонок (звуковой сигнал) после передачи каждого файла.

exit 0

```

Пример А-6. copy-cd: Копирование компакт-дисков с данными

```

#!/bin/bash
# copy-cd.sh: copying a data CD

CDROM=/dev/cdrom          # устройство CD ROM
OF=/home/bozo/projects/cdimage.iso # промежуточный файл
# /xxxx/xxxxxxx/        измените для своей системы.

```

```

BLOCKSIZE=2048
SPEED=2                                # Можно задать более высокую скорость,
если поддерживается.

echo; echo "Вставьте исходный CD, но *НЕ* монтируйте его."
echo "Нажмите ENTER, когда будете готовы. "
read ready                               # Ожидание.

echo; echo "Создается промежуточный файл $OF."
echo "Это может занять какое-то время. Пожалуйста подождите."

dd if=$CDROM of=$OF bs=$BLOCKSIZE       # Копирование.

echo; echo "Выньте исходный CD."
echo "Вставьте чистую болванку CDR."
echo "Нажмите ENTER, когда будете готовы. "
read ready                               # Ожидание.

echo "Копируется файл $OF на болванку."

cdrecord -v -isoz speed=$SPEED dev=0,0 $OF
# Используется пакет Joerg Schilling -- "cdrecord" .
# http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html

echo; echo "Копирование завершено."

echo "Желаете удалить промежуточный файл (y/n)? " # Наверняка большой файл
получился.
read answer

case "$answer" in
[yY]) rm -f $OF
      echo "Файл $OF удален."
      ;;
*)    echo "Файл $OF не был удален.";;
esac

echo

# Упражнение:
# Добавьте в оператор "case" возможность обработки, введенных пользователем, "yes" и
"Yes".

exit 0

```

Пример А-7. Последовательности Коллаца (Collatz)

```

#!/bin/bash
# collatz.sh

# Широко известная последовательность Коллаца (Collatz) (гипотеза Коллаца).
# -----
# 1) Принимает из командной строки "начальное" целое число.
# 2) ЧИСЛО <--- НАЧАЛЬНОЕ ЗНАЧЕНИЕ
# 3) Вывести ЧИСЛО.
# 4) Если ЧИСЛО четное, разделить на 2,
# 5)+ Если не четное -- умножить на 3 и прибавить 1.
# 6) ЧИСЛО <--- РЕЗУЛЬТАТ
# 7) Повторить, начиная с п. 3, заданное число раз.
#
# Теоретически, такая последовательность должна сходиться,
#+ не зависимо от величины начального значения,
#+ к повторению циклов "4,2,1...",
#+ даже после значительных флуктуаций в самом начале.

MAX_ITERATIONS=200

```

```

# Для больших начальных значений (>32000), это значение придется увеличить.

h=${1:-$$}          # Начальное значение
                    # если из командной строки ничего не задано, то
берется $PID,

echo
echo "C($h) --- $MAX_ITERATIONS итераций"
echo

for ((i=1; i<=MAX_ITERATIONS; i++))
do

echo -n "$h      "
#      ^^^^^
#      табуляция

    let "remainder = h % 2"
    if [ "$remainder" -eq 0 ] # Четное?
    then
        let "h /= 2"          # Разделить на 2.
    else
        let "h = h*3 + 1"     # Умножить на 3 и прибавить 1.
    fi

COLUMNS=10         # Выводить по 10 значений в строке.
let "line_break = i % $COLUMNS"
if [ "$line_break" -eq 0 ]
then
    echo
fi

done

echo

exit 0

```

Пример A-8. days-between: Подсчет числа дней между двумя датами

```

#!/bin/bash
# days-between.sh: Подсчет числа дней между двумя датами.
# Порядок использования: ./days-between.sh [M]M/[D]D/YYYY [M]M/[D]D/YYYY

ARGS=2              # Ожидается два аргумента из командной строки.
E_PARAM_ERR=65     # Ошибка в числе ожидаемых аргументов.

REFYR=1600         # Начальный год.
CENTURY=100
DIY=365
ADJ_DIY=367       # Корректировка на високосный год + 1.
MIY=12
DIM=31
LEAPCYCLE=4

MAXRETVAL=255     # Максимально возможное возвращаемое значение
                  # для положительных чисел.

diff=              # Количество дней между датами.
value=            # Абсолютное значение.
day=              # день, месяц, год.
month=
year=

Param_Error ()    # Ошибка в параметрах командной строки.
{
    echo "Порядок использования: `basename $0` [M]M/[D]D/YYYY [M]M/[D]D/YYYY"
}

```

```

echo "          (даты должны быть после 1/3/1600)"
exit $E_PARAM_ERR
}

Parse_Date ()          # Разбор даты.
{
    month=${1%/**}
    dm=${1%/**}        # День и месяц.
    day=${dm#*/}
    let "year = `basename $1`" # Хотя это и не имя файла, но результат тот же.
}

check_date ()         # Проверка даты.
{
    [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] || [ "$year" -lt "$REFYR" ] &&
Param_Error
    # Выход из сценария при обнаружении ошибки.
    # Используется комбинация "ИЛИ-списка / И-списка".
    #
    # Упражнение: Реализуйте более строгую проверку даты.
}

strip_leading_zero () # Удалить ведущий ноль
{
    val=${1#0}        # иначе Bash будет считать числа
    return $val       # восьмеричными (POSIX.2, sect 2.9.2.1).
}

day_index ()         # Формула Гаусса:
{                   # Количество дней от 3 Янв. 1600 до заданной даты.

    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"
    # Более подробное объяснение алгоритма вы найдете в
    # http://home.t-online.de/home/berndt.schwerdtfeger/cal.htm

    if [ "$Days" -gt "$MAXRETVL" ] # Если больше 255,
    then                          # то поменять знак
        let "dindex = 0 - $Days"  # чтобы функция смогла вернуть полное значение.
    else let "dindex = $Days"
    fi

    return $dindex
}

calculate_difference () # Разница между двумя датами.
{

```

```

    let "diff = $1 - $2"          # Глобальная переменная.
}

abs ()                          # Абсолютное значение
{                               # Используется глобальная переменная "value".
    if [ "$1" -lt 0 ]          # Если число отрицательное
    then                       # то
        let "value = 0 - $1"   # изменить знак,
    else                       # иначе
        let "value = $1"      # оставить как есть.
    fi
}

if [ $# -ne "$ARGS" ]         # Требуется два аргумента командной строки.
then
    Param_Error
fi

Parse_Date $1
check_date $day $month $year  # Проверка даты.

strip_leading_zero $day      # Удалить ведущие нули
day=$?                       # в номере дня и/или месяца.
strip_leading_zero $month
month=$?

day_index $day $month $year
date1=$?

abs $date1                    # Абсолютное значение
date1=$value

Parse_Date $2
check_date $day $month $year

strip_leading_zero $day
day=$?
strip_leading_zero $month
month=$?

day_index $day $month $year
date2=$?

abs $date2                    # Абсолютное значение
date2=$value

calculate_difference $date1 $date2

abs $diff                    # Абсолютное значение
diff=$value

echo $diff

exit 0
# Сравните этот сценарий с реализацией формулы Гаусса на C
# http://buschencrew.hypermart.net/software/datedif

```

Пример А-9. Создание "словаря"

```

#!/bin/bash
# makedict.sh [создание словаря]

# Модификация сценария /usr/sbin/mkdict.
# Авторские права на оригинальный сценарий принадлежат Alec Muffett.
#
# Этот модифицированный вариант включен в документ на основе

```

```

#+ документа "LICENSE" из пакета "Crack"
#+ с которым распространяется оригинальный сценарий.

# Этот скрипт обрабатывает текстовые файлы и создает отсортированный список
#+ слов, найденных в этих файлах.
# Он может оказаться полезным для сборки словарей
#+ и проведения лексикографического анализа.

E_BADARGS=65

if [ ! -r "$1" ]          # Необходим хотя бы один аргумент --
then                    #+ имя файла.
    echo "Порядок использования: $0 имена_файлов"
    exit $E_BADARGS
fi

# SORT="sort"           # Необходимость задания ключей сортировки
отпала.                #+ Изменено, по отношению к оригинальному
сценарию.

cat $* |                # Выдать содержимое файлов на stdout.
    tr A-Z a-z |        # Преобразовать в нижний регистр.
    tr ' ' '\012' |    # Новое: заменить пробелы символами перевода
строки.                #
#    tr -cd '\012[a-z][0-9]' | # В оригинальном сценарии: удалить все символы,
#                               #+ которые не являются буквами или цифрами.
    tr -c '\012a-z' '\012' | # Вместо удаления
#                               #+ неалфавитно-цифровые символы заменяются на
перевод строки.
    sort |
    uniq |              # Удалить повторяющиеся слова.
    grep -v '^#' |     # Удалить строки, начинающиеся с "#".
    grep -v '^$'       # Удалить пустые строки.

exit 0

```

Пример A-10. Расчет индекса "созвучности"

```

#!/bin/bash
# soundex.sh: Расчет индекса "созвучности"

# =====
#     Сценарий Soundex
#     Автор
#     Mendel Cooper
#     thegrendel@theriver.com
#     23 Января 2002 г.
#
#     Условия распространения: Public Domain.
#
# Несколько отличающаяся версия этого сценария была опубликована
#+ Эдом Шэфером (Ed Schaefer) в Июле 2002 года в колонке "Shell Corner"
#+ "Unix Review" on-line,
#+ http://www.unixreview.com/documents/uni1026336632258/
# =====

ARGCOUNT=1            # Требуется аргумент командной строки.
E_WRONGARGS=70

if [ $# -ne "$ARGCOUNT" ]
then
    echo "Порядок использования: `basename $0` имя"
    exit $E_WRONGARGS
fi

```

```

assign_value ()                                # Присвоить числовые значения
{                                               #+ символам в имени.

    val1=bfpv                                  # 'b,f,p,v' = 1
    val2=cgjkqsxz                             # 'c,g,j,k,q,s,x,z' = 2
    val3=dt                                    # и т.п.
    val4=l
    val5=mn
    val6=r

# Попробуйте разобраться в том, что здесь происходит.

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 )

# Символам в имени присваиваются числовые значения.
# Удаляются повторяющиеся числа, если они не разделены гласными.
# Гласные игнорируются, если они не являются разделителями, которые удаляются в
последнюю очередь.
# Символы 'w' и 'h' удаляются в первую очередь.
}

input_name="$1"
echo
echo "Имя = $input_name"

# Перевести все символы в имени в нижний регистр.
# -----
name=$( echo $input_name | tr A-Z a-z )
# -----

# Начальный символ в индекса "созвучия": первая буква в имени.
# -----

char_pos=0                                    # Начальная позиция в имени.
prefix0=${name:$char_pos:1}
prefix=`echo $prefix0 | tr a-z A-Z`          # Первую букву в имени -- в верхний регистр.

let "char_pos += 1"                           # Передвинуть "указатель" на один символ.
name1=${name:$char_pos}

# ++++++ Искключение отдельных ситуаций ++++++
# Теперь мы передвинулись на один символ вправо.
# Если второй символ в имени совпадает с первым
#+ то его нужно отбросить.
# Кроме того, мы должны проверить -- не является ли первый символ
#+ гласной, 'w' или 'h'.

char1=`echo $prefix | tr A-Z a-z`           # Первый символ -- в нижний регистр.

assign_value $name
s1=$value
assign_value $name1
s2=$value
assign_value $char1
s3=$value
s3=9$s3                                     # Если первый символ в имени -- гласная буква

```



```

#+ или 'w' или 'h',
#+ то ее "значение" нужно отбросить.
#+ Поэтому ставим 9, или другое
#+ неиспользуемое значение, которое можно будет

```

проверить.

```

if [[ "$s1" -ne "$s2" || "$s3" -eq 9 ]]
then
  suffix=$s2
else
  suffix=${s2:$char_pos}
fi
# ++++++ Конец исключения отдельных ситуаций ++++++
+++++

padding=000          # Дополнить тремя нулями.

soun=$prefix$suffix$padding  # Нули добавить в конец получившегося индекса.

MAXLEN=4             # Ограничить длину индекса 4-мя символами.
soundex=${soun:0:$MAXLEN}

echo "Индекс созвучия = $soundex"

echo

# Индекс "созвучия" - это метод индексации и классификации имен
#+ по подобию звучания.
# Индекс "созвучия" начинается с первого символа в имени,
#+ за которым следуют 3-значный расчетный код.
# Имена, которые произносятся примерно одинаково, имеют близкие индексы "созвучия".

# Например:
# Smith и Smythe -- оба имеют индекс "созвучия" "S530".
# Harrison = H625
# Hargison = H622
# Harriman = H655

# Как правило эта методика дает неплохой результат, но имеются и аномалии.
#
#
# Дополнительную информацию вы найдете на
#+ "National Archives and Records Administration home page",
#+ http://www.nara.gov/genealogy/soundex/soundex.html

# Упражнение:
# -----
# Упростите блок "Исключение отдельных ситуаций" .

exit 0

```

Пример А-11. "Игра "Жизнь""

```

#!/bin/bash
# life.sh: Игра "Жизнь"

# ##### #
# Это Bash-версия известной игры Джона Конвея (John Conway) "Жизнь". #
# ----- #
# Прямоугольное игровое поле разбито на ячейки, в каждой ячейке может #
#+ располагаться живая особь. #
# Соответственно, ячейка с живой особью отмечается точкой, #
#+ не занятая ячейка -- остается пустой. #
# Изначально, ячейки заполняются из файла -- #

```

```

#+ это первое поколение, или "поколение 0" #
# Воспроизводство особей, в каждом последующем поколении, #
#+ определяется следующими правилами #
# 1) Каждая ячейка имеет "соседей" #
#+ слева, справа, сверху, снизу и 4 по диагоналям. #
# 123 #
# 4*5 #
# 678 #
# #
# 2) Если живая особь имеет 2 или 3 живых соседей, то она остается жить. #
# 3) Если пустая ячейка имеет 3 живых соседей -- #
#+ в ней "рождается" новая особь #
SURVIVE=2 #
BIRTH=3 #
# 4) В любом другом случае, живая особь "погибает" #
# ##### #

startfile=gen0 # Начальное поколение из файла по-умолчанию -- "gen0".
# если не задан другой файл, из командной строки.
#
if [ -n "$1" ] # Проверить аргумент командной строки -- файл с "поколением 0".
then
  if [ -e "$1" ] # Проверка наличия файла.
  then
    startfile="$1"
  fi
fi

ALIVE1=.
DEAD1=_
# Представление "живых" особей и пустых ячеек в файле с "поколением
0".

# Этот сценарий работает с игровым полем 10 x 10 grid (может быть увеличено,
#+ но большое игровое поле будет обрабатываться очень медленно).
ROWS=10
COLS=10

GENERATIONS=10 # Максимальное число поколений.

NONE_ALIVE=80 # Код завершения на случай,
#+ если не осталось ни одной "живой" особи.

TRUE=0
FALSE=1
ALIVE=0
DEAD=1

avar= # Текущее поколение.
generation=0 # Инициализация счетчика поколений.

# =====

let "cells = $ROWS * $COLS"
# Количество ячеек на игровом поле.

declare -a initial # Массивы ячеек.
declare -a current

display ()
{
  alive=0 # Количество "живых" особей.
# Изначально -- ноль.

  declare -a arr
  arr=( `echo "$1"` ) # Преобразовать аргумент в массив.

```

```

element_count=${#arr[*]}

local i
local rowcheck

for ((i=0; i<$element_count; i++))
do

    # Символ перевода строки -- в конец каждой строки.
    let "rowcheck = $i % ROWS"
    if [ "$rowcheck" -eq 0 ]
    then
        echo                # Перевод строки.
        echo -n "          " # Выравнивание.
    fi

    cell=${arr[i]}

    if [ "$cell" = . ]
    then
        let "alive += 1"
    fi

    echo -n "$cell" | sed -e 's/_/ /g'
    # Вывести массив, по пути заменяя символы подчеркивания на пробелы.
done

return

}

IsValid ()                # Проверка корректности координат ячейки.
{
    if [ -z "$1" -o -z "$2" ]                # Проверка наличия входных аргументов.
    then
        return $FALSE
    fi

    local row
    local lower_limit=0                # Запрет на отрицательные координаты.
    local upper_limit
    local left
    local right

    let "upper_limit = $ROWS * $COLS - 1" # Номер последней ячейки на игровом поле.

    if [ "$1" -lt "$lower_limit" -o "$1" -gt "$upper_limit" ]
    then
        return $FALSE                # Выход за границы массива.
    fi

    row=$2
    let "left = $row * $ROWS"                # Левая граница.
    let "right = $left + $COLS - 1"         # Правая граница.

    if [ "$1" -lt "$left" -o "$1" -gt "$right" ]
    then
        return $FALSE                # Выход за нижнюю строку.
    fi

    return $TRUE                # Координаты корректны.
}

IsAlive ()                # Проверка наличия "живой" особи в ячейке.

```

```

# Принимает массив и номер ячейки в качестве входных
аргументов.
{
  GetCount "$1" $2      # Подсчитать кол-во "живых" соседей.
  local nhbd=$?

  if [ "$nhbd" -eq "$BIRTH" ] # "Живая".
  then
    return $ALIVE
  fi

  if [ "$3" = "." -a "$nhbd" -eq "$SURVIVE" ]
  then
    return $ALIVE      # "Живая" если перед этим была "живая".
  fi

  return $DEAD        # По-умолчанию.
}

GetCount ()          # Подсчет "живых" соседей.
                   # Необходимо 2 аргумента:
                   # $1) переменная-массив
                   # $2) cell 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"    # Номера соседних ячеек.
  let "center = $cell_number - 1"
  let "bottom = $cell_number + $COLS - 1"
  let "r = $cell_number / $ROWS"

  for ((i=0; i<$ROW_NHBD; i++))          # Просмотр слева-направо.
  do
    let "t_top = $top + $i"
    let "t_cen = $center + $i"
    let "t_bot = $bottom + $i"

    let "row = $r"                        # Пройти по соседям в средней строке.
    IsValid $t_cen $row                  # Координаты корректны?
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_cen]} = "$ALIVE1" ] # "Живая"?
      then
        let "count += 1"                # Да!
        # Нарастить счетчик.
      fi
    fi

    let "row = $r - 1"                    # По верхней строке.
    IsValid $t_top $row
    if [ $? -eq "$TRUE" ]
    then

```

```

        if [ ${array[$t_top]} = "$ALIVE1" ]
        then
            let "count += 1"
        fi
    fi

    let "row = $r + 1" # По нижней строке.
    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" # Убедиться, что сама проверяемая ячейка
fi #+ не была подсчитана.

return $count
}

next_gen () # Обновить массив, в котором содержится информация о новом
"поколении".
{
    local array
    local i=0

    array=( `echo "$1" ` ) # Преобразовать в массив.

    while [ "$i" -lt "$cells" ]
    do
        IsActive "$1" $i ${array[$i]} # "Живая"?
        if [ $? -eq "$ALIVE" ]
        then
            array[$i]=. # Если "живая", то
            #+ записать точку.
        else
            array[$i]="_" # Иначе -- символ подчеркивания
            #+ (который позднее заменится на пробел).
        fi
        let "i += 1"
    done

    # let "generation += 1" # Увеличить счетчик поколений.

    # Подготовка переменных, для передачи в функцию "display".
    avar=`echo ${array[@]} ` # Преобразовать массив в строку.
    display "$avar" # Вывести его.
    echo; echo
    echo "Поколение $generation -- живых особей $alive"

    if [ "$alive" -eq 0 ]
    then
        echo
        echo "Преждевременное завершение: не осталось ни одной живой особи!"
        exit $NONE_ALIVE # Нет смысла продолжать
    fi #+ если не осталось ни одной живой особи
}

```



```

#
# Mark Moraes (moraes@csri.toronto.edu), Feb 1, 1989
#

# ==> Эти комментарии добавлены автором документа.

# PATH=/local/bin:/usr/ucb:/usr/bin:/bin
# export PATH
# ==> Первые две строки в оригинальном сценарии вероятно излишни.

TMPFILE=/tmp/ftp.$$
# ==> Создан временный файл

SITE=`domainname`.toronto.edu
# ==> 'domainname' подобен 'hostname'

usage="Порядок использования: $0 [-h удаленный_сервер] [-d удаленный_каталог]... [-f
удаленный_файл:локальный_файл]... \
      [-c локальный_каталог] [-m шаблон_имен_файлов] [-v]"
ftpflags="-i -n"
verbflag=
set -f          # разрешить подстановку имен файлов (globbing) для опции -m
set x `getopt vh:d:c:m:f: $*`
if [ $? != 0 ]; then
    echo $usage
    exit 65
fi
shift
trap 'rm -f ${TMPFILE} ; exit' 0 1 2 3 15
echo "user anonymous ${USER-gnu}@${SITE} > ${TMPFILE}"
# ==> Добавлены кавычки (рекомендуется).
echo binary >> ${TMPFILE}
for i in $*    # ==> Разбор командной строки.
do
    case $i in
        -v) verbflag=-v; echo hash >> ${TMPFILE}; shift;;
        -h) remhost=$2; shift 2;;
        -d) echo cd $2 >> ${TMPFILE};
            if [ x${verbflag} != x ]; then
                echo pwd >> ${TMPFILE};
            fi;
            shift 2;;
        -c) echo lcd $2 >> ${TMPFILE}; shift 2;;
        -m) echo mget "$2" >> ${TMPFILE}; shift 2;;
        -f) f1=`expr "$2" : "\([^:]*\) *.*"`; f2=`expr "$2" : "[^:]*\(.*\)"`;
            echo get ${f1} ${f2} >> ${TMPFILE}; shift 2;;
        --) shift; break;;
    esac
done
if [ $# -ne 0 ]; then
    echo $usage
    exit 65    # ==> В оригинале было "exit 2", изменено в соответствии со
стандартами.
fi
if [ x${verbflag} != x ]; then
    ftpflags="${ftpflags} -v"
fi
if [ x${remhost} = x ]; then
    remhost=prep.ai.mit.edu
    # ==> Здесь можете указать свой ftp-сервер по-умолчанию.
fi
echo quit >> ${TMPFILE}
# ==> Все команды сохранены во временном файле.

ftp ${ftpflags} ${remhost} < ${TMPFILE}
# ==> Теперь обработать пакетный файл.

rm -f ${TMPFILE}

```



```
# ==> В заключение, удалить временный файл (можно скопировать его в системный журнал).
```

```
# ==> Упражнения:
```

```
# ==> -----
```

```
# ==> 1) Добавьте обработку ошибок.
```

```
# ==> 2) Добавьте уведомление звуковым сигналом.
```

Пример A-15. Указание на авторские права

Следующее соглашение об авторских правах относится к двум, включенным в книгу, сценариям от Mark Moraes: "behead.sh" и "ftpget.sh"

```
/*
 * Copyright University of Toronto 1988, 1989.
 * Автор: Mark Moraes
 *
 * Автор дает право на использование этого программного обеспечения
 * его изменение и распространение со следующими ограничениями:
 *
 * 1. Автор и Университет Торонто не отвечают
 * за последствия использования этого программного обеспечения,
 * какими ужасными бы они ни были,
 * даже если они вызваны ошибками в нем.
 *
 * 2. Указание на происхождение программного обеспечения не должно подвергаться
изменениям,
 * явно или по оплошности. Так как некоторые пользователи обращаются к исходным
текстам,
 * они обязательно должны быть включены в документацию.
 *
 * 3. Измененная версия должна содержать явное упоминание об этом и не должна
 * выдаваться за оригинал. Так как некоторые пользователи обращаются к исходным
текстам,
 * они обязательно должны быть включены в документацию.
 *
 * 4. Это соглашение не может удаляться и/или изменяться.
 */
+
```

Antek Sawicki предоставил следующий сценарий, который демонстрирует операцию подстановки параметров, обсуждавшуюся в [Section 9.3](#).

Пример A-16. password: Генератор случайного 8-ми символьного пароля

```
#!/bin/bash
# Для старых систем может потребоваться указать #!/bin/bash2.
#
# Генератор случайных паролей для bash 2.x
# Автор: Antek Sawicki <tenox@tenox.tc>,
# который великодушно позволил использовать его в данном документе.
#
# ==> Комментарии, добавленные автором документа ==>

MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
LENGTH="8"
# ==> 'LENGTH' можно увеличить, для генерации более длинных паролей.

while [ "${n:=1}" -le "$LENGTH" ]
# ==> Напоминаю, что ":= " -- это оператор "подстановки значения по-умолчанию".
# ==> Таким образом, если 'n' не инициализирована, то в нее заносится 1.
do
    PASS="$PASS${MATRIX:$((($RANDOM%${#MATRIX}))):1}"
```

```

# ==> Хитро, хитро....

# ==> Начнем с самых внутренних скобок...
# ==> ${#MATRIX} -- возвращает длину массива MATRIX.

# ==> $RANDOM%${#MATRIX} -- возвращает случайное число
# ==> в диапазоне 1 .. ДЛИНА_МАССИВА(MATRIX) - 1.

# ==> ${MATRIX:$(( $RANDOM%${#MATRIX} )):1}
# ==> возвращает символ из MATRIX, из случайной позиции (найденной выше).
# ==> См. подстановку параметров {var:pos:len} в Разделе 3.3.1
# ==> и примеры в этом разделе.

# ==> PASS=... -- добавление символа к строке PASS, полученной на предыдущих
итерациях.

# ==> Чтобы детальнее проследить ход работы цикла, раскомментируйте следующую
строку
# ==>             echo "$PASS"
# ==> Вы увидите, как на каждом проходе цикла,
# ==> к строке PASS добавляется по одному символу.

let n+=1
# ==> Увеличить 'n' перед началом следующей итерации.
done

echo "$PASS"      # ==> Или перенаправьте в файл, если пожелаете.

exit 0
+

```

James R. Van Zandt предоставил следующий сценарий, который демонстрирует применение именованных каналов, по его словам, "на самом деле -- упражнение на применение кавычек и на экранирование".

Пример А-17. fifo: Создание резервных копий с помощью именованных каналов

```

#!/bin/bash
# ==> Автор:James R. Van Zandt
# ==> используется с его разрешения.

# ==> Комментарии, добавленные автором документа.

HERE=`uname -n`      # ==> hostname
THERE=bilbo
echo "начало создания резервной копии на $THERE, за `date +%r`"
# ==> `date +%r` возвращает время в 12-ти часовом формате, т.е. "08:08:34 PM".

# убедиться в том, что /pipe -- это действительно канал, а не простой файл
rm -rf /pipe
mkfifo /pipe        # ==> Создание "именованного канала", с именем "/pipe".

# ==> 'su xyz' -- запускает команду от имени пользователя "xyz".
# ==> 'ssh' -- вызов secure shell (вход на удаленную систему).
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
# ==> Именованный канал /pipe, используется для передачи данных между процессами:
# ==> 'tar/gzip' пишет в /pipe, а 'ssh' -- читает из /pipe.

# ==> В результате будет получена резервная копия всех основных каталогов.

# ==> В чем состоит преимущество именованного канала, в данной ситуации,
# ==> перед неименованным каналом "|" ?
# ==> Будет ли работать неименованный канал в данной ситуации?

```

```
exit 0
```

+

Stephane Chazelas предоставил следующий сценарий, который демонстрирует возможность генерации простых чисел без использования массивов.

Пример A-18. Генерация простых чисел, с использованием оператора деления по модулю (остаток от деления)

```
#!/bin/bash
# primes.sh: Генерация простых чисел, без использования массивов.
# Автор: Stephane Chazelas.

# Этот сценарий не использует классический алгоритм "Решето Эратосфена",
#+ вместо него используется более понятный метод проверки каждого кандидата в простые
числа
#+ путем поиска делителей, с помощью оператора нахождения остатка от деления "%".

LIMIT=1000                                # Простые от 2 до 1000

Primes()
{
  (( n = $1 + 1 ))                          # Перейти к следующему числу.
  shift                                     # Следующий параметр в списке.
  # echo "_n=$n i=$i_"

  if (( n == LIMIT ))
  then echo $*
  return
  fi

  for i; do
    # echo "-n=$n i=$i-"                    # "i" устанавливается в "@", предыдущее значение $n.
    (( i * i > n )) && break                 # Оптимизация.
    (( n % i )) && continue                 # Отсечь составное число с помощью оператора "%".
    Primes $n $@                          # Рекурсивный вызов внутри цикла.
  done

  Primes $n $@ $n                          # Рекурсивный вызов за пределами цикла.
                                           # Последовательное накопление позиционных параметров.
                                           # в "$@" накапливаются простые числа.
}

Primes 1

exit 0

# Раскомментируйте строки 16 и 24, это поможет понять суть происходящего.

# Сравните скоростные характеристики этого сценария и сценария (ex68.sh),
# реализующего алгоритм "Решето Эратосфена".

# Упражнение: Попробуйте реализовать этот сценарий без использования рекурсии.
# Это даст некоторый выигрыш в скорости.
```

+

Jordi Sanfeliu дал согласие на публикацию своего сценария *tree*.

Пример A-19. tree: Вывод дерева каталогов

```

#!/bin/sh
#      @(#) tree      1.1  30/11/95      by Jordi Sanfeliu
#                                          email: mikaku@fiwix.org
#
#      Начальная версия:  1.0  30/11/95
#      Следующая версия:  1.1  24/02/97  Now, with symbolic links
#      Исправления       :  Ian Kjos, поддержка недоступных каталогов
#                                          email: beth13@mail.utexas.edu
#
#      Tree -- средство просмотра дерева каталогов (очевидно :- )
#
# ==> Используется в данном документе с разрешения автора сценария, Jordi Sanfeliu.
# ==> Комментарии, добавленные автором документа.
# ==> Добавлено "окавычивание" аргументов.

search () {
  for dir in `echo *`
  # ==> `echo *` список всех файлов в текущем каталоге, без символов перевода
  строки.
  # ==> Тот же эффект дает      for dir in *
  # ==> но "dir in `echo *`" не обработает файлы, чьи имена содержат пробелы.
  do
    if [ -d "$dir" ] ; then # ==> Если это каталог (-d)...
      zz=0 # ==> Временная переменная, для сохранения уровня вложенности
каталога.
      while [ $zz != $deep ] # Keep track of inner nested loop.
      do
        echo -n "|  " # ==> Показать символ вертикальной связи,
# ==> с 2 пробелами и без перевода строки.
        zz=`expr $zz + 1` # ==> Нарастить zz.
      done
      if [ -L "$dir" ] ; then # ==> Если символическая ссылка на каталог...
        echo "+---$dir" `ls -l $dir | sed 's/^\.*'$dir' //'`
# ==> Показать горизонтальный соединитель и имя связанного каталога,
но...
# ==> без указания даты/времени.
      else
        echo "+---$dir" # ==> Вывести горизонтальный соединитель...
# ==> и название каталога.
        if cd "$dir" ; then # ==> Если можно войти в каталог...
          deep=`expr $deep + 1` # ==> Нарастить уровень вложенности.
          search # рекурсия ;-)
          numdirs=`expr $numdirs + 1` # ==> Нарастить счетчик каталогов.
        fi
      fi
    done
    cd .. # ==> Подняться на один уровень вверх.
    if [ "$deep" ] ; then # ==> Если depth = 0 (возвращает TRUE)...
      swfi=1 # ==> выставить признак окончания поиска.
    fi
    deep=`expr $deep - 1` # ==> Уменьшить уровень вложенности.
  }

# - Main -
if [ $# = 0 ] ; then
  cd `pwd` # ==> Если аргумент командной строки отсутствует, то используется
текущий каталог.
else
  cd $1 # ==> иначе перейти в заданный каталог.
fi
echo "Начальный каталог = `pwd`"
swfi=0 # ==> Признак завершения поиска.
deep=0 # ==> Уровень вложенности.
numdirs=0
zz=0

```

```

while [ "$swfi" != 1 ] # Пока поиск не закончен...
do
    search # ==> Вызвать функцию поиска.
done
echo "Всего каталогов = $numdirs"

exit 0
# ==> Попробуйте разобраться в том как этот сценарий работает.

```

Noah Friedman дал разрешение на публикацию своей библиотеки *функций для работы со строками*, которая, по сути, воспроизводит некоторые библиотечные функции языка C.

Пример A-20. Функции для работы со строками

```

#!/bin/bash

# string.bash --- эмуляция библиотеки функций string(3)
# Автор: Noah Friedman <friedman@prep.ai.mit.edu>
# ==>      Используется с его разрешения.
# Дата создания: 1992-07-01
# Дата последней модификации: 1993-09-29
# Public domain

# Преобразование в синтаксис bash v2 выполнил Chet Ramey

# Комментарий:
# Код:

#:docstring strcat:
# Порядок использования: strcat s1 s2
#
# Strcat добавляет содержимое переменной s2 к переменной s1.
#
# Пример:
#   a="foo"
#   b="bar"
#   strcat a b
#   echo $a
#   => foobar
#
#:end docstring:

###;;;autoload
function strcat ()
{
    local s1_val s2_val

    s1_val=${!1} # косвенная ссылка
    s2_val=${!2}
    eval "$1"="\${s1_val}${s2_val}"\`
    # ==> eval $1='${s1_val}${s2_val}' во избежание проблем,
    # ==> если одна из переменных содержит одиночную кавычку.
}

#:docstring strncat:
# Порядок использования: strncat s1 s2 $n
#
# Аналог strcat, но добавляет не более n символов из
# переменной s2. Результат выводится 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=${!s1}                # ==> косвенная ссылка
    s2_val=${!s2}

    if [ ${#s2_val} -gt $n ]; then
        s2_val=${s2_val:0:$n}    # ==> выделение подстроки
    fi

    eval "$s1"="\${s1_val}${s2_val}"
    # ==> eval $1='${s1_val}${s2_val}' во избежание проблем,
    # ==> если одна из переменных содержит одиночную кавычку.
}

#:docstring strcmp:
# Порядок использования: strcmp $s1 $s2
#
# Strcmp сравнивает две строки и возвращает число меньше, равно
# или больше нуля, в зависимости от результатов сравнения.
#:end docstring:

###;;;autoload
function strcmp ()
{
    [ "$1" = "$2" ] && return 0

    [ "${1}" '<' "${2}" ] > /dev/null && return -1

    return 1
}

#:docstring strncmp:
# Порядок использования: strncmp $s1 $s2 $n
#
# Подобна strcmp, но сравнивает не более n символов
#: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:
# Порядок использования: strlen s
#
# возвращает количество символов в строке s.
#:end docstring:

```

```

###;;;autoload
function strlen ()
{
    eval echo "\${#${1}}"
    # ==> Возвращает длину переменной,
    # ==> чье имя передается как аргумент.
}

#:docstring strstr:
# Порядок использования: strstr $s1 $s2
#
# Strstr выводит подстроку первого вхождения строки $s2
# в строке $s1, или ничего не выводит, если подстрока $s2 в строке $s1 не найдена.
# Если $s2 содержит строку нулевой длины, то strstr выводит строку $s1.
#:end docstring:

###;;;autoload
function strstr ()
{
    # Сброс содержимого переменной IFS позволяет обрабатывать пробелы как обычные
    # символы.
    local IFS=
    local result="${1%[!${2}]*}"

    echo ${#result}
}

#:docstring strspn:
# Порядок использования: strspn $s1 $s2
#
# Strspn возвращает максимальную длину сегмента в строке $s1,
# который полностью состоит из символов строки $s2.
#:end docstring:

###;;;autoload
function strspn ()
{
    # Сброс содержимого переменной IFS позволяет обрабатывать пробелы как обычные
    # символы.
    local IFS=
    local result="${1%[!${2}]*}"

    echo ${#result}
}

#:docstring strcspn:
# Порядок использования: strcspn $s1 $s2
#
# Strcspn возвращает максимальную длину сегмента в строке $s1,
# который полностью не содержит символы из строки $s2.
#:end docstring:

###;;;autoload
function strcspn ()
{
    # Сброс содержимого переменной IFS позволяет обрабатывать пробелы как обычные
    # символы.
    local IFS=
    local result="${1%[${2}]*}"

    echo ${#result}
}

#:docstring strstr:
# Порядок использования: strstr $s1 $s2
#
# Strstr выводит подстроку первого вхождения строки $s2
# в строке $s1, или ничего не выводит, если подстрока $s2 в строке $s1 не найдена.
# Если $s2 содержит строку нулевой длины, то strstr выводит строку $s1.
#:end docstring:

###;;;autoload
function strstr ()
{
    # Если $s2 -- строка нулевой длины, то вывести строку $s1
    [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }

    # не выводить ничего, если $s2 не найдена в $s1
    case "$1" in
    *$2*) ;;
    *) return 1;;
    esac

    # использовать шаблон, для удаления всех несоответствий после $s2 в $s1
    first=${1/$2*/}

    # Затем удалить все несоответствия с начала строки
    echo "${1##$first}"
}

```

```

}

#:docstring strtok:
# Порядок использования: strtok s1 s2
#
# Strtok рассматривает строку s1, как последовательность из 0, или более,
# лексем (токенов), разделенных символами строки s2
# При первом вызове (с непустым аргументом s1)
# выводит первую лексему на stdout.
# Функция запоминает свое положение в строке s1 от вызова к вызову,
# так что последующие вызовы должны производиться с пустым первым аргументом,
# чтобы продолжить выделение лексем из строки s1.
# После вывода последней лексемы, все последующие вызовы будут выводить на stdout
# пустое значение. Строка-разделитель может изменяться от вызова к вызову.
#:end docstring:

###;;;autoload
function strtok ()
{
:
}

#:docstring strtrunc:
# Порядок использования: strtrunc $n $s1 {$s2} {$...}
#
# Используется многими функциями, такими как strncmp, чтобы отсечь "лишние" символы.
# Выводит первые n символов в каждой из строк s1 s2 ... на stdout.
#:end docstring:

###;;;autoload
function strtrunc ()
{
n=$1 ; shift
for z; do
echo "${z:0:$n}"
done
}

# provide string

# string.bash конец библиотеки

# ===== #
# ==> Все, что находится ниже, добавлено автором документа.

# ==> Чтобы этот сценарий можно было использовать как "библиотеку", необходимо
# ==> удалить все, что находится ниже и "source" этот файл в вашем сценарии.

# strcat
string0=one
string1=two
echo
echo "Проверка функции \"strcat\" :"
echo "Изначально \"string0\" = $string0"
echo "\"string1\" = $string1"
strcat string0 string1
echo "Теперь \"string0\" = $string0"
echo

# strlen
echo
echo "Проверка функции \"strlen\" :"
str=123456789
echo "\"str\" = $str"
echo -n "Длина строки \"str\" = "
strlen str
echo

```



```
# Упражнение:
# -----
# Добавьте проверку остальных функций.
```

```
exit 0
```

Michael Zick предоставил очень сложный пример работы с массивами и утилитой [md5sum](#), используемой для кодирования сведений о каталоге.

От переводчика:

К своему стыду вынужден признаться, что перевод комментариев оказался мне не "по зубам", поэтому оставляю этот сценарий без перевода.

Пример A-21. Directory information

```
#!/bin/bash
# directory-info.sh
# Parses and lists directory information.

# NOTE: Change lines 273 and 353 per "README" file.

# Michael Zick is the author of this script.
# Used here with his permission.

# Controls
# If overridden by command arguments, they must be in the order:
#   Arg1: "Descriptor Directory"
#   Arg2: "Exclude Paths"
#   Arg3: "Exclude Directories"
#
# Environment Settings override Defaults.
# Command arguments override Environment Settings.

# Default location for content addressed file descriptors.
MD5UCFS=${1:-${MD5UCFS:-'/tmpfs/ucfs'}}

# Directory paths never to list or enter
declare -a \
  EXCLUDE_PATHS=${2:-${EXCLUDE_PATHS:-'(/proc /dev /devfs /tmpfs)'}}

# Directories never to list or enter
declare -a \
  EXCLUDE_DIRS=${3:-${EXCLUDE_DIRS:-'(ucfs lost+found tmp wtmp)'}}

# Files never to list or enter
declare -a \
  EXCLUDE_FILES=${3:-${EXCLUDE_FILES:-'(core "Name with Spaces")'}}
```

Here document used as a comment block.

```
: << LSfieldsDoc
# # # # # List Filesystem Directory Information # # # # #
#
#     ListDirectory "FileGlob" "Field-Array-Name"
# or
#     ListDirectory -of "FileGlob" "Field-Array-Filename"
#     '-of' meaning 'output to filename'
# # # # #
```

String format description based on: ls (GNU fileutils) version 4.0.36

Produces a line (or more) formatted:
inode permissions hard-links owner group ...

```
32736 -rw----- 1 mszick mszick
```

```
size    day month date hh:mm:ss year path
2756608 Sun Apr 20 08:53:06 2003 /home/mszick/core
```

Unless it is formatted:

```
inode permissions hard-links owner group ...
266705 crw-rw---- 1 root uucp
```

```
major minor day month date hh:mm:ss year path
4, 68 Sun Apr 20 09:27:33 2003 /dev/ttyS4
NOTE: that pesky comma after the major number
```

NOTE: the 'path' may be multiple fields:

```
/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]
```

If that isn't enough to keep your parser guessing, either or both of the path components may be relative:

```
../Built-Shared -> Built-Static
../linux-2.4.20.tar.bz2 -> ../../../../SRCS/linux-2.4.20.tar.bz2
```

The first character of the 11 (10?) character permissions field:

```
's' Socket
'd' Directory
'b' Block device
'c' Character device
'l' Symbolic link
```

NOTE: Hard links not marked - test for identical inode numbers on identical filesystems.

All information about hard linked files are shared, except for the names and the name's location in the directory system.
NOTE: A "Hard link" is known as a "File Alias" on some systems.
'-' An undistinguished file

Followed by three groups of letters for: User, Group, Others

```
Character 1: '-' Not readable; 'r' Readable
Character 2: '-' Not writable; 'w' Writable
Character 3, User and Group: Combined execute and special
 '-' Not Executable, Not Special
 'x' Executable, Not Special
 's' Executable, Special
 'S' Not Executable, Special
Character 3, Others: Combined execute and sticky (tacky?)
 '-' Not Executable, Not Tacky
 'x' Executable, Not Tacky
 't' Executable, Tacky
 'T' Not Executable, Tacky
```

Followed by an access indicator

Haven't tested this one, it may be the eleventh character or it may generate another field

```
' ' No alternate access
 '+' Alternate access
LSfieldsDoc
```

ListDirectory()

```
{
    local -a T
    local -i of=0          # Default return in variable
#    OLD_IFS=$IFS         # Using BASH default ' \t\n'

    case "$#" in
```

```

3)      case "$1" in
        -of)      of=1 ; shift ;;
        * )      return 1 ;;
        esac ;;
2)      : ;;          # Poor man's "continue"
*)      return 1 ;;
esac

# NOTE: the (ls) command is NOT quoted (")
T=( $(ls --inode --ignore-backups --almost-all --directory \
--full-time --color=none --time=status --sort=none \
--format=long $1) )

case $of in
# Assign T back to the array whose name was passed as $2
0) eval $2=\( \"\${T[@]}\\" \) ;;
# Write T into filename passed as $2
1) echo "${T[@]}" > "$2" ;;
esac
return 0
}

```

```

##### Is that string a legal number? #####
#
#      IsNumber "Var"
##### There has to be a better way, sigh...

```

```

IsNumber()
{
    local -i int
    if [ $# -eq 0 ]
    then
        return 1
    else
        (let int=$1) 2>/dev/null
        return $?      # Exit status of the let thread
    fi
}

```

```

##### Index Filesystem Directory Information #####
#
#      IndexList "Field-Array-Name" "Index-Array-Name"
# or
#      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

```

Walk an array of directory fields produced by ListDirectory

Having suppressed the line breaks in an otherwise line oriented report, build an index to the array element which starts each line.

Each line gets two index entries, the first element of each line (inode) and the element that holds the pathname of the file.

The first index entry pair (Line-Number==0) are informational:
Index-Array-Name[0] : Number of "Lines" indexed
Index-Array-Name[1] : "Current Line" pointer into Index-Array-Name

The following index pairs (if any) hold element indexes into the Field-Array-Name per:

Index-Array-Name[Line-Number * 2] : The "inode" field element.

NOTE: This distance may be either +11 or +12 elements.

Index-Array-Name[(Line-Number * 2) + 1] : The "pathname" element.

NOTE: This distance may be a variable number of elements.

Next line index pair for Line-Number+1.

IndexListDoc

```

IndexList()
{
    local -a LIST                                # Local of listname passed
    local -a -i INDEX=( 0 0 )                  # Local of index to return
    local -i Lidx Lcnt
    local -i if=0 of=0                          # Default to variable names

    case "$#" in
        0) return 1 ;;
        1) return 1 ;;
        2) : ;;                                  # Poor man's 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

    # Make local copy of list
    case "$if" in
        0) eval LIST=\( \"\${LIST[@]}\\" \) ;;
        1) LIST=( $(cat $1) ) ;;
    esac

    # Grok (grope?) the array
    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]}           # Hard Links field
            ft=${LIST[$Lidx+1]:0:1}           # Fast-Stat
            case $ft in
                b) ((Lidx+=12)) ;;             # Block device
                c) ((Lidx+=12)) ;;             # Character device
                *) ((Lidx+=11)) ;;             # Anything else
            esac
            name=Lidx
            case $ft in
                -) ((Lidx+=1)) ;;              # The easy one
                b) ((Lidx+=1)) ;;              # Block device
                c) ((Lidx+=1)) ;;              # Character device
                d) ((Lidx+=1)) ;;              # The other easy one
                l) ((Lidx+=3)) ;;              # At LEAST two more fields
            esac
            # A little more elegance here would handle pipes,
            #+ sockets, deleted files - later.
            *) until IsNumber ${LIST[$Lidx]} || ((Lidx >= Lcnt))
            do
                ((Lidx+=1))
            done
            ;;
            # Not required
        esac
        INDEX[${#INDEX[*]}]=$inode
        INDEX[${#INDEX[*]}]=$name
        INDEX[0]=${INDEX[0]}+1                # One more "line" found
        # echo "Line: ${INDEX[0]} Type: $ft Links: $m Inode: \
        # ${LIST[$inode]} Name: ${LIST[$name]}"

    else
        ((Lidx+=1))
    fi
done
}

```

```

fi
done
case "$of" in
    0) eval $2=\( \"\${INDEX[@]\}\\" \) ;;
    1) echo "${INDEX[@]}" > "$2" ;;
esac
return 0 # What could go wrong?
}

```

```

# # # # # Content Identify File # # # # #
#
# DigestFile Input-Array-Name Digest-Array-Name
# or
# DigestFile -if Input-FileName Digest-Array-Name
# # # # #

```

```

# Here document used as a comment block.
: <<DigestFilesDoc

```

The key (no pun intended) to a Unified Content File System (UCFS) is to distinguish the files in the system based on their content. Distinguishing files by their name is just, so, 20th Century.

The content is distinguished by computing a checksum of that content. This version uses the md5sum program to generate a 128 bit checksum representative of the file's contents. There is a chance that two files having different content might generate the same checksum using md5sum (or any checksum). Should that become a problem, then the use of md5sum can be replaced by a cryptographic signature. But until then...

The md5sum program is documented as outputting three fields (and it does), but when read it appears as two fields (array elements). This is caused by the lack of whitespace between the second and third field. So this function gropes the md5sum output and returns:

```

[0] 32 character checksum in hexadecimal (UCFS filename)
[1] Single character: ' ' text file, '*' binary file
[2] Filesystem (20th Century Style) name
Note: That name may be the character '-' indicating STDIN read.

```

DigestFilesDoc

DigestFile()

```

{
    local if=0 # Default, variable name
    local -a T1 T2

    case "$#" in
        3) case "$1" in
            -if) if=1 ; shift ;;
            * ) return 1 ;;
        esac ;;
        2) : ;; # Poor man's "continue"
        *) return 1 ;;
    esac

    case $if in
        0) eval T1=\( \"\${T1[@]\}\\" \)
            T2=( $(echo ${T1[@]} | md5sum -) )
            ;;
        1) T2=( $(md5sum $1) )
            ;;
    esac

    case ${#T2[@]} in
        0) return 1 ;;
        1) return 1 ;;
    esac
}

```

```

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) : ;; # Assume it worked
*) return 1 ;;
esac

local -i len=${#T2[0]}
if [ $len -ne 32 ] ; then return 1 ; fi
eval $2=\( \"\$T2[@]\)" \)
}

# # # # # Locate File # # # # #
#
#     LocateFile [-l] FileName Location-Array-Name
# or
#     LocateFile [-l] -of FileName Location-Array-FileName
# # # # #

# A file location is Filesystem-id and inode-number

# Here document used as a comment block.
: <<StatFieldsDoc
  Based on stat, version 2.2
  stat -t and stat -lt fields
  [0]     name
  [1]     Total size
          File - number of bytes
          Symbolic link - string length of pathname
  [2]     Number of (512 byte) blocks allocated
  [3]     File type and Access rights (hex)
  [4]     User ID of owner
  [5]     Group ID of owner
  [6]     Device number
  [7]     Inode number
  [8]     Number of hard links
  [9]     Device type (if inode device) Major
  [10]    Device type (if inode device) Minor
  [11]    Time of last access
          May be disabled in 'mount' with noatime
          atime of files changed by exec, read, pipe, utime, mknod (mmap?)
          atime of directories changed by addition/deletion of files
  [12]    Time of last modification
          mtime of files changed by write, truncate, utime, mknod
          mtime of directories changed by addition/deletion of files
  [13]    Time of last change
          ctime reflects time of changed inode information (owner, group
          permissions, link count

- *- *- Per:
  Return code: 0
  Size of array: 14
  Contents of array
  Element 0: /home/mszick
  Element 1: 4096
  Element 2: 8
  Element 3: 41e8
  Element 4: 500
  Element 5: 500
  Element 6: 303
  Element 7: 32385
  Element 8: 22
  Element 9: 0
  Element 10: 0

```

```
Element 11: 1051221030
Element 12: 1051214068
Element 13: 1051214068
```

```
For a link in the form of linkname -> realname
stat -t linkname returns the linkname (link) information
stat -lt linkname returns the realname information
```

```
stat -tf and stat -ltf fields
```

```
[0]      name
[1]      ID-0?          # Maybe someday, but Linux stat structure
[2]      ID-0?          # does not have either LABEL nor UUID
                                # fields, currently information must come
                                # from file-system specific utilities
```

```
These will be munged into:
```

```
[1]      UUID if possible
[2]      Volume Label if possible
```

```
Note: 'mount -l' does return the label and could return the UUID
```

```
[3]      Maximum length of filenames
[4]      Filesystem type
[5]      Total blocks in the filesystem
[6]      Free blocks
[7]      Free blocks for non-root user(s)
[8]      Block size of the filesystem
[9]      Total inodes
[10]     Free inodes
```

```
-*-*- Per:
```

```
Return code: 0
Size of array: 11
Contents of array
Element 0: /home/mszick
Element 1: 0
Element 2: 0
Element 3: 255
Element 4: ef53
Element 5: 2581445
Element 6: 2277180
Element 7: 2146050
Element 8: 4096
Element 9: 1311552
Element 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
}
```

```

# More Sanscrit-2.0.5
# LOC1=( $(stat -t $lk $1) )
# LOC2=( $(stat -tf $lk $1) )
# Uncomment above two lines if system has "stat" command installed.
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
# Which yields (if you are lucky, and have "stat" installed)
# -*-*- Location Descriptor -*-*-
# Return code: 0
# Size of array: 15
# Contents of array
# Element 0: /home/mszick          20th Century name
# Element 1: 41e8                  Type and Permissions
# Element 2: 500                    User
# Element 3: 500                    Group
# Element 4: 303                    Device
# Element 5: 32385                  inode
# Element 6: 22                     Link count
# Element 7: 0                      Device Major
# Element 8: 0                      Device Minor
# Element 9: 1051224608             Last Access
# Element 10: 1051214068            Last Modify
# Element 11: 1051214068            Last Status
# Element 12: 0                     UUID (to be)
# Element 13: 0                     Volume Label (to be)
# Element 14: ef53                  Filesystem type
}

```

And then there was some test code

```

ListArray() # ListArray Name
{
    local -a Ta

    eval Ta=\( \"\${1[@]}\\" \)
    echo
    echo "-*-*- List of Array -*-*- "
    echo "Size of array $1: ${#Ta[*]}"
    echo "Contents of array $1:"
    for (( i=0 ; i<${#Ta[*]} ; i++ ))
    do
        echo -e "\tElement $i: ${Ta[$i]}"
    done
    return 0
}

```

```

declare -a CUR_DIR
# For small arrays
ListDirectory "${PWD}" CUR_DIR
ListArray CUR_DIR

```

```

declare -a DIR_DIG
DigestFile CUR_DIR DIR_DIG
echo "The new \"name\" (checksum) for ${CUR_DIR[9]} is ${DIR_DIG[0]}"

```

```

declare -a DIR_ENT
# BIG_DIR # For really big arrays - use a temporary file in 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
# Small (should) be able to parallize IndexList & DigestFile
# Large (should) be able to parallize IndexList & DigestFile & the assignment
echo "The \"name\" (checksum) for the contents of ${PWD} is ${IDX_DIG[0]}"

declare -a FILE_LOC
LocateFile ${PWD} FILE_LOC
ListArray FILE_LOC

exit 0

```

Stephane Chazelas демонстрирует возможность объектно ориентированного подхода к программированию в Bash-сценариях.

Пример А-22. Объектно ориентированная база данных

```

#!/bin/bash
# obj-oriented.sh: Объектно ориентированный подход к программированию в сценариях.
# Автор: Stephane Chazelas.

```

```

person.new()          # Очень похоже на объявление класса в C++.
{
    local obj_name=$1 name=$2 firstname=$3 birthdate=$4

    eval "$obj_name.set_name() {
        eval \"\$obj_name.get_name() {
            echo \$1
        }\"
    }"

    eval "$obj_name.set_firstname() {
        eval \"\$obj_name.get_firstname() {
            echo \$1
        }\"
    }"

    eval "$obj_name.set_birthdate() {
        eval \"\$obj_name.get_birthdate() {
            echo \$1
        }\"
        eval \"\$obj_name.show_birthdate() {
            echo \"\$(date -d \"1/1/1970 0:0:\$1 GMT\""
        }\"
        eval \"\$obj_name.get_age() {
            echo \"\$( ( (\$(date +%s) - \$1) / 3600 / 24 / 365 ) )\"
        }\"
    }"

    $obj_name.set_name $name
    $obj_name.set_firstname $firstname
    $obj_name.set_birthdate $birthdate
}

echo

person.new self Bozeman Bozo 101272413
# Создается экземпляр класса "person.new" (фактически -- вызов функции с
аргументами).

self.get_firstname          #      Bozo

```

```
self.get_name      #   Bozeman
self.get_age       #   28
self.get_birthdate #   101272413
self.show_birthdate #  Sat Mar 17 20:13:33 MST 1973

echo

# typeset -f
# чтобы просмотреть перечень созданных функций.

exit 0
```

Приложение В. Маленький учебник по Sed и Awk

В этом приложении содержится очень краткое описание приемов работы с утилитами обработки текста **sed** и **awk**. Здесь будут рассмотрены лишь несколько базовых команд, которых, в принципе, будет достаточно, чтобы научиться понимать простейшие конструкции **sed** и **awk** внутри сценариев на языке командной оболочки.

sed: неинтерактивный редактор текстовых файлов

awk: язык обработки шаблонов с C-подобным синтаксисом

При всех своих различиях, эти две утилиты обладают похожим синтаксисом, они обе умеют работать с [регулярными выражениями](#), обе, по-умолчанию, читают данные с устройства `stdin` и обе выводят результат обработки на устройство `stdout`. Обе являются утилитами UNIX-систем, и прекрасно могут взаимодействовать между собой. Вывод от одной может быть перенаправлен, по конвейеру, на вход другой. Их комбинирование придает сценариям, на языке командной оболочки, мощь и гибкость языка Perl.



Одно важное отличие состоит в том, что в случае с **sed**, сценарий легко может передавать дополнительные аргументы этой утилите, в то время, как в случае с **awk** (см. [Пример 33-3](#) и [Пример 9-22](#)), это более сложная задача .

В.1. Sed

Sed -- это неинтерактивный строчный редактор. Он принимает текст либо с устройства `stdin`, либо из текстового файла, выполняет некоторые операции над строками и затем выводит результат на устройство `stdout` или в файл. Как правило, в сценариях, **sed** используется в конвейерной обработке данных, совместно с другими командами и утилитами.

Sed определяет, по заданному *адресному пространству*, над какими строками следует выполнить операции. [\[66\]](#) Адресное пространство строк задается либо их порядковыми номерами, либо шаблоном. Например, команда `3d` заставит **sed** удалить третью строку, а команда `/windows/d` означает, что все строки, содержащие "windows", должны быть удалены.

Из всего разнообразия операций, мы остановимся на трех, используемых наиболее

часто. Это **p** -- печать (на stdout), **d** -- удаление и **s** -- замена.

Таблица В-1. Основные операции sed

Операция	Название	Описание
[диапазон строк] /p	print	Печать [указанного диапазона строк]
[диапазон строк] /d	delete	Удалить [указанный диапазон строк]
s/pattern1/pattern2/	substitute	Заменить первое встреченное соответствие шаблону pattern1, в строке, на pattern2
[диапазон строк] /s/pattern1/pattern2/	substitute	Заменить первое встреченное соответствие шаблону pattern1, на pattern2, в указанном <i>диапазоне строк</i>
[диапазон строк] /y/pattern1/pattern2/	transform	заменить любые символы из шаблона pattern1 на соответствующие символы из pattern2, в указанном <i>диапазоне строк</i> (эквивалент команды tr)
g	global	Операция выполняется над <i>всеми</i> найденными соответствиями внутри каждой из заданных строк



Без оператора **g** (*global*), операция замены будет производиться только для первого найденного совпадения, с заданным шаблоном, в каждой строке.

В отдельных случаях, операции sed необходимо заключать в кавычки.

```
sed -e '/^$/d' $filename
# Ключ -e говорит о том, что далее следует строка, которая должна интерпретироваться
#+ как набор инструкций редактирования.
# (При передаче одной инструкции, ключ "-e" является необязательным.)
# "Строгие" кавычки (') предотвращают интерпретацию символов регулярного выражения,
#+ как специальных символов, командным интерпретатором.
#
# Действия производятся над строками, содержащимися в файле $filename.
```

В отдельных случаях, команды редактирования не работают в одиночных кавычках.

```
filename=file1.txt
pattern=BEGIN

sed "/^$pattern/d" "$filename" # Результат вполне предсказуем.
# sed '/^$pattern/d' "$filename" дает иной результат.
# В данном случае, в "строгих" кавычках (' ... '),
#+ не происходит подстановки значения переменной "$pattern".
```



Sed использует ключ **-e** для того, чтобы определить, что следующая строка является инструкцией, или набором инструкций, редактирования. Если инструкция является единственной, то использование этого ключа не является обязательным.

```
sed -n '/xzy/p' $filename
# Ключ -n заставляет sed вывести только те строки, которые совпадают с указанным
шаблоном.
# В противном случае (без ключа -n), будут выведены все строки.
# Здесь, ключ -e не является обязательным, поскольку здесь стоит единственная
команда.
```

Таблица В-2. Примеры операций в sed

Операция	Описание
8d	Удалить 8-ю строку.
/^\$/d	Удалить все пустые строки.
1,/^\$/d	Удалить все строки до первой пустой строки, включительно.
/Jones/p	Вывести строки, содержащие "Jones" (с ключом -n).
s/Windows/Linux/	В каждой строке, заменить первое встретившееся слово "Windows" на слово "Linux".
s/BSOD/stability/g	В каждой строке, заменить все встретившиеся слова "BSOD" на "stability".
s/ *\$//	Удалить все пробелы в конце каждой строки.
s/00*/0/g	Заменить все последовательности ведущих нулей одним символом "0".
/GUI/d	Удалить все строки, содержащие "GUI".
s/GUI//g	Удалить все найденные "GUI", оставляя остальную часть строки без изменений.

Замена строки пустой строкой, эквивалентна удалению части строки, совпадающей с шаблоном. Остальная часть строки остается без изменений. Например, `s/GUI//`, изменит следующую строку

`The most important parts of any application are its GUI and sound effects`

на

`The most important parts of any application are its and sound effects`

Символ обратного слэша представляет символ перевода строки, как символ замены. В этом случае, замещающее выражение продолжается на следующей строке.

```
s/^ */\n/g
```

Эта инструкция заменит начальные пробелы в строке на символ перевода строки. Ожидаемый результат -- замена отступов в начале параграфа пустыми строками.

Указание диапазона строк, предшествующее одной, или более, инструкции может потребовать заключения инструкций в фигурные скобки, с соответствующими символами перевода строки.

```
/[0-9A-Za-z]/,/^$/{
/^$/d
}
```

В этом случае будут удалены только первые из нескольких, идущих подряд, пустых строк. Это может использоваться для установки однострочных интервалов в файле, оставляя, при этом, пустые строки между параграфами.

 Быстрый способ установки двойных межстрочных интервалов в текстовых файлах -- `sed G filename`.

Примеры использования sed в сценариях командной оболочки, вы найдете в:

1. [Пример 33-1](#)
2. [Пример 33-2](#)
3. [Пример 12-2](#)
4. [Пример A-3](#)
5. [Пример 12-12](#)
6. [Пример 12-20](#)
7. [Пример A-13](#)
8. [Пример A-19](#)
9. [Пример 12-24](#)
10. [Пример 10-9](#)
11. [Пример 12-33](#)
12. [Пример A-2](#)
13. [Пример 12-10](#)
14. [Пример 12-8](#)
15. [Пример A-11](#)
16. [Пример 17-11](#)

Ссылки на дополнительные сведения о sed, вы найдете в разделе [Литература](#).

B.2. Awk

Awk -- это полноценный язык обработки текстовой информации с синтаксисом, напоминающим синтаксис языка C. Он обладает довольно широким набором возможностей, однако, мы рассмотрим лишь некоторые из них -- наиболее употребимые в сценариях командной оболочки.

Awk "разбивает" каждую строку на отдельные *поля*. По-умолчанию, поля -- это последовательности символов, отделенные друг от друга [пробелами](#), однако имеется возможность назначения других символов, в качестве разделителя полей. Awk анализирует и обрабатывает каждое поле в отдельности. Это делает его идеальным инструментом для работы со структурированными текстовыми файлами, особенно с таблицами.

Внутри сценариев командной оболочки, код awk, заключается в "строгие" (одиночные)

кавычки и фигурные скобки.

```
awk '{print $3}' $filename  
# Выводит содержимое 3-го поля из файла $filename на устройство stdout.
```

```
awk '{print $1 $5 $6}' $filename  
# Выводит содержимое 1-го, 5-го и 6-го полей из файла $filename.
```

Только что, мы рассмотрели действие команды **print**. Еще, на чем мы остановимся -- это переменные. Awk работает с переменными подобно сценариям командной оболочки, но более гибко.

```
{ total += ${column_number} }
```

Эта команда добавит содержимое переменной *column_number* к переменной "total". Чтобы, в завершение вывести "total", можно использовать команду **END**, которая открывает блок кода, обрабатывающий после того, как будут обработаны все входные данные.

```
END { print total }
```

Команде **END**, соответствует команда **BEGIN**, которая открывает блок кода, обрабатывающий перед началом обработки входных данных.

Примеры использования awk в сценариях командной оболочки, вы найдете в:

1. [Пример 11-10](#)
2. [Пример 16-7](#)
3. [Пример 12-24](#)
4. [Пример 33-3](#)
5. [Пример 9-22](#)
6. [Пример 11-16](#)
7. [Пример 27-1](#)
8. [Пример 27-2](#)
9. [Пример 10-3](#)
10. [Пример 12-42](#)
11. [Пример 9-26](#)
12. [Пример 12-3](#)
13. [Пример 9-12](#)
14. [Пример 33-11](#)

Это все, что я хотел рассказать об awk. Дополнительные ссылки на информацию об awk, вы найдете в разделе [Литература](#).

Приложение С. Коды завершения, имеющие predeterminedный смысл

Таблица С-1. "Зарезервированные" коды завершения

Код завершения	Смысл	Пример	Примечание
1	разнообразные ошибки	let "var1 = 1/0"	различные ошибки, такие как "деление на ноль" и пр.
2	согласно документации к Bash -- неверное использование встроенных команд		Встречаются довольно редко, обычно код завершения возвращается равным 1
126	вызываемая команда не может быть выполнена		возникает из-за проблем с правами доступа или когда вызван на исполнение неисполняемый файл
127	"команда не найдена"		Проблема связана либо с переменной окружения \$PATH, либо с неверным написанием имени команды
128	неверный аргумент команды exit	exit 3.14159	команда exit может принимать только целочисленные значения, в диапазоне 0 - 255
128+n	фатальная ошибка по сигналу "n"	kill -9 \$PPID сценария	\$? вернет 137 (128 + 9)
130	завершение по Control-C		Control-C -- это выход по сигналу 2, (130 = 128 + 2, см. выше)
255*	код завершения вне допустимого диапазона	exit -1	exit может принимать только целочисленные значения, в диапазоне 0 - 255

Согласно этой таблице, коды завершения 1 - 2, 126 - 165 и 255 [\[67\]](#) имеют predeterminedное значение, поэтому вам следует избегать употребления этих кодов для своих нужд. Завершение сценария с кодом возврата **exit 127**, может привести в замешательство при поиске ошибок в сценарии (действительно ли он означает ошибку "команда не найдена"? Или это предусмотренный программистом код завершения?). В большинстве случаев, программисты вставляют **exit 1**, в качестве реакции на ошибку. Так как код завершения 1 подразумевает целый "букет" ошибок, то в данном случае трудно говорить о какой либо двусмысленности, хотя и об информативности -- тоже.

Не раз предпринимались попытки систематизировать коды завершения (см. `/usr/include/sysexits.h`), но эта систематизация предназначена для программистов, пишущих на языках C и C++. Автор документа предлагает ограничить коды завершения, определяемые пользователем, диапазоном 64 - 113 (и, само собой разумеется -- 0, для обозначения успешного завершения), в соответствии со стандартом C/C++. Это сделало бы поиск ошибок более простым.

Все сценарии, прилагаемые к данному документу, приведены в соответствие с этим стандартом, за исключением случаев, когда существуют отменяющие обстоятельства, например в [Пример 9-2](#).

- ☞ Обращение к переменной `$?`, из командной строки, после завершения работы сценария, дает результат, в соответствии с таблицей, приведенной выше, но только для Bash или *sh*. Под управлением *csh* или *tcsh* значения могут в некоторых случаях отличаться.

Приложение D. Подробное введение в операции ввода-вывода и перенаправление ввода-вывода

написано Stephane Chazelas и дополнено автором документа

Практически любая команда предполагает доступность 3-х [файловых дескрипторов](#). Первый -- 0 (стандартный ввод, `stdin`), доступный для чтения. И два других -- 1 (`stdout`) и 2 (`stderr`), доступные для записи.

Запись, типа `ls 2>&1`, означает временное перенаправление вывода, с устройства `stderr` на устройство `stdout`.

В соответствии с соглашениями, команды принимают ввод из файла с дескриптором 0 (`stdin`), выводят результат работы в файл с дескриптором 1 (`stdout`), а сообщения об ошибках -- в файл с дескриптором 2 (`stderr`). Если какой либо из этих трех дескрипторов окажется закрытым, то могут возникнуть определенные проблемы:

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

К примеру, когда пользователь запускает **xterm**, то он сначала выполняет процедуру инициализации, а затем, перед запуском командной оболочки, **xterm** трижды открывает терминальные устройства (`/dev/pts/<n>`, или нечто подобное).

После этого, командная оболочка наследует эти три дескриптора, и любая команда, запускаемая в этой оболочке, так же наследует их. Термин [перенаправление](#) -- означает переназначение одного файлового дескриптора на другой (канал (конвейер) или что-то другое). Переназначение может быть выполнено локально (для отдельной команды, для группы команд, для подоболочки, для операторов [while, if, case, for...](#)) или глобально (с помощью [exec](#)).

`ls > /dev/null` -- означает запуск команды `ls` с файловым дескриптором 1, присоединенным к устройству `/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
```

Упражнение : Проанализируйте следующий сценарий.

```
#!/usr/bin/env bash

mkfifo /tmp/fifo1 /tmp/fifo2
while read a; do echo "FIF01: $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 "FIF02: $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, to stdout
      sleep 1
      echo 2nd, to stderr >&2
      sleep 1
      echo 3rd, to fd 3 >&3
      sleep 1
      echo 4th, to fd 4 >&4
      sleep 1
      echo 5th, to fd 5 >&5
      sleep 1
      echo 6th, through a pipe | sed 's/./PIPE: &, to fd 5/' >&5
      sleep 1
      echo 7th, to fd 6 >&6
      sleep 1
      echo 8th, to fd 7 >&7
      sleep 1
      echo 9th, to 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
```

```
# Выясните, куда переназначены файловые дескрипторы каждой команды и подоболочки.
```

```
exit 0
```

Приложение Е. Локализация

Возможность локализации сценариев Bash нигде в документации не описана.

Локализованные сценарии выводят текст на том языке, который используется системой, в соответствии с настройками. Пользователь Linux, живущий в Берлине (Германия), будет видеть сообщения на немецком языке, в то время как другой пользователь, проживающий в Берлине штата Мэриленд (США) -- на английском.

Для создания локализованных сценариев можно использовать следующий шаблон, предусматривающий вывод всех сообщений на языке пользователя (сообщения об ошибках, приглашения к вводу и т.п.).

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

E_CDERROR=65

error()
{
    printf "$@" >&2
    exit $E_CDERROR
}

cd $var || error $"Can't cd to %s." "$var"
read -p $"Enter the value: " var
# ...
```

```
bash$ bash -D localized.sh
"Can't cd to %s."
"Enter the value: "
```

Это список всех текстовых сообщений, которые подлежат локализации. (Ключ -D выводит список строк в двойных кавычках, которым предшествует символ \$, без запуска сценария на исполнение.)

```
bash$ bash --dump-po-strings localized.sh
#: a:6
msgid "Can't cd to %s."
msgstr ""
#: a:7
msgid "Enter the value: "
msgstr ""
```

Ключ `--dump-po-strings` в Bash напоминает ключ `-D`, но выводит строки в формате "po", с помощью утилиты [gettext](#).

Теперь построим файл `language.po`, для каждого языка, на которые предполагается перевести сообщения сценария. Например:

Файл `ru.po` сделан переводчиком, в оригинальном документе локализация выполнена на примере французского языка

`ru.po`:

```
#: a:6
msgid "Can't cd to %s."
msgstr "Невозможно перейти в каталог %s."
#: a:7
msgid "Enter the value: "
msgstr "Введите число: "
```

Затем запустите **msgfmt**.

```
msgfmt -o localized.sh.mo ru.po
```

Перепишите получившийся файл `localized.sh.mo` в каталог `/usr/share/locale/ru/LC_MESSAGES` и добавьте в начало сценария строки:

```
TEXTDOMAINDIR=/usr/share/locale
TEXTDOMAIN=localized.sh
```

Если система корректно настроена на русскую локаль, то пользователь, запустивший сценарий, будет видеть сообщения на русском языке.



В старых версиях Bash или в других командных оболочках, потребуется воспользоваться услугами утилиты [gettext](#), с ключом `-s`. В этом случае наш сценарий будет выглядеть так:

```
#!/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
# ...
```

А переменные `TEXTDOMAIN` и `TEXTDOMAINDIR`, необходимо будет экспортировать в окружение.

Автор этого приложения: Stephane Chazelas.

Приложение F. История команд

Командная оболочка Bash предоставляет в распоряжение пользователя инструментарий командной строки, позволяющий управлять *историей команд*. История команд -- это, прежде всего, очень удобный инструмент, сокращающий ручной ввод.

История команд Bash:

1. **history**
2. **fc**

```
bash$ history
1  mount /mnt/cdrom
2  cd /mnt/cdrom
3  ls
   ...
```

Внутренние переменные Bash, связанные с историей команд:

1. \$HISTCMD
2. \$HISTCONTROL
3. \$HISTIGNORE
4. \$HISTFILE
5. \$HISTFILESIZE
6. \$HISTSIZE
7. !!
8. !\$
9. !#
10. !N
11. !-N
12. !STRING
13. !?STRING?
14. ^STRING^string^

К сожалению, инструменты истории команд, в Bash, совершенно бесполезны в

сценариях.

```
#!/bin/bash
# history.sh
# Попытка воспользоваться 'историей' команд в сценарии.
```

```
history
```

```
# На экран ничего не выводится.
# История команд не работает в сценариях.
```

```
bash$ ./history.sh
(ничего не выводится)
```

Приложение G. Пример файла `.bashrc`

Файл `~/ .bashrc` определяет поведение командной оболочки. Внимательное изучение этого примера поможет вам значительно продвинуться в понимании Bash.

[Emmanuel Rouat](#) представил следующий, очень сложный, файл `.bashrc`, написанный для операционной системы Linux. Предложения и замечания приветствуются.

Внимательно изучите этот файл. Отдельные участки этого файла вы свободно можете использовать в своем собственном `.bashrc` или, даже в своих сценариях!

Пример G-1. Пример файла `.bashrc`

```
#=====
#
# ЛИЧНЫЙ ФАЙЛ $HOME/.bashrc для bash-2.05a (или выше)
#
# Время последней модификации: Втр Апр 15 20:32:34 CEST 2003
#
# Этот файл содержит настройки интерактивной командной оболочки.
# Здесь размещены определения псевдонимов, функций
# и других элементов Bash, таких как prompt (приглашение к вводу).
#
# Изначально, этот файл был создан в операционной системе Solaris,
# но позднее был переделан под Redhat
# --> Модифицирован под Linux.
# Большая часть кода, который находится здесь, была взята из
# Usenet (или Интернет).
# Этот файл содержит слишком много определений -- помните, это всего лишь пример.
#
#
#-----
# --> Комментарии, добавленные автором HOWTO.
# --> И дополнены автором сценария Emmanuel Rouat :-)
#-----
# Глобальные определения
#-----

if [ -f /etc/bashrc ]; then
```

```

. /etc/bashrc # --> Прочитать настройки из /etc/bashrc, если таковой
имеется.
fi

#-----
# Настройка переменной $DISPLAY (если еще не установлена)
# Это срабатывает под linux - в вашем случае все может быть по другому....
# Проблема в том, что различные типы терминалов
# дают разные ответы на запрос 'who am i'.....
# я не нашел 'универсального' метода
#-----

function get_xserver ()
{
    case $TERM in
        xterm )
            XSERVER=$(who am i | awk '{print $NF}' | tr -d ' ' '(' )
            XSERVER=${XSERVER%:*}
            ;;
        aterm | rxvt)
            # добавьте здесь свой код.....
            ;;
    esac
}

if [ -z ${DISPLAY:=} ]; then
    get_xserver
    if [[ -z ${XSERVER} || ${XSERVER} == $(hostname) || ${XSERVER} == "unix" ]];
then
        DISPLAY=":0.0" # для локального хоста
    else
        DISPLAY=${XSERVER}:0.0 # для удаленного хоста
    fi
fi

export DISPLAY

#-----
# Некоторые настройки
#-----

ulimit -S -c 0 # Запрет на создание файлов coredump
set -o notify
set -o noclobber
set -o ignoreeof
set -o nounset
#set -o xtrace # полезно для отладки

# Разрешающие настройки:
shopt -s cdspell
shopt -s cdable_vars
shopt -s checkhash
shopt -s checkwinsize
shopt -s mailwarn
shopt -s sourcepath
shopt -s no_empty_cmd_completion # только для bash>=2.04
shopt -s cmdhist
shopt -s histappend histreedit histverify
shopt -s extglob

# Запрещающие настройки:
shopt -u mailwarn
unset MAILCHECK # Я не желаю, чтобы командная оболочка сообщала мне о
прибытии почты

export TIMEFORMAT='\nreal %3R\tuser %3U\tsys %3S\tcpu %P\n'
export HISTIGNORE="&:bg:fg:ll:h"
export HOSTFILE=$HOME/.hosts # Поместить список удаленных хостов в файл ~/.hosts

```

```

#-----
# Greeting, motd etc...
#-----

# Для начала определить некоторые цвета:
red='\e[0;31m'
RED='\e[1;31m'
blue='\e[0;34m'
BLUE='\e[1;34m'
cyan='\e[0;36m'
CYAN='\e[1;36m'
NC='\e[0m'          # No Color (нет цвета)
# --> Прекрасно. Имеет тот же эффект, что и "ansi.sys" в DOS.

# Лучше выглядит на черном фоне....
echo -e "${CYAN}This is BASH ${RED}${BASH_VERSION%.*}${CYAN} - DISPLAY on ${RED}
${DISPLAY}${NC}\n"
date
if [ -x /usr/games/fortune ]; then
    /usr/games/fortune -s      # сделает наш день более интересным.... :- )
fi

function _exit()          # функция, запускающаяся при выходе из оболочки
{
    echo -e "${RED}Аста ла виста, бэби ${NC}"
}
trap _exit EXIT

#-----
# Prompt
#-----

if [[ "${DISPLAY}${HOST}" != ":0.0" && "${DISPLAY}" != ":0" ]]; then
    HILIT=${red}    # на удаленной системе: prompt будет частично красным
else
    HILIT=${cyan}  # на локальной системе: prompt будет частично циановым
fi

# --> Замените \W на \w в функциях ниже
#+ --> чтобы видеть в оболочке полный путь к текущему каталогу.

function fastprompt()
{
    unset PROMPT_COMMAND
    case $TERM in
        *term | rxvt )
            PS1="${HILIT}[\h]$NC \W > \[\033]0;\${TERM} [\u@\h] \w\007\" ;;
        linux )
            PS1="${HILIT}[\h]$NC \W > " ;;
        *)
            PS1="[\h] \W > " ;;
    esac
}

function powerprompt()
{
    _powerprompt()
    {
        LOAD=$(uptime|sed -e "s/.*: \([^,]*\)*/\1/" -e "s/ //g")
    }

    PROMPT_COMMAND=_powerprompt
    case $TERM in
        *term | rxvt )
            PS1="${HILIT}[\A \${LOAD}]$NC\n[\h \#] \W > \[\033]0;\${TERM} [\u@\h]
\w\007\" ;;

```

```

linux )
    PS1="${HILIT}[\A - \${LOAD}]$NC\n[\h \#] \w > " ;;
* )
    PS1="[\A - \${LOAD}]\n[\h \#] \w > " ;;
esac
}

powerprompt      # это prompt по-умолчанию - может работать довольно медленно
                  # Если это так, то используйте fastprompt....

#=====
#
# ПСЕВДОНИМЫ И ФУНКЦИИ
#
# Возможно некоторые из функций, приведенных здесь, окажутся для вас слишком
# большими,
# но на моей рабочей станции установлено 512Mb ОЗУ, так что.....
# Если пожелаете уменьшить размер этого файла, то можете оформить эти функции
# в виде отдельных сценариев.
#
# Большинство функций были взяты, почти без переделки, из примеров
# к bash-2.04.
#
#=====

#-----
# Псевдонимы
#-----

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 r='rlogin'
alias which='type -all'
alias ..='cd ..'
alias path='echo -e ${PATH//:/\\n}'
alias print='/usr/bin/lp -o nobanner -d $LPDEST' # Предполагается, что LPDEST
определен
alias pjet='enscript -h -G -fCourier9 -d $LPDEST' # Печать через enscript
alias background='xv -root -quit -max -rmode 5' # Положить картинку в качестве
фона
alias du='du -kh'
alias df='df -kTh'

# Различные варианты 'ls' (предполагается, что установлена GNU-версия ls)
alias la='ls -Al' # показать скрытые файлы
alias ls='ls -hF --color' # выделить различные типы файлов цветом
alias lx='ls -lXB' # сортировка по расширению
alias lk='ls -lSr' # сортировка по размеру
alias lc='ls -lcr' # сортировка по времени изменения
alias lu='ls -lur' # сортировка по времени последнего обращения
alias lr='ls -lR' # рекурсивный обход подкаталогов
alias lt='ls -ltr' # сортировка по дате
alias lm='ls -al |more' # вывод через 'more'
alias tree='tree -Csu' # альтернатива 'ls'

# подготовка 'less'
alias more='less'
export PAGER=less
export LESSCHARSET='latin1'
export LESSOPEN='|/usr/bin/lesspipe.sh %s 2>&-' # 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:-...'
```



```

# проверка правописания - настоятельно рекомендую :-)
alias xs='cd'
alias vf='cd'
alias moer='more'
alias moew='more'
alias kk='ll'

#-----
# добавим немножко "приятностей"
#-----

function xtitle ()
{
    case "$TERM" in
        *term | rxvt)
            echo -n -e "\033]0;${*\007" ;;
        *)
            ;;
    esac
}

# псевдонимы...
alias top='xtitle Processes on $HOST && top'
alias make='xtitle Making $(basename $PWD) ; make'
alias ncftp="xtitle ncFTP ; ncftp"

# .. и функции
function man ()
{
    for i ; do
        xtitle The $(basename $1|tr -d .[:digit:]) manual
        command man -F -a "$i"
    done
}

function ll(){ ls -l "$@" | egrep "^d" ; ls -lXB "$@" 2>&- | egrep -v "^d|total " ; }
function te() # "обертка" вокруг хемакс/gnuserv
{
    if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
        gnuclient -q "$@";
    else
        ( хемакс "$@" &);
    fi
}

#-----
# Функции для работы с файлами и строками:
#-----

# Поиск файла по шаблону:
function ff() { find . -type f -iname '*$*' -ls ; }
# Поиск файла по шаблону в $1 и запуск команды в $2 с ним:
function fe() { find . -type f -iname '*$1*' -exec "${2:-file}" {} \; ; }
# поиск строки по файлам:
function fstr()
{
    OPTIND=1
    local case=""
    local usage="fstr: поиск строки в файлах.
Порядок использования: fstr [-i] \"шаблон\" [\"шаблон_имени_файла\"] "
    while getopts :it opt
    do
        case "$opt" in
            i) case="-i " ;;
            *) echo "$usage"; return;;
        esac
    done
    shift $(( $OPTIND - 1 ))
    if [ "$#" -lt 1 ]; then

```

```

        echo "$usage"
        return;
    fi
    local SMSO=$(tput smso)
    local RMSO=$(tput rmso)
    find . -type f -name "${2:-*}" -print0 | xargs -0 grep -sn "${case}" "$1" 2>&- | \
sed "s/$1/${SMSO}\0${RMSO}/gI" | more
}

function cuttail() # удалить последние n строк в файле, по-умолчанию 10
{
    nlines=${2:-10}
    sed -n -e :a -e "1,${nlines}!{P;N;D;};N;ba" $1
}

function lowercase() # перевести имя файла в нижний регистр
{
    for file ; do
        filename=${file##*/}
        case "$filename" in
            /*) dirname==${file%/*} ;;
            *) dirname=.;;
        esac
        nf=$(echo $filename | tr A-Z a-z)
        newname="${dirname}/${nf}"
        if [ "$nf" != "$filename" ]; then
            mv "$file" "$newname"
            echo "lowercase: $file --> $newname"
        else
            echo "lowercase: имя файла $file не было изменено."
        fi
    done
}

function swap() # меняет 2 файла местами
{
    local TMPFILE=tmp.$$
    mv "$1" $TMPFILE
    mv "$2" "$1"
    mv $TMPFILE "$2"
}

#-----
# Функции для работы с процессами/системой:
#-----

function my_ps() { ps @$@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }
function pp() { my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }

# Эта функция является грубым аналогом 'killall' в linux
# но не эквивалентна (насколько я знаю) 'killall' в Solaris
function killps() # "Прибить" процесс по его имени
{
    local pid pname sig="-TERM" # сигнал, рассылаемый по-умолчанию
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
        echo "Порядок использования: killps [-SIGNAL] шаблон_имени_процесса"
        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 "Послать сигнал $sig процессу $pid <$pname>?"
            then kill $sig $pid
        fi
    done
}

function my_ip() # IP адрес

```

```

{
    MY_IP=$(/sbin/ifconfig ppp0 | awk '/inet/ { print $2 } ' | sed -e s/addr://)
    MY_ISP=$(/sbin/ifconfig ppp0 | awk '/P-t-P/ { print $3 } ' | sed -e s/P-t-P://)
}

function ii() # Дополнительные сведения о системе
{
    echo -e "\nВы находитесь на ${RED}$HOST"
    echo -e "\nДополнительная информация:$NC " ; uname -a
    echo -e "\n${RED}В системе работают пользователи:$NC " ; w -h
    echo -e "\n${RED}Дата:$NC " ; date
    echo -e "\n${RED}Время, прошедшее с момента последней перезагрузки :$NC " ;
uptime
    echo -e "\n${RED}Память :$NC " ; free
    my_ip 2>&- ;
    echo -e "\n${RED}IP адрес:$NC" ; echo ${MY_IP:-"Соединение не установлено"}
    echo -e "\n${RED}Адрес провайдера (ISP):$NC" ; echo ${MY_ISP:-"Соединение не
установлено"}
    echo
}

# Разные утилиты:

function repeat() # повторить команду n раз
{
    local i max
    max=$1; shift;
    for ((i=1; i <= max ; i++)); do # --> C-подобный синтаксис
        eval "$@";
    done
}

function ask()
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}

#####
#
# ПРОГРАММНЫЕ ДОПОЛНЕНИЯ - ТОЛЬКО НАЧИНАЯ С ВЕРСИИ BASH-2.04
# Большая часть дополнений взята из документации к bash 2.05 и из
# пакета 'Bash completion' (http://www.caliban.org/bash/index.shtml#completion)
# автор -- Ian McDonalds
# Фактически, у вас должен стоять bash-2.05a
#
#####

if [ "${BASH_VERSION%.*}" \< "2.05" ]; then
    echo "Вам необходимо обновиться до версии 2.05"
    return
fi

shopt -s extglob # необходимо
set +o nounset # иначе некоторые дополнения не будут работать

complete -A hostname rsh rcp telnet rlogin r 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
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

# Архивация
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
# Postscript, pdf, dvi....
complete -f -o default -X '!*.ps' gs ghostview ps2pdf ps2ascii
complete -f -o default -X '!*.dvi' dvips dvi2pdf xdvi dvi2ps dvi2ps
complete -f -o default -X '!*.pdf' acroread pdf2ps
complete -f -o default -X '!*.*(pdf|ps)' gv
complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
complete -f -o default -X '!*.tex' tex latex slite
complete -f -o default -X '!*.lyx' lyx
complete -f -o default -X '!*.*(htm*|HTML*)' lynx html2ps
# Multimedia
complete -f -o default -X '!*.*(jp*g|gif|xpm|png|bmp)' xv gimp
complete -f -o default -X '!*.*(mp3|MP3)' mpg123 mpg321
complete -f -o default -X '!*.*(ogg|OGG)' ogg123

complete -f -o default -X '!*.pl' perl perl5

# Эти 'универсальные' дополнения работают тогда, когда команды вызываются
# с, так называемыми, 'длинными ключами', например: 'ls --all' вместо 'ls -a'

_get_longopts ()
{
    $1 --help | sed -e '/--!/d' -e 's/.*--\([^[:space:]]*\).*/--\1/' | \
grep ^"$2" |sort -u ;
}

_longopts_func ()
{
    case "${2:-*}" in
        -*) ;;
        *) return ;;
    esac

    case "$1" in
        \~*) eval cmd="$1" ;;
        *) cmd="$1" ;;
    esac
    COMPREPLY=( $(_get_longopts $1 $2) )
}

complete -o default -F _longopts_func configure bash
complete -o default -F _longopts_func wget id info a2ps ls recode

_make_targets ()
{
    local mdef makef gcmd cur prev i

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    # Если аргумент prev это -f, то вернуть возможные варианты имен файлов.
    # будем великодушны и вернем несколько вариантов

```

```

# `makefile Makefile *.mk'
case "$prev" in
  -*f)    COMPREPLY=( $(compgen -f $cur ) ); return 0;;
esac

# Если запрошены возможные ключи, то вернуть ключи posix
case "$cur" in
  -)      COMPREPLY=(-e -f -i -k -n -p -q -r -S -s -t); return 0;;
esac

# попробовать передать make `makefile' перед тем как попробовать передать
`Makefile'
if [ -f makefile ]; then
  mdef=makefile
elif [ -f Makefile ]; then
  mdef=Makefile
else
  mdef=*.mk
fi

# прежде чем просмотреть "цели", убедиться, что имя makefile было задано
# ключом -f
for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
  if [[ ${COMP_WORDS[i]} == -*f ]]; then
    eval makef=${COMP_WORDS[i+1]}
    break
  fi
done

[ -z "$makef" ] && makef=$mdef

# Если задан шаблон поиска, то ограничиться
# этим шаблоном
if [ -n "$2" ]; then gcmd='grep "^$2"' ; else gcmd=cat ; fi

# если мы не желаем использовать *.mk, то необходимо убрать cat и использовать
# test -f $makef с перенаправлением ввода
COMPREPLY=( $(cat $makef 2>/dev/null | awk 'BEGIN {FS=":"} /^[^.#  ][^=]*:/{print $1}' | tr -s ' ' '\012' | sort -u | eval $gcmd ) )
}

complete -F _make_targets -X '+(($*|*.[cho])' make gmake pmake

# cvs(1) completion
_cvs ()
{
  local cur prev
  COMPREPLY=()
  cur=${COMP_WORDS[COMP_CWORD]}
  prev=${COMP_WORDS[COMP_CWORD-1]}

  if [ $COMP_CWORD -eq 1 ] || [ "${prev:0:1}" = "-" ]; then
    COMPREPLY=( $( compgen -W 'add admin checkout commit diff \
      export history import log rdiff release remove rtag status \
      tag update' $cur ))
  else
    COMPREPLY=( $( compgen -f $cur ))
  fi
  return 0
}
complete -F _cvs cvs

_killall ()
{
  local cur prev
  COMPREPLY=()
  cur=${COMP_WORDS[COMP_CWORD]}

```

```

# получить список процессов
COMP_REPLY=( $( /usr/bin/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

# Функция обработки мета-команд
# В настоящее время недостаточно отказоустойчива (например, mount и umount
# обрабатываются некорректно), но все еще актуальна. Автор Ian McDonald, изменена
# мной.

_my_command()
{
    local cur func cline cspec

    COMP_REPLY=(
    cur=${COMP_WORDS[COMP_CWORD]}

    if [ $COMP_CWORD = 1 ]; then
        COMP_REPLY=( $( compgen -c $cur ) )
    elif complete -p ${COMP_WORDS[1]} &>/dev/null; then
        cspec=$( complete -p ${COMP_WORDS[1]} )
        if [ "${cspec%-F *}" != "${cspec}" ]; then
            # complete -F <function>
            #
            # COMP_CWORD and COMP_WORDS() доступны на запись,
            # так что мы можем установить их перед тем,
            # как передать их дальше

            # уменьшить на 1 текущий номер лексемы
            COMP_CWORD=$(( $COMP_CWORD - 1 ))
            # получить имя функции
            func=${cspec#*-F }
            func=${func%% *}
            # получить командную строку, исключив первую команду
            cline="${COMP_LINE#$1 }"
            # разбить на лексемы и поместить в массив
            COMP_WORDS=( $cline )
            $func $cline
        elif [ "${cspec#*-[abcdefgjkvu]}" != "" ]; then
            # complete -[abcdefgjkvu]
            #func=$( echo $cspec | sed -e 's/^.*\(-[abcdefgjkvu]\).*$/\1/' )
            func=$( echo $cspec | sed -e 's/^complete//' -e 's/[^]*$//' )
            COMP_REPLY=( $( eval compgen $func $cur ) )
        elif [ "${cspec#*-A}" != "$cspec" ]; then
            # complete -A <type>
            func=${cspec#*-A }
            func=${func%% *}
            COMP_REPLY=( $( compgen -A $func $cur ) )
        fi
    else
        COMP_REPLY=( $( compgen -f $cur ) )
    fi
}

complete -o default -F _my_command nohup exec eval trace truss strace sotruss gdb
complete -o default -F _my_command command type which man nice

# Локальные переменные:
# mode:shell-script
# sh-shell:bash
# Конец:

```

Приложение Н. Преобразование пакетных (*.bat) файлов DOS в сценарии командной оболочки

Большое число программистов начинало изучать скриптовые языки на PC, работающих под управлением DOS. Даже на этом "калеке" удавалось создавать неплохие сценарии, хотя это и требовало значительных усилий. Иногда еще возникает потребность в переносе пакетных файлов DOS на платформу UNIX, в виде сценариев командной оболочки. Обычно это не сложно, поскольку набор операторов, доступных в DOS, представляет из себя ограниченное подмножество эквивалентных команд, доступных в командной оболочке.

Таблица Н-1. Ключевые слова/переменные/операторы пакетных файлов DOS и их аналоги командной оболочки

Операторы пакетных файлов	Эквивалентные команды в UNIX	Описание
%	\$	префикс аргументов командной строки
/	-	признак ключа (опции)
\	/	разделитель имен каталогов в пути
==	=	(равно) сравнение строк
!==!	!=	(не равно) сравнение строк
		конвейер (канал)
@	set +v	не выводить текущую команду
*	*	"шаблонный символ" в имени файла
>	>	перенаправление (с удалением существующего файла)
>>	>>	перенаправление (с добавлением в конец существующего файла)
<	<	перенаправление ввода stdin
%VAR%	\$VAR	переменная окружения
REM	#	комментарий
NOT	!	отрицание последующего условия
NUL	/dev/null	"черная дыра" для того, чтобы "спрятать" вывод команды
ECHO	echo	вывод (в Bash имеет большое число опций)
ECHO .	echo	вывод пустой строки
ECHO OFF	set +v	не выводить последующие команды
FOR %%VAR IN (LIST) DO	for var in [list]; do	цикл "for"

Операторы пакетных файлов	Эквивалентные команды в UNIX	Описание
: LABEL	эквивалент отсутствует (нет необходимости)	метка
GOTO	эквивалент отсутствует (используйте функции)	переход по заданной метке
PAUSE	sleep	пауза, или ожидание, в течение заданного времени
CHOICE	case или select	выбор из меню
IF	if	условный оператор if
IF EXIST FILENAME	if [-e filename]	проверка существования файла
IF !%N==!	if [-z "\$N"]	Проверка: параметр "N" отсутствует
CALL	source или . (оператор "точка")	"подключение" другого сценария
COMMAND /C	source или . (оператор "точка")	"подключение" другого сценария (то же, что и CALL)
SET	export	установить переменную окружения
SHIFT	shift	сдвиг списка аргументов уомандной строки влево
SGN	-lt или -gt	знак (целого числа)
ERRORLEVEL	\$?	код завершения
CON	stdin	"консоль" (stdin)
PRN	/dev/lp0	устройство принтера
LPT1	/dev/lp0	устройство принтера
COM1	/dev/ttyS0	первый последовательный порт

Пакетные файлы обычно содержат вызовы команд DOS. Они должны быть заменены эквивалентными командами UNIX.

Таблица Н-2. Команды DOS и их эквиваленты в UNIX

Команды DOS	Эузивалент в UNIX	Описание
ASSIGN	ln	ссылка на файл или каталог
ATTRIB	chmod	изменить атрибуты файла (права доступа)
CD	cd	сменить каталог
CHDIR	cd	сменить каталог
CLS	clear	очистить экран
COMP	diff, comm, cmp	сравнить файлы
COPY	cp	скопировать файл
Ctl-C	Ctl-C	прервать исполнение сценария
Ctl-Z	Ctl-D	EOF (конец-файла)
DEL	rm	удалить файл(ы)
DELTREE	rm -rf	удалить каталог с подкаталогами

Команды DOS	Эквивалент в UNIX	Описание
DIR	ls -l	вывести содержимое каталога
ERASE	rm	удалить файл(ы)
EXIT	exit	завершить текущий процесс
FC	comm, cmp	сравнить файлы
FIND	grep	найти строку в файлах
MD	mkdir	создать каталог
MKDIR	mkdir	создать каталог
MORE	more	постраничный вывод
MOVE	mv	переместить
PATH	\$PATH	путь поиска исполняемых файлов
REN	mv	переименовать (переместить)
RENAME	mv	переименовать (переместить)
RD	rmdir	удалить каталог
RMDIR	rmdir	удалить каталог
SORT	sort	отсортировать файл
TIME	date	вывести системное время
TYPE	cat	вывести содержимое файла на stdout
XCOPY	cp	(расширенная команда) скопировать файл



Фактически, команды и операторы командной оболочки UNIX имеют огромное количество дополнительных опций, расширяющих их функциональность, по сравнению с их эквивалентами в DOS. В большинстве своем, пакетные файлы DOS предполагают наличие вспомогательных утилит, таких как **ask.com** ("увечный" аналог UNIX-ового [read](#)).

DOS поддерживает крайне ограниченный набор шаблонных символов, участвующих в операциях [подстановки имен файлов](#), распознавая только два символа -- * и ?.

Преобразование пакетных файлов DOS в сценарии командной оболочки, обычно не вызывает затруднений, а результат такого преобразования читается гораздо лучше, чем оригинал.

Пример Н-1. VIEWDATA.BAT: пакетный файл DOS

```

REM VIEWDATA

REM INSPiRED BY AN EXAMPLE IN "DOS POWERTOOLS"
REM                                     BY 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
```

Результат преобразования в сценарий командной оболочки, немного улучшенный.

Пример N-2. viewdata.sh: Результат преобразования VIEWDATA.BAT в сценарий командной оболочки

```
#!/bin/bash
# Результат преобразования пакетного файла VIEWDATA.BAT в сценарий командной
оболочки.
```

```
DATAFILE=/home/bozo/datafiles/book-collection.data
ARGNO=1
```

```
# @ECHO OFF          Эта команда здесь не нужна.
```

```
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
```

```
# операторы перехода GOTO, метки и прочий "мусор" больше не нужны.
# Результат преобразования стал короче, чище и понятнее,
```

На сайте Тэда Дэвиса (Ted Davis) [Shell Scripts on the PC](#), вы найдете большое число руководств по созданию пакетных файлов в DOS. Определенно, его изобретательность будет вам полезна, при создании ваших сценариев.

Приложение I. Упражнения

I.1. Анализ сценариев

Просмотрите следующие сценарии. Попробуйте запустить их, затем объясните -- что они делают. Расставьте комментарии, затем попробуйте записать их в более компактном 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 произойдет, если закомментировать эту строку? Почему?

done

echo "Число = $nr"

exit 0
```

Читатель прислал следующий кусок кода.

```
while read LINE
do
    echo $LINE
done < `tail -f /var/log/messages`
```

Он предполагал написать сценарий, который отслеживал бы изменения в системном журнале `/var/log/messages`. К сожалению, этот код "зависает" и не делает ничего полезного. Почему? Найдите ошибку и исправьте ее (подсказка: вместо операции [перенаправления stdin в цикл](#), попробуйте использовать [конвейерную обработку](#)).

Просмотрите сценарий [Пример A-11](#), попробуйте изменить его таким образом, чтобы он выглядел проще и логичнее. Удалите все "лишние" переменные и попытайтесь оптимизировать сценарий по скорости исполнения.

Измените сценарий таким образом, чтобы он мог принимать начальную установку "поколения 0" из любого текстового файла. Сценарий должен считать первые `$ROW*$COL` символов, и на место гласных вставлять "живые особи". Подсказка: не забудьте преобразовать пробелы в символы подчеркивания.

I.2. Создание сценариев

Напишите сценарии для выполнения повседневных задач.

Простые задания

Содержимое домашнего каталога

Выполните рекурсивный обход домашнего каталога и сохраните информацию в файл. Сожмите файл. Попросите пользователя вставить дискету и нажать клавишу **ENTER**. Запишите сжатый файл на дискету.

Замена цикла [for](#) циклами [while](#) и [until](#)

Замените *циклы for* в [Пример 10-1](#) на *while*. Подсказка: запишите данные в [массив](#) и пройдите в цикле по элементам массива.

Выполнив эту "тяжелую работу", замените циклы, в этом примере, на циклы *until*.

Изменение межстрочного интервала в текстовом файле

Напишите сценарий, который будет читать текст из заданного файла, и выводить, построчно, на `stdout`, добавляя при этом дополнительные пустые строки так, чтобы в результате получился вывод с *двойным межстрочным интервалом*.

Добавьте код, который будет выполнять проверку наличия файла, передаваемого как аргумент.

Когда сценарий будет отлажен, измените его так, чтобы он выводил текстовый файл с *тройным межстрочным интервалом*.

И наконец, напишите сценарий, который будет удалять пустые строки из заданного файла.

Вывод "задом-на-перед"

Напишите сценарий, который будет выводить себя на `stdout`, но в *обратном порядке*.

Автоматическое разархивирование

Для каждого файла, из заданного списка, сценарий должен определить тип архиватора, которым был создан тот или иной файл (с помощью утилиты [file](#)). Затем сценарий должен выполнить соответствующую команду разархивации (**`gunzip`**, **`bunzip2`**, **`unzip`**, **`uncompress`** или что-то иное). Если файл не является архивом, то сценарий должен оповестить пользователя об этом и ничего не делать с этим файлом.

Уникальный идентификатор системы

Сценарий должен сгенерировать "уникальный" 6-ти разрядный шестнадцатиричный идентификатор системы. *Не* пользуйтесь дефектной утилитой [hostid](#). Подсказка: [md5sum](#) / `etc/passwd`, затем отберите первые 6 цифр.

Резервное копирование

Сценарий должен создать архив (`*.tar.gz`) всех файлов в домашнем каталоге пользователя (`/home/user-name`), которые изменялись в течение последних 24 часов. Подсказка: воспользуйтесь утилитой [find](#).

Простые числа

Сценарий должен вывести (на `stdout`) все простые числа, в диапазоне от 60000 до 63000. Вывод должен быть отформатирован по столбцам (подсказка: воспользуйтесь командой [printf](#)).

Лототрон

Сценарий должен имитировать работу лототрона -- извлекать 5 случайных неповторяющихся чисел в диапазоне 1 - 50. Сценарий должен предусматривать как вывод на `stdout`, так и запись чисел в файл, кроме того, вместе с числами должны выводиться дата и время генерации данного набора.

Задания повышенной сложности

Управление дисковым пространством

Сценарий должен отыскивать в домашнем каталоге пользователя `/home/username` файлы, имеющие размер больше 100К. Каждый раз предоставляя пользователю возможность удалить или сжать этот файл, затем переходить к поиску следующего файла.

Безопасное удаление

Напишите сценарий "безопасного" удаления файлов -- `srn.sh`. Файлы, с именами, передаваемыми этому сценарию, не должны удаляться, вместо этого, файлы следует сжать утилитой [gzip](#), если они еще не сжаты (не забывайте про утилиту [file](#)), и переместить в каталог `/home/username/trash`. При старте, сценарий должен удалять из каталога "trash" файлы, которые были созданы более 48 часов тому назад.

Размен монет

Как более рационально собрать сумму в \$1.68, используя только монеты, с номиналом не выше 25с? Это будет шесть 25-ти центовых монет, одна десятицентовая, одна пятицентовая и три монеты достоинством в 1 цент.

Учитывая возможность произвольного ввода суммы в долларах и центах (`$*.??`), найдите такую комбинацию, которая требовала бы наименьшее число монет. Если вы проживаете не в США, то можете использовать свою денежную единицу и номиналы монет. Подсказка: взгляните на [Пример 22-4](#).

Корни квадратного уравнения

Напишите сценарий, который находил бы корни "квадратного" уравнения, вида: $Ax^2 + Bx + C = 0$. Сценарий должен получать коэффициенты уравнения A , B и C , как аргументы командной строки, и находить корни, с точностью до четвертого знака после запятой.

Подсказка: воспользуйтесь [bc](#), для нахождения решения по хорошо известной формуле: $x = (-B \pm \sqrt{B^2 - 4AC}) / 2A$.

Сумма чисел

Найдите сумму всех пятизначных чисел (в диапазоне 10000 - 99999), которые содержат *точно две* цифры из следующего набора: { 4, 5, 6 }.

Примеры чисел, удовлетворяющих данному условию: 42057, 74638 и 89515.

Счастливым билет

"Счастливым" считается такой билет, в котором последовательное сложение цифр номера дает число 7. Например, 62431 -- номер "счастливого" билета ($6 + 2 + 4 + 3 + 1 = 16$, $1 + 6 = 7$). Найдите все "счастливые" номера, располагающиеся в диапазоне

1000 - 10000.

Синтаксический анализ

Проанализируйте файл `/etc/passwd` и выведите его содержимое в табличном виде.

Просмотр файла с данными

Некоторые базы данных и электронные таблицы используют формат CSV (*comma-separated values*), для хранения данных в файлах. Зачастую, эти файлы должны анализироваться другими приложениями.

Пусть файл содержит следующие данные:

```
Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989  
Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612  
...
```

Прочитайте данные и выведите их на `stdout` в виде колонок с заголовками.

Выравнивание

Текст вводится с устройства `stdin` или из файла. Его необходимо вывести на `stdout`, с выравниванием по ширине, используя задаваемую пользователем ширину строк.

Список рассылки

Напишите сценарий, который использовал бы команду [mail](#), для управления простым списком рассылки. Сценарий должен брать текст ежемесячного информационного бюллетеня из заданного файла, список адресатов из другого файла и выполнять рассылку новостей по электронной почте.

Пароли

Сгенерируйте псевдослучайные 8-ми символьные пароли, используя символы из диапазона `[0-9]`, `[A-Z]`, `[a-z]`. Каждый пароль должен содержать не менее 2-х цифр.

Сложные задания

Регистрация обращений к файлам

Попробуйте отследить все попытки обращения к файлам в каталоге `/etc`, в течение дня. Сведения, которые включают в себя время обращения, имя файла, имя пользователя (если имели место какие либо изменения в файлах, то они тоже должны быть отмечены), запишите в виде аккуратно отформатированных записей в логфайл.

Удаление комментариев

Удалите все комментарии из сценария, имя которого задается с командной строки. При этом, строка `#!/bin/bash` не должна удаляться.

Преобразование в HTML

Преобразуйте заданный текстовый файл в HTML формат. Этот сценарий должен автоматически вставлять необходимые теги HTML в тело файла.

Удаление тегов HTML

Удалите все теги HTML из заданного HTML файла, затем переформатируйте его так, чтобы строки не были короче 60 и длиннее 75 символов. Предусмотрите оформление параграфов. Преобразуйте таблицы HTML в их приблизительный текстовый эквивалент.

Преобразование XML файлов

Преобразуйте файл из формата XML в формат HTML и в простой текстовый файл.

Борьба со спамом

Напишите сценарий, который анализировал бы входящие почтовые сообщения на принадлежность к спаму и отыскивал бы в DNS имена узлов сети, по IP адресам из заголовка письма. Сценарий должен отправлять найденные спамерские сообщения ответственным за спам провайдером (ISP). Естественно, вы должны отфильтровать *свой собственный IP адрес*, чтобы не случилось так, что вы жалуетесь на самого себя.

По мере необходимости, используйте соответствующие [команды для работы с сетью](#).

Азбука Морзе

Преобразуйте текстовый файл в код Морзе. Символы из файла должны быть представлены в виде, соответствующих им, кодов Морзе, состоящих из точек и тире, и разделенных пробелами. Например, "script" ==> "... _ _ . _".

Шестнадцатиричный дамп

Выведите, в виде шестнадцатиричного дампа, содержимое бинарного файла, передаваемого в сценарий, как аргумент командной строки. Вывод должен производиться в четкой табличной форме, первое поле таблицы -- адрес, далее должны следовать 8 полей, содержащие 4-х байтовые шестнадцатиричные числа, а завершать строку должно поле, содержащее эквивалентное отображение 8-ми предшествующих полей, в виде ASCII-символов.

Эмуляция сдвигового регистра

Используя [Пример 25-9](#), как образец, напишите сценарий, который эмулировал бы 64-х битный сдвиговый регистр в виде [массива](#). Реализуйте функции *загрузки* значения в регистр, *сдвиг влево* и *сдвиг вправо*. В заключение, напишите функцию, которая интерпретировала бы содержимое "регистра" как восемь 8-ми битных символов ASCII.

Детерминант (определитель)

Найдите детерминант (определитель) матрицы 4 x 4.

Анаграммы

Сценарий должен запросить у пользователя 4-х символьное слово, и найти

анаграммы для этого слова. Например, анаграммы к слову *word: do or rod row word*. Для поиска анаграмм можете использовать файл `/usr/share/dict/linux.words`.

Индекс сложности текста

"Индекс сложности текста" оценивает трудность понимания текста, как некое число, которое грубо соответствует количеству лет обучения в общеобразовательной школе. Например, индекс равный 8-ми говорит о том, что текст доступен для понимания человеку, окончившему 8-й класс общеобразовательной школы.

Вычисление индекса ведется по следующему алгоритму.

1. Выберите кусок текста, длиной не менее 100 слов.
2. Сосчитайте количество предложений.
3. Найдите среднее число слов в предложении.

$$\text{СРЕДНЕЕ_ЧИСЛО_СЛОВ} = \frac{\text{ОБЩЕЕ_ЧИСЛО_СЛОВ}}{\text{ЧИСЛО_ПРЕДЛОЖЕНИЙ}}$$

4. Сосчитайте количество "трудных" слов -- которые содержат не менее 3-х слогов. Разделите это число на общее количество слов, в результате вы получите пропорцию сложных слов.

$$\text{ПРОПОРЦИЯ_СЛОЖНЫХ_СЛОВ} = \frac{\text{ЧИСЛО_ДЛИННЫХ_СЛОВ}}{\text{ОБЩЕЕ_ЧИСЛО_СЛОВ}}$$

5. Индекс сложности текста рассчитывается как сумма двух этих чисел, умноженная на 0.4 и округленная до ближайшего целого.

$$\text{ИНДЕКС_СЛОЖНОСТИ} = \text{int} (0.4 * (\text{СРЕДНЕЕ_ЧИСЛО_СЛОВ} + \text{ПРОПОРЦИЯ_СЛОЖНЫХ_СЛОВ}))$$

4-й пункт -- самый сложный. Существуют различные алгоритмы подсчета слогов в словах. В данном же случае, вы можете ограничиться подсчетом сочетаний "гласный-согласный".

Строго говоря, при расчете индекса сложности не следует считать составные слова и имена собственные как "сложные" слова, но это слишком усложнит сценарий.

Вычисление числа пи по алгоритму "Игла Баффона"

В 18 веке, французский математик де Баффон (de Buffon) проделывал эксперимент, который заключался в бросании иглы, длиной "n", на деревянный пол, собранный из длинных и узких досок. Ширина всех досок пола одинакова и равна "d". Оказалось, что отношение общего числа бросков, к числу бросков, когда игла ложилась на щель, кратно числу пи.

Пользуясь [Пример 12-35](#), напишите сценарий, который использовал бы метод Монте Карло для эмуляции "Иглы Баффона". Для простоты примите длину иглы равной ширине досок, $n = d$.

Подсказка: особое значение здесь имеют переменные, которые будут вычисляться как расстояние от центра иглы до ближайшей щели и величина угла между иглой и щелью. Для выполнения расчетов можно воспользоваться утилитой [bc](#).

Шифрование по алгоритму Playfair

Напишите сценарий, реализующий алгоритм шифрования Playfair (Wheatstone).

В соответствии с этим алгоритмом, текст шифруется путем замены каждой 2-х символьной последовательности -- "диграммы". Традиционно, в качестве ключа, используется матрица символов алфавита 5 x 5.

```
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
```

Матрица содержит все символы алфавита, за исключением символа "J", который представляет символ "I". Первая строка матрицы -- произвольно выбранное слово, в данном случае -- "CODES", далее следуют символы алфавита, исключая те, которые входят в состав первой строки.

Шифрование производится по следующему алгоритму: для начала, текст сообщения разбивается на диграммы (группы по 2 символа). Если в диграмму попадают два одинаковых символа, то второй символ удаляется, и формируется новая диграмма. Если в последней группе остается один символ, то такая "неполная" диграмма дополняется "пустым" символом, обычно "X".

```
THIS IS A TOP SECRET MESSAGE
```

```
TH IS IS AT OP SE CR ET ME SA GE
```

Каждая диграмма может подпадать под одно из следующих определений:.

- 1) Оба символа находятся в одной строке ключа. Тогда, каждый из них заменяется символом, стоящим справа в той же строке. Если это последний символ строки ключа, то он заменяется первым символом в той же строке ключа.
- 2) Оба символа находятся в одном столбце ключа. Тогда каждый из них заменяется на символ, стоящий ниже, в этом же столбце. Если это последний символ в столбце ключа, то он заменяется первым символом в том же столбце ключа.
- 3) Символы диграммы стоят в вершинах прямоугольника. Тогда каждый из них заменяется символом из соседнего, по горизонтали, угла.

Диграмма "TH" соответствует 3-му определению.

```
G H
M N
T U      (Прямоугольник с вершинами "T" и "H")
```

```
T --> U
H --> G
```

Диграмма "SE" соответствует 1-му определению.

```
C O D E S      (Строка содержит оба символа "S" и "E")
```

S --> C (замена на первый символ в строке ключа)
E --> S

Дешифрация выполняется обратной процедурой, для случаев 1 и 2 -- замена символом стоящим левее/выше. Для случая 3 -- аналогично шифрации, т.е. заменяется символом из соседнего, по горизонтали, угла. Helen Fouche Gaines, в своей классической работе "Elementary Cryptanalysis" (1939), приводит подробное описание алгоритма Playfair и методы его реализации.

Этот сценарий должен иметь три основных раздела

- I. Генерация "ключевой матрицы", основывающейся на слове, которое вводит пользователь.
- II. Шифрование "плоского" текста сообщения.
- III. Дешифрование зашифрованного текста.

Широкое применение, в этом сценарии, найдут [массивы](#) и [функции](#).

--

Пожалуйста, не присылайте автору свои варианты решения упражнений. Если вы хотите впечатлить его своим умом и сообразительностью -- присылайте обнаруженные вами ошибки и предложения по улучшению этой книги.

Приложение J. Авторские права

Авторские права на книгу "Advanced Bash-Scripting Guide", принадлежат Менделю Куперу (Mendel Cooper). Этот документ может распространяться исключительно на условиях Open Publication License (версия 1.0 или выше), <http://www.opencontent.org/openpub/>. Соблюдение следующих пунктов лицензии обязательно.

1. Распространение существенно измененных версий этого документа, запрещено без явного разрешения держателя прав.
2. Запрещено распространение твердых (бумажных) копий книги, или ее производных, без явного согласия держателя прав.

Пункт 1, выше, явно запрещает вставлять в текст документа логотипы компаний или навигационные элементы, за исключением

1. Некоммерческих организаций, таких как [Linux Documentation Project](#) и [Sunsite](#).
2. Не "запятнавших" себя дистрибутивостроителей Linux, таких как Debian, Red Hat, Mandrake и других.

Практически, вы можете свободно распространять *неизмененную* электронную версию этой книги. Вы должны получить явное разрешение автора на распространение

измененных версий книги или ее производных. Цель этого ограничения состоит в том, чтобы сохранить художественную целостность данного документа и предотвратить появление побочных "ветвей".

Это очень либеральные условия и они не должны препятствовать законному распространению и использованию этой книги. Автор особенно поощряет использование этой книги в учебных целях.

Права на коммерческое распространение книги могут быть получены у [автора](#).

Автор произвел этот документ в соответствии с буквой и духом [LDP Manifesto](#).

Hyun Jin Cha завершил [перевод на Корейский язык](#) версию 1.0.11 этой книги. Переводы на Испанский, Португальский, Французский, Немецкий, Итальянский и Китайский языки находятся на стадии реализации. Если вы изъявите желание перевести этот документ на другой язык, то можете свободно выполнить этот перевод, основываясь на условиях, заявленных выше. В этом случае, автор хотел бы, чтобы его поставили в известность.

Linux -- это торговая марка, принадлежащая Линусу Торвальдсу (Linus Torvalds).

Unix и UNIX -- это торговая марка, принадлежащая Open Group.

MS Windows -- это торговая марка, принадлежащая Microsoft Corp.

Все другие коммерческие торговые марки, упомянутые в данном документе, принадлежат их владельцам.

Примечания

[1] Их так же называют [встроенными](#) конструкциями языка командной оболочки shell.

[2] Многие особенности *ksh88* и даже *ksh93* перекочевали в Bash.

[3] В соответствии с соглашениями, имена файлов с shell-скриптами, такими как Bourne shell и совместимыми, имеют расширение `.sh`. Все стартовые скрипты, которые вы найдете в `/etc` следуют этому соглашению.

[4] Некоторые разновидности UNIX (основанные на 4.2BSD) требуют, чтобы эта последовательность состояла из 4-х байт, за счет добавления пробела после `#!` `/bin/sh`.

[5] В shell-скриптах последовательность `#!` должна стоять самой первой и задает интерпретатор (`bash`). Интерпретатор, в свою очередь, воспринимает эту строку как комментарий, поскольку начинается с символа `#`.

Если в сценарии имеются еще такие же строки, то они воспринимаются как обычный комментарий.

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

```
echo "Первая часть сценария."  
a=1
```

```
#!/bin/bash  
# Это *НЕ* означает запуск нового сценария.
```

```
echo "Вторая часть сценария."
```

```
echo $a # Значение переменной $a осталось равно 1.
```

[6] Эта особенность позволяет использовать различные хитрости.

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

# Этот скрипт ничего не делает -- только уничтожает себя.

WHATEVER=65

echo "Эта строка никогда не будет напечатана."

exit $WHATEVER # Не имеет смысла, поскольку работа сценария завершается не здесь.
```

Попробуйте запустить файл README с сигнатурой `#!/bin/more` (предварительно не забудьте сделать его исполняемым).

[7] **Portable Operating System Interface**, попытка стандартизации UNIX-подобных операционных

[8] Внимание: вызов Bash-скрипта с помощью команды `sh scriptname` отключает спецификацию Bash расширения, что может привести к появлению ошибки и аварийному завершению работы сценария.

[9] Сценарий должен иметь как право на исполнение, так и право на *чтение*, поскольку shell дает возможность прочитать скрипт.

[10] Почему бы не запустить сценарий просто набрав название файла `scriptname`, если сценарий находится в текущем каталоге? Дело в том, что из соображений безопасности, путь к текущему каталогу "." не включен в переменную окружения `$PATH`. Поэтому необходимо явно указывать текущему каталогу, в котором находится сценарий, т.е. `./scriptname`.

[11] Интерпретатор, встретив фигурные скобки, раскрывает их и возвращает полученный список команд, которые затем и исполняет.

[12] Исключение: блок кода, являющийся частью конвейера, *может* быть запущен в дочернем процессе (`subshell-e`).

```
ls | { read firstline; read secondline; }
# Ошибка! Вложенный блок будет запущен в дочернем процессе,
# таким образом, вывод команды "ls" не может быть записан в переменные
# находящиеся внутри блока.
echo "Первая строка: $firstline; вторая строка: $secondline" # Не работает!

# Спасибо S.C.
```

[13] Аргумент `$0` устанавливается вызывающим процессом. В соответствии с соглашениями, этот аргумент содержит имя файла скрипта. См. страницы руководства для `execv` (man execv).

[14] Символ "!", помещенный в двойные кавычки, порождает сообщение об ошибке, если команда не найдена *с командной строки*. Вероятно это связано с тем, что этот символ интерпретируется как опция обращения к *истории команд*. Однако внутри сценариев такой прием проблем не вызывает.

Не менее любопытно поведение символа "\", употребляемого внутри двойных кавычек.

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

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

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

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

(Спасибо Wayne Pollock за пояснения.)

[15] "Разбиение на слова", в данном случае это означает разделение строки символов на некоторые аргументов.

[16] С флагом *suid*, на двоичных исполняемых файлах, надо быть очень осторожным, поскольку они могут быть небезопасным. Установка флага *suid* на файлы-сценарии не имеет никакого эффекта.

[17] В современных UNIX-системах, "sticky bit" больше не используется для файлов, только для каталогов.

[18] Как указывает S.C., даже заключение строки в кавычки, при построении сложных условий может оказаться недостаточным. [`-n "$string" -o "$a" = "$b"`] в некоторых версиях Bash такая проверка может вызвать сообщение об ошибке, если строка `$string` пустая. В смысле отказоустойчивости, было бы добавить какой-либо символ к, возможно пустой, строке: [`$string" != x -o "x$a" = "x$b"`] (символ "x" не учитывается).

[19] PID текущего процесса хранится в переменной `$$`.

[20] Слова "аргумент" и "параметр" очень часто используются как синонимы. В тексте данного документа они применяются для обозначения одного и того же понятия, будь то аргумент, передаваемый в функцию из командной строки или входной параметр функции.

[21] Применяется к аргументам командной строки или входным параметрам [функций](#).

[22] Если `$parameter` "пустой", в неинтерактивных сценариях, то это будет приводить к завершению с кодом возврата [127](#) ("command not found").

[23] Эти команды являются [встроенными командами](#) языка сценариев командной оболочки (shell), в то время как [while](#), [case](#) и т.п. -- являются [зарезервированными словами](#).

[24] Исключение из правил -- команда [time](#), которая в официальной документации к Bash называется ключевым словом.

[25] Опция `--` это аргумент, который управляет поведением сценария и может быть либо включен, либо выключен. Аргумент, который объединяет в себе несколько опций (ключей), определяет поведение сценария в соответствии с отдельными опциями, объединенными в данном аргументе..

[26] Как правило, исходные тексты подобных библиотек, на языке C, располагаются в каталоге

1 /usr/share/doc/bash-?.??.functions.

Обратите внимание: ключ -f команды **enable** может отсутствовать в некоторых системах.

[27] Тот же эффект можно получить с помощью [typeset -fu](#).

1
[28] Скрытыми считаются файлы, имена которых начинаются с точки, например, ~/.Xdefaults. Такие файлы не выводятся простой командой **ls**, и не могут быть удалены командой **rm -rf ***. Как и с другими скрытыми делаются конфигурационные файлы в домашнем каталоге пользователя.

[29] Это верно только для GNU-версии команды **tr**, поведение этой команды, в коммерческих UNIX-системах, может несколько отличаться.

[30] Команда **tar czvf archive_name.tar.gz *** *включит* в архив все скрытые файлы (имена которых начинаются с точки) из *вложенных подкаталогов*. Это недокументированная "особенность" версии **tar**.

[31] Она реализует алгоритм симметричного блочного шифрования, в противоположность алгоритму шифрования с "открытым ключом", из которых широко известен **pgp**.

[32] *Демон* -- это некий фоновый процесс, не привязанный ни к одной из терминальных сессий. Такие процессы предназначены для выполнения определенного круга задач либо через заданные промежутки времени, либо по наступлению какого-либо события.

Слово "демон" ("daemon"), в греческой мифологии, употреблялось для обозначения призрака или чего-то мистического, сверхъестественного. В мире UNIX -- под словом демон подразумевается процесс, который "тихо" и "незаметно" выполняет свою работу.

[33] Фактически -- это сценарий, заимствованный из дистрибутива Debian Linux.

[34] *Очередь печати* -- это группа заданий "ожидających вывода" на принтер.

[35] Эта тема прекрасно освещена в статье, которую написал Andy Vaught, [Introduction to Namespaces](#), опубликованная в сентябре 1997 для [Linux Journal](#).

[36] EBCDIC (произносится как "ebb-sid-ic") -- это аббревиатура от Extended Binary Coded Decimal Interchange Code (Расширенный Двоично-Десятичный Код Обмена Информацией). Это формат представления данных от IBM, не нашедший широкого применения. Не совсем обычное предложение опции `conv=ebcdic` -- это использовать **dd** для быстрого и легкого, но слабого, шифрования файлов.

```
cat $file | dd conv=swab,ebcdic > $file_encrypted
# Зашифрованный файл будет выглядеть как "абракадабра".
# опция swab добавлена для внесения большей неразберихи.
```

```
cat $file_encrypted | dd conv=swab,ascii > $file_plaintext
# Декодирование.
```

[37] *макроопределение* -- это идентификатор, символическая константа, которая представляет последовательность команд, операций и параметров.

[38] Команда **userdel** завершится неудачей, если удаляемый пользователь в этот момент работает.

системой

[39] Дополнительную информацию по записи компакт-дисков, вы найдете в статье Алекса Уизера (Alex Wither): [Creating CDs](#), в октябрьском выпуске журнала [Linux Journal](#) за 1999 год.

[40] Утилита [mke2fs](#), с ключом `-c`, так же производит поиск поврежденных блоков.

[41] Пользователи небольших, десктопных Linux-систем предпочитают утилиты попроще, например [find](#).

[42] NAND -- логическая операция "И-НЕ". В общих чертах она напоминает вычитание.

[43] *Замещающая команда* может быть внешней системной командой, внутренней (встроенной) или даже функцией в сценарии.

[44] *дескриптор файла* -- это просто число, по которому система идентифицирует открытые файлы. Рассматривайте его как упрощенную версию указателя на файл.

[45] При использовании *дескриптора с номером 5* могут возникать проблемы. Когда Bash запускает дочерний процесс, например командой [exec](#), то дочерний процесс наследует дескриптор 5 "открытый" (см. архив почты Чета Рамея (Chet Ramey), [SUBJECT: RE: File descriptor 5 is held](#)). Поэтому, лучше не использовать этот дескриптор.

[46] В качестве простейшего регулярного выражения можно привести строку, не содержащую никаких метасимволов.

[47] Поскольку с помощью [sed](#), [awk](#) и [grep](#) обрабатываются одиночные строки, то обычно символ перевода строки не принимается во внимание. В тех же случаях, когда производится разбор многострочного текста, метасимвол "точка" будет соответствовать символу перевода строки.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF # Встроенный документ
line1
line2
EOF
# OUTPUT:
# [line1
# line2]

echo

awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
line 1
line 2
EOF
# OUTPUT:
# line
# 1

# Спасибо S.C.

exit 0
```

[48] Подстановка таких имен файлов *возможна*, но только при условии, что символ точки будет

присутствовать в шаблоне.

```
~/.[.]bashrc      # Не будет соответствовать имени ~/.bashrc
~/?.bashrc       # То же самое.
                  # Метасимволы не могут соответствовать символу точки при подстановке имени

~/.[b]ashrc      # Имя ~/.bashrc будет соответствовать данному шаблону
~/?.ba?hrc       # Аналогично.
~/?.bashr*       # Аналогично.

# Установка ключа "dotglob" отключает такое поведение интерпретатора.
# Спасибо S.C.
```

[49] Имеет тот же эффект, что и [именованные каналы](#) (временный файл), фактически, именованные каналы некогда использовались в операциях подстановки процессов.

[50] Механизм [косвенных ссылок](#) на переменные (см. [Пример 34-2](#)) слишком неудобен для переименования аргументов по ссылке.

```
#!/bin/bash

ITERATIONS=3 # Количество вводимых значений.
icount=1

my_read () {
    # При вызове my_read varname,
    # выводит предыдущее значение в квадратных скобках,
    # затем просит ввести новое значение.

    local local_var

    echo -n "Введите новое значение переменной "
    eval 'echo -n "[$'$1'] "' # Прежнее значение.
    read local_var
    [ -n "$local_var" ] && eval $1=\$local_var

    # Последовательность "And-list": если "local_var" не пуста, то ее значение переписывается
    "$1".
}

echo

while [ "$icount" -le "$ITERATIONS" ]
do
    my_read var
    echo "Значение #$icount = $var"
    let "icount += 1"
    echo
done

# Спасибо Stephane Chazelas за этот поучительный пример.

exit 0
```

[51] Команда **return** -- это [встроенная](#) команда Bash.

[52] [Herbert Mayer](#) определяет *рекурсию*, как "...описание алгоритма с помощью более простой версии самого алгоритма..." Рекурсивной называется функция, которая вызывает самого себя.

[53] Слишком глубокая рекурсия может вызвать крах сценария.

]

```
#!/bin/bash

recursive_function ()
{
(( $1 < $2 )) && recursive_function $(( $1 + 1 )) $2;
# Увеличивать 1-й параметр до тех пор,
#+ пока он не станет равным, или не превысит, второму параметру.
}

recursive_function 1 50000 # Глубина рекурсии = 50,000!
# Само собой -- Segmentation fault.

# Рекурсия такой глубины может "обрушить" даже программу, написанную на C,
#+ по исчерпанию памяти, выделенной под сегмент стека.

# Спасибо S.C.

exit 0 # Этот сценарий завершает работу не здесь, а в результате ошибки Segmentation
```

[54] Однако, псевдонимы могут "раскрывать" позиционные параметры.

]

[55] Это не относится к таким оболочкам, как **csh**, **tcsh** и другим, которые не являются производными классической Bourne shell (**sh**).

]

[56] Каталог /dev содержит специальные файлы -- точки монтирования физических и виртуальных устройств. Они занимают незначительное пространство на диске.

Некоторые из устройств, такие как /dev/null, /dev/zero или /dev/urandom -- являются виртуальными. Они не являются файлами физических устройств, система эмулирует эти устройства программным способом.

[57] *Блочное устройство* читает и/или пишет данные целыми блоками, в отличие от *символьных устройств*, которые читают и/или пишут данные по одному символу. Примером блочного устройства может служить жесткий диск, CD-ROM. Примером символьного устройства -- клавиатура.

[58] Отдельные системные команды, такие как [procinfo](#), [free](#), [vmstat](#), [lsdev](#) и [uptime](#) делают это иными образом.

]

[59] [Bash debugger](#) (автор: Rocky Bernstein) частично возмещает этот недостаток.

]

[60] В соответствии с соглашениями, сигнал с номером 0 соответствует команде [exit](#).

]

[61] Установка этого бита на файлы сценариев не имеет никакого эффекта.

]

[62] ANSI -- аббревиатура от American National Standards Institute.

]

[63] См. статью Marius van Oers, [Unix Shell Scripting Malware](#), а также ссылку на *Denning* в разделе [Литература](#).

]

[64] Chet Ramey обещал ввести в Bash ассоциативные массивы (они хорошо знакомы программистам, работающим с языком Perl) в одном из следующих релизов Bash.

]

[65] Кто может -- тот делает. Кто не может... тот получает сертификат MCSE.

]

[66] Если адресное пространство не указано, то, по-умолчанию, к обработке принимаются все с

]

[67] Указание кода завершения за пределами установленного диапазона, приводит к возврату с

]

кодов. Например, **exit 3809** вернет код завершения, равный 225.