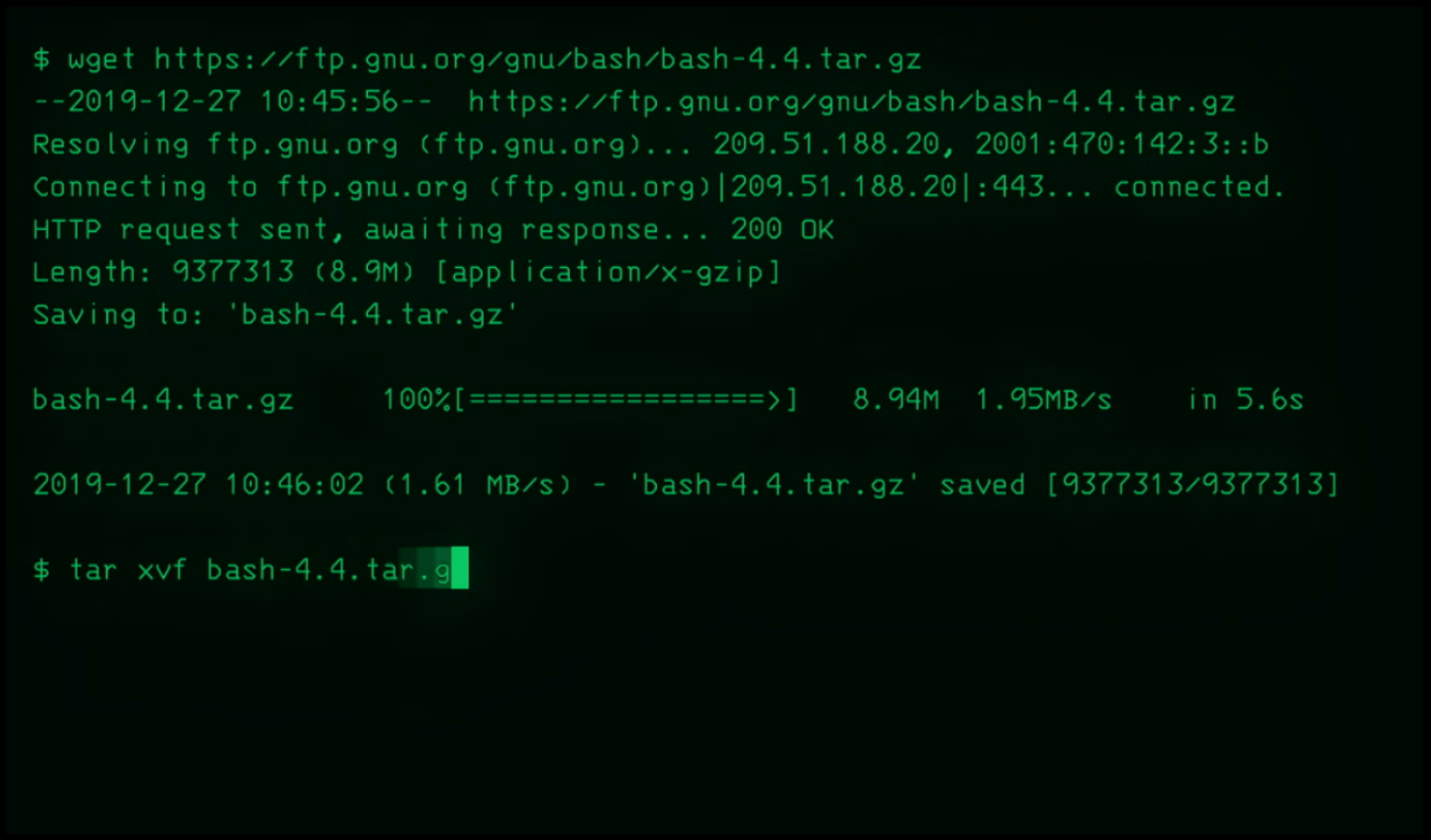
Программирование на Bash с нуля



Илья Шпигорь

Программирование на Bash с нуля

Илья Шпигорь

Эта книга предназначена для продажи на http://leanpub.com/programming-from-scratch

Эта версия была опубликована 2021-04-24

Это книга с Leanpub book. Leanpub позволяет авторам и издателям участвовать в так называемом Lean Publishing - процессе, при котором электронная книга становится доступна читателям ещё до её завершения. Это помогает собрать отзывы и пожелания для скорейшего улучшения книги. Мы призываем авторов публиковать свои работы как можно раньше и чаще, постепенно улучшая качество и объём материала. Тем более, что с нашими удобными инструментами этот процесс превращается в удовольствие.

Твитните эту книгу!

Пожалуйста, помогите Илья Шпигорь в распространении информации о книге в Twitter!

Предлагаемый хештег для этой книги #programming-from-scratch.

Узнайте, что другие люди пишут об этой книге, кликнув на ссылку на странцу поиска по этому хештегу в Twitter:

#programming-from-scratch

Оглавление

Вступление **1**

Общая информация **3**
Операционные системы 3
Компьютерная программа 27

Командный интерпретатор Bash **41**
Инструменты для разработки 41
Командный интерпретатор 46
Навигация по файловой системе 49
Информация о командах 76
Действия над файлами и каталогами 83
Дополнительные возможности Bash 96

Разработка Bash-скриптов **120**
Инструменты для разработки 120
Зачем нужны скрипты? 126
Переменные и параметры 143
Условные операторы 172
Арифметические выражения 195
Операторы цикла 229
Функции 254

Пакетный менеджер **273**
Репозиторий 273
Работа с пакетами 273

Заключение **277**

Благодарности **280**

Список терминов **281**
А 281
Б 281
В 281
Д 282

З . . . 282
И . . . 282
К . . . 283
Л . . . 283
О . . . 284
П . . . 285
Р . . . 286
С . . . 287
У . . . 287
Ф . . . 288
Х . . . 288
Ш . . . 289
Я . . . 289
A . . . 289
B . . . 290
C . . . 291
E . . . 291
F . . . 292
G . . . 292
I . . . 292
L . . . 292
P . . . 292
Q . . . 293
S . . . 293
T . . . 294
U . . . 294
W . . . 294

Ответы . . . 295
Общая информация . . . 295
Командный интерпретатор Bash . . . 295
Разработка Bash-скриптов . . . 301

Ссылки на ресурсы . . . 319
Общая информация . . . 319
Bash . . . 319
Unix-окружение . . . 320

Вступление

Научиться программировать непросто. Ещё сложнее если вы начинаете с нуля и учитесь самостоятельно, не прибегая к помощи профессиональных преподавателей. Это выполнимая задача, но многое будет зависеть от вас самих.

Изучать новый и сложный предмет невозможно без хорошей мотивации. Поэтому прежде чем продолжить читать эту книгу разберитесь со своими целями. Чего вы ждёте от своих новых навыков? Какие задачи будут решать ваши программы? Ответы на эти вопросы помогут выбрать эффективный путь обучения.

Если вы твёрдо намерены стать профессиональным программистом в кратчайшие сроки, то без посторонней помощи вам не обойтись. Запишитесь на очные курсы или на онлайн лекции в интернете. Возможность напрямую общаться с наставником, задавать вопросы и прояснять непонятные моменты значительно ускорит ваш прогресс.

Без наставника можно обойтись, если вы интересуетесь программированием из любопытства как хобби или для расширения своего кругозора. В этом случае самообучение с книгой уже принесёт практическую пользу. Ведь базовые навыки программирования пригодятся любому, кто работает с компьютером. Возможно, эта книга послужит отправной точкой и поможет вам определиться с направлением для дальнейшего развития.

Сегодня в онлайн магазинах и библиотеках доступно множество книг по программированию для читателей с различным уровнем подготовки. Есть ли смысл в ещё одной книге?

Программирование — это сугубо практическая область. Конечно, в ней есть много теории, как например в математике. Но знание основополагающих принципов не сделает вас программистом. Чтобы им стать, вам придётся самостоятельно написать очень много кода. Сначала этот код просто не будет работать. Потом он будет содержать ошибки. Постепенно вы научитесь их предвидеть и устранять заранее. Показателем вашего прогресса будет не растущий объём знаний по конкретному языку, а оценка вашего старого кода. Когда вы прочитаете его спустя месяц или два и заметите в нём ошибки, это станет подтверждением вашего прогресса.

Так что же не так с существующими книгами? Многие из них посвящены конкретному языку или технологии. Выбранная тема рассматривается во всех тонкостях и нюансах. При этом выполнению практических заданий уделяется недостаточно внимания. Новичку такой объём узкоспециализированных теоретических знаний ни к чему. Кроме того он вызывает ложное представление о том, как надо учиться программировать. Такие книги по конкретным технологиям редко кто читает от корки до корки. Чаще ими пользуются в качестве справочника, когда возникает какой-то практический вопрос.

Другие книги предлагают освоить какой-то язык программирования на примерах. Обучение с ними пойдёт намного продуктивнее. Проблема в том, что не все читатели находят моти-

вацию работать над примерами. Авторы предлагают прочитать и разобраться в большом объёме кода. К сожалению, если этот код только демонстрирует некий принцип и не несёт никакой практической пользы, он будет мало интересен читателю. Современные языки общего назначения сложны. Это значит, что прежде чем перейти к примерам из реальной жизни, вам понадобиться узнать о языке многое. Получается замкнутый круг: примеры неинтересны, потому что бесполезны, но полезные примеры ещё слишком сложны для понимания.

Эта книга предлагает другой подход. Она начинается, как это принято, с общей теории о том, как устроен и работает компьютер. При этом внимание уделяется причинам тех или иных технических решений прошлых лет, которые определили основные функции современного компьютера. Знание этих причин поможет вам лучше запомнить материал и понять предмет.

Общие знания о компьютерной технике окажутся полезны, когда вы начнёте программировать на конкретном языке. Вы обязательно столкнётесь с проблемами. Например, программа работает слишком медленно или постоянно завершается с ошибкой. Знание устройства компьютера поможет вам понять причины такого поведения.

Дальше мы познакомимся с языком программирования Bash. Вопреки распространённому мнению, это сложный предметно-ориентированный язык. Однако, ряд задач решается на нём сравнительно легко и лаконично. На примере этих задач мы изучим базовые концепции программирования.

Прежде всего мы рассмотрим Bash как замену графическому интерфейсу пользователя. Вы научитесь выполнять основные операции над файлами и каталогами с помощью текстовых команд. Таким образом вы освоите минимальный синтаксис. Его будет достаточно для написания программ на Bash.

Не стоит рассматривать изучение языка Bash, как необходимое, но бесполезное на практике упражнение. Поверьте, каждый профессиональный программист сталкивается в своей работе с автоматизацией рутинных задач и выполнением команд на Unix-системах. В обоих случаях без знания Bash не обойтись.

Если у вас не получается запустить какой-то пример или выполнить упражнение, не расстраивайтесь. Это означает только то, что материал не был раскрыт в должной мере. Напишите[1] об этом мне, и мы вместе его разберём.

В конец книги приводится список терминов. В нём вы можете уточнить незнакомое понятие, которое встретилось при чтении этой книги.

[1] mailto:petrsum@gmail.com

Общая информация

В этом разделе мы рассмотрим основные принципы работы компьютера. Начнём с операционных систем и истории их возникновения. Познакомимся с возможностями и семействами современных ОС. Затем рассмотрим, что представляет собой компьютерная программа, как она запускается и исполняется.

Операционные системы

Предпосылки возникновения ОС

Большинство пользователей компьютера понимает, зачем нужна **операционная система**[2] (ОС). Покупая или загружая из интернета приложение, вы проверяете его системные требования. В них указаны минимальные параметры аппаратной части компьютера. Кроме этого в требованиях указана ОС, на которой приложение запустится. Получается, что ОС — это программная платформа на которой работают приложения. Но откуда взялось это требование? Почему нельзя просто купить компьютер и запустить на нём приложение без ОС?

Эти вопросы кажутся бессмысленными только на первый взгляд. Подумайте сами: современные ОС универсальны и предлагают пользователю множество функций. Большинство из них каждому конкретному пользователю не нужно. Но эти функции зачастую невозможно отключить. Для их обслуживания ОС активно использует ресурсы компьютера. В результате приложениям пользователя достаётся меньше ресурсов. Из-за этого они работают медленно и зависают.

Обратимся к истории, чтобы выяснить причины возникновения ОС. На самом деле первая коммерческая ОС GM-NAA I/O[3] появилась только в 1956 году для компьютера IBM 704[4]. Все ранние модели компьютеров обходились без ОС. Почему в них не было необходимости?

Главная причина в быстродействии. Например, рассмотрим первый **электромеханический компьютер**[5]. Его сконструировал Герман Холлерит[6] в 1890 году. Компьютер получил название табулятор. Для работы ему не нужна ОС и программы[7] в современном смысле этого слова. Табулятор выполняет ограниченный набор арифметических операций. Эти операции определяет конструкция компьютера. Данные для вычислений загружаются с **перфокарт**[8].

[2] https://ru.wikipedia.org/wiki/Операционная_система#Функции
[3] https://ru.wikipedia.org/wiki/GM-NAA_I/O
[4] https://ru.wikipedia.org/wiki/IBM_704
[5] http://chernykh.net/content/view/16/40/
[6] https://ru.wikipedia.org/wiki/Холлерит,_Герман
[7] https://ru.wikipedia.org/wiki/Компьютерная_программа
[8] https://ru.wikipedia.org/wiki/Перфокарта

Перфокарты представляют собой листки плотной бумаги с пробитыми отверстиями. Эти листки вручную подготавливаются и укладываются в специальные приёмные устройства. Там они нанизываются на иглы. В местах отверстий происходит замыкание электрической цепи. Каждое замыкание увеличивает механический счётчик. Счётчиком служит вращающийся цилиндр. Результаты вычислений выводятся на циферблаты, напоминающие часы.

Иллюстрация 1-1 демонстрирует табулятор, построенный Германом Холлеритом.

Иллюстрация 1-1. Табулятор Холлерита

По современным меркам табулятор работает очень медленно. На это есть несколько причин. Прежде всего данные для вычислений подготавливаются вручную. Во времена табулятора не было способа автоматически пробивать перфокарты. Далее загрузка перфокарт в компьютер также выполняется вручную. Сам табулятор содержит много механических частей: иглы для считывания данных, счётчики из вращающихся цилиндров, циферблаты для вывода результата. Вся эта механика работает медленно. Выполнение одной элементарной операции занимает порядка одной секунды. Никакая автоматизация не способна ускорить эти механические процессы.

Табуляторы выполняют вычисления с помощью вращающихся цилиндров. На смену им пришли компьютеры, работающие на **реле**[9]. Реле — это механический элемент. Он меняет своё состояние под воздействием электрического тока. Один из первых релейных компьютеров[10] Z2 сконструировал немецкий инженер Конрад Цузе[11] в 1939 году. Затем этот компьютер был усовершенствован в 1941 году и получил название Z3. Переход на реле сократил время выполнения одной элементарной операции с секунды до миллисекунд.

Кроме возросшей скорости вычислений, компьютеры Цузе отличает ещё одна особенность. В них появилось понятие программы. Теперь с помощью перфокарт вводятся не исходные данные задачи, а **алгоритмы**[12] по которым она решается. Для ввода данных используется

[9] https://ru.wikipedia.org/wiki/Реле
[10] https://habr.com/ru/company/ua-hosting/blog/386247
[11] https://ru.wikipedia.org/wiki/Цузе,_Конрад
[12] https://ru.wikipedia.org/wiki/Алгоритм

клавиатура. Она отдалённо напоминает печатную машинку. Такие компьютеры стали называться **программируемыми**[13] или универсальными.

Алгоритмом называется конечная последовательность инструкций для выполнения какого-либо вычисления или решения задачи.

Появление программируемых компьютеров стало важным шагом в развитии вычислительной техники. До этого машины выполняли только узкоспециализированные задачи. Это было слишком дорого и неэффективно. Поэтому многие инвесторы избегали вкладывать деньги в проекты по конструированию новых компьютеров. Эти проекты ограничивались только военными разработками в годы Второй мировой войны.

Следующим большим шагом стало создание компьютера **ENIAC**[14] (см. иллюстрацию 1-2) в 1946 году Джоном Эккертом[15] и Джоном Мокли[16]. В качестве рабочих элементов в нём используются не реле, а **электровакуумные лампы**[17]. То есть электромеханические компоненты с большим временем отклика заменили на более быстрые электронные. Это увеличило быстродействие компьютера на порядок. Время выполнения одной элементарной операции сократилось до 200 микросекунд.

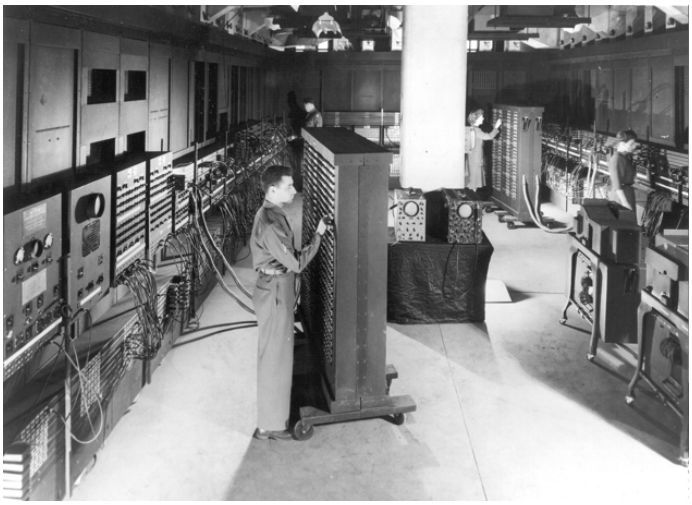


Иллюстрация 1-2. ENIAC

В среде инженеров долго сохранялось скептическое отношение к электровакуумным лампам. Они были известны своей низкой надёжностью и высоким энергопотреблением. Никто не верил, что сконструированная на них машина вообще сможет работать. В ENIAC использовалось около 18 000 ламп. Они часто выходили из строя. Но между их отказами компьютер успешно справлялся с вычислениями. ENIAC стал положительным примером использования ламп. Он переубедил многих конструкторов.

[13] https://ru.wikipedia.org/wiki/Компьютер_общего_назначения
[14] https://ru.wikipedia.org/wiki/ЭНИАК
[15] https://ru.wikipedia.org/wiki/Эккерт,_Джон_Преспер
[16] https://ru.wikipedia.org/wiki/Мокли,_Джон
[17] https://ru.wikipedia.org/wiki/Электронная_лампа

ENIAC — это программируемый компьютер. В нём алгоритм вычислений задаётся с помощью комбинации переключателей и перемычек на коммутационных панелях. Такое программирование долго и трудозатратно. Нужна одновременная работа нескольких человек. На иллюстрации 1-3 изображена одна из панелей для программирования ENIAC.

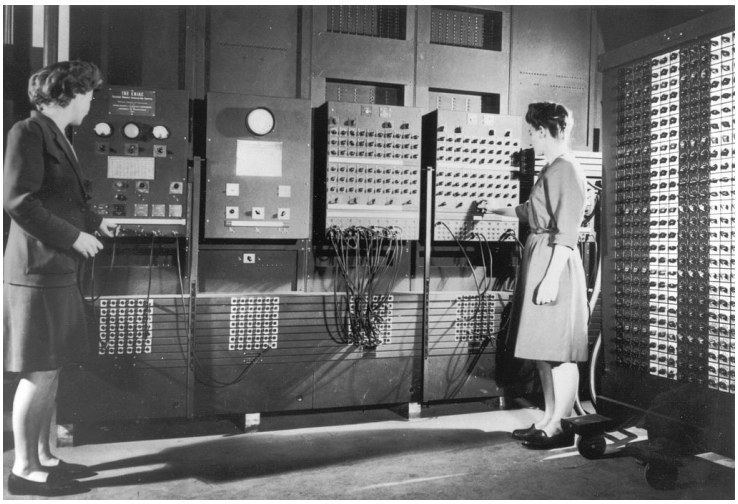


Иллюстрация 1-3. Панель программирования ENIAC

ENIAC использует перфокарты для ввода исходных данных и вывода результатов. Так же работали и предыдущие модели компьютеров. Но в ENIAC появилась новая возможность. Перфокарты могли хранить промежуточные расчёты. Если исходная задача из-за своей сложности не решалась сразу, она разбивалась на несколько подзадач. После выполнения каждой подзадачи её результаты выгружались на перфокарты. Затем компьютер перепрограммировался. После этого перфокарты загружались обратно в качестве входных данных.

Опыт эксплуатации ENIAC показал, что производительность компьютера ограничивают все механические операции. Например, ручное перепрограммирование с помощью переключателей и перемычек, а также чтение и пробивание перфокарт. ENIAC обладал небывалой по тем временам производительностью. Но несмотря на это, прикладные задачи решались на нём медленно. Большую часть времени компьютер простаивал, ожидая программы или входных данных. Опыт работы с ENIAC привел к разработке новых средств ввода и вывода данных.

Вычислительная мощность компьютеров увеличилась на порядок после перехода с электровакуумных ламп на **транзисторы**[18]. Вместе с усовершенствованными средствами ввода-вывода это привело к более интенсивной эксплуатации компьютеров и их частому перепрограммированию. К этому времени вычислительные машины распространились за пределы военных проектов и стали использоваться крупными банками и корпорациями. В результате возросло число и разнообразие запускаемых на них программ.

Часто программы исполнялись друг за другом без задержек. Это исключало простой оборудования. Для автоматизации загрузки программ и вывода их результатов потребовались

[18] https://ru.wikipedia.org/wiki/Транзистор

специальные решения. Именно для управления выполнением программ и была разработана первая ОС GM-NAA I/O.

Интенсивное использование компьютеров и разнообразие программ привело к проблеме управления их исполнением. Была и другая проблема. Дело в том, что функции компьютера определялись загруженной в него программой. Например, если она включает в себя код для управления устройствами ввода-вывода, они доступны. В противном случае устройства не работают. Для конкретной модели компьютера подключенное к нему оборудование менялось редко. Поэтому код для работы с ним копировался. Он кочевал из одной программы в другую, занимая лишнее место на устройствах хранения. Со временем этот код стали выносить в отдельную служебную программу. Она загружалась в компьютер вместе с основной. Постепенно эти служебные программы вошли в состав первых ОС.

Вернёмся к нашему вопросу о необходимости операционных систем. Мы выяснили, что приложения могут работать и без них. Такие программы используются и сегодня. Например, это **утилиты**[19] проверки памяти и разбивки диска, а также некоторые антивирусы. Однако, разработка таких программ требует больше времени и сил. В них приходится включать код для поддержки оборудования, который обычно предоставляет ОС. Разработчики предпочитают использовать возможности ОС. Это уменьшает объём работы и ускоряет выпуск программы.

Возможности ОС

Почему мы начали изучение программирования с рассмотрения ОС? Иллюстрация 1-4 демонстрирует схему взаимодействия ОС с **прикладными программами**[20] и **аппаратным обеспечением**[21]. Прикладные программы — это приложения, которые решают задачи пользователя (например, текстовый редактор, калькулятор, браузер). Аппаратным обеспечением называются все электронные и механические компоненты компьютера (например, клавиатура, монитор, центральный процессор, видеокарта).

[19] https://ru.wikipedia.org/wiki/Утилита
[20] https://ru.wikipedia.org/wiki/Прикладное_программное_обеспечение
[21] https://ru.wikipedia.org/wiki/Аппаратное_обеспечение

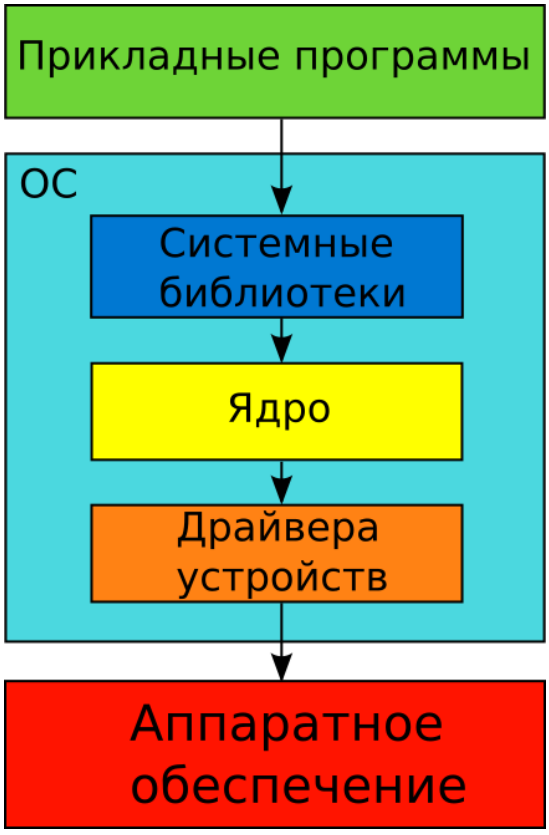


Иллюстрация 1-4. Схема взаимодействия ОС с программами и аппаратным обеспечением

Рассмотрим схему. Приложения получают доступ к аппаратным ресурсам не напрямую, а через **системные библиотеки**[22] ОС. Чтобы получить этот доступ, приложение должно следовать строгим правилам. Например, доступны только те возможности устройств, которые поддерживает ОС. Правила работы с устройствами определяет **интерфейс прикладного программирования**[23]. Он также известен как API (Application Programming Interface). API предоставляют системные библиотеки.

Интерфейс — это набор соглашений о взаимодействии компонентов информационной системы. Интерфейсы описаны в стандартах. Такие стандарты гарантируют совместимость компонентов системы.

API определяет все аспекты взаимодействия приложения с ОС. Например, он отвечает на следующие вопросы:

1. Какую операцию выполнит ОС при вызове конкретной системной функции?
2. Какие данные функция ожидает на вход?
3. Какие данные функция вернёт в качестве результата?

Следовать интерфейсу должна как ОС, так и приложение. Например, в документации сказано: “ОС создаёт файл при вызове функции X”. Не зависимо от версии ОС, она всегда должна

[22] https://ru.wikipedia.org/wiki/Библиотека_(программирование)
[23] (https://ru.wikipedia.org/wiki/API)

следовать этому правилу. Это гарантирует совместимость разных версий приложений и ОС. Такая совместимость невозможна без хорошо документированного и стандартизированного интерфейса.

Мы уже выяснили, что приложения могут работать без ОС. Однако, ОС предлагает готовые решения для взаимодействия с аппаратными ресурсами компьютера. Без этих решений разработчики приложений должны сами отвечать за работу с оборудованием. Это огромная работа. Представьте всё разнообразие комплектующих современных компьютеров. Приложение должно поддерживать все модели устройств (например, видеокарт). В противном случае оно не заработает у некоторых пользователей.

Выясним, какие возможности предоставляет ОС через API интерфейс. Все электронные и механические компоненты компьютера можно рассматривать как ресурсы. Программы используют эти ресурсы для вычислений. Другими словами аппаратное обеспечение выполняет программы. API отражает возможности оборудования, которые доступны программе. Также интерфейс определяет порядок взаимодействия между несколькими программами и оборудованием.

Рассмотрим пример. Две программы не могут одновременно записывать данные на жёсткий диск[24] в одну и ту же область. Во-первых, запись выполняется единственной магнитной головкой жёсткого диска. Во-вторых, после записи данных первым приложением их может затереть второе приложение. Поэтому одновременные запросы программ на запись нужно упорядочить. Обычно их помещают в очередь и исполняют друг за другом. За это отвечает ОС, а точнее её **ядро**[25] (см. иллюстрацию 1-4). В ядре реализована **файловая система**[26]. Похожим образом ОС упорядочивает доступ ко всем **периферийным**[27] и внутренним устройствам компьютера. Этот доступ предоставляется через специальные программы. Они называются **драйверами устройств**[28] (см. иллюстрацию 1-4).

Что такое периферийные устройства, и чем они отличаются от внутренних? К периферийным относятся все устройства, отвечающие за ввод-вывод информации и её постоянное хранение. Примеры: клавиатура, мышь, микрофон, монитор, колонки, жёсткий диск. Внутренние устройства отвечают за обработку информации, то есть непосредственное исполнение программ. К ним относятся **центральный процессор**[29] (central processing unit, CPU), **оперативная память**[30] (random-access memory, RAM), **видеокарта**[31] (graphics processing unit, GPU).

ОС предоставляет не только интерфейс доступа к аппаратным ресурсам. Кроме аппаратных есть ещё и программные ресурсы самой ОС. Это повторяющийся код, ставший со временем служебными программами. Впоследствии его оформили в системные библиотеки (см. иллюстрацию 1-4). Некоторые из библиотек обслуживают устройства. Другие выполняют

[24] https://ru.wikipedia.org/wiki/Жёсткий_диск#Технологии_записи_данных
[25] https://ru.wikipedia.org/wiki/Ядро_операционной_системы
[26] https://ru.wikipedia.org/wiki/Файловая_система
[27] https://ru.wikipedia.org/wiki/Периферийное_устройство
[28] https://ru.wikipedia.org/wiki/Драйвер
[29] https://ru.wikipedia.org/wiki/Центральный_процессор
[30] https://ru.wikipedia.org/wiki/Оперативная_память
[31] https://ru.wikipedia.org/wiki/Видеокарта

полезные операции над входными данными. Например, компонент Windows под названием **интерфейс графических устройств**[32] (Graphical Device Interface или GDI). С его помощью приложения манипулируют графическими объектами. Используя GDI, разработчики создают пользовательский интерфейс для своих программ. К программным ресурсам относятся все компоненты ОС, установленные на компьютере. Кроме них ОС также предоставлять доступ к алгоритмам сторонних приложений или библиотек.

ОС не только управляет ресурсами. Она организует совместную работу запущенных приложений. Запуск приложения — это нетривиальная задача. Её выполняет специальная служебная программа ОС. После запуска приложения, ОС контролирует его выполнение. Если нарушается какое-то ограничение, приложение завершается. Пример нарушения — чтение недоступной памяти. В следующем разделе мы подробно рассмотрим процесс запуска и исполнения программы.

Если ОС многопользовательская, она контролирует доступ к данным. Благодаря этому, каждый пользователь работает только со своими файлами и каталогами.

Подведём итог. ОС выполняет следующие функции:

1. Предоставляет и упорядочивает доступ к аппаратным ресурсам компьютера.
2. Предоставляет программные ресурсы в виде системных библиотек.
3. Запускает приложения, а также отвечает за ввод данных для них и вывод результата.
4. Организует взаимодействие приложений друг с другом.
5. Контролирует доступ пользователей к данным.

Посмотрите внимательно на эти функции ОС. Наверное вы догадались, что без ОС нельзя запустить несколько приложений одновременно. Проблема в том, что их разработчики не знают, в каком сочетании программы будут выполняться. Только ОС имеет достаточно информации, чтобы эффективно распределить ресурсы компьютера в реальном времени.

Современные ОС

Мы познакомились с основными возможностями ОС. Теперь учтём новые знания и рассмотрим современные ОС. Их функции во многом аналогичны. Основные отличия заключаются в способах реализации этих функций. Эти особенности реализации и решения, которые к ним привели, называются **архитектурой**[33].

У современных ОС есть две особенности. Они определяют их поведение и способ взаимодействия с пользователем. Речь идёт о многозадачности и графическом интерфейсе. Рассмотрим их подробнее.

[32] https://ru.wikipedia.org/wiki/GDI
[33] https://ru.wikipedia.org/wiki/Архитектура_программного_обеспечения

Многозадачность

Большинство современных ОС **многозадачны**[34]. Это означает, что они исполняют несколько программ одновременно. Почему это свойство оказалось важным? Системы с этим свойством вытеснили ОС без него.

В 1960-е годы появилась проблема эффективного использования компьютеров. В то время они стоили дорого. Поэтому ценилась каждая минута их работы. Позволить купить себе мейнфрейм могли только крупные компании и университеты. Они считали неприемлемым любой его простой.

Ранние операционные системы исполняли программы друг за другом без задержек. В таких ОС программы и их входные данные подготавливались заранее. Они записывались на устройство хранения (например, магнитную ленту). Эта лента подавалась на устройство чтения компьютера. Он последовательно исполнял программы и выводил их результаты на устройство вывода (например, принтер). Такой режим работы называется **пакетная обработка**[35] (batch processing). Он экономит время на переключение компьютера с одной задачи на другую.

Пакетная обработка увеличила эффективность использования мейнфреймов. Она автоматизировала загрузку программ и частично исключила из этого процесса человека-оператора. Однако, у системы осталось **узкое место**[36]. Вычислительная мощность процессоров значительно выросла. Скорость же работы периферийных устройств почти не изменилась. Поэтому CPU часто простаивал, ожидая ввода-вывода данных.

Узкое место (bottleneck) — компонент или ресурс информационной системы, который ограничивает её общую производительность или пропускную способность.

Рассмотрим пример. Представьте, что мейнфрейм последовательно выполняет программы. Данные для них считываются с магнитной ленты, а результаты печатаются на принтере. ОС загружает каждую программу и исполняет её инструкции, затем загружает следующую и так далее. Проблема возникает на этапах чтения данных и печати результата. Время доступа к данным на магнитной ленте огромно в масштабах центрального процессора. Между двумя операциями чтения, он успел бы выполнить ряд вычислений. Но он этого не делает. Все ресурсы компьютера использует только программа, загруженная сейчас в память. То же происходит с выводом результатов на печать. Принтер — это чисто механическое устройство. Он работает очень медленно.

Проблема простоя центрального процессора привела к идее **мультипрограммирования**[37]. Это означает одновременную загрузку сразу нескольких программ в память компьютера. Первая из них выполняется до тех пор, пока доступны все необходимые ей ресурсы. Как

[34] https://ru.wikipedia.org/wiki/Многозадачность
[35] https://ru.wikipedia.org/wiki/Пакетное_задание
[36] https://ru.wikipedia.org/wiki/Узкое_место
[37] https://ru.wikipedia.org/wiki/Мультипрограммирование

только один из ресурсов оказывается занят, выполнение программы останавливается. Например, ей нужны данные, хранящиеся на жёстком диске. Пока контроллер диска читает первую часть данных, он занят и не может обработать запрос на чтение следующей части. В этом случае ОС прекращает выполнение первой программы и переключается на вторую. Она в свою очередь исполняется до конца или до момента, когда нужный ей ресурс окажется занят. После этого опять происходит переключение задач.

Мультипрограммирование стало прототипом многозадачности, которая реализована в современных ОС. Мультипрограммирование хорошо справляется с режимом пакетной обработки. Однако, этот подход распределения нагрузки не подходит для систем с **интерактивным взаимодействием**[38]. В таких системах действия пользователя являются событиями. Например, нажатие клавиши. Каждое событие обрабатывается сразу. Например, добавление нового символа в текстовый документ. Время отклика системы должно быть минимальным. Иначе пользователь заметит зависания программы.

Проблема мультипрограммирования в том, что момент переключения задач непредсказуем. Это случится только при её завершении программы или её обращении к занятому ресурсу. При это действия пользователя не будут обработаны до момента переключения задач.

Многозадачность решает проблему быстрого отклика при интерактивной работе с компьютером. Способ её реализации постепенно развивался и усложнялся. В современных ОС применяется **вытесняющая многозадачность**[39] с псевдопараллельной обработкой задач. Это означает, что ОС самостоятельно решает, какую программу выполнять в данный момент. При выборе учитываются приоритеты работающих приложений. То есть более приоритетные приложения будут получать аппаратные ресурсы чаще, чем низкоприоритетные. Механизм переключения задач реализован в ядре ОС и называется **планировщиком задач**[40].

Псевдопараллельность обработки означает, что в каждый момент времени выполняется только одна задача. При этом ОС переключается между задачами настолько быстро, что пользователь этого не замечает. Ему кажется, что компьютер выполняет одновременно несколько программ. При это графический интерфейс сразу реагирует на любое действие. Но на самом деле, каждая программа и компонент ОС получают аппаратные ресурсы в строго определённые моменты времени.

Одновременное выполнение программ возможно только на компьютерах с несколькими процессорами или с многоядерными[41] процессорами. На таких компьютерах число одновременно работающих программ примерно равно числу ядер всех процессоров. При этом всё равно работает механизм вытесняющей многозадачности с постоянным переключением задач. Он универсален и балансирует нагрузку на любых системах, независимо от числа ядер. Так выдерживается приемлемое время отклика на действия пользователя.

[38] https://ru.wikipedia.org/wiki/Интерактивность
[39] https://ru.wikipedia.org/wiki/Вытесняющая_многозадачность
[40] https://ru.wikipedia.org/wiki/Диспетчер_операционной_системы
[41] https://ru.wikipedia.org/wiki/Ядро_микропроцессора

Интерфейс пользователя

Современные ОС решают разные задачи. Эти задачи определяет тип компьютера, на котором запускается ОС. Основные типы следующие:

- Персональные компьютеры[42] (ПК) и ноутбуки.
- Мобильные телефоны и планшеты.
- Сервера.
- Встраиваемые системы[43].

Мы рассмотрим только ОС для ПК и ноутбуков. Помимо многозадачности они предоставляют **графический интерфейс пользователя**[44] (graphical user interface или GUI). В данном случе интерфейс — это способ взаимодействия с системой. Через него пользователь запускает приложения, настраивает устройства компьютера и компоненты ОС. Рассмотрим подробнее историю возникновения графического интерфейса.

Пользователи коммерческих компьютеров впервые узнали об интерактивном режиме работы в 1960 году. Его поддерживал новый мини-компьютер[45] PDP-1[46] от компании Digital Equipment Corporation[47]. Почему производители и пользователи компьютеров вообще заинтересовались интерактивностью? В 1950-е годы на рынке мейнфреймов доминировали компьютеры IBM. Они работали в режиме пакетной обработки и хорошо справлялись с вычислительными задачами. Их операционные системы с поддержкой мультипрограммирования автоматизировали загрузку программ и обеспечивали высокую производительность.

Идея интерактивной работы с компьютером появилась в военном проекте SAGE. Он выполнялся по заказу ВВС США. В проекте разрабатывалась автоматизированная система ПВО для обнаружения советских бомбардировщиков. При разработке конструкторы столкнулись с проблемой обработки данных с радаров. Согласно требованиям, компьютер должен был выводить данные в реальном времени. Человек-оператор реагировал на них максимально быстро и отдавал команды. Существующие тогда методы работы с компьютером для этой задачи не подходили. Поскольку в системе ПВО важна скорость реакции на угрозу.

Проект SAGE привёл к созданию первого интерактивного компьютера AN/FSQ-7[48] (см иллюстрацию 1-5). Он выводил данные на **электронно-лучевой монитор**[49]. Команды вводились оператором с помощью **светового пера**[50].

[42] https://ru.wikipedia.org/wiki/Персональный_компьютер
[43] https://ru.wikipedia.org/wiki/Встраиваемая_система
[44] https://ru.wikipedia.org/wiki/Графический_интерфейс_пользователя
[45] https://ru.wikipedia.org/wiki/Мини-компьютер
[46] https://ru.wikipedia.org/wiki/PDP-1
[47] https://ru.wikipedia.org/wiki/Digital_Equipment_Corporation
[48] https://en.wikipedia.org/wiki/AN/FSQ-7_Combat_Direction_Central
[49] https://ru.wikipedia.org/wiki/Кинескоп
[50] https://ru.wikipedia.org/wiki/Световое_перо

Иллюстрация 1-5. Компьютер AN/FSQ-7

Метод интерактивной работы с компьютером стал известен в научных кругах. Он быстро набрал популярность. Пакетная обработка успешно справлялась с выполнением программ. Однако, их разработка и отладка была неудобной. Программист писал алгоритм и записывал его на устройство хранения. Дальше разработчик помещал свою задачу в очередь на выполнение на мейнфрейме. Ожидание в очереди занимало часы. Если после исполнения программы обнаруживалась ошибка, программист её исправлял и снова помещал свою задачу в очередь. В результате исправление всех ошибок даже в небольшой программе занимало дни.

В интерактивном режиме работы программист запускает программу, ожидает её завершения и видит на экране результат. Это в разы увеличивает скорость разработки и отладки приложений. Теперь работа, требующая несколько дней с пакетной обработкой, выполнялась за несколько часов.

Интерактивный режим принёс новые задачи. Этот режим имел смысл, только если система сразу реагировала на действия пользователя. Для этого требовался новый механизм балансирования нагрузки. С этим требованием справился режим многозадачности новых ОС.

Интерактивный режим поддерживают не только многозадачные ОС, но и однозадачные. Пример такой ОС — MS-DOS[51]. Совмещение интерактивности и однозадачности было

[51] https://ru.wikipedia.org/wiki/MS-DOS

нецелесообразно во времена дорогих мейнфреймов. Тогда ресурсами одного компьютера пользовались сразу несколько пользователей. Их программы исполнялись параллельно и независимо друг от друга. Такой режим работы получил название **разделение времени**[52] (time-sharing). Совместить же однозадачность и разделение времени невозможно.

Когда появились первые относительно дешевые персональные компьютеры, на них устанавливались однозадачные ОС. Они требовали меньше аппаратных ресурсов чем их старшие аналоги для мейнфреймов. Несмотря на свою простоту, однозадачные ОС поддерживали интерактивную работу. Для пользователей ПК этот режим стал особенно привлекательным.

Интерактивный режим поставил не только задачу балансировки нагрузки системы. Нужны были новые способы взаимодействия пользователя и компьютера. Существующие в 1960-е годы магнитные ленты и принтеры для этого не подходили.

Телетайп[53] (teletype) стал прототипом устройства для интерактивной работы с компьютером. Иллюстрация 1-6 демонстрирует телетайп Model 33. Он представляет собой электромеханическую печатную машинку. С помощью проводов она подключается к такой же машинке. После соединения двух телетайпов операторы могут передавать друг другу текстовые сообщения. Отправитель набирает текст на своём устройстве. Нажатия клавиш передаются на устройство получателя. Оно печатает каждую принятую букву на бумаге.

[52] https://ru.wikipedia.org/wiki/Разделение_времени
[53] https://ru.wikipedia.org/wiki/Телетайп

Иллюстрация 1-6. Телетайп Model 33

Телетайпы стали подключать к мейнфреймам. Такое устройство называлось **терминалом**[54]. На его клавиатуре пользователь набирал команды. Мейнфрейм их получал, исполнял и отправлял результат обратно. Терминал распечатывал полученные данные на бумаге. Позднее устройство печати заменил монитор. В результате получился **интерфейс командной строки**[55] (command-line interface или CLI). Принцип его работы напоминает телетайп. Пользователь вводит команды одну за другой. Компьютер последовательно их исполняет и выводит на экран результаты.

Иллюстрация 1-7 демонстрирует современный интерфейс командной строки. Это окно **эмулятора терминала**[56] Terminator[57]. В нём запущен интерпретатор командной строки Bash. В окне выведены результаты работы программ ping и ls.

[54] https://ru.wikipedia.org/wiki/Компьютерный_терминал
[55] https://ru.wikipedia.org/wiki/Интерфейс_командной_строки
[56] https://ru.wikipedia.org/wiki/Эмулятор_терминала
[57] https://en.wikipedia.org/wiki/Terminator_(terminal_emulator)

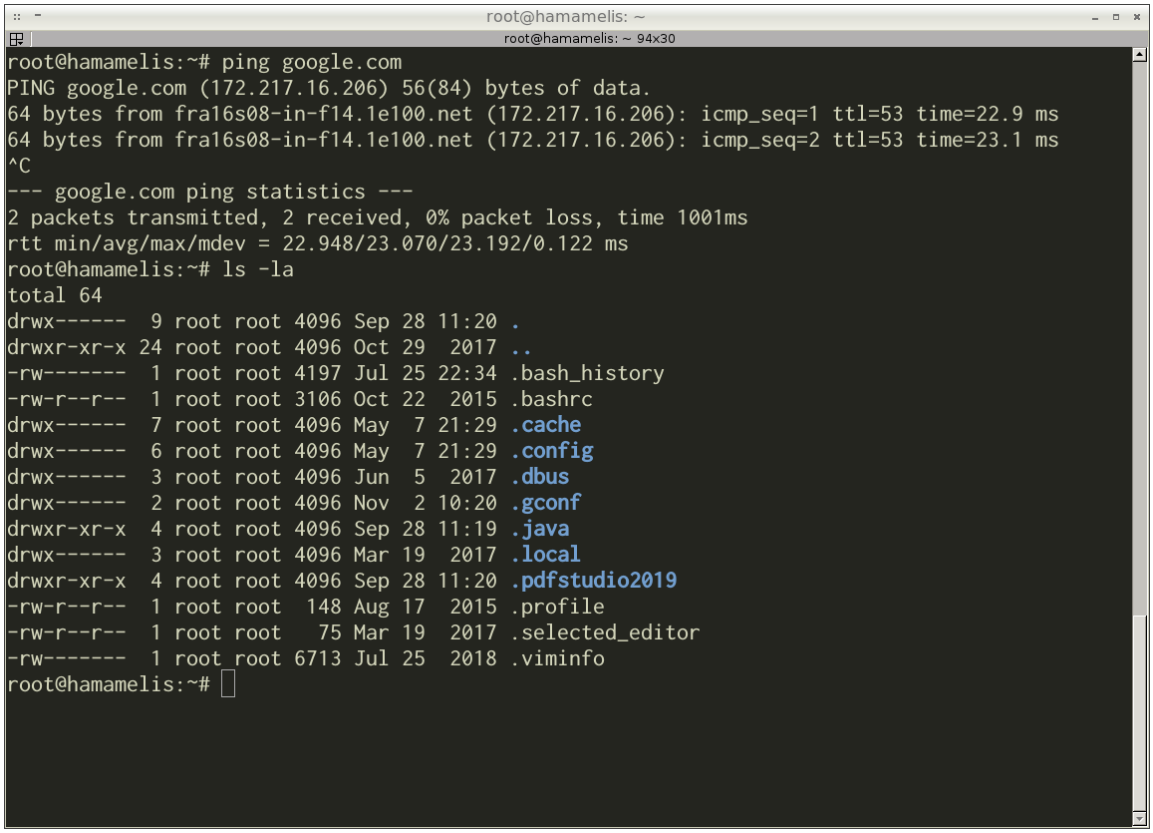


Иллюстрация 1-7. Интерфейс командной строки

Интерфейс командной строки востребован и сегодня. Он имеет ряд преимуществ по сравнению с графическим интерфейсом. Главное достоинство CLI в его нетребовательности к вычислительным ресурсам. Он работает одинаково стабильно и без задержек как на низкопроизводительных встраиваемых компьютерах, так и на мощных серверах. Если применять CLI для удалённого доступа к компьютеру, качество канала связи и его пропускная способность могут быть низкими. Даже с медленным соединением сервер получит команды.

У интерфейса командной строки есть и недостатки. Главная его проблема в сложности освоения. Пользователю доступны сотни команд с различными входными параметрами, которые определяют их режим работы. Требуется немало времени, чтобы запомнить хотя бы часто используемые команды.

Проблему наглядного представления доступных команд решил **текстовый интерфейс пользователя**[58] (textual user interface или TUI). В нём наряду с буквенными и цифровыми символами используется **псевдографика**[59]. Псевдографикой называются специальные символы, с помощью которых на экране отображаются графические примитивы. Примитивы — это линии, прямоугольники, треугольники и т.д. Иллюстрация 1-8 демонстрирует типичный текстовый интерфейс. Это вывод статистики использования системных ресурсов программой htop.

[58] https://ru.wikipedia.org/wiki/Текстовый_интерфейс_пользователя
[59] https://ru.wikipedia.org/wiki/Псевдографика

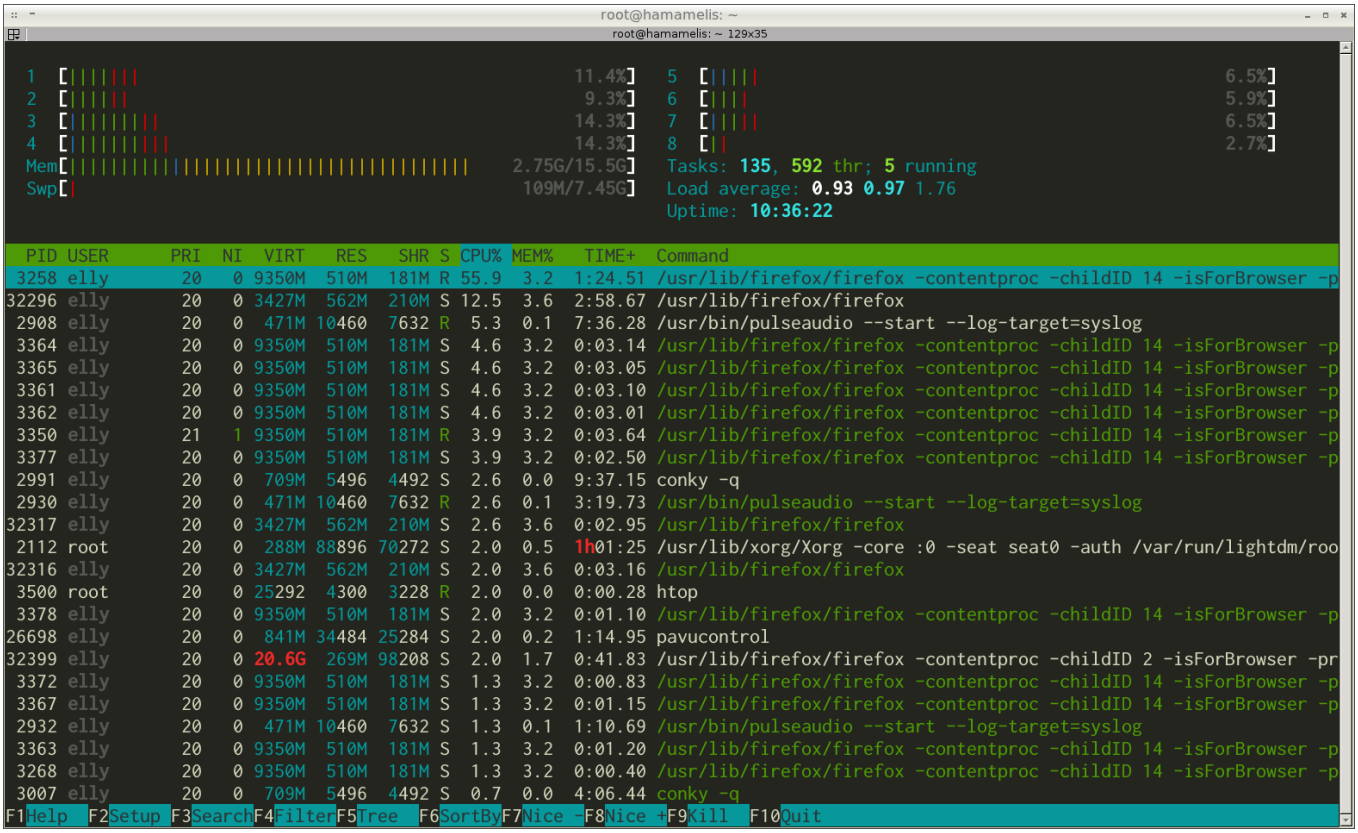


Иллюстрация 1-8. Текстовый интерфейс пользователя

Дальнейший рост производительности компьютеров позволил заменить псевдографику на реальные графические элементы. Примеры таких элементов: окна, иконки, кнопки и т.д. В результате возник полноценный графический интерфейс. Он применяется в современных ОС.

Графический интерфейс ОС Windows приведён на иллюстрации 1-9. Это скриншот рабочего стола. На нём развёрнуты окна трёх приложений: Проводник, Блокнот и Калькулятор. Они работают одновременно.

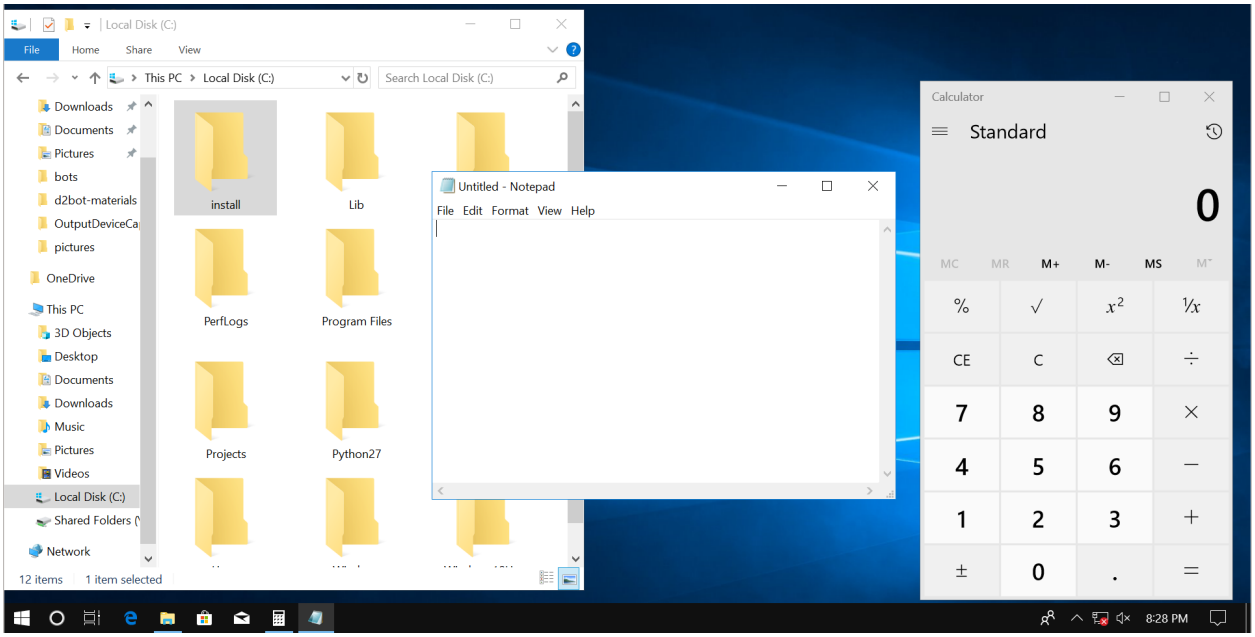


Иллюстрация 1-9. Графический интерфейс пользователя

Первый графический интерфейс предназначался для мини-компьютера Xerox Alto[60] (см. иллюстрацию 1-10). Его разработали в 1973 году в исследовательском центре Xerox PARC[61]. Однако, интерфейс не получил широкого распространения вплоть до 1980-х годов. Он требовал много памяти и высокой производительности компьютера. В то время такие ПК стоили слишком дорого для рядовых пользователей.

Компания Apple выпустила на рынок первый ПК Lisa с графическим интерфейсом только в 1983 году.

[60] https://ru.wikipedia.org/wiki/Xerox_Alto
[61] https://ru.wikipedia.org/wiki/Xerox_PARC

Иллюстрация 1-10. Мини-компьютер Xerox Alto

Семейства ОС

Сегодня на рынке персональных компьютеров доминируют три семейства ОС:

- Windows[62]
- Linux[63]
- macOS[64]

Термин “семейство” означает ряд версий ОС, которые следуют одним и тем же архитектурным решениям. Кроме того большинство функций в этих версиях ОС реализованы одинаково.

Разработчики ОС придерживаются одной и той же архитектуры. Они не предлагают что-то принципиально новое в следующих версиях своего продукта. Почему?

На самом деле изменения в современных ОС происходят, но постепенно и медленно. Причина этого в **обратной совместимости**[65]. Такая совместимость означает, что новые версии

[62] https://ru.wikipedia.org/wiki/Windows
[63] https://ru.wikipedia.org/wiki/Linux
[64] https://ru.wikipedia.org/wiki/MacOS
[65] https://ru.wikipedia.org/wiki/Обратная_совместимость

ОС повторяют функции старых версий. Эти функции нужны для работы существующих программ. На первый взгляд это требование кажется необязательным. Но это серьёзное ограничение для разработки программного обеспечения. Давайте разберёмся, почему это так.

Представьте, что вы написали программу для Windows и продаёте её. Иногда пользователи обнаруживают в программе ошибки. Вы их исправляете. Время от времени вы добавляете новые функции.

Теперь представьте, что вышла новая версия Windows. На ней ваша программа не работает. У ваших пользователей есть два решения:

- Ждать обновления вашей программы, которое совместимо с новой Windows.
- Отказаться от обновления Windows.

Если ваша программа нужна пользователям для ежедневной работы, они откажутся от обновления Windows.

Предположим, что новая Windows принципиально отличается от предыдущей. Это значит, что вашу программу придётся полностью переписать. Посчитайте всё время, которое вы уже потратили на исправление ошибок и добавление новых функций. Эту работу в полном объёме придётся повторить. Скорее всего вы откажетесь от этой идеи и предложите пользователям оставаться на старой версии Windows.

Таких программ как ваша много. Их разработчики придут к тому же решению, что и вы. В результате новая версия Windows окажется никому не нужна. В этом суть проблемы обратной совместимости. Из-за неё и существуют семейства ОС.

Влияние приложений на развитие ОС велико. Например, ОС Windows и персональные компьютеры от IBM обязаны своим успехом табличному процессору Lotus 1-2-3[66]. Он запускался только на ПК от IBM с ОС Windows. Ради Lotus 1-2-3 пользователи покупали и ПК, и ОС. Такие популярные приложения, выводящие платформу на широкий рынок, получили название **killer application**[67] (букв. убойное приложение).

Похожее влияние оказал табличный процессор VisiCalc[68]. Он содействовал распространению компьютеров Apple II[69]. Точно так же бесплатные компиляторы языков C, Fortran и Pascal подогрели интерес к Unix в университетских кругах. За каждой из трёх доминирующих сегодня ОС стоит killer application. Далее эти ОС распространялись, благодаря сетевому эффекту[70]. Разработчики новых приложений выбирали программную платформу, которая уже была у большинства пользователей.

Вернёмся к списку семейств ОС. Windows и Linux примечательны тем, что не привязаны к конкретной аппаратной платформе. Это значит, что вы без проблем установите их на любой

[66] https://ru.wikipedia.org/wiki/Lotus_1-2-3
[67] https://ru.wikipedia.org/wiki/Killer_application
[68] https://ru.wikipedia.org/wiki/VisiCalc
[69] https://ru.wikipedia.org/wiki/Apple_II
[70] https://ru.wikipedia.org/wiki/Сетевой_эффект

персональный компьютер или ноутбук. macOS же запускается только на устройствах Apple. Чтобы установить macOS на другую аппаратную платформу, понадобится неофициальная модифицированная версия[71] ОС.

Совместимость с аппаратной платформой — это пример архитектурного решения. Таких решений много. Все вместе они определяют особенности каждого семейства.

ОС определяет инфраструктуру, доступную программисту. Она диктует инструменты разработки. Например, IDE, компилятор, система сборки. Также ОС навязывает архитектурные решения самим приложениям. Принято говорить о сложившейся культуре написания программ под конкретную ОС. Это важный момент: под разные ОС программы принято разрабатывать по-разному. Постарайтесь его учитывать.

Рассмотрим различие культур разработки программ на примере Windows и Linux.

Windows

Windows — это проприетарная[72] ОС. Исходные коды проприетарных программ закрыты. Вы не сможете их прочитать и изменить. Другими словами нет законного способа узнать про такое ПО больше, чем расскажет его документация.

Чтобы установить Windows на компьютер, вы должны купить её у компании Microsoft. Однако, эта ОС часто предустанавливается на новые компьютеры и ноутбуки. Поэтому её цена включена в конечную стоимость устройства.

Целевой платформой Windows были и остаются относительно дешёвые ПК и ноутбуки. Многие могут позволить себе купить такое устройство. Поэтому рынок потенциальных пользователей огромен. Microsoft стремится сохранить конкурентное преимущество на этом рынке. Компания опасается появления аналогов Windows с такими же возможностями. Microsoft заботится о защите своей интеллектуальной собственности не только техническими, но и юридическими путями. Например, пользовательское соглашение запрещает вам исследовать внутреннее устройство ОС.

Для семейства Windows было написано много прикладных программ. Первые приложения разработала сама компания Microsoft. Например, это пакет офисных приложений Microsoft Office[73] и стандартные приложения Windows[74]. Для сторонних разработчиков они стали образцом для подражания.

Microsoft придерживалась одного и того же принципа, разрабатывая и ОС, и приложения для неё. Это принцип закрытости: исходные коды недоступны пользователям, форматы данных недокументированны, сторонние утилиты не имеют доступа к возможностям ПО. Эти решения защищают интеллектуальную собственность Microsoft.

[71] https://ru.wikipedia.org/wiki/OSx86
[72] https://ru.wikipedia.org/wiki/Проприетарное_программное_обеспечение
[73] https://ru.wikipedia.org/wiki/Microsoft_Office
[74] https://ru.wikipedia.org/wiki/Категория:Стандартные_приложения_Windows

Разработчики программ последовали примеру Microsoft. Они придерживались той же философии закрытости. В результате их приложения получались самодостаточными и независимы друг от друга. Форматы их данных закодированы и недокументированны.

Если вы опытный пользователь компьютера, то сразу узнаете типичное Windows-приложение. Это окно с элементами интерфейса[75] вроде кнопок, полей ввода, вкладок и т.д. В этом окне пользователь манипулирует данными. Например, это текст, изображение или звуковая запись. Результат работы сохраняется на жёсткий диск. Его можно открыть снова в этом же приложении. Если вы напишете собственную Windows-программу, она будет выглядеть и работать похожим образом. Такая преемственность решений и называется культурой разработки под ОС.

Linux

Linux заимствовал идеи и решения ОС Unix[76]. Обе ОС следуют набору стандартов POSIX[77] (Portable Operating System Interface). POSIX определяет интерфейсы взаимодействия прикладных программ с ОС. Следование Linux и Unix одному стандарту привело к их похожему поведению.

ОС Unix возникла в конце 1960-х годов. Её создали два инженера из компании Bell Labs. Это был хобби-проект Кена Томпсона[78] и Денниса Ритчи[79]. В рабочее время они разрабатывали ОС **Multics**[80]. Это был совместный проект Массачусетского технологического института (MIT), компании General Electric (GE) и Bell Labs. Multics планировалась как ОС для нового мейнфрейма GE-645 (см. иллюстрацию 1-11) компании General Electric.

Иллюстрация 1-11. Мейнфрейм модели GE-645

[75] https://ru.wikipedia.org/wiki/Элемент_интерфейса
[76] https://ru.wikipedia.org/wiki/Unix
[77] https://ru.wikipedia.org/wiki/POSIX
[78] https://ru.wikipedia.org/wiki/Томпсон,_Кен
[79] https://ru.wikipedia.org/wiki/Ритчи,_Деннис
[80] https://ru.wikipedia.org/wiki/Multics

В Multics разработчики применили несколько инновационных решений. Одно из них — это разделение времени. То есть мейнфрейм GE-645 стал первым компьютером, на котором одновременно могли работать несколько пользователей. При этом за разделение ресурсов между ними отвечала многозадачность.

ОС Multics оказалась слишком сложной. На её разработку потребовалось больше времени и денег, чем планировалось изначально. Поэтому компания Bell Labs решила выйти из проекта. Но проект был интересен с технической стороны. Поэтому многие инженеры Bell Labs хотели продолжать работу над ним. На этой волне Кен Томпсон решил создать собственную ОС для компьютера GE-645. Он начал писать ядро системы и продублировал некоторые механизмы Multics. Однако, General Electric скоро потребовала вернуть свой GE-645. Ведь Bell Labs получила его только во временное пользование. В результате Кен Томпсон остался без аппаратной платформы для разработки.

Параллельно с работой над аналогом Multics Кен писал компьютерную игру Space Travel[81]. Она запускалась на мейнфрейме General Electric прошлого поколения GE-635. Этот компьютер работал под управлением ОС GECOS[82]. GE-635 представлял собой шкафы с электроникой и стоил около 7 500 000$. Его активно использовали инженеры Bell Labs. Поэтому Кен редко мог с ним работать.

Кен решил портировать свою игру на относительно недорогой и реже используемый коллегами мини-компьютер PDP-7[83] (см. иллюстрацию 1-12). Он стоил около 72 000$. Но возникла одна проблема. Игра Space Travel использовала возможности ОС GECOS. ОС PDP-7 их не предоставляла. К Кену присоединился коллега Деннис Ритчи. Вместе они реализовали возможности GECOS для PDP-7. Это был набор библиотек и систем. Со временем они развились в самостоятельную ОС Unix.

Иллюстрация 1-12. Мини-компьютер PDP-7

[81] https://ru.wikipedia.org/wiki/Space_Travel
[82] https://ru.wikipedia.org/wiki/GCOS
[83] https://ru.wikipedia.org/wiki/PDP-7

Кен и Деннис не собирались продавать свои разработки. Поэтому вопроса о защите интеллектуальной собственности никогда не стояло. Они писали Unix для собственных нужд и распространяли её с открытым исходным кодом. Все желающие могли скопировать и использовать ОС в своих проектах. Изначально круг пользователей ограничивался сотрудниками компании Bell Labs. Позднее AT&T, которой принадлежала Bell Labs, предоставила исходный код ОС высшим учебным заведениям США. Таким образом развитие Unix продолжилось уже в университетских кругах.

ОС Linux создал Линус Торвальдс[84] в 1991 году. В это время он учился в Хельсинкском университете. Линус решал чисто практическую проблему. Ему нужна была полноценная Unix-совместимая ОС для ПК, которой в то время не было.

В Хельсинкском университете студенты выполняли учебные задания на мини-компьютере MicroVAX под управлением Unix. Дома у них были ПК, но Unix на них не запускалась. Для домашнего использования у Unix была альтернатива. Это ОС Minix[85], которую разработал Эндрю Таненбаумом в 1987 году для IBM ПК с процессорами Intel 80268. Minix создавалась исключительно для учебных целей. Поэтому Эндрю отказывался вносить в неё изменения для поддержки более современных компьютеров. Эти изменения привели бы к усложнению системы и сделали бы её непригодной для обучения студентов.

Линус задался целью написать Unix-совместимую ОС для своего нового компьютера IBM с процессором Intel 80386. Её прототипом стала Minix. Как и у создателей Unix, у него не было коммерческих интересов и продавать результаты своего труда он не собирался. Линус разрабатывал ОС для собственных нужд и свободно делился ею со всеми желающими. Так получилось, что Linux стала бесплатной. Она свободно распространялась с исходным кодом через интернет.

На самом деле Linux — это только ядро ОС. Оно предоставляет функции для работы с памятью, файловой системой, периферийными устройствами и управлением процессорным временем. Основные функций системы доступны через свободные пользовательские компоненты GNU[86]. Эти компоненты разрабатывал Ричард Столлман[87] в Массачусетском технологическом институте. Они тоже распространялись бесплатно. Поэтому Линус включил их в первый дистрибутив[88] своей ОС.

У первых версий Linux не было графической подсистемы. Пользователь запускал все приложения из командной строки. Только некоторые сложные приложения имели текстовый интерфейс. Со временем в Linux появилась оконная система X Window System[89]. С её помощью разработчики стали добавлять графический интерфейс своим приложениям.

Культуру написания приложений для Unix и Linux определили условия, в которых развивались эти ОС. Обе системы развивались в университетских кругах. Их пользователями были

[84] https://ru.wikipedia.org/wiki/Торвальдс,_Линус
[85] https://ru.wikipedia.org/wiki/Minix
[86] https://ru.wikipedia.org/wiki/Проект_GNU
[87] https://ru.wikipedia.org/wiki/Столлман,_Ричард_Мэттью
[88] https://ru.wikipedia.org/wiki/Дистрибутив_Linux
[89] https://ru.wikipedia.org/wiki/X_Window_System

преподаватели и студенты ИТ специальностей. Они понимали, как работает ОС и охотно вносили в неё свои исправления.

В культуре Unix предпочитают узкоспециализированные утилиты командной строки. Для каждой задачи есть своя утилита. Она хорошо написана, многократно протестирована и работает максимально эффективно. Если решается сложная задача, одна узкоспециализированная утилита не в состоянии с ней справится. Но если скомбинировать несколько утилит, задача решается быстро и эффективно. Поэтому утилиты принимают входные данные и выводят результаты в открытом формате. Как правило, это текстовые[90] данные. Исходный код утилит всегда доступен для изучения и исправления.

Культура разработки Linux во многом повторяет традиции Unix. Она отличается от стандартов, принятых в Windows. В Windows каждое приложение монолитно и самостоятельно выполняет все свои задачи. Оно не полагается на сторонние утилиты, которые могут оказаться платными или недоступными для пользователя. Каждый разработчик полагается только на себя. Он не в праве требовать от пользователя купить что-то дополнительное для работы его приложения. В Linux же большинство утилит бесплатны, взаимозаменяемы и легко доступны через интернет. Поэтому естественно, что одно приложение потребует загрузить и установить недостающие ему системные компоненты или другое приложение.

Взаимодействие программ принципиально важно в Linux. Даже монолитные графические приложения в Linux обычно предоставляют дополнительный интерфейс командной строки. Таким образом они органично вписываются в экосистему и легко интегрируются с другими утилитами и приложениями.

В Linux сложный вычислительный процесс часто строится на сочетании узкоспециализированных программ. Чтобы это было эффективным, нужны средства для составления общего алгоритма вычислений. Именно для этого была создана командная оболочка[91] Bourne shell[92] и её потомок Bash[93]. В этой книге мы рассмотрим только Bash. Он вытеснил Bourne shell на всех современных Linux-системах.

Нельзя отдать однозначное предпочтение культуре Linux или Windows. Их сравнение давно служит поводом для бесконечных споров. Каждая из культур имеет свои достоинства и недостатки. Например, типичные для Windows монолитные приложения лучше справляются с задачами, требующими интенсивных расчётов. При комбинации узкоспециализированных Linux-утилит в этом случае появляются накладные расходы. Они связаны с запуском утилит и передачей данных между ними. Всё это требует дополнительного времени. В результате задача выполняется дольше.

Сегодня происходит синтез культур Windows и Linux. Всё больше коммерческих приложений портируются на Linux: браузеры, инструменты для разработки программ, игры, мессенджеры и т.д. При этом их разработчики часто не готовы вносить изменения, продиктованные Linux-культурой. Такие изменения требуют времени и сил. Кроме того они усложняют

[90] https://ru.wikipedia.org/wiki/Текстовые_данные
[91] https://ru.wikipedia.org/wiki/Командная_оболочка_Unix
[92] https://ru.wikipedia.org/wiki/Bourne_shell
[93] https://ru.wikipedia.org/wiki/Bash

сопровождение продукта. Вместо одного приложения получается два: под каждую платформу разная версия. Намного проще портировать приложение в том же виде, в каком оно уже работает под Windows. В результате под Linux всё чаще встречаются приложения, выполненные в типичном Windows-стиле. О плюсах и минусах этого процесса можно спорить. Но одно очевидно: чем больше приложений запускается на ОС, тем популярнее она становится благодаря сетевому эффекту.

Подробнее о культуре разработки в Unix и Linux вы узнаете из книги Эрика Реймонда "Искусство программирования в Unix"[94].

Компьютерная программа

Мы познакомились с операционными системами. Они отвечают за запуск и выполнение компьютерных программ. Программы решают прикладные задачи пользователя. Например, текстовый редактор позволяет работать с текстом.

Программа представляет собой набор элементарных шагов или инструкций. Компьютер последовательно выполняет эти шаги. Так он справляется со сложными задачами. Рассмотрим подробнее, как происходит запуск и исполнение программы.

Память компьютера

Инструкции компьютерной программы хранятся на жёстком диске или другом носителе информации в виде файла. Чтобы начать их исполнение, ОС загружает содержимое этого файла в оперативную память. Затем ОС выделяет процессорное время на исполнение программы. В заданные интервалы процессор выполняет инструкции программы.

Разберёмся, как ОС загружает программу в оперативную память. Начнём с общего устройства памяти компьютера.

Память компьютера измеряется в **байтах**[95]. Байт — это минимальный блок информации, на который может ссылаться процессор и загружать в свою память. Процессор способен оперировать и меньшими объёмами информации — битами. **Бит**[96] — это минимальная единица информации, которую нельзя разложить на составные части. Бит представляет собой логическое состояние с двумя возможными значениями. Эти значения интерпретируются разными способами: 0 или 1, истина или ложь, да или нет, + или —, включено или выключено. Представьте себе бит, как выключатель лампы. Он либо замыкает цепь и лампа горит, либо размыкает и лампа выключена. Восемь битов составляют блок в один байт.

[94] https://ru.wikipedia.org/wiki/Философия_Unix#Реймонд:_Искусство_программирования_в_Unix
[95] https://ru.wikipedia.org/wiki/Байт
[96] https://ru.wikipedia.org/wiki/Бит

Упаковка битов в байты вызывает вопросы. Операции над отдельными битами возможны. Почему тогда нельзя ссылаться на конкретный бит в памяти? У этого ограничения есть исторические причины. Первые компьютеры использовались преимущественно для арифметических вычислений. Например, для расчёта баллистических таблиц[97]. Компьютеры оперировали целыми и дробными числами. Чтобы сохранить число в памяти, одного бита недостаточно. Поэтому понадобились блоки памяти, крупнее битов. Такими блоками стали байты. Дальше объединение битов в байты отразилось на архитектуре процессоров. Разработчики процессоров ожидали, что большая часть вычислений выполняется над числами. Поэтому загрузка и обработка всех битов числа за раз увеличивает производительность компьютера на порядок. Так получилось, что процессоры работают только с байтами.

Ещё один вопрос. Почему байт состоит именно из восьми бит? В первых компьютерах размер байта равнялся шести битам[98]. Такого блока хватало для кодирования всех символов английского алфавита в верхнем и нижнем регистре, цифр, знаков пунктуации и математических операций. Со временем этого размера стало недостаточно. Байт расширили до семи битов. Этот момент совпал с появлением ASCII-таблицы[99]. Она стала стандартом для кодирования символов. Именно поэтому ASCII-таблица определяет символы для кодов от 0 до 127, то есть до максимального семибитного числа. Позднее IBM выпустила мейнфрейм IBM System/360[100]. В нём размер байта составлял восемь битов. Такой размер позволял поддерживать старые кодировки символов из прошлых проектов IBM. Такая упаковка битов стала стандартом в отрасли благодаря популярности и широкому распространению IBM System/360.

Часто используемые единицы объёма информации приведены в таблице 1-1.

Таблица 1-1. Единицы объёма информации

Название	Сокращение	Число байтов	Число битов
килобайт	Кбайт	1000	8000
мегабайт	Мбайт	1000000	8000000
гигабайт	Гбайт	1000000000	8000000000
терабайт	Тбайт	1000000000000	8000000000000

В таблице 1-2 приведены распространённые устройства хранения информации и их объёмы.

[97] https://ru.wikipedia.org/wiki/Баллистическая_таблица
[98] https://ru.wikipedia.org/wiki/Шестибитные_кодировки
[99] https://ru.wikipedia.org/wiki/ASCII
[100] https://ru.wikipedia.org/wiki/IBM_System/360

Таблица 1-2. Устройства хранения информации

Устройство хранения	Объём
Дискета 3.5”[101]	1.44 Мбайт
Компакт-диск[102]	700 МБайт
DVD-диск[103]	до 17 Гбайт
USB-флеш-накопитель[104]	до 2 Тбайт
Жёсткий диск[105]	до 16 Тбайт
Твердотельный накопитель[106]	до 100 Тбайт

Мы познакомились с единицами измерения памяти. Теперь вернёмся к исполнению программы. Зачем загружать её в оперативную память? Ведь процессор мог бы читать инструкции программы напрямую с жёсткого диска.

В современном компьютере вся память делится на четыре уровня[107]. Они изображены на иллюстрации 1-13 красными прямоугольниками. Это физическое разделение памяти. То есть каждому уровню соответствуют разные устройства. Единственное исключение — процессор. В кристалле процессора находятся и регистры, и кэш память. Но это разные модули кристалла.

Стрелки на иллюстрации 1-13 означают потоки данных. Передача происходит только между соседними уровнями памяти. Процессор работает только с данными из своих регистров. Если ему нужны данные с дисковой памяти, их загрузка произойдёт так:

1. Дисковая память -> Оперативная память
2. Оперативная память -> Кэш процессора
3. Кэш процессора -> Регистры процессора

Данные записываются на диск в обратном порядке шагов.

[101] https://ru.wikipedia.org/wiki/Дискета
[102] https://ru.wikipedia.org/wiki/Компакт-диск
[103] https://ru.wikipedia.org/wiki/DVD
[104] https://ru.wikipedia.org/wiki/USB-флеш-накопитель
[105] https://ru.wikipedia.org/wiki/Жёсткий_диск
[106] https://ru.wikipedia.org/wiki/Твердотельный_накопитель
[107] https://ru.wikipedia.org/wiki/Иерархия_памяти

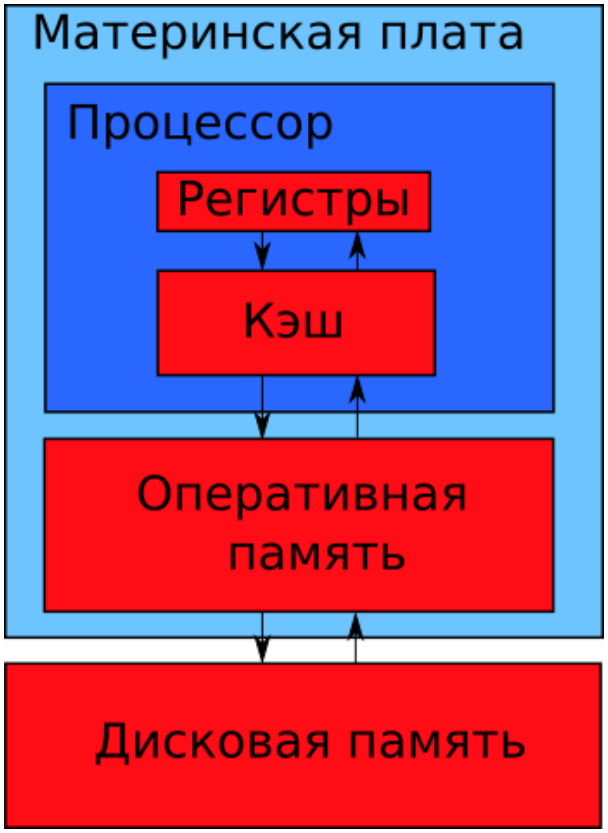


Иллюстрация 1-13. Уровни памяти персонального компьютера

Уровни памяти отличаются друг от друга следующими параметрами:

1. **Скорость доступа** — сколько данных читается или пишется на носитель в единицу времени. Единицы измерения — байты в секунду (байт/с).
2. **Объём** — максимальное количество данных, которое хранит носитель. Измеряется в байтах.
3. **Стоимость** — цена носителя в соотношении к его объёму. Измеряется в долларах или центах за байт или бит.
4. **Время доступа** — время между моментами, когда данные понадобились процессору и когда они стали ему доступны. Измеряется в **тактовых сигналах**[108] процессора.

Таблица 1-3 приводит соотношение параметров разных типов памяти.

[108] https://ru.wikipedia.org/wiki/Тактовый_сигнал

Таблица 1-3. Уровни памяти персонального компьютера

Уровень	Память	Объём	Скорость доступа	Время доступа	Стоимость
1	**Регистры**[109] процессора.	до тысячи байтов	—	1 такт	—
2	**Кэш**[110] память процессора.	от одного килобайта до нескольких мегабайтов	от 700 до 100 гигабайт/сек	от 2 до 100 тактов	—
3	Оперативная память	десятки гигабайтов	10 гигабайт/сек	до 1000 тактов	$10-^{9}$/байт
4	Дисковая память (**жёсткие диски**[111] и **твёрдотельные накопители**[112])	терабайты	2000 мегабайт/сек	до 10000000 тактов	$10-^{12}$/байт

Таблица 1-3 вызывает вопросы. Скорость доступа к дисковой памяти огромна. Почему нельзя сразу читать данные с диска в регистры? На самом деле скорость чтения не так важна. Главное — как долго простаивает процессор, дожидаясь доступа к запрошенным данным. Это время доступа к памяти измеряется в числе тактовых сигналов или тактах. Такт синхронизирует выполнение всех операций процессора. Одна инструкция программы выполняется в течении одного или нескольких тактов.

Предположим, что процессор читает инструкции программы напрямую с жёсткого диска. В этом случае простейшие алгоритмы выполнялись бы неделями. Причём большую часть этого времени процессор бы простаивал в ожидании операций чтения. Иерархическая организация памяти на порядки ускоряет доступ к данным, необходимым процессору.

Представьте, что процессор исполняет программу. Она читает файл с диска и выводит его содержимое на экран. Согласно иллюстрации 1-13, данные с диска сначала загружаются в оперативную память. Затем по частям они загружаются в кэш процессора и оттуда в его регистры. После этого CPU вызывает функцию из системной библиотеки ОС. В неё он передаёт содержимое файла. Функция обращается к драйверу видеокарты. Он выводит данные на экран.

Проблема может возникнуть, когда процессор вызвал функцию и передаёт в неё данные. Если они ещё не загружены в регистры из кэша, то CPU проведёт в ожидании от 2 до 100 тактов (согласно таблице 1-3). Аналогично, если данные ещё не загружены из RAM в кэш,

[109] https://ru.wikipedia.org/wiki/Регистр_процессора
[110] https://ru.wikipedia.org/wiki/Кэш_процессора
[111] https://ru.wikipedia.org/wiki/Жёсткий_диск
[112] https://ru.wikipedia.org/wiki/Твердотельный_накопитель

то время ожидания вырастет на порядок (до 1000 тактов). Предположим, что читаемый файл оказался слишком большим. Он не поместился целиком в оперативную память. Тогда CPU может обратиться к части файла, которая ещё не загружена в RAM. В этом случае время простоя CPU увеличится на 4 порядка (до 10000000 тактов). Для сравнения за это время процессор мог бы выполнить около 1000000 инструкций программы.

За загрузку данных в кэш процессора отвечает механизм кэширования. Пример с чтением файла показал, как дорого обходится каждая ошибка этого механизма. Она называется **промахом**. Помните об иерархии памяти и учитывайте её, разрабатывая алгоритмы. Некоторые алгоритмы и структуры данных приводят к большему числу промахов, чем другие.

Чем меньше время доступа к памяти, тем ближе она к процессору. Это демонстрирует иллюстрация 1-14. Например, внутренняя память CPU (регистры и кэш) находится внутри его кристалла. Оперативная память (RAM) расположена на **материнской плате**[113] рядом с процессором. Они соединяются высокочастотной **шиной данных**[114]. Дисковая память подключается к материнской плате через относительно медленную шину. Пример такой шины — **SATA**[115].

За загрузку данных из RAM в кэш процессора отвечает системный контроллер. Он называется **северный мост**[116]. В ранних версиях персональных компьютеров это был отдельный чип на материнской плате. Технология изготовления процессоров развивалась. В результате северный мост стали встраивать в кристалл процессора.

За чтение данных с жёсткого диска в оперативную память отвечает контроллер под названием **южный мост**[117].

[113] https://ru.wikipedia.org/wiki/Материнская_плата
[114] https://ru.wikipedia.org/wiki/Шина_данных
[115] https://ru.wikipedia.org/wiki/SATA
[116] https://ru.wikipedia.org/wiki/Северный_мост_(компьютер)
[117] https://ru.wikipedia.org/wiki/Южный_мост_(компьютер)

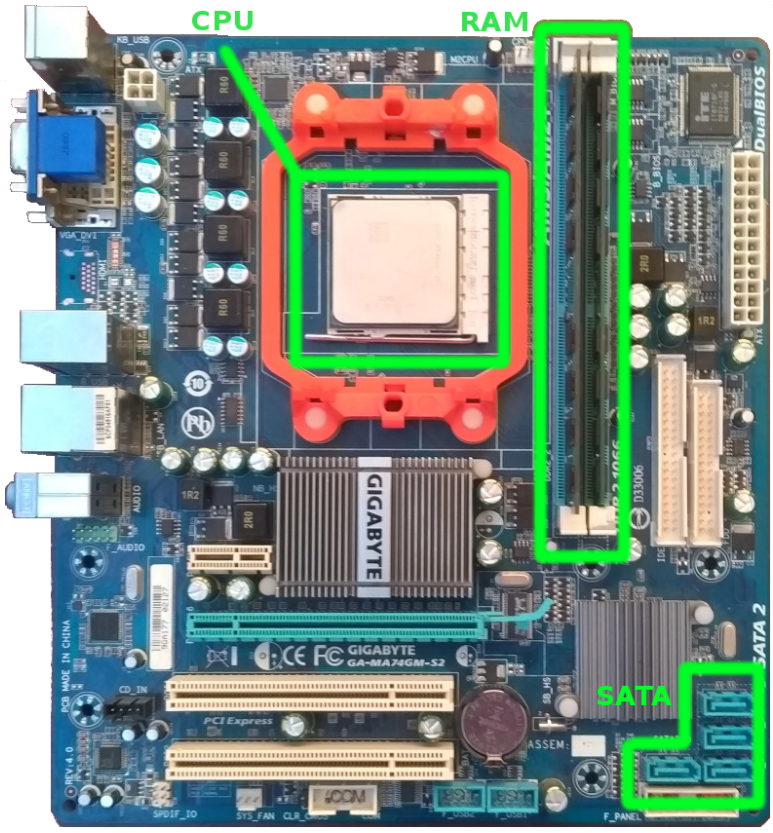


Иллюстрация 1-14. Материнская плата ПК

Машинный код

Предположим, что ОС загрузила содержимое исполняемого файла в оперативную память. В этом файле хранятся не только инструкции программы, но и данные для её работы. Примеры данных: текстовые строки, иконки, картинки, предопределённые константы и т.д.

Инструкции программы называются **машинным кодом**[118]. У процессора есть разные логические блоки. Каждый блок выполняет свой тип инструкций. Набор блоков определяет, какие операции поддерживает CPU. Если процессор не имеет отдельного блока для выполнения операции, она выполняется комбинацией блоков. Такое исполнение займёт больше времени и ресурсов. Одна инструкция считается элементарной операцией над данными в регистрах CPU.

После загрузки программы в оперативную память CPU начинает её исполнять. Исполнение программы называется **вычислительным процессом**[119] (process). К процессу также относятся ресурсы, которые использует работающая программа. Это область памяти и объекты ОС.

[118] https://ru.wikipedia.org/wiki/Машинный_код
[119] https://ru.wikipedia.org/wiki/Процесс_(информатика)

Есть специальные программы для чтения и редактирования исполняемых файлов. Они называются **Hex-редакторами**[120]. Такие редакторы представляют машинный код программы в **шестнадцатеричной системе счисления**[121]. На самом деле в исполняемом файле хранится **двоичный код**[122]. Этот код представляет собой последовательность нулей и единиц. Hex-редактор для удобства чтения переводит их в шестнадцатеричный формат. Именно в двоичном коде процессор получает инструкции и данные.

Одно и то же число в разных системах счисления выглядит по-разному. Система счисления определяет, какие символы и в каком порядке используются при записи числа. Например, двоичная система допускает только символы 0 и 1.

Таблица 1-4 приводит соответствие чисел в двоичной (binary, BIN), десятичной (decimal, DEC) и шестнадцатеричной (hexadecimal, HEX) системах счисления.

Таблица 1-4. Числа в системах счисления DEC, HEX и BIN

Десятичная	Шестнадцатеричная	Двоичная
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Для перевода из одной системы счисления в другую используйте стандартный калькулятор Windows. Функция перевода доступна в **режиме “Программист”**[123].

Почему двоичная и шестнадцатеричная система используются в программировании? На двоичной системе и булевой алгебре строится современная **цифровая техника**[124]. В цифровой технике элементарный носитель информации — это электрический **сигнал**[125]. Простейший

[120] https://ru.wikipedia.org/wiki/Hex-редактор
[121] https://ru.wikipedia.org/wiki/Шестнадцатеричная_система_счисления
[122] https://ru.wikipedia.org/wiki/Двоичный_код#Примеры_двоичных_чисел
[123] https://ru.wikipedia.org/wiki/Калькулятор_(Windows)#Режим_«Программист»
[124] https://ru.wikipedia.org/wiki/Цифровые_технологии
[125] https://ru.wikipedia.org/wiki/Сигнал

способ его закодировать — различать состояния, когда он есть и когда его нет. Наличие сигнала кодируется единицей, а отсутствие — нулём. Для такого кодирования достаточно одного бита.

Базовый элемент в цифровой технике — это **логический вентиль**[126]. Он преобразовывает электрические сигналы. Роль логического вентиля способны выполнять разные элементы. Это могут быть электромагнитные реле, электровакуумные лампы или транзисторы. Все эти устройства работают одинаково с точки зрения обработки сигналов. Эта обработка состоит из двух действий:

1. Получить на вход один или более сигналов.
2. Передать результирующий сигнал на выход.

Такая обработка выполняется по правилам **булевой алгебры**[127]. Она также известна как **алгебра логики**[128]. Каждой операции булевой алгебры соответствует логический вентиль. Если соединить их последовательно, получается сложное преобразование сигналов. По сути центральный процессор — это огромная сеть логических вентилей.

Двоичная система счисления позволяет работать с цифровой техникой на самом низком уровне. Это уровень электрических сигналов. Получается, что устройство аппаратуры навязывает нам двоичную систему.

Аппаратура работает в двоичной системе счисления. Зачем тогда понадобилась шестнадцатеричная система? На самом деле программисты используют либо десятичную систему, либо двоичную. Первая удобна при написании высокоуровневой логики программы. Например, для подсчёта повторений одного и того же действия. Двоичная система нужна для взаимодействия с аппаратурой. Например, для подготовки и передачи данных на устройство. У двоичной системы есть проблема. Её неудобно записывать, читать, запоминать и произносить. Перевод же из десятичной в двоичную сложен. Проблему перевода решает шестнадцатеричная система. Она так же компактна и удобна, как и десятичная. Перевод из шестнадцатеричной в двоичную систему и обратно выполняется в уме.

Рассмотрим перевод числа из двоичной системы в шестнадцатеричную. Для этого разбейте число на группы по четыре разряда, начиная с конца. Если последняя группа оказалась меньше четырёх разрядов, дополните её слева нулями. Затем по таблице 1-4 каждую четвёрку разрядов замените на шестнадцатеричное число. Вот пример перевода:

```
1 110010011010111 = 110 0100 1101 0111 = 0110 0100 1101 0111 = 6 4 D 7 = 64D7
```

[126] https://ru.wikipedia.org/wiki/Логический_вентиль
[127] https://ru.wikipedia.org/wiki/Булева_алгебра
[128] https://ru.wikipedia.org/wiki/Алгебра_логики

Упражнение 1-1. Перевод чисел из BIN в HEX

```
Переведите следующие числа из двоичной системы в шестнадцатеричную:
* 10100110100110
* 10110001110101000010011
* 11111011100010010101001100000000110101101
```

Упражнение 1-2. Перевод чисел из HEX в BIN

```
Переведите следующие числа из шестнадцатеричной системы в двоичную:
* FF00AB02
* 7854AC1
* 1E5340ACB38
```

В последнем разделе книги вы найдёте ответы на все упражнения. Если вы не уверены в своём результате, сверьтесь с ответами.

Вернёмся к нашему исполняемому файлу. Кроме него в оперативную память загружаются все необходимые для работы приложения библиотеки (в том числе системные). За эту процедуру отвечает **загрузчик программ Windows**[129]. Благодаря предварительной загрузке библиотек, процессору не приходится ждать, когда программа к ним обращается. Код нужной библиотеки уже находится в памяти и доступен CPU в течении нескольких сотен тактов. После окончания работы загрузчика Windows программа считается процессом. CPU исполняет её, начиная с первой инструкции.

Пока программа выполняется, её инструкции, ресурсы и библиотеки занимают область RAM. После завершения программы эта область памяти очищается. С этого момента её могут использовать другие приложения.

Исходный код

Машинный код — это низкоуровневое представление программы. Такой формат инструкций и данных удобен для процессора. Однако, человеку писать программу в таком виде неудобно. Эта проблема стала ещё актуальнее с увеличением мощности компьютеров и усложнением программ. Проблему решают специальные приложения двух типов: **компиляторы**[130] и **интерпретаторы**[131]. Они выполняют одну и ту же задачу: переводят текст программы с удобного человеку языка в машинный код.

Сегодня программы пишут на **языках программирования**[132]. Они отличаются от **естественных языков**[133], на которых общаются люди. Языки программирования очень ограничены. На них можно выразить только действия, которые способен выполнить компьютер.

[129] https://ru.wikipedia.org/wiki/Загрузчик_программ
[130] https://ru.wikipedia.org/wiki/Компилятор
[131] https://ru.wikipedia.org/wiki/Интерпретатор
[132] https://ru.wikipedia.org/wiki/Язык_программирования
[133] https://ru.wikipedia.org/wiki/Естественный_язык

Также эти языки отличаются строгими правилами. Например, допустим небольшой набор слов. Сочетать слова можно только в определённом порядке. Текст программы, записанный на языке программирования, называется **исходным кодом**[134].

Компилятор и интерпретатор работают с исходным кодом по-разному. Отличие в моменте, когда машинный код генерируется из исходного. Компилятор читает текст программы целиком, генерирует инструкции процессора и сохраняет результат в файл на диске. При этом программа не исполняется. Интерпретатор читает исходный код по частям, генерирует инструкции процессора и сразу же их исполняет. Результат работы интерпретатора временно хранится в оперативной памяти.

Рассмотрим пример компиляции программы. Предположим, что вы написали и сохранили её исходный код в файл на жёстком диске. Дальше вы запускаете подходящий компилятор. Для каждого языка программирования есть свой компилятор или интерпретатор. Результат компиляции программы сохраняется в исполняемый файл на диске. Файл содержит машинный код. Он соответствует исходному коду программы. Теперь для выполнения программы достаточно запустить её исполняемый файл.

Иллюстрация 1-15 демонстрирует процесс компиляции программы, написанной на языке C или C++.

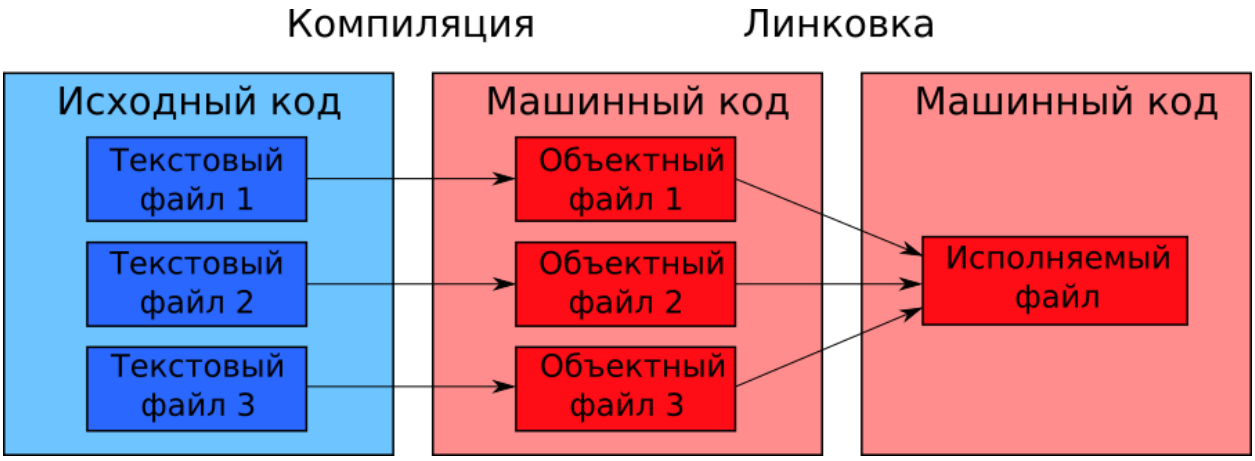


Иллюстрация 1-15. Компиляция программы

Компиляция состоит из двух этапов. Первый этап выполняет компилятор. Второй этап называется **линковка**. Его выполняет специальная программа **компоновщик**[135].

Компилятор создаёт промежуточные **объектные файлы**. Их использует линковщик, чтобы создать исполняемый файл. Почему нельзя объединить компилятор и линковщик в одну программу? У такого решения есть несколько проблемы. Первая связана с ограниченным размером оперативной памяти. Исходный код программы принято разбивать на несколько файлов. Компилятор обрабатывает их по отдельности и записывает результаты на диск в объектные файлы. Это промежуточные результаты компиляции. Если объединить компилятор

[134] https://ru.wikipedia.org/wiki/Исходный_код
[135] https://ru.wikipedia.org/wiki/Компоновщик

и линковщик, сохранять промежуточные результаты на диск не получится. Всю программу придётся компилировать целиком за раз. В этом случае может не хватить оперативной памяти.

Вторая проблема заключается в разрешении зависимостей. В исходном коде есть блоки команд, которые обращаются друг к другу. Чтобы сопоставить такие перекрёстные вызовы, компилятору нужны дополнительные проходы по всему коду программы. Это увеличивает время компиляции в разы. Линковщик решает эту задачу быстрее.

Возможны случаи, когда в исходном коде вызываются функции из библиотеки. Тогда она подаётся на вход линковщика вместе с объектными файлами. Компилятор не может обработать библиотеку. Поскольку, она поставляется в машинном, а не в исходном коде. Разделение компиляции на два этапа решает эту проблему.

Теперь предположим, что для перевода исходного кода программы вы выбрали интерпретатор. В этом случае файл с исходным кодом уже готов для исполнения. Чтобы его запустить, ОС сначала загружает интерпретатор. Далее интерпретатор читает файл с исходным кодом с диска в оперативную память. Затем он исполняет файл строка за строкой. При этом преобразования команд исходного кода в машинный выполняются в оперативной памяти. Некоторые интерпретаторы сохраняют на диск файлы с промежуточным представлением программы. Это нужно для оптимизации. Но так или иначе программу всегда исполняет интерпретатор.

Процесс интерпретации программы приведён на иллюстрации 1-16.

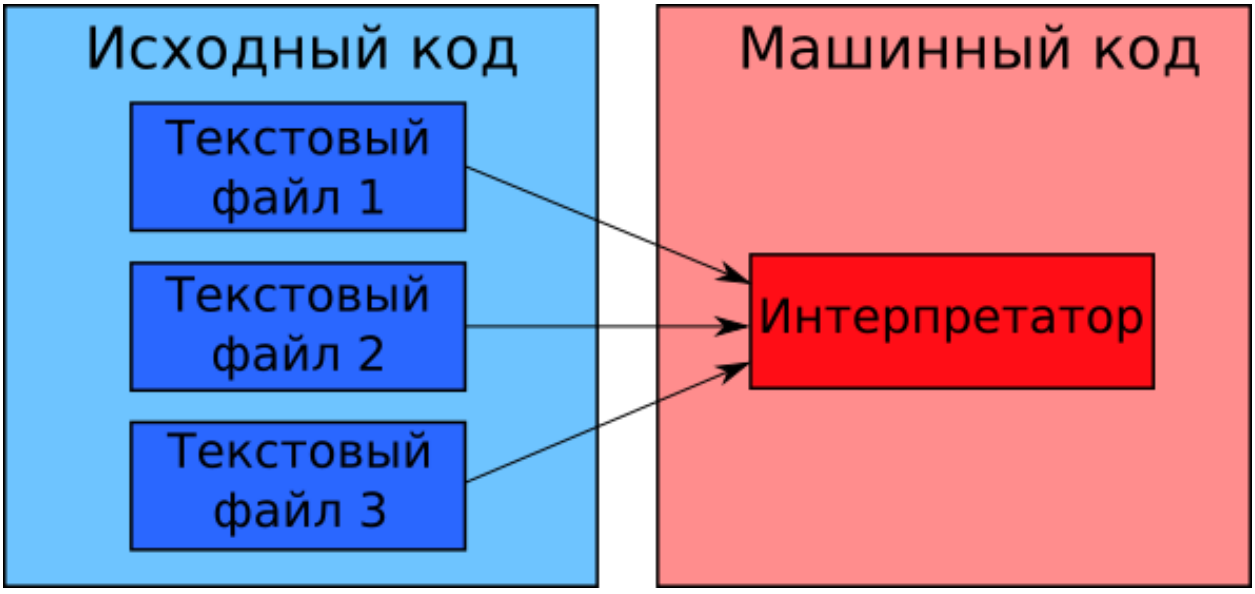


Иллюстрация 1-16. Интерпретация программы

Схема на иллюстрации 1-16 выглядит так, словно интерпретатор работает как компилятор, объединённый с линковщиком. Интерпретатор загружает текстовые файлы в оперативную память и переводит их в машинный код. Почему при этом не возникает проблем с объёмом

RAM и перекрёстными вызовами блоков кода?

Интерпретатор обрабатывает исходный код не так как компилятор. Он читает и выполняет программу строка за строкой. Это значит, что ему не нужно хранить в памяти машинный код всего приложения. Достаточно обрабатывать текстовые файлы с исходным кодом по мере надобности. Для экономии оперативной памяти, интерпретатор периодически удаляет уже выполненные команды.

Все интерпретаторы работают медленно. Загрузка исходного кода программы с диска в RAM приводит к простоям процессора. Согласно таблице 1-3, эта загрузка занимает до 10000000 тактов. Кроме того сам интерпретатор — это сложная программа. Для работы она требует часть аппаратных ресурсов компьютера. Получается, что в дополнение к вашей программе компьютер параллельно выполняет инструкции интерпретатора. Это лишние накладные расходы. Они замедляют работу программы.

Интерпретация программ обходится дорого. А что насчёт компиляции? Компилятор генерирует исполняемый файл с машинным кодом. Поэтому скорость выполнения скомпилированной программы такая же, как и написанной вручную на машинном коде. Однако, вы платите за удобство языка программирования на этапе компиляции. Чтобы скомпилировать небольшую программу, достаточно пары секнуд и нескольких мегабайтов RAM. Но компиляция больших проектов (например, ядро ОС) занимает несколько часов. Помните про накладные расходы, при выборе языка программирования. Одни задачи лучше решает интерпретатор, другие — компилятор.

Стоит ли вообще использовать языки программирования? Время на компиляцию проекта можно потратить на написание программы в машинном коде. Кажется, что так вы сэкономите время и ресурсы компьютера. Пример поможет оценить преимущество языков программирования. Листинг 1-1 демонстрирует исходный код программы на языке C. Она выводит на экран текст “Hello world!”.

Листинг 1-1. Исходный код программы на языке C

```c
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
}
```

Листинг 1-2 приводит ту же программу в виде машинного кода в шестнадцатеричном представлении.

Листинг 1-2. Машинный код программы

```
BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9
CD 20 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21
```

Даже если вы не знаете язык C, код из листинга 1-1 выглядит понятнее, чем из листинга 1-2. Его проще прочитать и отредактировать. Возможно, профессионал сможет быстро написать машинный код для небольшой программы. Но чтобы разобраться в нём, другому программисту понадобиться много времени и сил.

Язык программирования удобнее и выразительнее машинного кода. С его помощью программы проще разрабатывать и поддерживать.

Командный интерпретатор Bash

Программирование — это прикладной навык. Чтобы его освоить, выберем язык программирования и начнём решать на нём практические задачи. Так вы получите некоторые навыки и познакомитесь с базовыми принципами.

В этой книге мы будем писать на Bash. Он удобен для автоматизации задач администрирования компьютера. Например, с ним вы сможете:

- Создать резервные копии данных.
- Выполнить операции с каталогами[136] и файлами.
- Запускать программы и передавать данные между ними.

Bash появился в ОС Unix. Он несёт на себе отпечаток Unix-философии[137]. Bash также доступен на Windows и macOS.

Инструменты для разработки

Для запуска примеров из этой главы нужны интерпретатор Bash и эмулятор терминала. Они устанавливаются на все современные ОС. Рассмотрим, как это сделать.

Интерпретатор Bash

Bash — это **скриптовый язык программирования**[138]. Такие языки:

1. Интерпретируются, а не компилируются.
2. Оперируют готовыми программами или высокоуровневыми командами.
3. Интегрированы в командную оболочку или ОС.

Язык Bash интегрирован в Linux и macOS. Если вы используете Windows, установите минимальное Unix-окружение. Оно нужно Bash для корректной работы. Рассмотрим два способа установки окружения.

Термины “Unix-окружение” и “Linux-окружение” означают программную среду, совместимую со стандартами POSIX.

[136] https://ru.wikipedia.org/wiki/Каталог_(файловая_система)
[137] https://ru.wikipedia.org/wiki/Философия_Unix
[138] https://ru.wikipedia.org/wiki/Сценарный_язык

Первый вариант — установить набор инструментов **MinGW**[139]. Помимо интерпретатора Bash он предоставляет свободный набор компиляторов GCC[140]. Для примеров этой книги будет достаточно компонента MinGW под названием MSYS (Minimal SYStem). Этот компонент включает: интерпретатор Bash, эмулятор терминала и утилиты командной строки GNU[141]. Вместе они составляют минимальное Unix-окружение.

Установим версию Unix-окружения под названием MSYS2[142]. Перед её установкой уточните разрядность вашей ОС Windows. Для этого выполните следующее:

1. Если на вашем рабочем столе есть иконка “Компьютер”, нажмите на неё правой кнопкой мыши и выберите пункт “Свойства”.
2. Если на рабочем столе нет иконки “Компьютер”, нажмите кнопку “Пуск”. Найдите в открывшемся меню пункт “Компьютер”. Нажмите на него правой кнопкой мыши и выберите “Свойства”.
3. В открывшемся окне “Система” найдите пункт “Тип системы” как на иллюстрации 2-1. В этом окне написана разрядность вашей Windows.

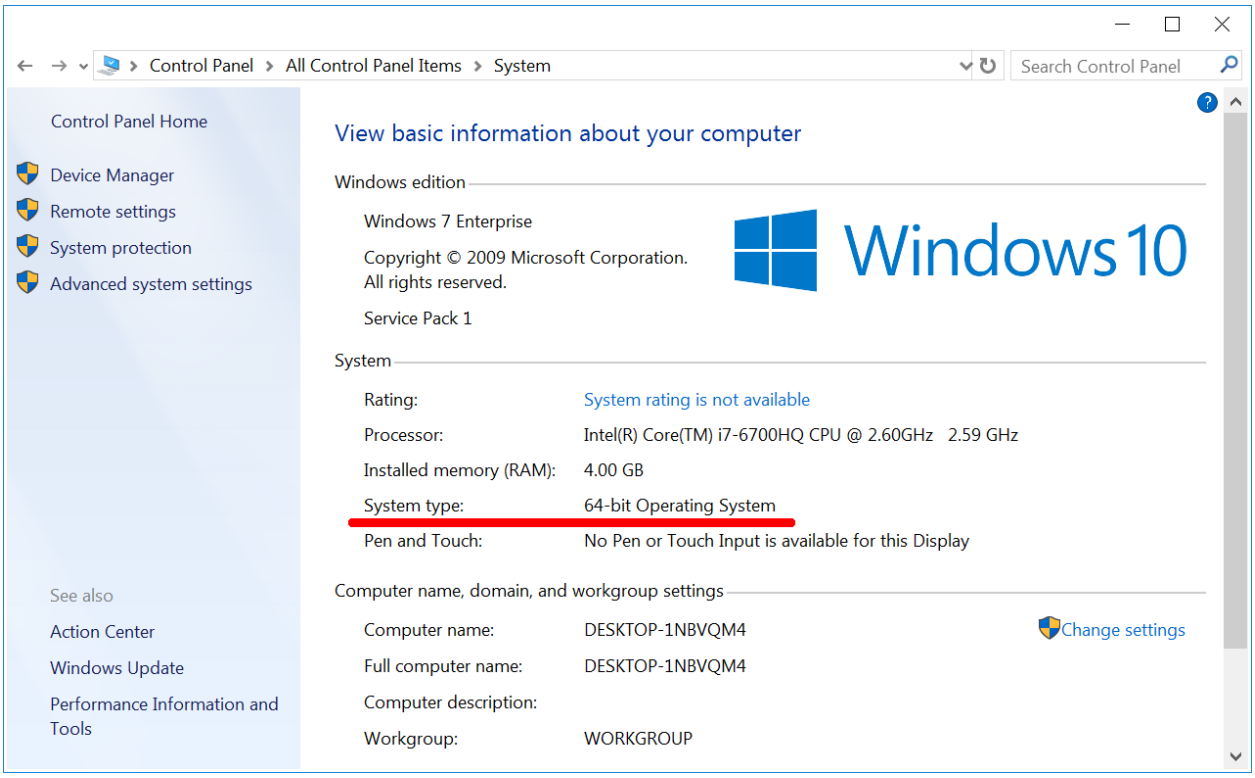


Иллюстрация 2-1. Тип системы

[139] https://ru.wikipedia.org/wiki/MinGW#Компоненты_MinGW
[140] https://ru.wikipedia.org/wiki/GNU_Compiler_Collection
[141] https://ru.wikipedia.org/wiki/GNU_Coreutils
[142] https://www.msys2.org

Скачайте установщик MSYS2 с официального сайта[143]. Файл установщика зависит от разрядности вашей системы:

- `msys2-i686-20190524.exe` для 32-разрядной Windows.
- `msys2-x86_64-20190524.exe` для 64-разрядной Windows.

Число `20190524` в имени файла означает версию MSYS2. Выберите последнюю доступную версию.

Теперь установим MSYS2. Для этого сделайте следующее:

1. Запустите установщик. Откроется окно, как на иллюстрации 2-2.

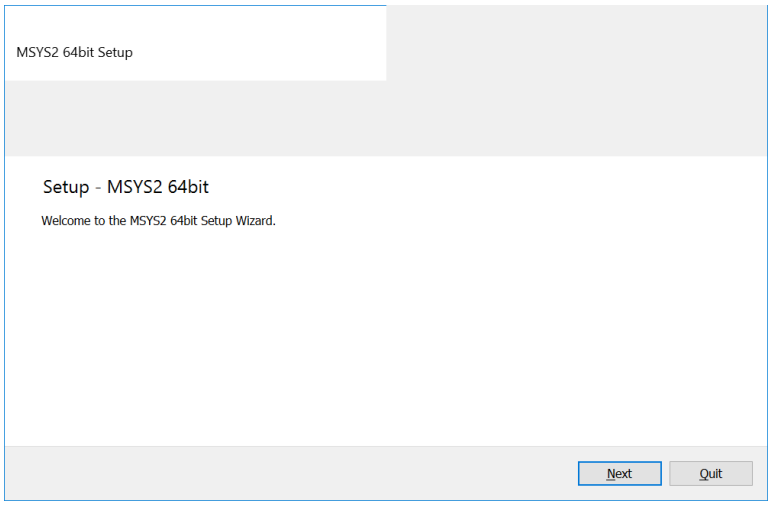


Иллюстрация 2-2. Диалог установки MSYS2

2. Нажмите кнопку “Next” (далее). В новом окне (см. иллюстрацию 2-3) выберите **путь**[144] установки и нажмите кнопку “Next”.

[143] https://www.msys2.org
[144] https://ru.wikipedia.org/wiki/Путь_к_файлу

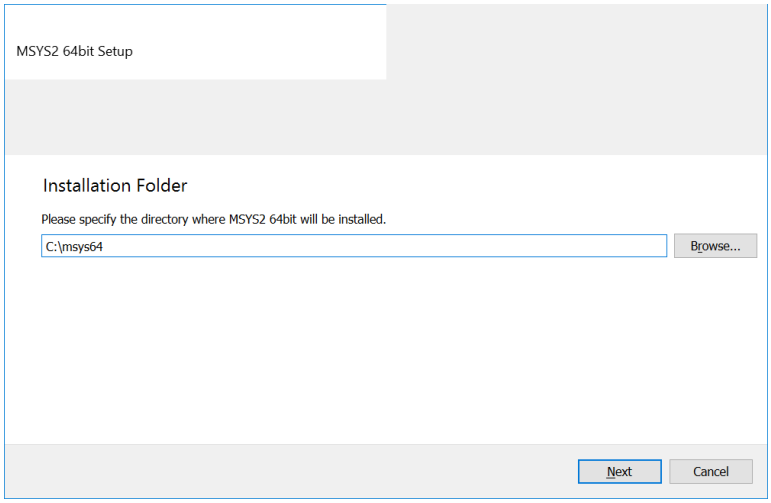


Иллюстрация 2-3. Выбор пути установки

3. Следующее окно предлагает выбрать имя приложения для меню “Пуск”. Оставьте его без изменений и нажмите “Next”. После этого начнётся процесс установки.
4. После завершения установки нажмите кнопку “Finish” (завершить). Окно закроется.

Unix-окружение MSYS2 установлено на ваш жёсткий диск. Его файлы находятся в каталоге `C:\msys64`, если вы оставили путь установки по умолчанию. Перейдите в этот каталог и запустите приложение `msys2.exe`. Откроется окно интерпретатора командной строки Bash как на иллюстрации 2-4.

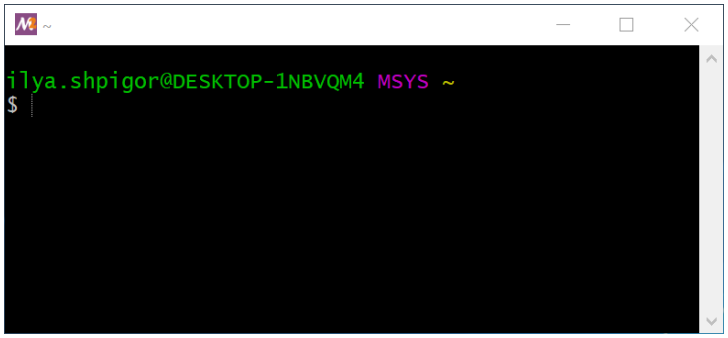


Иллюстрация 2-4. Окно интерпретатора командной строки Bash

Второй вариант — установить Unix-окружение от Microsoft под названием Windows-подсистема для Linux[145] (Windows subsystem for Linux или WSL). Это окружение доступно только для Windows 10. Оно не заработает на Windows 8 и 7. Инструкция установки WSL доступна на сайте Microsoft[146].

Пользователям Linux и macOS устанавливать Bash не надо. Он доступен в этих системах по умолчанию.

[145] https://ru.wikipedia.org/wiki/Windows_Subsystem_for_Linux
[146] https://docs.microsoft.com/ru-ru/windows/wsl/install-win10

Чтобы открыть окно интерпретатора Bash в Linux, нажмите комбинацию клавиш Ctrl+Alt+T.

Для запуска Bash в macOS сделайте следующее:

1. Запустите программу поиска Spotlight. Для этого нажмите на иконку лупы в правом верхнем углу экрана.
2. Появится диалог. Введите в нём текст “Terminal”.
3. В открывшемся списке приложений щёлкните мышью на первой строчке с именем “Terminal”.

Эмулятор терминала

После запуска приложения `msys2.exe` откроется окно эмулятора терминала. **Эмулятор**[147] — это программа, которая имитирует поведение другой программы, ОС или устройства. Эмуляторы нужны для совместимости. Например, вы хотите запустить Windows-программу на Linux. Для этого установите на Linux эмулятор Windows-окружения под названием Wine[148]. Wine предоставляет свою версию системных библиотек Windows. Благодаря эмулятору, Windows-программа запустится на Linux.

Эмулятор терминала также решает задачу совместимости. Приложения с интерфейсом командной строки предназначены для работы через устройство терминал. Сегодня это устройство нигде не используется. Его вытеснили персональные компьютеры и ноутбуки. Чтобы запустить программу, работающую только через терминал, вам нужен эмулятор терминала. Благодаря эмулятору, программа сможет передавать команды в командную оболочку и выводить на экран результат их исполнения.

Иллюстрация 2-5 демонстрирует взаимодействие устройств ввода-вывода, эмулятора терминала, оболочки и программы.

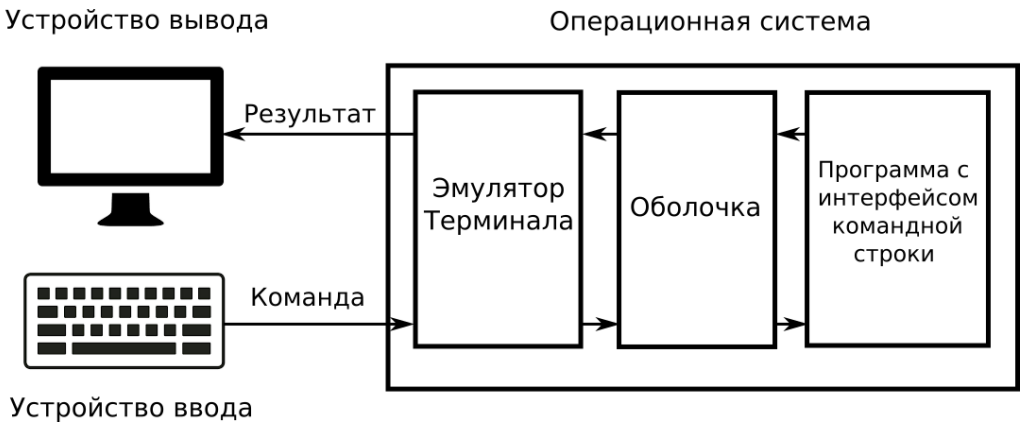


Иллюстрация 2-5. Роль эмулятора терминала

В окне терминала сразу после запуска выводятся две строчки (см. иллюстрацию 2-4):

[147] https://ru.wikipedia.org/wiki/Эмуляция
[148] https://ru.wikipedia.org/wiki/Wine

```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Первая строка начинается с имени пользователя. В моём случае это `ilya.shpigor`. Далее через символ @ следует имя компьютера: `DESKTOP-1NBVQM4`. Его можно изменить в настройках Windows. Затем через пробел идёт слово `MSYS`. Это название платформы, на которой запущен Bash. В конце строки стоит символ ~. Это **абсолютный путь** до текущего каталога. Остановимся на этом пункте подробнее.

Есть два типа путей до объектов файловой системы: абсолютные и относительные. Пути первого типа выводит Проводник Windows в адресной строке. Абсолютным называется путь к одному и тому же объекту файловой системы вне зависимости от текущего каталога. **Относительный путь** напротив указывается по отношению к текущему каталогу.

Относительные пути короче абсолютных. Поэтому их быстрее набирать и удобнее применять в командных интерпретаторах. Чтобы отличить тип пути в Unix-окружении, есть простое правило. Абсолютные пути начинаются с символа слеш /. Например, `/c/Windows/system32`. Относительные пути начинаются с имени каталога. Например, `Windows/system32`.

Командный интерпретатор

Интерпретаторы работают в двух режимах: не интерактивном и интерактивном. В первом режиме интерпретатор исполняет программу. Программа загружается с диска в оперативную память. Затем она построчно исполняется.

В интерактивном режиме пользователь вводит команды в окне эмулятора терминала[149]. После нажатия клавиши Enter, команда исполняется. Интерпретатор с интерактивным режимом работы называется **командной оболочкой**[150] (shell).

Командная оболочка часто встраивается в ОС. Она предоставляет доступ к настройкам и функциям ОС. В Linux через оболочку запускаются программы и системные сервисы, управляются периферийные и внутренние устройства, происходят обращения к ядру ОС.

Востребованность

Зачем сегодня изучать интерфейс командной строки (CLI)? Его создавали 40 лет назад для компьютеров, которые в тысячи раз медленнее современных. На ПК и ноутбуках давно доминирует графический интерфейс.

Кажется, что CLI — устаревшая технология, давно отжившая свой век. Это утверждение ошибочно. Не просто так Bash входит во все дистрибутивы[151] macOS и Linux. В Windows

[149] https://ru.wikibooks.org/wiki/Введение_в_администрирование_UNIX/Командная_строка_UNIX#Терминал_и_командная_строка
[150] https://ru.wikipedia.org/wiki/Интерпретатор_командной_строки
[151] https://ru.wikipedia.org/wiki/Дистрибутив_операционной_системы

тоже есть командный интерпретатор Cmd.exe[152]. В 2006 году компания Microsoft заменила его на новый PowerShell[153]. Задумайтесь над этим фактом. Разработчик самой популярной ОС для ПК создаёт новую командную оболочку. Значит, польза от CLI всё-таки есть.

Какие задачи решает командная оболочка в современных ОС? Прежде всего это инструмент для администрирования системы[154]. В состав ОС кроме ядра входят программные модули: библиотеки, сервисы и утилиты. У них есть настройки и специальные режимы работы. Большинство настроек и режимов не нужны рядовому пользователю. Это дополнительные возможности. Поэтому графический интерфейс не даёт к ним доступа.

Если ОС выходит из строя, дополнительные возможности нужны для её восстановления. Кроме того при сбое системы графический интерфейс часто не работает. Поэтому все утилиты восстановления имеют интерфейс командной строки.

Интерпретатор Bash интегрирован в Linux и macOS. Через него доступны функции этих ОС. С Windows Bash не работает. Вместо него Microsoft предлагает свою оболочку PowerShell. В этой книге мы изучаем Bash. Во-первых, он совместим со стандартом POSIX. Во-вторых, Bash встречается чаще чем PowerShell и доступен на всех современных ОС.

Кроме задач администрирования командный интерфейс нужен для подключения к компьютерам по сети[155]. Для такого подключения есть графические программы: TeamViewer, Remote Desktop и другие. Но они требуют стабильного и быстрого сетевого соединения. Если соединение ненадёжное, эти программы работают медленно и связь постоянно теряется. Командный интерфейс не требователен к качеству соединения. Даже с медленным каналом связи удалённый компьютер получит и исполнит команду.

Знание командного интерфейса полезно не только администраторам, но и рядовым пользователям. С командной оболочкой намного быстрее выполнять ежедневные задачи. Например, следующие:

- Операции над файлами и каталогами.
- Создание резервных копий данных.
- Загрузка файлов из интернета.
- Сбор статистики об использовании ресурсов компьютера.

Рассмотрим пример. Предположим, вы переименовываете файлы на диске. К их именам добавляется один и тот же суффикс. Для десятка файлов это легко сделать через графическое приложение Проводник Windows[156]. Но допустим, что файлов несколько тысяч. Тогда работа

[152] https://ru.wikipedia.org/wiki/Cmd.exe
[153] https://ru.wikipedia.org/wiki/PowerShell
[154] https://ru.wikipedia.org/wiki/Системный_администратор
[155] https://ru.wikipedia.org/wiki/Программы_удалённого_администрирования
[156] https://ru.wikipedia.org/wiki/Проводник_Windows

через Проводник займёт целый день. С помощью оболочки эта задача решается одной командой за пару секунд.

Пример с переименованием файлов показал сильную сторону CLI — масштабируемость. Масштабируемость в общем смысле означает, что одно и то же решение одинаково хорошо справляется с большим и малым объёмом входных данных. В случае командной оболочки решение — это команда. Она одинаково быстро обрабатывает десять файлов и тысячу.

Знание командного интерфейса пригодится программисту. Разрабатывая сложное приложение, приходится изменять много файлов с исходным кодом. Для работы с ними есть редакторы с графическим интерфейсом. Но иногда одно и то же изменение надо внести в несколько файлов. Например, изменить заголовок с информацией о лицензии. В этом случае работа с редактором неэффективна. Утилиты командной строки решат эту задачу намного быстрее.

Навыки работы с CLI нужны для запуска компиляторов и интерпретаторов. Эти программы, как правило, не имеют графического интерфейса. Они запускаются через командную строку. На вход они принимают имена файлов с исходным кодом. Файлов может быть много, поэтому графический интерфейс плохо подходит для их обработки.

Интегрированные среды разработки[157] (integrated development environment или IDE) компилируют программы через графический интерфейс. На самом деле IDE — это только обёртка над командным интерфейсом компилятора. Он вызывается, когда вы нажимаете кнопку в IDE. Чтобы изменить режим работы компилятора в этом случае, придётся столкнуться с интерфейсом командной строки.

Если вы опытный программист, командная строка поможет вам разработать вспомогательные утилиты. Дело в том, что приложение с текстовым интерфейсом быстрее и проще писать чем аналогичное с GUI. Скорость разработки важна, когда решаются одноразовые задачи.

Допустим, у вас появилась задача. Она решается многократным повторением одного и того же действия. В этом случае посчитайте: сколько времени нужно на ручную работу и сколько на написание утилиты для автоматизации. Писать утилиту с графическим интерфейсом долго. Решить задачу вручную окажется быстрее. Но если выбрать командный интерфейс, автоматизировать задачу станет выгоднее. Так вы сэкономите время и исключите ошибки из-за ручной работы.

Стоит ли изучать интерфейс командной строки, решать вам. Я только привёл примеры из практики, которые говорят о пользе этого навыка. Перейти с графического интерфейса на командную строку тяжело. Придётся заново научиться многим вещам, которые вы привыкли делать через Проводник Windows. Но освоившись с командной оболочкой, вы удивитесь насколько продуктивнее стала ваша работа на компьютере.

[157] https://ru.wikipedia.org/wiki/Интегрированная_среда_разработки

Навигация по файловой системе

Знакомство с Unix-окружением и Bash мы начнём с **файловой системы**[158] (ФС). Файловой системой называется способ хранения и чтения информации с дисков. Сначала рассмотрим отличия структуры каталогов в Unix и Windows. Затем познакомимся с Bash-командами для навигации по файловой системе.

Структура каталогов

В верхней части окна Windows Проводника находится адресная строка. Она выводит абсолютный путь к текущему каталогу. В терминологии Windows каталоги называются **папками**[159]. Оба названия обозначают один и тот же объект файловой системы.

Иллюстрация 2-6 приводит окно Проводника. В нём открыт путь `This PC › Local Disk (C:) › msys64`. Это каталог `msys64` на диске `C`. Буква `C` обозначает локальный системный диск. Локальный означает физически подключённый к компьютеру. Системный диск — это тот, на который установлена ОС Windows. Если перевести адресную строку Проводника в абсолютный путь, получим `C:\msys64`.

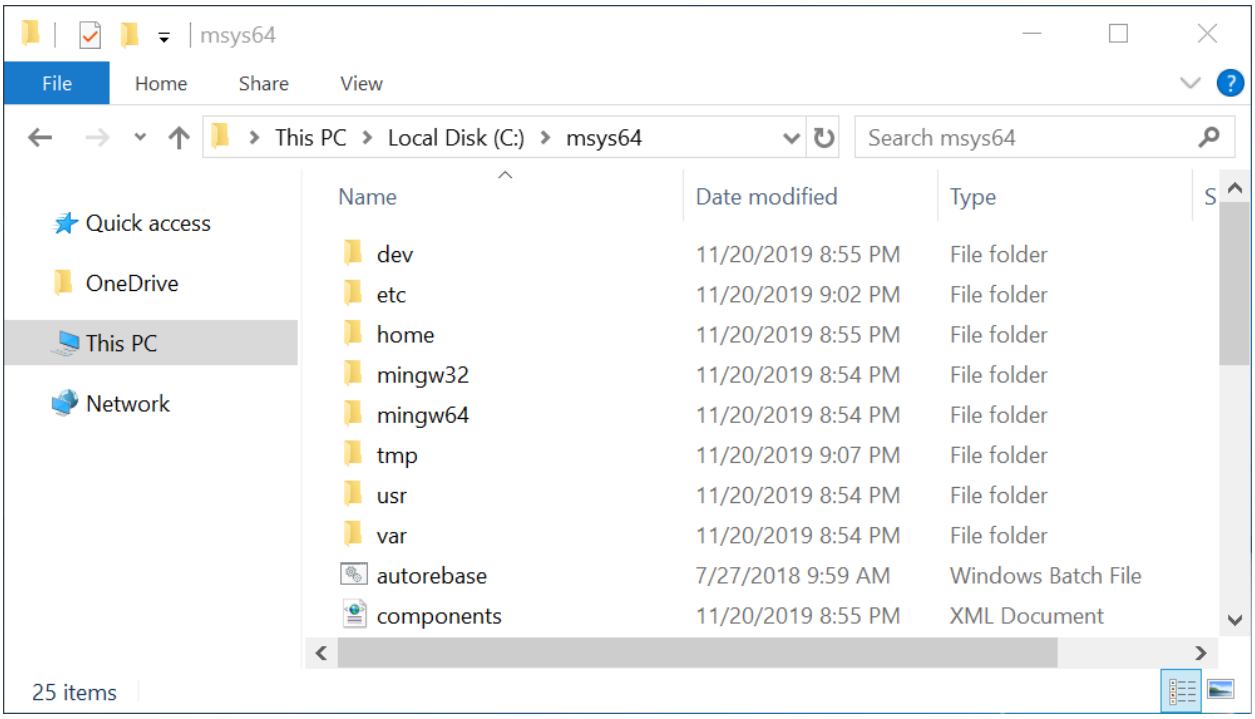


Иллюстрация 2-6. Окно Проводника Windows

[158] https://ru.wikipedia.org/wiki/Файловая_система
[159] https://ru.wikipedia.org/wiki/Каталог_(файловая_система)#Термин_«Папка»

В окне терминала выводится текущий абсолютный путь. Это работает так же как адресная строка Проводника. Но пути в терминале и Проводнике различаются. Причина в отличии структуры каталогов Unix-окружения и Windows.

В Windows каждому диску соответствует буква латинского алфавита. Диск открывается через Проводник как обычная папка. Тогда можно работать с его содержимым. Для примера рассмотрим системный диск `C`. Windows в процессе установки создаёт на нём стандартный набор каталогов[160]:

- `Windows`
- `Program Files`
- `Program Files (x86)`
- `Users`
- `PerfLogs`

В этих каталогах хранятся компоненты ОС и их временные файлы.

Помимо системного диска к компьютеру можно подключить дополнительные диски. Windows обозначает их следующими буквами латинского алфавита: `D`, `E`, `F` и т. д. Структуру каталогов на дополнительных дисках задаёт пользователь. Windows не накладывает на неё никаких ограничений.

Структуру каталогов Windows определяет файловая система File Allocation Table[161] (FAT). Компания Microsoft разработала её для ОС MS-DOS[162]. Впоследствии принципы работы FAT легли в основу стандарта ECMA-107[163]. Система NTFS[164] сменила устаревшую FAT в современных версиях Windows. Но из-за требований обратной совместимости структура каталогов в NTFS осталась без изменений.

Структуру каталогов Unix определяет стандарт POSIX[165]. Согласно стандарту, в системе есть каталог самого верхнего уровня. Он называется **корневым каталогом**[166] и обозначается символом слэш /. Каталоги и файлы всех подключенных к компьютеру дисков находятся внутри корневого каталога.

Чтобы получить доступ к содержимому диска, его надо смонтировать. **Монтированием** называется встраивание содержимого диска в корневой каталог системы. После монтирования содержимое диска становится доступно по какому-то пути. Этот путь называется **точкой монтирования**[167]. Если перейти в точку монтирования, вы окажетесь в файловой системе диска.

[160] https://en.wikipedia.org/wiki/Directory_structure#Windows_10
[161] https://ru.wikipedia.org/wiki/FAT
[162] https://ru.wikipedia.org/wiki/MS-DOS
[163] http://www.ecma-international.org/publications/standards/Ecma-107.htm
[164] https://ru.wikipedia.org/wiki/NTFS
[165] https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap10.html#tag_10
[166] https://ru.wikipedia.org/wiki/Каталог_(файловая_система)#Корневой_каталог
[167] https://ru.wikipedia.org/wiki/Точка_монтирования

Сравним структуру каталогов Windows и Unix на примере. Предположим, что к компьютеру с Windows подключены два локальных диска `C` и `D`. Тогда структура каталогов первого уровня иерархии выглядит так, как в листинге 2-1.

Листинг 2-1. Структура каталогов первого уровня в ОС Windows

```
C:\
    PerfLogs\
    Windows\
    Program Files\
    Program Files (x86)\
    Users\

D:\
    Documents\
    Install\
```

В Unix эта же иерархия каталогов выглядит иначе. Её демонстрирует листинг 2-2.

Листинг 2-2. Структура каталогов в ОС Unix

```
/
    c/
        PerfLogs/
        Windows/
        Program Files/
        Program Files (x86)/
        Users/

    d/
        Documents/
        Install/
```

Запустив терминал MSYS2, вы попадаете в Unix-окружение. В нём Windows-пути не работают. Вместо них используйте Unix-пути. Например, каталог `C:\Windows` теперь доступен по пути `/c/Windows`.

В Unix-окружении важен регистр символов[168]. Это значит, что строки `Documents` и `documents` не равны. В Windows нет чувствительности к регистру. Поэтому если в адресной строке Проводника написать путь `c:\windows`, вы перейдёте в системный каталог `C:\Windows`. В Unix-окружении это не сработает. Все символы надо вводить в правильном регистре.

Кроме регистра символов есть ещё одно отличие. В Unix имена каталогов и файлов в пути разделяет слэш /. В Windows для этого используют обратный слэш .

[168] https://ru.wikipedia.org/wiki/Чувствительность_к_регистру_символов

Команды навигации по файловой системе

Как выполнить команду в эмуляторе терминала? Для этого переключитесь на его окно, наберите текст команды и нажмите клавишу Enter. Оболочка обработает ваш ввод. Когда она готова к вводу, на экран выводится **приглашение командной строки**. Приглашение — это специальный символ или строка символов. Если приглашения нет, оболочка занята и не может выполнить команду.

На иллюстрации 2-4 в приглашение обозначается символом доллара $.

Для навигации по файловой системе через Проводник Windows, возможны следующие действия:

- Вывести текущий каталог.
- Перейти в указанный каталог.
- Найти каталог или файл на диске.

Эти же действия доступны через интерфейс командной строки. Каждое из них выполняет специальная команда. Эти команды приведены в таблице 2-1.

Таблица 2-1. Команды и утилиты для навигации по файловой системе

Команда	Описание	Примеры
ls	Вывести на экран содержимое каталога. Если каталог не указан, выводится содержимое текущего.	`ls` `ls /c/Windows`
pwd	Вывести на экран путь до текущего каталога. Ключ комнды `-W` выводит путь в структуре каталогов Windows	`pwd` `pwd -W`

cd	Перейти в каталог по относительному или абсолютному пути.	`cd tmp`
		`cd /c/Windows`
		`cd ..`

mount	Смонтировать диск в корневую файловую систему. При запуске без параметров выводит список всех смонтированных дисков.	`mount`
find	Найти файл или каталог. Первый параметр команды — это каталог, начиная с которого ведётся поиск. Если он не указан, используется текущий каталог.	`find . -name vim` `find /c/Windows -name *vim*`

grep	Найти файл по его содержимому.	`grep "PATH" *`
		`grep -Rn "PATH" .`
		`grep "PATH" * .*`

Следующие команды из таблицы 2-1 Bash выполняет самостоятельно:

- pwd
- cd

Эти команды называются **встроенными в интерпретатор**[169]. Если Bash не может выполнить команду сам, он ищет подходящую утилиту или программу.

В окружение MSYS2 входит набор GNU-утилит. Это вспомогательные узкоспециализированные программы. Они дают доступ к функциям ОС. Также через них пользователь работает с файловой системой. Следующие команды из таблицы 2-1 выполняются GNU-утилитами:

- ls
- mount
- find
- grep

Часто различий между командами и утилитами не делают. Любой текст после приглашения командной строки называют командой.

pwd

Рассмотрим команды из таблицы 2-1. Мы только что запустили терминал. Первым делом узнаем текущий каталог. Терминал MSYS2 выводит его перед приглашением $. Этот вывод зависит от конфигурации терминала. Если вы работаете на ОС Linux или macOS, текущий каталог не выводится без дополнительной настройки.

После запуска терминала MSYS2 откроется домашний каталог текущего пользователя. Для сокращения он обозначается символом тильда . Этот символ вы видите перед приглашением командной строки. С сокращением можно работать так же, как с любым абсолютным путём.

Чтобы вывести текущий каталог, выполните встроенную команду интерпретатора pwd. Иллюстрация 2-7 демонстрирует результат её выполнения. Команда вывела абсолютный путь до домашнего каталога пользователя: `/home/ilya.shpigor`.

Если к вызову pwd добавить **опцию** `-W`, команда выведет путь в структуре каталогов Windows. Это полезно, если вы создали файл в окружении MSYS2 и собираетесь открыть его в Windows-приложении. Результат вывода pwd с опцией `-W` приведён на иллюстрации 2-7.

[169] https://ru.wikipedia.org/wiki/Bash#Внутренние_команды

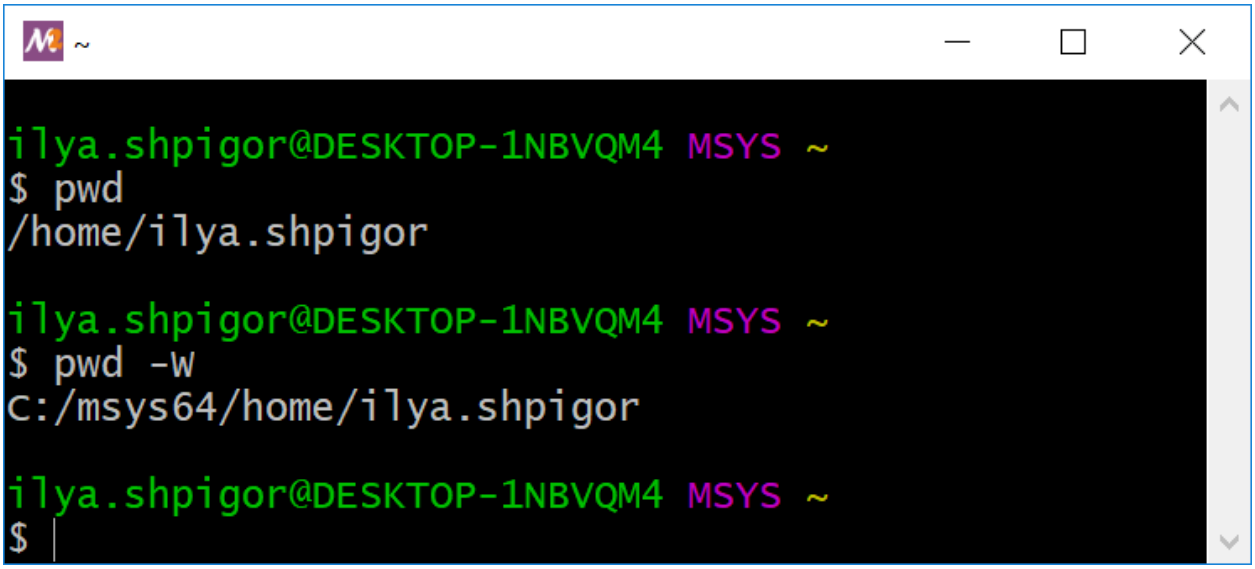


Иллюстрация 2-7. Вывод команды pwd

Что такое опция команды? Если у приложения только текстовый интерфейс, способы взаимодействия с ним ограничены. При этом ему нужны входные данные для работы. Например, путь до файла или каталога. Командный интерпретатор предлагает простой способ передать эту информацию. Она указывается через пробел после команды запуска приложения. **Параметром**[170] или аргументом программы называются слово или символ, которые передаются ей на вход. **Опцией** или ключом называется аргумент, который переключает режим работы программы. Формат опций стандартизован. Обычно они начинаются с тире - или двойного тире –.

Встроенные команды интерпретатора вызываются так же как и программы. У них тоже есть параметры и опции.

Набирать длинные команды неудобно. Поэтому в Bash есть функция автодополнения. Она вызывается по нажатию клавиши Tab. Наберите первые буквы команды и нажимаете Tab. Если Bash сможет найти команду по первым буквам, он допишет её за вас. Если несколько команд начинаются одинаково, автодополнение не произойдёт. В этом случае нажмите Tab повторно. Bash выведет список всех доступных команд.

Иллюстрация 2-8 демонстрирует список доступных команд. Bash вывел его после ввода текста `pw` и двойного нажатия Tab.

[170] https://ru.wikipedia.org/wiki/Интерфейс_командной_строки#Формат_команды

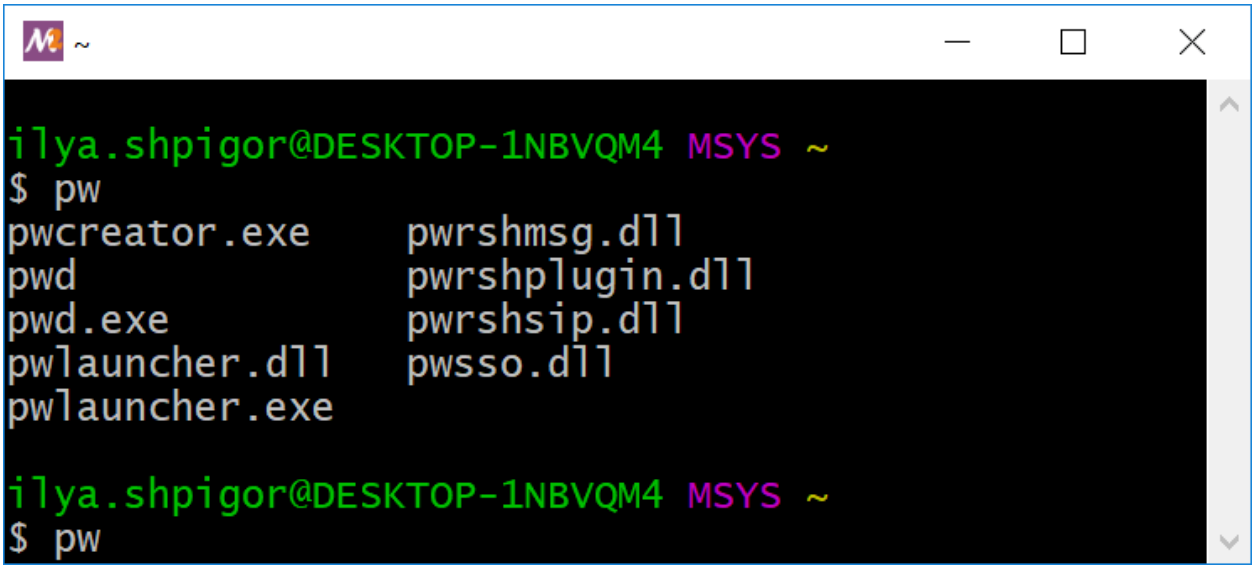


Иллюстрация 2-8. Автодополнение для команды `pw`

ls

Мы узнали текущий каталог. Теперь выведем его содержимое. Для этого есть утилита ls. Предположим, вы только что установили окружение MSYS2. Вызовите ls без параметров в домашнем каталоге пользователя. Утилита ничего не выведет. Этот результат демонстрирует вторая строчка на иллюстрации 2-9. Обычно это означает, что каталог пуст.

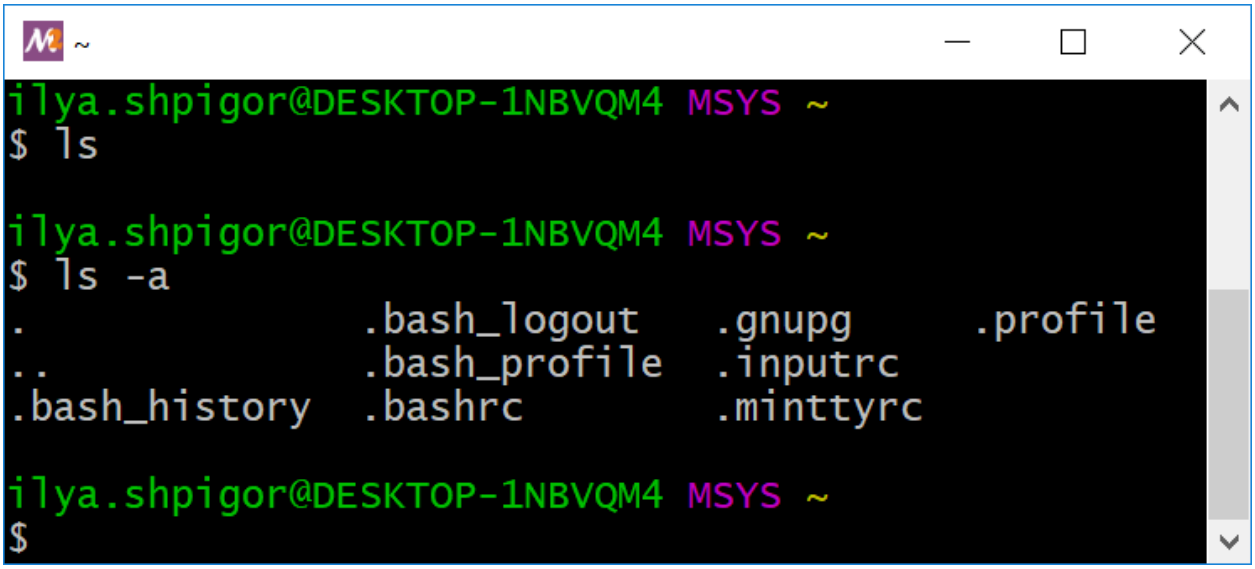


Иллюстрация 2-9. Вывод утилиты ls

В Windows есть понятие скрытых файлов и каталогов. Они есть и в Unix-окружении. Такие файлы создают приложения и ОС для своих нужд. Например, в них хранится конфигурация

или временная информация. В обычном режиме работы Проводник Windows их не отображает. Чтобы увидеть скрытые файлы, измените настройки Проводника[171].

В Unix-окружении имена скрытых файлов и каталогов начинаются с точки. Утилита ls их не отображает по-умолчанию. Чтобы изменить это поведение, запустите утилиту с опцией `-a`. Тогда в домашнем каталоге вы увидите восемь файлов. Все они начинаются с точки, как на иллюстрации 2-9.

Утилита ls может вывести содержимое каталога без перехода в него. Для этого передайте в неё абсолютный или относительный путь до каталога. Иллюстрация 2-10 демонстрирует вывод команды “ls /”. Это содержимое корневого каталога.

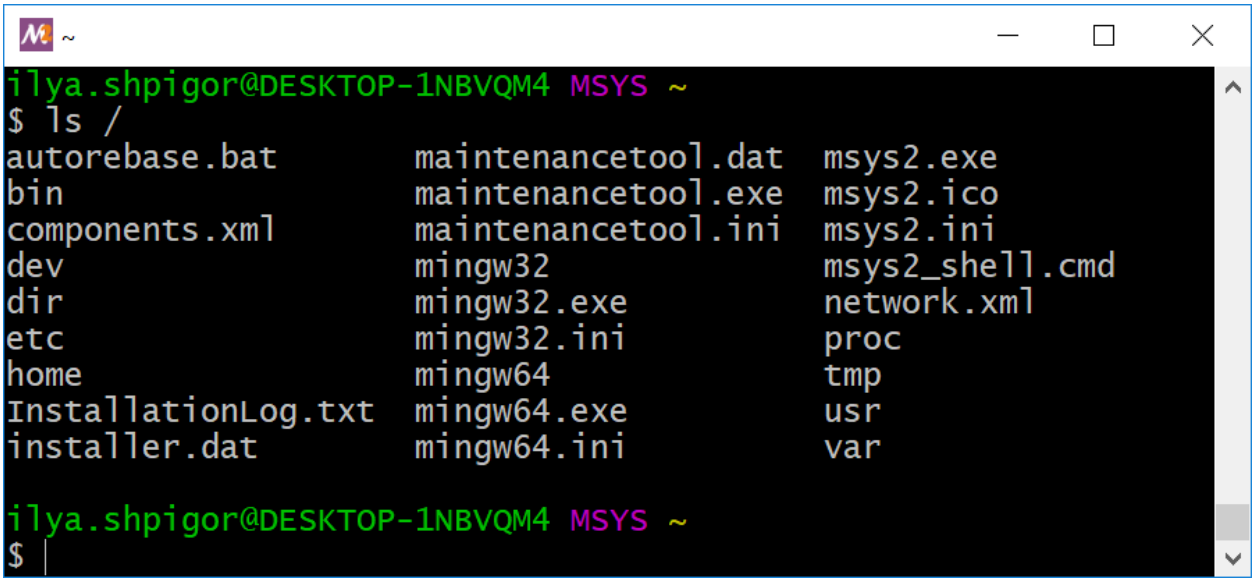


Иллюстрация 2-10. Вывод утилиты ls

Обратите внимание, что в выводе команды “ls /” нет каталогов `/c` и `/d`. Согласно листингу 2-2, это точки монтирования дисков C и D. Они находятся в корневом каталоге. Почему их не выводит ls? Проблема в том, что в файловой системе Windows нет понятия точек монтирования. Поэтому в ней нет каталогов `/c` и `/d`. Они создаются только в Unix-окружении. Через эти каталоги вы получаете доступ к содержимому дисков. Утилита ls читает содержимое каталогов в файловой системе Windows. Поэтому точки монтирования она не отображает. В Linux и macOS такой проблемы нет. Там ls корректно выводит все точки монтирования.

mount

Если к компьютеру подключено несколько дисков, полезно вывести на экран их точки монтирования. Это делает утилита mount[172]. Запустите её без параметров. Она выведет список точек монтирования как на иллюстрации 2-11.

[171] https://support.microsoft.com/ru-ru/help/14201/windows-show-hidden-files
[172] https://ru.wikipedia.org/wiki/Mount

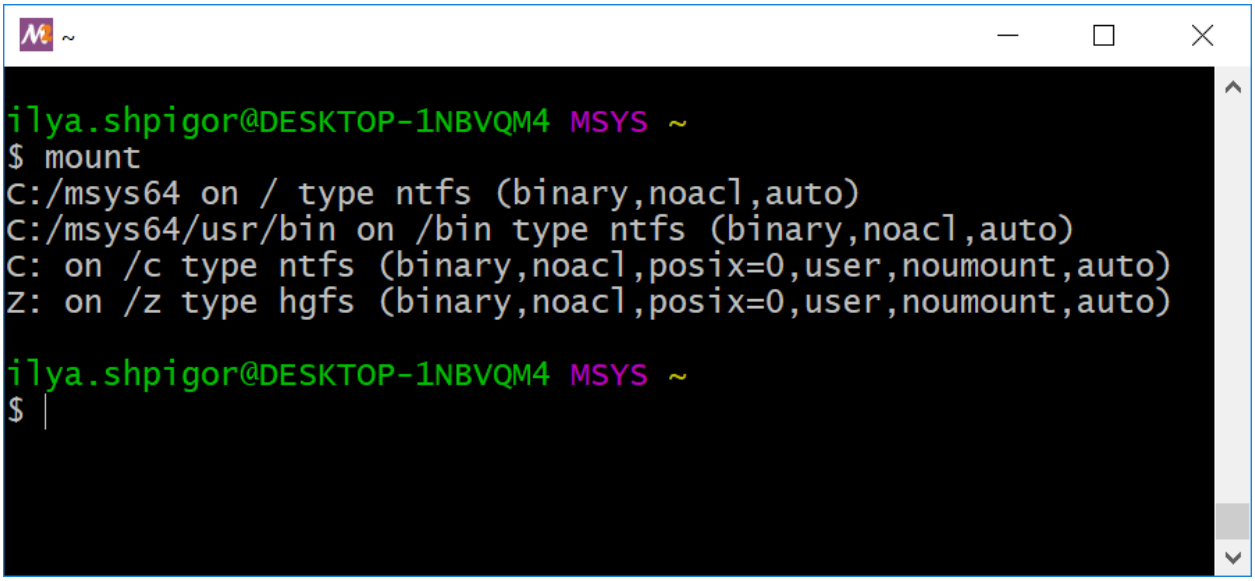


Иллюстрация 2-11. Вывод утилиты mount

Рассматривайте этот вывод как таблицу, состоящую из четырёх столбцов. Их значения следующие:

1. Диск, его раздел или каталог. Это то, что монтируется в корневую файловую систему.
2. Точка монтирования. Это путь, по которому доступен смонтированный диск.
3. Тип файловой системы диска.
4. Параметры монтирования. Например, права доступа к диску.

Таблица 2-2 демонстрирует вывод утилиты mount с иллюстрации 2-11. Вывод разделён на столбцы.

Таблица 2-2. Вывод утилиты mount

Монтируемый раздел	Точка монтирования	Тип ФС	Параметры монтирования
`C:/msys64`	`/`	`ntfs`	`binary,noacl,auto`
`C:/msys64/usr/bin`	`/bin`	`ntfs`	`binary,noacl,auto`
`C:`	`/c`	`ntfs`	`binary,noacl,posix=0,user,noumount,auto`
`Z:`	`/z`	`hgfs`	`binary,noacl,posix=0,user,noumount,auto`

Таблица 2-2 вызовет недоумение у Windows-пользователей. В качестве корневого каталога в Unix-окружении монтируется каталог `C:/msys64`. Далее в него монтируются диски C и Z по путям `/c` и `/z`. С точки зрения Unix-окружения диск C находится по пути `C:/msys64/c`. Но в файловой системе Windows зависимость обратная. Там `C:/msys64` — это подкаталог диска C.

В Unix-окружении это противоречие не вызывает проблем. Путь `/c` является точкой монтирования. Она существует только в окружении Unix. Её нет файловой системе Windows.

Представьте, что каталог `/c` в MSYS2 — это ярлык[173] для диска C.

Вывод утилиты mount занял на экране много места. Чтобы очистить окно терминала, нажмите комбинацию клавиш Ctrl+L.

Бывает, что запущенная команда долго выполняется или зависла. Чтобы прервать её выполнение, нажмите комбинацию клавиш Ctrl+C.

cd

Мы знаем текущий каталог. Теперь перейдём по нужному нам пути. Для примера, найдём документацию по интерпретатору Bash. Проверим системный каталог `/usr`. Там хранятся файлы установленных приложений. Для перехода в `/usr` наберите команду cd так:

```
cd /usr
```

Не забывайте про автодополнение. Оно работает как для имени команды, так и для её параметров. Достаточно набрать `cd /u` и нажать клавишу Tab. Имя каталога `usr` Bash добавит автоматически. Результат выполнения команды приводит иллюстрация 2-12.

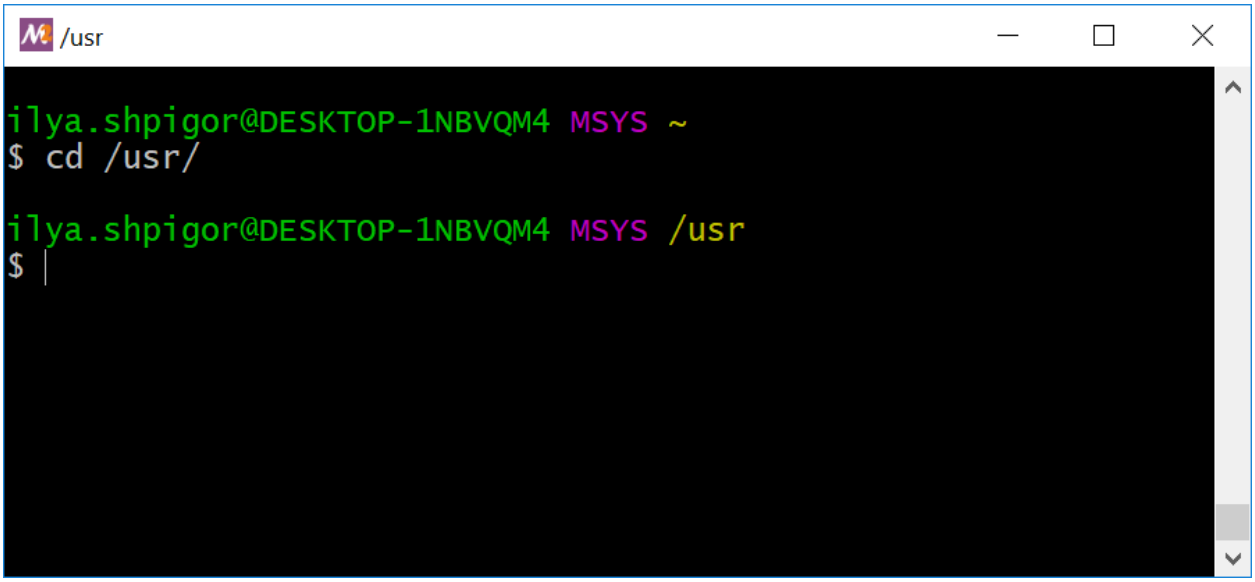


Иллюстрация 2-12. Результат выполнения команды cd

При успешном выполнении команда cd ничего не выводит. Она только меняет текущий каталог. Выполните команду и проверьте вывод перед приглашением командной строки. Теперь текущим каталогом стал `/usr`.

Команда cd принимает на вход и абсолютные пути, и относительные. Относительные пути короче и быстрее в наборе. Поэтому их чаще используют для навигации по файловой системе.

[173] https://ru.wikipedia.org/wiki/Ярлык_(компьютер)

Мы перешли в каталог `/usr`. Теперь можно вывести его подкаталоги и перейти в один из них. Предположим, что вместо этого вам надо перейти на уровень выше в корневой каталог. Для этого есть два способа: перейти по абсолютному пути `/` или по специальному относительному пути `..`. Путь `..` всегда указывает на родительский каталог для текущего. Команда перехода по этому пути выглядит так:

```
cd ..
```

Кроме `..` есть еще один специальный путь `.`. Он означает текущий каталог. Если выполнить команду “cd .”, ничего не произойдёт. Вы останетесь в там же. Путь `.` нужен для запуска программ из текущего каталога.

Мы находимся в каталоге `/usr`. Выполним здесь утилиту ls. В её выводе есть подкаталог `share`. В нём — подкаталог `doc` с документацией по установленным приложениям. Полный путь документации по Bash такой: `share/doc/bash`. Перейдём в него следующей командой:

```
cd share/doc/bash
```

Теперь текущим каталогом стал `/usr/share/doc/bash`. Выполним команду ls. Среди прочего она выведет файл с именем `README`. Это и есть документация по Bash, которую мы ищем.

Выведите содержимое файла `README` на экран с помощью утилиты cat. Для этого выполните команду:

```
cat README
```

Иллюстрация 2-13 демонстрирует её результат.

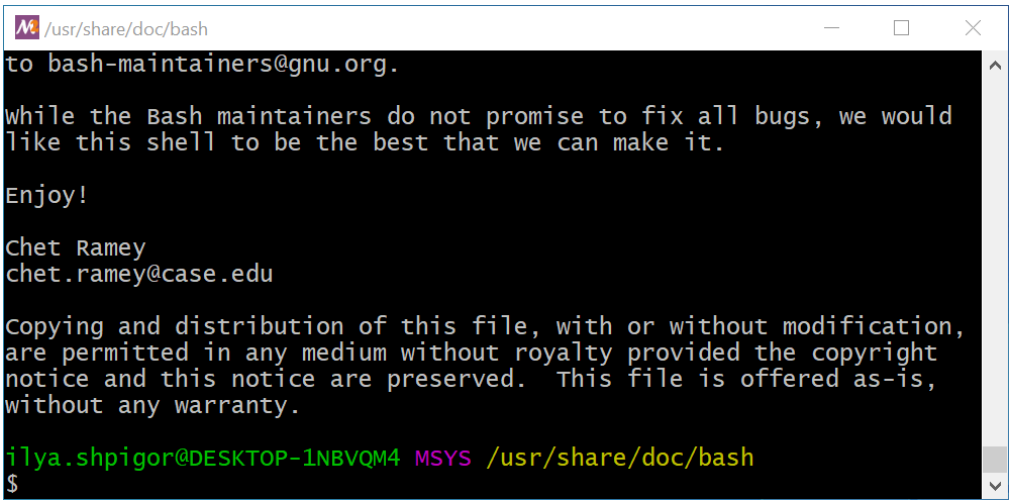


Иллюстрация 2-13. Вывод утилиты cat

Авторы некоторых руководств по Bash не рекомендуют выводить содержимое файла утилитой cat. Её назначение — объединять несколько файлов и выводить результат в стандартный поток вывода. Вместо вызова cat для одного файла они рекомендуют комбинировать команду echo и перенаправление потоков. Вот пример такого подхода:

```
echo "$(< README.txt)"
```

На иллюстрации 2-13 приводится не весь файл `README`, а только его последние строки. Этот файл большой. Поэтому вывод утилиты cat не поместился в окно терминала. Чтобы просмотреть начало файла, используйте полосу прокрутки в правой части окна. Для прокрутки по страницам[174] используйте горячие клавиши Shift+PageUp и Shift+PageDown. Для прокрутки по строкам — Shift+↑ и Shift+↓.

История команд

Каждая выполненная в терминале команда сохраняется в истории команд[175]. Чтобы повторить предыдущую команду, нажмите стрелку вверх ↑ и Enter. Нажмите стрелку вверх несколько раз, чтобы прокрутить историю дальше к началу. Для перехода к следующей команде в истории нажмите стрелку вниз ↓.

Например, вы только что ввели команду “cat README”. Чтобы её повторить, нажмите стрелку вверх и Enter.

Комбинация клавиш Ctrl+R вызывает поиск по истории. Нажмите Ctrl+R и начните набирать текст. Bash предложит вам последнюю введённую команду, которая начинается также. Чтобы исполнить её, просто нажмите Enter.

Для вывода на экран всей истории командой наберите:

```
history
```

В историю команд попадают только выполненные команды. В неё не попадают команды, которые вы набрали, а затем стёрли.

Что делать, есть в историю надо сохранить команду без её исполнения? Например, вы собираетесь исполнить её позже. Для этого есть трюк с комментарием. Если команда начинается с символа решётка #, Bash обработает её как комментарий. По нажатию Enter она попадёт в историю, но не исполнится. С этим трюком вызов утилиты cat станет таким:

[174] https://en.wikipedia.org/wiki/Page_Up_and_Page_Down_keys
[175] https://ru.wikipedia.org/wiki/History

```
#cat README
```

Теперь исполним команду. Для этого найдите её в истории и сотрите символ решётка в начале. Затем нажмите Enter.

В большинстве терминалов трюк с комментарием выполняет комбинация клавиш Alt+Shift+3. Работает она так:

1. Наберите команду, но не нажимайте Enter.
2. Нажмите Alt+Shift+3.
3. Команда сохранится в истории без исполнения.

Как скопировать текст из терминала? Предположим, что часть файла `README` нужна в другом документе. Для копирования используйте **буфером обмена**[176]. Это временное хранилище для строк. В нём сохраняется выделенный в терминале текст. Его можно вставить в любое другое окно.

Для копирования текста из терминала:

1. Выделите мышью нужный текст. Для этого зажмите левую кнопку мыши и проведите курсором по тексту.
2. Для вставки текста из буфера обмена в окно терминала нажмите среднюю кнопку мыши. Текст будет вставлен в текущую позицию курсора.
3. Для вставки текста в другое приложение нажмите правую кнопку мыши и выберите пункт “Вставить”.

find

Искать нужный файл или каталог командами cd и ls неудобно. Для этого есть специальная утилита find.

Если запустить утилиту find без параметров, она выведет содержимое текущего каталога и его подкаталогов. В вывод попадут и скрытые объекты. На иллюстрации 2-14 результат запуска find для домашнего каталога пользователя ~.

[176] https://ru.wikipedia.org/wiki/Буфер_обмена

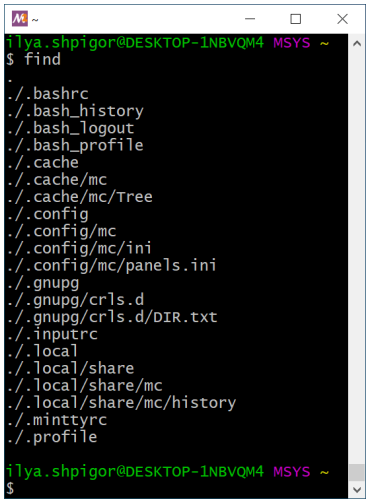


Иллюстрация 2-14. Вывод утилиты find

Первый параметр утилиты find — это каталог, в котором надо искать. Утилита принимает относительный или абсолютный путь. Например, вот команда для поиска в корневом каталоге:

```
find /
```

Со второго параметра утилиты начинаются условия поиска. Если найденный объект не удовлетворяет условиям, он не выводится на экран. Условия сочетаются друг с другом и составляют единое выражение. Для обработки этого выражения в утилиту встроен специальный интерпретатор. Например, условием поиска может быть имя файла. Тогда в вывод find попадут только файлы с этим именем.

Таблица 2-3 приводит часто используемые условия для утилиты find.

Таблица 2-3. Часто используемые условия утилиты find

Условие	Значение	Пример
`-type f`	Искать только файлы.	`find -type f`
`-type d`	Искать только каталоги.	`find -type d`

`-name шаблон`	Поиск файла или каталога по **шаблону имени**[177]. Шаблон чувствителен к регистру.	`find -name README`
		`find -name READ*`
		`find -name READ??`

`-iname шаблон`	Поиск файла или каталога по шаблону имени. Шаблон нечувствителен к регистру.	`find -iname readme`

[177] https://ru.wikipedia.org/wiki/Шаблон_поиска

Таблица 2-3. Часто используемые условия утилиты find

Условие	Значение	Пример
`-path шаблон`	Поиск по шаблону пути к файлу или каталогу. Шаблон чувствителен к регистру.	`find -path */doc/bash/*`
`-ipath шаблон`	Поиск по шаблону пути к файлу или каталогу. Шаблон нечувствителен к регистру.	`find . -ipath */DOC/BASH/*`
`-a` или `-and`	Скомбинировать несколько условий с помощью логического И. В вывод попадут только файлы и каталоги, удовлетворяющие всем условиям.	`find -name README -a -path */doc/bash/*`
`-o` или `-or`	Скомбинировать несколько условий с помощью логического ИЛИ. Если файл или каталог соответствует хотя бы одному условию, он попадёт в вывод.	`find -name README -o -path */doc/bash/*`
`!` или `-not`	Логическое отрицание (НЕ) последующего условия. В вывод попадут только файлы и каталоги, которые не удовлетворяют условию.	`find -not -name README` `find ! -name README`

Шаблоном называется поисковый запрос. В него вместе с обычными символами входят **символы подстановки**[178] (wildcard character). Всего в Bash три таких символа: *, ? и [. Звёздочка означает любое количество любых символов. Знак вопроса — один любой символ. Например, строка `README` соответствует следующим шаблонам:

- `*ME`
- `READM?`
- `*M?`
- `R*M?`

Квадратные скобки указывают набор символов в определённой позиции строки. Например, шаблон `[cb]at.txt` соответствует файлам `cat.txt` и `bat.txt`. Вот вызов утилиты find с поиском по этому шаблону:

```
find . -name "[cb]at.txt"
```

[178] https://ru.wikipedia.org/wiki/Символ_подстановки

Упражнение 2-1. Шаблоны поиска

Какая из следующих строк соответствует шаблону "*ME.??" ?

* 00_README.txt
* README
* README.md

Упражнение 2-2. Шаблоны поиска

Какая из следующих строк соответствует шаблону "*/doc?openssl*" ?

* /usr/share/doc/openssl/IPAddressChoice_new.html
* /usr/share/doc_openssl/IPAddressChoice_new.html
* doc/openssl
* /doc/openssl

Применим поиск по шаблонам на практике. Вернёмся к задаче с документацией по Bash. Найдём файл `README` с помощью утилиты find. Предположим, что нам неизвестен диск, где хранится файл. Тогда передадим в find первым параметром корневой каталог. Так она будет искать файл на всех смонтированных дисках. В Unix-окружении документы хранятся в каталогах с именем `doc`. Учитывая это, вызовем следующую команду поиска:

```
find / -path */doc/*
```

Команда выведет список всех файлов с документацией на всех смонтированных дисках. Этот список слишком длинный. Сократим его с помощью дополнительных условий поиска. Добавим слово `bash` в путь искомого файла. Получится следующая команда:

```
find / -path */doc/* -path */bash/*
```

Иллюстрация 2-15 демонстрирует результат. То же самое выведет следующая команда:

```
find / -path */doc/* -a -path */bash/*
```

Отличие команд в опции `-a` между условиями. Она означает логическое И. Если между условиями не указывать логическую операцию, по умолчанию применяется И.

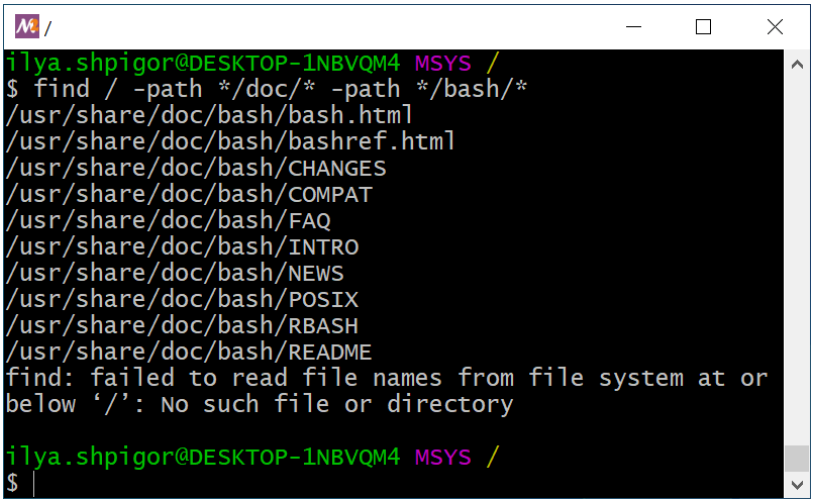


Иллюстрация 2-15. Результат поиска утилиты find

В конце вывода утилита find сообщает об ошибке. Проблема в том, что некоторые подкаталоги / являются точками монтирования Windows-дисков. Например, диск C смонтирован в /c. Утилита не может получить доступ к их содержимому при поиске с корневого каталога. Чтобы избежать ошибки, начните поиск с точки монтирования диска C:

```
find /c -path */doc/* -a -path */bash/*
```

Альтернативное решение — исключить точки монтирования из поиска. Для этого передайте опцию -mount:

```
find / -mount -path */doc/* -a -path */bash/*
```

В результате find выведет небольшой список документов. Среди них легко найти нужный README ƒайл.

Искать ƒайл документации можно и по-другому. Предположим, что его имя известно. Тогда укажем имя вместе с предполагаемым путём. Получим следующую команду

```
find / -path */doc/* -name README
```

Иллюстрация 2-16 демонстрирует результат поиска.

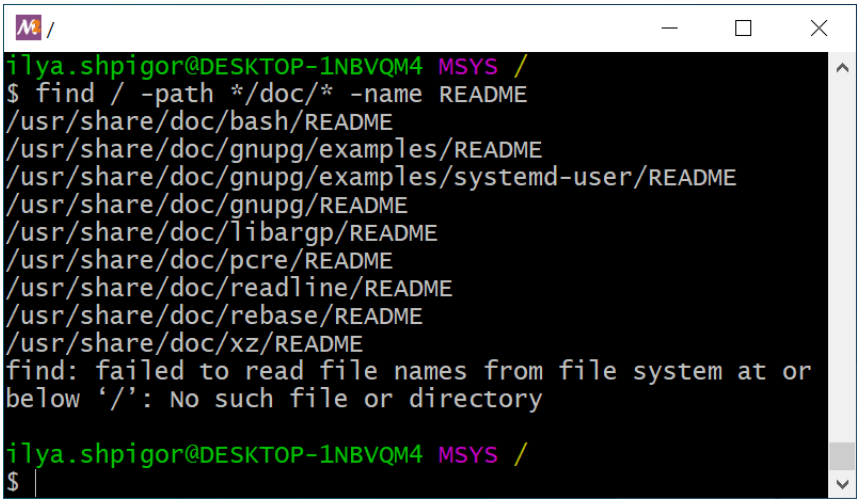


Иллюстрация 2-16. Результат поиска утилиты find

Перед нами снова небольшой список файлов, в котором легко найти нужный.

Условия утилиты find можно группировать. Для этого используйте экранированные[179] круглые скобки. Например, найдём файлы `README` с путём `*/doc/*` или файлы `LICENSE` с произвольным путём. Это сделает следующая команда:

```
find / \( -path */doc/* -name README \) -o -name LICENSE
```

Зачем экранировать скобки в выражении утилиты find? Дело в том, что скобки — это часть синтаксиса Bash. Они используются в конструкциях языка. Встретив их в вызове утилиты, Bash выполнит **подстановку**. Подстановкой называется замена части команды на что-то другое. С помощью экранирования мы заставляем интерпретатор игнорировать скобки. В этом случае он передаст их как есть в утилиту find.

Утилита find умеет не только искать, но и обрабатывать файлы и каталоги. После условия поиска, можно указать действие. Утилита применит его к каждому найденному объекту.

Опции для указания действий приведены в таблице 2-4.

Таблица 2-4. Опции для указания действий над найденными объектами

Опция	Значение	Пример
`-exec команда {} \;`	Выполнить указанную команду над каждым найденным объектом.	`find -name README -type f -exec cp {} ~ \;`
`-exec команда {} +`	Выполнить указанную команду один раз над всеми найденными объектами. Команда получит все объекты на вход.	`find -type d -exec cp -t ~ {} +`

[179] https://ru.wikipedia.org/wiki/Экранирование_символов

Таблица 2-4. Опции для указания действий над найденными объектами

Опция	Значение	Пример
`-delete`	Удалить каждый из найденных файлов. Каталоги удаляются, только если они пустые.	`find -name README -type f -delete`

Есть два варианта действия `-exec`. Они отличаются символами на конце: экранированная точка с запятой `\;` или плюс `+`. Действие с плюсом сработает только, если вызываемая команда способна обработать несколько входных параметров. Большинство GNU-утилит с этим справятся. Если команда принимает только один параметр, она обработает только первый найденный объект.

Применим действие `-exec` для решения практической задачи. Скопируем файлы документации по Bash с расширением HTML в домашний каталог. Для начала найдём эти файлы утилитой find. Её вызов выглядит так:

```
find / -path "*/doc/bash/*" -name "*.html"
```

Передавая шаблоны в find, заключайте их в двойные кавычки “. Кавычки делают то же, что и обратный слэш перед круглыми скобками. Они запрещают Bash интерпретировать шаблоны. Тогда утилита find получает их как есть и интерпретирует сама.

Результат поиска HTML документов приведён на иллюстрации 2-17.

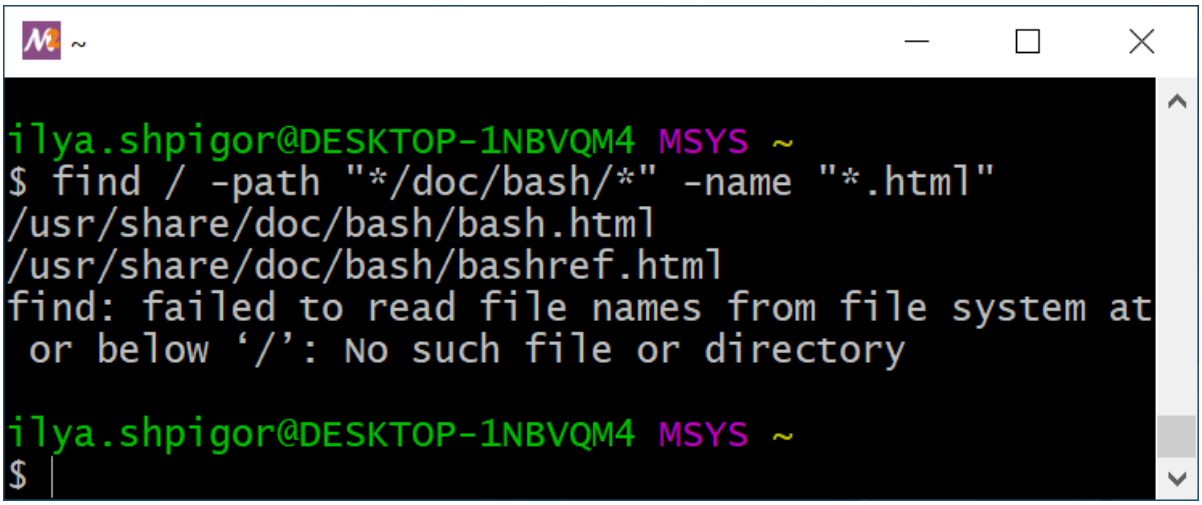


Иллюстрация 2-17. Результат работы утилиты find

Добавим к команде поиска действие `-exec`. Оно вызывает утилиту cp. Утилита копирует файлы и каталоги в указанный путь. Первым параметром cp принимает копируемый объект. Второй параметр — путь, куда копировать. Вызов find с действием выглядит так:

```
find / -path "*/doc/bash/*" -name "*.html" -exec cp {} ~ \;
```

Это команда выведет только ошибку с точками монтирования.

Разберёмся, что сделала команда. Она вызвала утилиту cp для каждого найденного HTML файла. Первым параметром cp получила путь до файла. Утилита find подставила путь вместо фигурных скобок {}. Она нашла два файла. Поэтому cp вызывалась дважды так:

```
1 cp ./usr/share/doc/bash/bash.html ~
2 cp ./usr/share/doc/bash/bashref.html ~
```

Каждый вызов копирует один HTML файл в домашний каталог пользователя.

Только что мы написали первую программу на языке интерпретатора утилиты find. Она работает по следующему алгоритму:

1. Найти файлы с расширением HTML на всех дисках. Их пути соответствуют шаблону `*/doc/bash/*`.
2. Скопировать каждый найденный файл в каталог пользователя.

Алгоритм программы состоит всего из двух шагов. Но это масштабируемое решение для поиска и копирования файлов. Программа обработает десятки HTML файлов так же быстро, как и два.

Действия `-exec` комбинируются точно так же как и условия поиска. Для примера выведем содержимое каждого из найденных HTML файлов и подсчитаем количество строк в них. С первой задачей справится утилита cat. Утилита wc подсчитает число строк. На вход wc принимает имя файла для обработки. Команда вызова find в этом случае выглядит так:

```
find / -path "*/doc/bash/*" -name "*.html" -exec cat {} \; -exec wc -l {} \;
```

Мы не указали логическую операцию между действиями `-exec`. По умолчанию используется логическое И. Это означает, что второе действие выполняется только при успешном исполнении первого. Если заменить логическую операцию на ИЛИ, второе действие будет выполняться всегда, независимо от результата первого.

Действия `-exec` группируются с помощью экранированных круглых скобок `\(` и `\)`. Это работает так же как группирование условий поиска.

Упражнение 2-3. Поиск файлов утилитой find

Напишите команду вызова find для поиска текстовых файлов в Unix-окружении. Дополните команду, чтобы вывести общее число строк в этих файлах.

Логические выражения

Условия поиска утилиты find представляют собой **логические выражения**[180]. Логическим выражением называется конструкция языка программирования. Выражение можно вычислить. В результате получится одно из двух значений: “истина” или “ложь”.

Условие поиска find — это конструкция встроенного в утилиту интерпретатора. Если для найденного объекта условие выполняется, его вычисление даст результат “истина”. Если условие не выполняется, его результат — “ложь”. Если условий несколько, они объединяются в составное логическое выражение.

Мы сталкивались с алгеброй логики, когда знакомились с двоичной системой счисления. Этот раздел математики изучает **логические операции**[181]. Они отличаются от привычной арифметики сложения, вычитания, умножения и деления.

Вычисление логического выражения даёт только два возможных значения. Поэтому арифметические действия над двумя выражениями тривиальны и ничего не дают. Если же применить к ним логические операции, получим условия со строгими правилами вывода результата. Например, для утилиты find так составляется условие для поиска файла с заданным именем и путём. Комбинация таких условий с действиями даёт сложное поведение программы.

Операндом называется то, над чем выполняется логическая операция. Операндом может быть выражение или отдельное значение.

Для простоты разберём логические выражения на примере, не связанном с утилитой find. Представьте, что мы программируем робота для склада. Его задача — перевозить коробки из точки А в точку Б. Для этого зададим ему следующий прямолинейный алгоритм:

1. Двигайся в точку А.
2. Возьми коробку в точке А.
3. Двигайся в точку Б.
4. Положи коробку в точке Б.

В этом алгоритме нет никаких условий. Это значит, что робот выполняет каждый его шаг независимо от внешних событий.

Теперь представьте, что на пути робота в точку Б окажется препятствие. Например, другой робот. В этом случае исполнение алгоритма приведёт к столкновению. Чтобы этого не случилось, добавим условие:

[180] https://ru.wikipedia.org/wiki/Логическое_выражение
[181] https://ru.wikipedia.org/wiki/Логическая_операция

1. Двигайся в точку А.
2. Возьми коробку в точке А.
3. Если нет препятствия, двигайся в точку Б. Иначе остановись.
4. Положи коробку в точке Б.

Третий шаг алгоритма называется **условным оператором**[182]. Все современные языки программирования имеют такую конструкцию.

Алгоритм работы условного оператора выглядит так:

1. Вычислить значение операнда.
2. Если результат “истина”, выполнить первое действие.
3. Если результат “ложь”, выполнить второе действие.

В нашем примере робот вычисляет значение логического выражения “нет препятствия”. Если препятствие есть, выражение будет ложно и робот остановится. В противном случае он продолжит движение в точку Б.

Логические операции позволяют скомбинировать несколько выражений. Например, робот пробует взять коробку в точке А, но её там нет. Тогда ему нет смысла двигаться в точку Б. Добавим это условие к уже существующему выражению с помощью операции **логического И**[183] (конъюнкция). Теперь наш алгоритм выглядит так:

1. Двигайся в точку А.
2. Возьми коробку в точке А.
3. Если есть коробка И нет препятствия, двигайся в точку Б. Иначе остановись.
4. Положи коробку в точке Б.

Вычисление логических операций тоже даёт истину или ложь. Результатом логического И будет “истина”, когда оба операнда истинны. То есть у робота есть коробка и нет препятствия. В любом другом случае результатом операции будет “ложь”. Тогда робот остановится.

Работая с утилитой find, мы познакомились с ещё двумя логическими операциями: **ИЛИ**[184] (дизъюнкция) и **НЕ**[185] (отрицание).

На самом деле в нашем алгоритме для робота мы уже применили НЕ, когда написали выражение “нет препятствия”. Это отрицание: “НЕ есть препятствие”. Укажем явно логическое НЕ в алгоритме:

1. Двигайся в точку А.
2. Возьми коробку в точке А.

[182] https://ru.wikipedia.org/wiki/Ветвление_(программирование)
[183] https://ru.wikipedia.org/wiki/Конъюнкция
[184] https://ru.wikipedia.org/wiki/Дизъюнкция
[185] https://ru.wikipedia.org/wiki/Отрицание

3. Если есть коробка И НЕ есть препятствие, двигайся в точку Б. Иначе остановись.
4. Положи коробку в точке Б.

В логическом выражении операцию И можно заменить на ИЛИ. Чтобы поведение робота не изменилось, добавим отрицание к первому условию и уберём его у второго. Также поменяем порядок действий условного оператора. Теперь если выражение будет истинным, робот остановится. Если ложным, продолжит двигаться к точке Б. В итоге получим следующий алгоритм:

1. Двигайся в точку А.
2. Возьми коробку в точке А.
3. Если НЕ есть коробка ИЛИ есть препятствие, остановись. Иначе двигайся в точку Б.
4. Положи коробку в точке Б.

Прочитайте внимательно новый условный оператор. Логика робота не изменилась. Он по-прежнему остановится, если у него нет коробки или на пути возникло препятствие.

В нашем примере логическое выражение записано в виде предложения на русском языке. Это предложение звучит неестественно. Его надо прочитать несколько раз, чтобы понять. Причина в том, что **естественный язык**[186] (язык общения людей) не подходит для описания логических выражений. Он недостаточно точен. Поэтому в алгебре логики применяют математическая запись.

Кроме логических И, ИЛИ, НЕ в программировании часто используются ещё три операции:

- Эквивалентность
- Не эквивалентность
- Исключающее ИЛИ

Таблице 2-5 приводит полный список логических операций.

Таблица 2-5. Логические операции в программировании

Операция	Результат вычисления выражения
И (AND)	“Истина”, когда оба операнда “истина”.
ИЛИ (OR)	“Истина”, когда любой из операндов “истина”. “Ложь”, когда все операнды “ложь”.
НЕ (NOT)	“Истина”, когда операнд “ложь” и наоборот.
Исключающее ИЛИ (XOR)	“Истина”, когда значения операндов отличаются (истина-ложь или ложь-истина). “Ложь”, когда они совпадают (истина-истина, ложь-ложь).

[186] https://ru.wikipedia.org/wiki/Естественный_язык

Таблица 2-5. Логические операции в программировании

Операция	Результат вычисления выражения
Эквивалентность	“Истина”, когда значения операндов совпадают.
Не эквивалентность	“Истина”, когда значения операндов отличаются.

Постарайтесь запомнить эту таблицу. Это несложно, если вы часто используете логические операции на практике.

grep

Утилита grep — это ещё один инструмент поиска. Она находит файлы по их содержимому.

Когда применять утилиту find, а когда grep? Используйте find, чтобы найти файл или каталог по имени, пути или **метаданным**[187]. Метаданными называется дополнительная информация об объекте файловой системы. Например, размер, время создания и последней модификации, права доступа. Ищите файл утилитой grep, если известно только его содержимое.

Рассмотрим выбор утилиты поиска на примере. Мы ищем файл с документацией. Известно, что в нём встречается фраза “free software” (свободное ПО). Если применить утилиту find, алгоритм поиска будет следующим:

1. Найти все файлы документации с именем `README` с помощью find.
2. Открыть каждый файл в текстовом редакторе и найти в нём фразу “free software”.

Проверять содержимое файлов в текстовом редакторе долго. Утилита grep автоматизирует эту операцию. Следующая команда найдёт строку “free software” в файле `README`:

```
grep "free software" /usr/share/doc/bash/README
```

Первый параметр утилиты — это искомая строка. Не забывайте про двойные кавычки. Они гарантируют, что grep получит строку без изменений. Без кавычек Bash разделит её пробелом на два отдельных параметра. Этот механизм разделения строк называется **word splitting**[188].

Вторым параметром grep принимает относительный или абсолютный путь к файлу. Если указать список файлов через пробелы, утилита обработает их все. В примере мы передали только один путь до файла `README`.

Иллюстрация 2-18 приводит результат вызова утилиты grep.

[187] https://ru.wikipedia.org/wiki/Метаданные
[188] http://mywiki.wooledge.org/WordSplitting

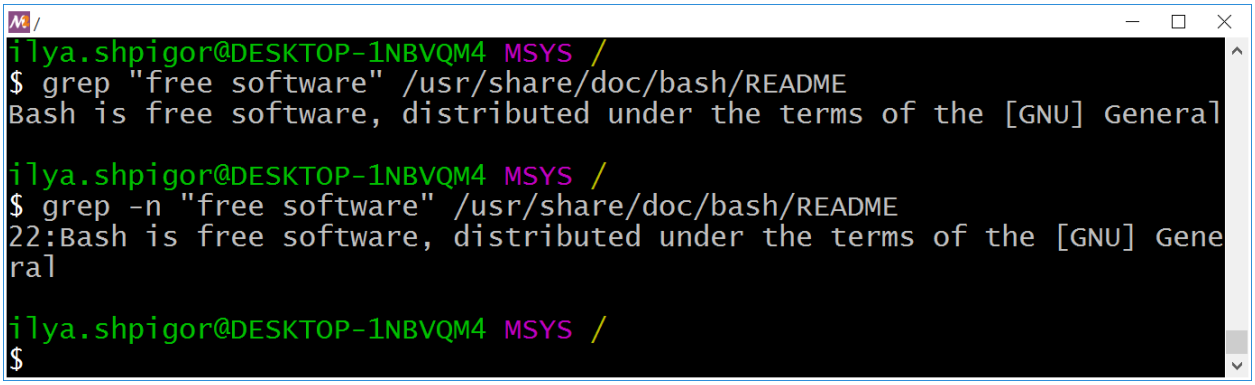


Иллюстрация 2-18. Результат вызова утилиты grep

Утилита выводит на экран все строки файла, в которых встречается искомая фраза. Вывод удобнее читать, если добавить в него номера найденных строк. Для этого укажите опцию `-n` перед первым параметром утилиты. Результат такого вызова приведён в нижней части иллюстрации 2-18.

Мы узнали, как с помощью grep найти строку в указанных файлах. Теперь применим утилиту для решения нашей задачи. Найдём файлы документации с фразой “free software”. Это можно сделать двумя способами:

- Использовать шаблоны поиска Bash.
- Использовать механизм перебора файлов самой утилиты grep.

Рассмотрим первый способ. Предположим, что в домашнем каталоге пользователя есть два текстовых файла: `bash.txt` и `xz.txt`. Это копии `README` документов программ Bash и xz. Найдём, в каком из них встречается фраза “free software”. Для этого выполним следующие две команды:

```
1 cd ~
2 grep "free software" *
```

Сначала мы переходим в домашний каталог пользователя. Затем вызываем утилиту grep.

В качестве пути до целевого файла мы указали символ подстановки — звёздочку. Этот шаблон поиска означает любую строку. Bash заменит шаблон на список всех файлов из домашнего каталога. Затем он вызовет утилиту. В результате команда запуска grep станет такой:

```
grep "free software" bash.txt xz.txt
```

Попробуйте выполнить обе команды: grep с шаблоном и со списком файлов через пробел. Утилита даст одинаковый результат.

Попробуем обойтись без команды cd для перехода в домашний каталог. Для этого добавим путь к каталогу в шаблон поиска. Получится такой вызов grep:

```
grep "free software" ~/*
```

Команда echo поможет проверить результат подстановки шаблонов поиска. Посмотрим, как Bash развернёт шаблоны из наших примеров:

```
1 echo *
2 echo ~/*
```

Выполните эти команды. Первая выведет список файлов в текущем каталоге, а вторая — в домашнем.

Шаблоны поиска нельзя заключать в двойные кавычки. Например, так:

```
grep "free software" "*"
```

Из-за кавычек Bash не развернёт шаблон, а передаст его как есть утилите grep. Она в отличие от find не умеет самостоятельно разворачивать шаблоны. Поэтому такой вызов приведёт к ошибке как на иллюстрации 2-19.

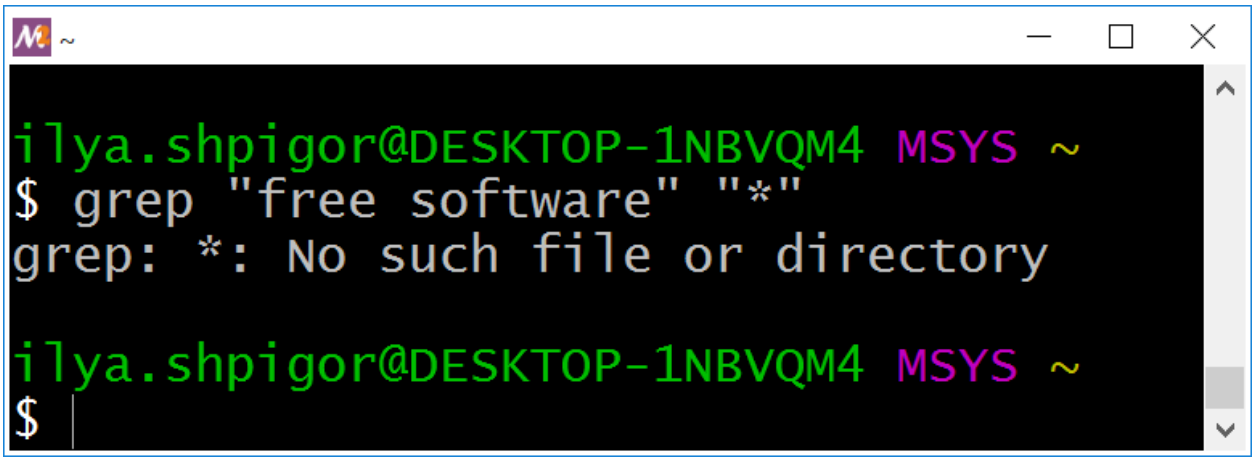


Иллюстрация 2-19. Результат обработки шаблона в grep

Bash не включает скрытые файлы и каталоги в подстановку шаблона *. В нашем примере это означает, что утилита grep не получит их на вход. Чтобы искать только по скрытым файлам, используйте шаблон .*. Чтобы искать по всем файлам сразу, укажите два шаблона через пробел. Например, так:

```
grep "free software" * .*
```

Вместо шаблонов поиска, можно использовать встроенный механизм grep. Он перебирает файлы в указанном каталоге. Опция -r включает этот режим работы утилиты. Если вы используете опцию, последним параметром укажите не имя файла, а каталог поиска.

Следующая команда найдёт строку “free software” в файлах текущего каталога:

```
grep -r "free software" .
```

Такой вызов grep обработает все файлы текущего каталога, включая скрытые. Но файлы из подкаталогов обработаны не будут. Чтобы включить их в поиск, замените опцию `-r` на `-R`. Например, так:

```
grep -R "free software" .
```

Целевой каталог для поиска указывается по относительному или абсолютному пути. Вот примеры для обоих случаев:

```
cd /home
grep -R "free software" ilya.shpigor/tmp
grep -R "free software" /home/ilya.shpigor/tmp
```

Предположим, нас интересует список файлов, в которых встречается фраза. В обычном режиме утилита grep выводит все вхождения искомой фразы. Этот вывод сейчас не нужен. Уберём его опцией `-l`. Например, так:

```
grep -Rl "free software" .
```

Иллюстрация 2-20 приводит результат этой команды.

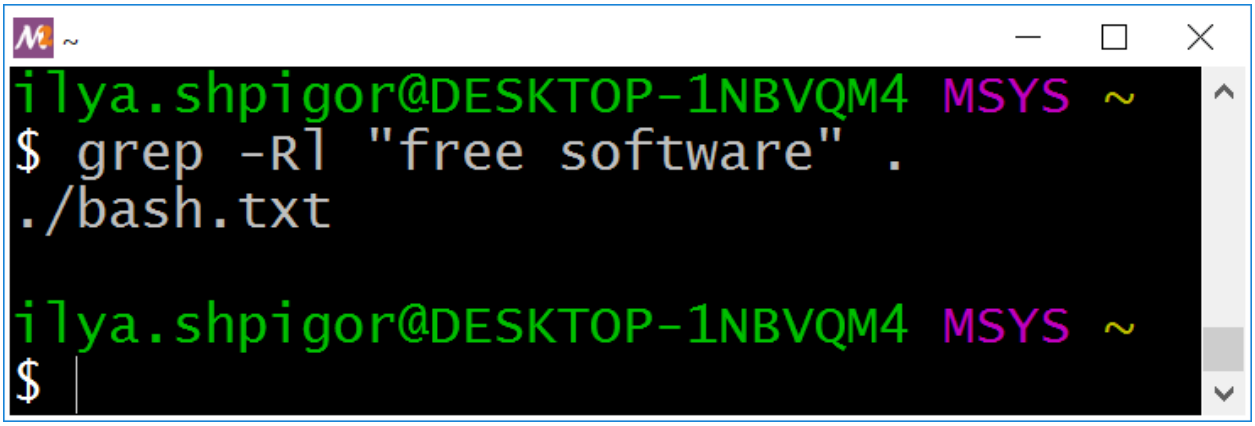


Иллюстрация 2-20. grep выводит только имена файлов

Мы получили список файлов, в которых фраза “free software” встречается хотя бы раз. Допустим, нам нужен противоположный результат: список файлов, где фразы нет. Для такого поиска используйте опцию `-L`. Например, так:

```
grep -RL "free software" .
```

Файлы с исходным кодом программ текстовые. Утилита grep работает только с текстовыми файлами. Поэтому grep хорошо справляется с поиском по исходному коду. Используйте её как дополнение к вашему редактору.

Возможно, вам понравилась утилита grep. Вы хотите обрабатывать ей документы в формате PDF[189] и MS Office. К сожалению, это не сработает. Формат этих файлов не текстовый. То есть данные в них закодированы. Для обработки таких файлов, вам понадобится другая утилита. Таблица 2-6 приводит альтернативы grep для не текстовых файлов.

Таблица 2-6. Утилиты для работы с PDF и MS Office файлами

Утилита	Функции
pdftotext[190]	Конвертирует PDF-файл в текстовый формат.
pdfgrep[191]	Ищет PDF-файл по его содержимому.
antiword[192]	Конвертирует файл MS Office в текстовый формат.
catdoc[193]	Конвертирует файл MS Office в текстовый формат.
xdoc2txt[194]	Конвертирует файлы PDF и MS Office в текстовый формат.

Некоторые из этих утилит устанавливаются в окружение MSYS2 пакетным менеджером pacman. В последней главе книги мы рассмотрим, как это сделать.

Упражнение 2-4. Поиск файлов утилитой grep

Напишите вызов утилиты grep для поиска системных утилит со свободной лицензией. Для открытого ПО предпочитают следующие лицензии:

1. GNU General Public License
2. MIT license
3. Apache license
4. BSD license

Информация о командах

Мы познакомились с командами навигации по файловой системе. Для каждой команды мы рассмотрели только часто используемые опции и параметры. Что делать, если вам нужны дополнительные функции?

[189] https://ru.wikipedia.org/wiki/Portable_Document_Format
[190] http://www.xpdfreader.com
[191] https://pdfgrep.org
[192] http://www.winfield.demon.nl
[193] https://www.wagner.pp.ru/~vitus/software/catdoc
[194] https://documentation.help/xdoc2txt/xdoc2txt_en.html

Все современные ОС и программы имеют встроенную документацию. Она редко бывает нужна благодаря наглядности графического интерфейса. Назначение иконок или кнопок обычно очевидно. Поэтому большинство пользователей документацией не пользуется.

При работе с интерфейсом командной строки режимы работы программ неочевидны. Но знать их надо, поскольку небрежность может привести к потере ваших данных.

Разработчики Unix сначала распространяли документацию в бумажном виде. Но это было неудобно. Объём информации быстро рос и скоро превысил размер одной книги. Чтобы сделать документацию доступнее, была разработана система man. Она предоставляет справку по любой установленной программе.

Система man — это центральное место для доступа к документации. Кроме этого каждая программа в Unix предоставляет краткую информацию о себе. Так например, у интерпретатора Bash есть своя система справки help. Рассмотрим её подробнее.

Чтобы вывести список встроенных команд Bash, выполните команду help без параметров. Иллюстрация 2-21 демонстрирует результат.

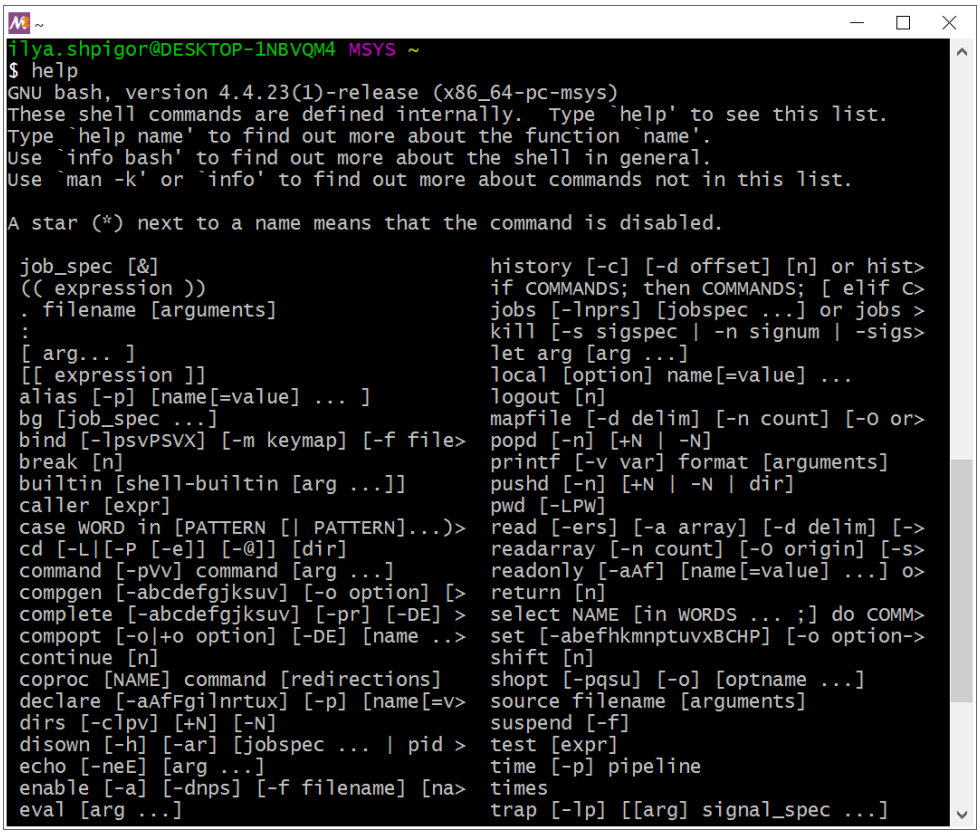


Иллюстрация 2-21. Результат выполнения команды `help`

Перед вами команды, которые Bash исполняет самостоятельно. Если команды нет в списке, она выполняется GNU-утилитой или сторонней программой.

Например, команда cd есть в списке help. Это значит, что Bash выполнит её самостоятельно.

Теперь предположим, что вы ввели команду find. Её в списке help нет. Поэтому Bash вызовет утилиту с именем find и передаст ей указанные вами параметры. Если найти find на жёстком диске не удалось, Bash сообщит об ошибке.

Где Bash ищет утилиты для выполнения команд? У него есть список системных путей, по которым устанавливаются программы. Этот список хранится в **переменной окружения** с именем `PATH`.

Переменная[195] — это область оперативной памяти. Обычно для обращения к области памяти нужно указать её адрес. Но переменная позволяет заменить адрес на имя. Обратившись по имени переменной, вы можете записать или прочитать её область памяти. С переменными окружения мы познакомимся в следующей главе.

Чтобы вывести доступные переменные окружения, вызовите утилиту env без параметров.

Представьте переменную как некоторое значение, у которого есть имя. Например, можно сказать: “время — 12 часов”. В этом случае “время” — это имя переменной, а “12 часов” — её значение. Значение хранится в памяти компьютера.

Теперь допустим, что вы читаете переменную из памяти. Для этого вы сообщаете компьютеру её имя “время”. Компьютер преобразует имя в адрес памяти. Он читает память, находит в ней значение “12 часов” и выдаёт его вам. Запись переменной происходит аналогично.

Команда echo выводит на экран не только строки, но и значения переменных. Например, следующая команда напечатает, чему равна переменная `PATH`:

```
echo "$PATH"
```

Зачем нам понадобился знак доллара $ перед именем переменной? Команда echo получает через входные параметры строки и выводит их на экран. Например, следующий вызов echo напечатает текст “123”:

```
echo 123
```

Знак доллара $ перед словом сообщает Bash, что это имя переменной. Когда Bash встречает знак доллара, он ищет следующее за ним слово в своём списке переменных. Если поиск удался, Bash заменяет имя на значение переменной. В противном случае имя заменяется на пустую строку.

[195] https://ru.wikipedia.org/wiki/Переменная_(программирование)

Заключать имена переменных в двойные кавычки " считается хорошей практикой[196]. Это предотвращает потенциальные ошибки. Представьте, что Bash подставляет значение переменной вместо её имени. Если в значении встречаются **управляющие символы**[197], интерпретатор их обработает. В результате подставленное значение будет отличаться от хранящегося в памяти. Это может привести к некорректному поведению программы.

Вернёмся к команде echo для вывода переменной PATH. Иллюстрация 2-22 демонстрирует результат её исполнения.

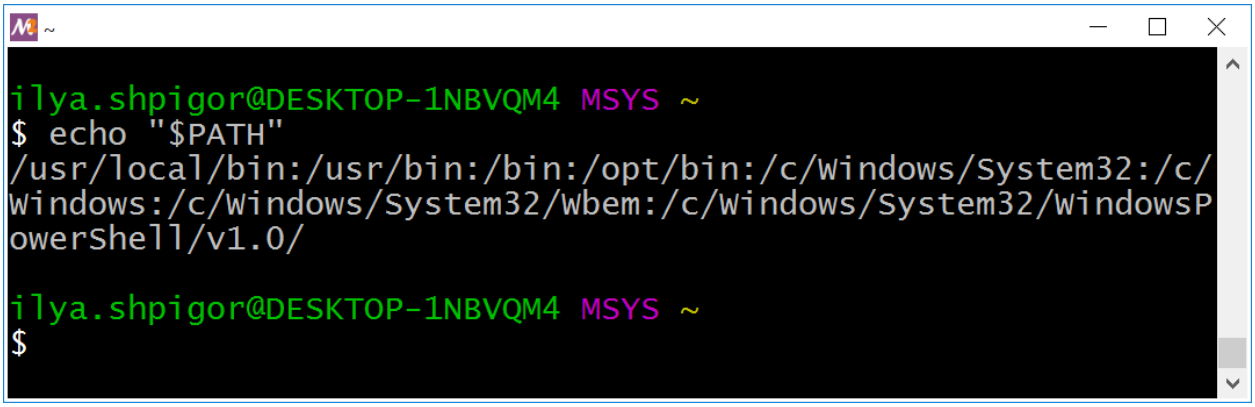


Иллюстрация 2-22. Значение переменной PATH

Что означает этот вывод? Перед нами список путей, разделённых двоеточиями. Если записать каждый путь с новой строки, получим следующее:

```
/usr/local/bin
/usr/bin
/bin
/opt/bin
/c/Windows/System32
/c/Windows
/c/Windows/System32/Wbem
/c/Windows/System32/WindowsPowerShell/v1.0/
```

Формат переменной PATH вызывает вопросы. Почему нельзя хранить пути в виде списка с переводом строки[198] вместо двоеточия? Тогда при выводе переменной на экран, её было бы удобнее читать. Короткий ответ: так проще программировать. Интерпретатор и некоторые GNU-утилиты обрабатывают символ перевода строки \n по-разному. Это может стать источником ошибок.

[196] https://www.tldp.org/LDP/abs/html/quotingvar.html
[197] https://ru.wikipedia.org/wiki/Управляющие_символы
[198] https://ru.wikipedia.org/wiki/Перевод_строки

Если вы ищете утилиту или программу на диске, переменная PATH подскажет, где искать. Кроме того не забывайте про утилиту для поиска find. Например, найдём её исполняемый файл такой командой:

```
find / -name find
```

Файл с именем find находится в каталогах /bin и /usr/bin.

Найти программу на диске можно быстрее. Для этого у Bash есть встроенная команда type. Передайте в неё имя интересующей вас программы. Команда type выведет абсолютный путь до её исполняемого файла как на иллюстрации 2-23.

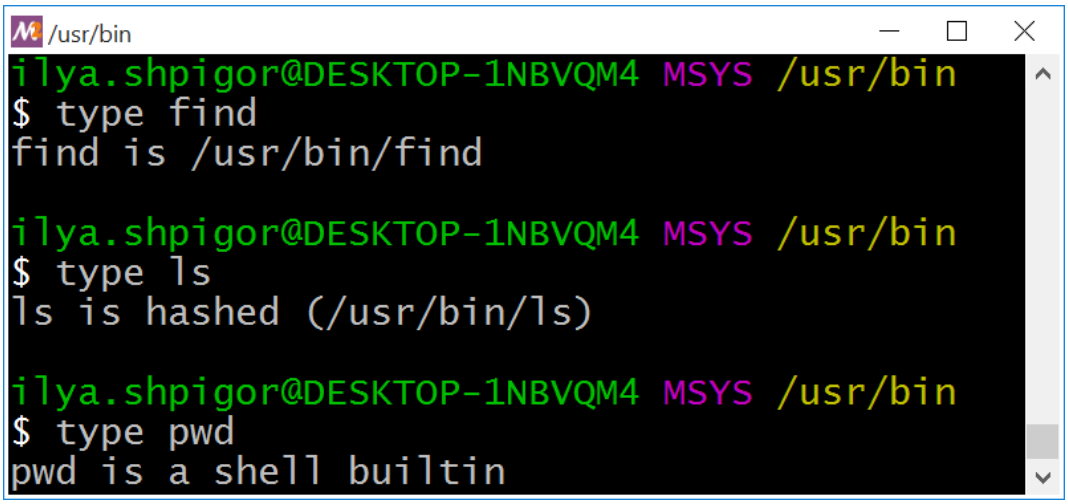


Иллюстрация 2-23. Результат выполнения команды type

Из иллюстрации 2-23 мы узнали, что исполняемые файлы утилит find и ls находятся в каталоге /usr/bin. Причём путь до утилиты ls **хэшируется**. Bash запоминает его и не ищет исполняемый файл ls при каждом вызове. Поэтому если перенести файл ls в другое место на диске, Bash его не найдёт.

В type можно передать встроенную команду интерпретатора. Тогда type сообщит, что оболочка исполняет эту команду самостоятельно. На иллюстрации 2-23 приведён пример для команды pwd.

Мы нашли исполняемый файл нужной утилиты. Как теперь узнать, какие входные параметры она принимает? Для вывода краткой справки вызовите утилиту с опцией --help. Вы получите вывод как на иллюстрации 2-24. Это краткая справка об утилите cat.

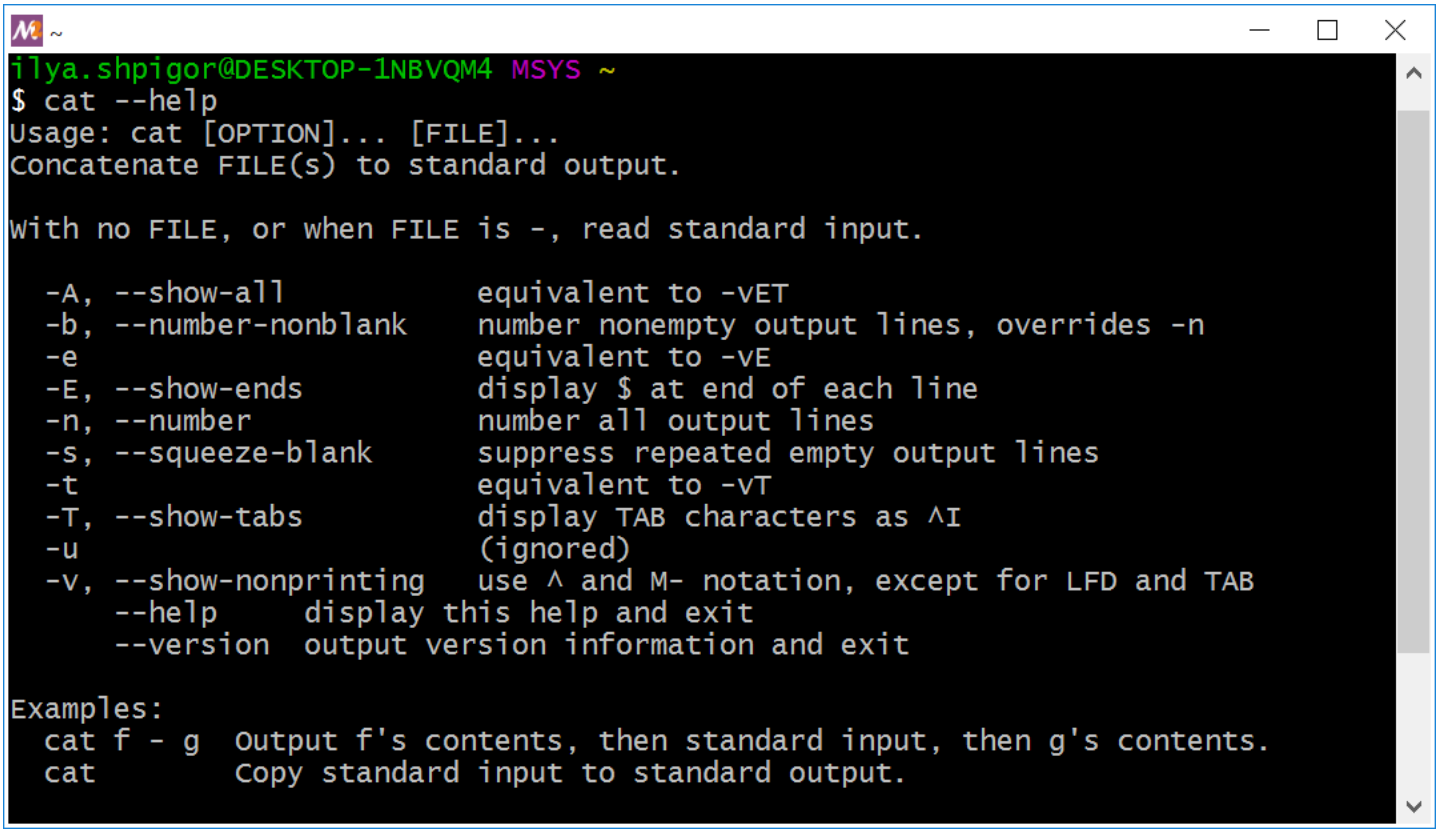


Иллюстрация 2-24. Краткая справка об утилите cat

Если язык вашей системы английский, справочная информация выводится на нём. Перевод справки на русский доступен в интернете[199]. Онлайн переводчик Google Translate[200] или DeepL[201] поможет перевести незнакомые английские слова. Если вы используйте Linux или macOS, переключите язык системы на русский. После этого документация станет на русском.

Если краткой справки недостаточно, обратитесь к системе документации info. Предположим, вы ищете примеры использования cat. Их выведет следующая команда:

```
info cat
```

Иллюстрация 2-25 демонстрирует результат команды info.

[199] https://www.opennet.ru/man.shtml?topic=cat&russian=0&category=&submit=%F0%CF%CB%C1%DA%C1%D4%D8+man
[200] https://translate.google.com
[201] https://www.deepl.com/translator

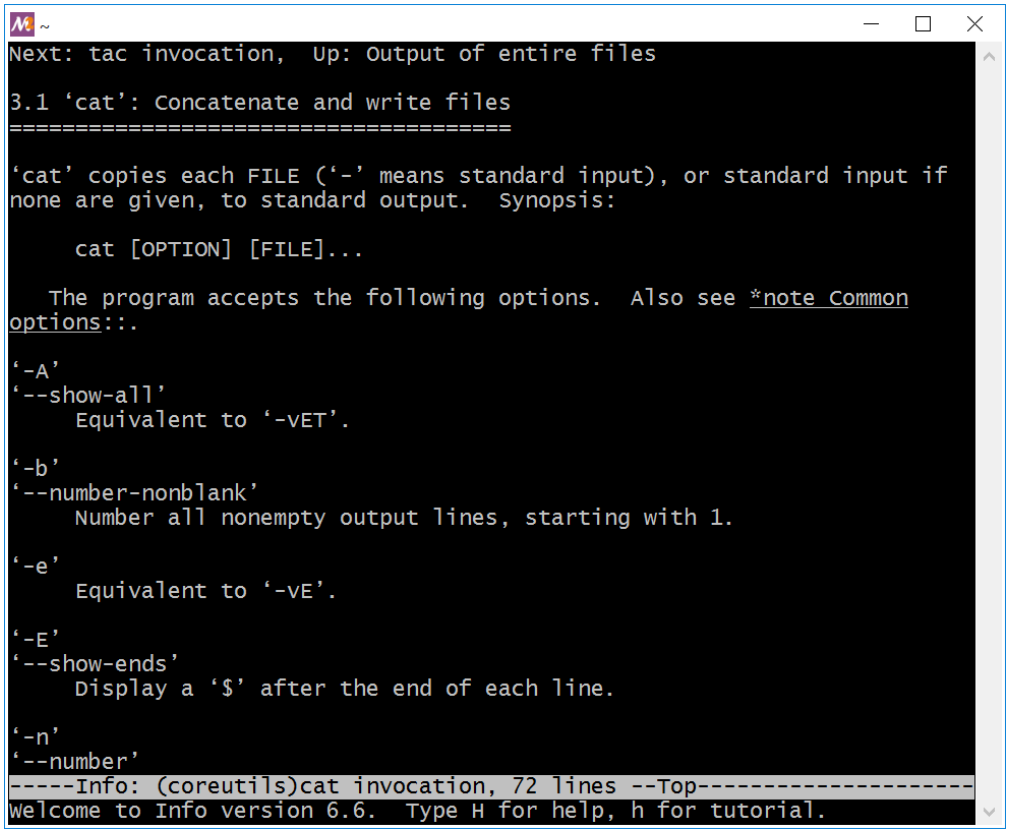


Иллюстрация 2-25. Справка info по утилите cat

Перед вами программа для чтения текстовых документов. Клавиши-стрелки либо PageUp и PageDown прокручивают текст вверх и вниз. Клавиша Q завершает программу.

Систему документацию info создали разработчики GNU-утилит. До неё все версии Unix использовали man. Возможности info и man похожи. В MSYS2 по умолчанию устанавливается современная система info.

В вашем Linux-дистрибутиве может быть установлена система man. Она выводит информацию так же, как info. Вот пример вызова справки для утилиты cat:

```
man cat
```

Когда имя нужной утилиты известно, о ней легко получить справку. Но что делать, если вы не знаете, какая утилита решает вашу задачу? Лучше искать ответ на этот вопрос в интернете. Советы по использованию командной строки лаконичнее инструкций для программ с графическим интерфейсом. Вам не нужны скриншоты и видеоролики с объяснениями каждого действия. Вместо этого достаточно одной строчки с командой, которая сделает всё необходимое.

Упражнение 2-5. Система документации

Найдите информацию по каждой встроенной команде и утилите из таблицы 2-1. Изучите параметры утилит `ls` и `find`, которые мы не рассмотрели.

Действия над файлами и каталогами

Мы узнали, как найти файл или каталог на диске. Теперь поговорим о действиях над ними. По опыту работы с графическим интерфейсом ОС вы знаете, что любой файл или каталог можно:

- Создать
- Удалить
- Скопировать
- Перенести или переименовать

Каждое из этих действий выполняется специальной GNU-утилитой. Таблица 2-7 приводит эти утилиты.

Таблица 2-7. Утилиты для работы с файлами и каталогами

Утилита	Действие	Примеры
`mkdir`	Создать каталог с указанным именем и путём.	`mkdir /tmp/docs` `mkdir -p tmp/docs/report`
`rm`	Удалить указанный файл или каталог по абсолютному или относительному пути.	`rm readme.txt` `rm -rf ~/tmp`
`cp`	Скопировать файл или каталог. Первым параметром передаётся текущий путь, а вторым — целевой.	`cp readme.txt tmp/readme.txt` `cp -r /tmp ~/tmp`
`mv`	Перенести или переименовать объект, указанный первым параметром.	`mv readme.txt documentation.txt.` `mv ~/tmp ~/backup`

У каждой утилиты есть опция `--help`. Она выводит краткую справку. Если нужный вам режим работы утилиты пропущен в книге, прочитайте о нём в документации. Если краткой справки окажется недостаточно, обратитесь к системе info или man.

Рассмотрим примеры использования утилит из таблицы 2-7.

mkdir

Утилита mkdir создаёт новый каталог по указанному абсолютному или относительному пути. Путь передаётся в первом параметре. Например, следующая команда создаёт каталог `docs` в домашнем каталоге пользователя:

```
mkdir ~/docs
```

Мы указали абсолютный путь до каталога `docs`. Его можно создать и по относительному пути. Для этого сначала перейдём в домашний каталог пользователя, а затем выполним mkdir:

```
cd ~
mkdir docs
```

Для создания вложенных каталогов у утилиты mkdir есть опция `-p`. Например, вы сохраняете отчёты за 2019 год по пути `~/docs/reports/2019`. Предположим, что каталогов `docs` и `reports` ещё нет. Сначала вам нужно создать их и только потом подкаталог `2019`. Это делает одна команда mkdir с опцией `-p`:

```
mkdir -p ~/docs/reports/2019
```

Если каталоги `docs` и `reports` уже существуют, утилита mkdir создаст только недостающую часть пути — подкаталог `2019`.

rm

Утилита rm удаляет файлы и каталоги. Они указываются по абсолютному или относительному пути.

Например, следующие две команды удаляют один и тот же файл `report.txt`:

```
rm report.txt
rm ~/docs/reports/2019/report.txt
```

Чтобы удалить несколько файлов за раз, передайте утилите rm их имена через пробел. Например, так:

```
rm report.txt ~/docs/reports/2019/report.txt
```

Предположим, вы удаляете десятки файлов. Перечислять их имена в вызове утилиты неудобно. В этом случае используйте шаблон поиска Bash. Для примера удалим текстовые файлы, имена которых начинаются со слова “report”. Вызов rm для них выглядит так:

```
rm ~/docs/reports/2019/report*.txt
```

При удалении защищённого от записи файла, утилита rm выведет предупреждение как на иллюстрации 2-26.

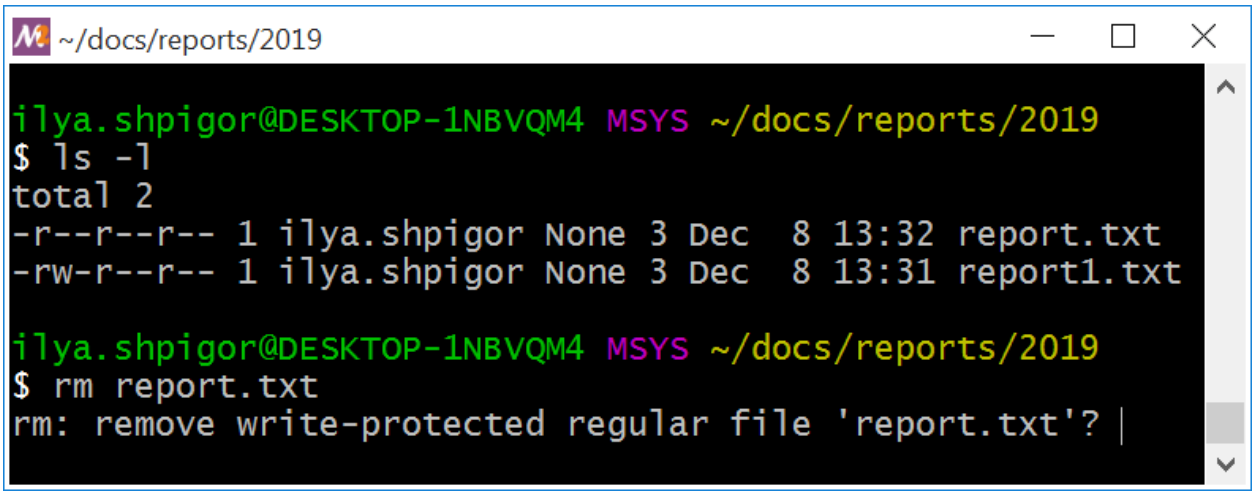


Иллюстрация 2-26. Предупреждение при удалении защищённого от записи файла

Чтобы удалить файл, нажмите клавишу Y (сокращение от yes) и Enter. Если вызвать утилиту с опцией `-f` или `--force`, она отработает без предупреждений. Вот пример такого вызова:

```
rm -f ~/docs/reports/2019/report*.txt
```

У опций GNU-утилит есть краткая и полная форма. Краткая состоит из одной буквы и начинается с тире `-`. Полная форма — это слово, перед которым стоит двойное тире `--`. Такой формат опций рекомендуется POSIX-стандартом[202] и GNU-расширением[203] к нему.

Утилита rm удаляет каталог, только если передать ей дополнительную опцию. Для удаления пустого каталога укажите опцию `-d` или `--dir`. Например, так:

```
rm -d ~/docs
```

Если в каталоге есть файлы или подкаталоги, вызовите утилиту с опцией `-r` или `--recursive`. Например:

[202] https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html
[203] https://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html

```
rm -r ~/docs
```

Опция `-r` удаляет и пустые каталоги тоже. Чтобы проще запомнить, используйте её для любых каталогов.

cp и mv

Утилиты для копирования и переименования работают по одному принципу. Первым параметром они принимают файл или каталог, над котором выполняется действие. Второй параметр — это новый путь, где окажется копируемый или переносимый объект.

Например, скопируем файл `report.txt` в текущем каталоге. Для этого вызовем утилиту cp:

```
cp report.txt report-2019.txt
```

Эта команда создаст новый файл с именем `report-2019.txt`. Его содержимое такое же как у `report.txt`.

Предположим, что старый файл с именем `report.txt` не нужен. После копирования его можно удалить утилитой rm. Но лучше вместо копирования перенести файл утилитой mv. Её вызов выглядит так:

```
mv report.txt report-2019.txt
```

Эта команда создаст новый файл с именем `report-2019.txt`. При этом она удалит старый файл `report.txt`. Таким образом утилита mv совмещает операции копирования и удаления.

Утилиты cp и mv работают с относительными и с абсолютными путями. Например, вы копируете файл `report.txt` из домашнего каталога в `~/docs/reports/2019`. Для этого выполните следующую команду:

```
cp ~/report.txt ~/docs/reports/2019
```

Тот же результат можно получить по-другому. Перейдите в домашний каталог и вызовите утилиту cp с относительными путями. Например, так:

```
1 cd ~
2 cp report.txt docs/reports/2019
```

Имя копии можно указать явно. Это полезно, когда оно должно отличаться от имени исходного файла. Для нашего примера вызов утилиты cp выглядит так:

```
cp ~/report.txt ~/docs/reports/2019/report-2019.txt
```

Перенос файлов из каталога в каталог работает точно так же как и копирование. Например, следующая команда перенесёт файл `report.txt`:

```
mv ~/report.txt ~/docs/reports/2019
```

А эта команда перенесёт файл и переименует:

```
mv ~/report.txt ~/docs/reports/2019/report-2019.txt
```

Переименовать каталог можно так же как и файл с помощью утилиты mv. Например:

```
mv ~/tmp ~/backup
```

Эта команда переименует каталог `tmp` в `backup`.

Утилита cp в обычном режиме работы не копирует каталоги. Предположим, вы копируете каталог `/tmp` с временными файлами в домашний каталог. Для этого вы выполняете такую команду:

```
cp /tmp ~
```

Команда завершится с ошибкой. Чтобы копирование сработало, укажите опцию `-r` или `--recursive`. Получится такой вызов:

```
cp -r /tmp ~
```

Предположим, вы копируете или переносите файл. Если по новому пути уже есть файл с таким же именем, утилиты cp и mv запросят подтверждение операции. В этом случае копируемый файл перезапишет существующий.

Если существующий файл не нужен, его можно перезаписать без подтверждения. Для этого используйте опцию `-f` или `--force`. Например:

```
1 cp -f ~/report.txt ~/tmp
2 mv -f ~/report.txt ~/tmp
```

Обе команды перезапишут существующий файл `report.txt` в каталоге `tmp`. Подтверждать операцию при этом не нужно.

Упражнение 2-6. Работа с файлами и каталогами

```
Упорядочьте свои фотографии за последние три месяца с помощью GNU-утилит.
Перед началом работы сделайте их резервную копию.
Разделите все фотографии по годам и месяцам.
Структура каталогов должна получиться следующая:

~/
  photo/
        2019/
             11/
             12/
        2020/
             01/
```

Права доступа

Утилита rm при удалении файла или каталога проверяет его **права доступа**[204]. Например, если файл защищён от записи, утилита выведет предупреждение. Что такое права доступа и для чего они нужны?

Права доступа ограничивают действия пользователя над файловой системой. За соблюдением этих прав следит операционная система. Благодаря этой функции, пользователю доступны только его файлы и каталоги. При этом доступ к данным других пользователей и компонентам ОС ограничен.

Права доступа позволяют нескольким пользователям работать на одном компьютере. До появления персональных компьютеров такой режим работы практиковался повсеместно. Вычислительные ресурсы были дороги. Поэтому эффективнее было решать несколько задач одновременно, чем ждать пока каждый пользователь закончит свою работу.

Обратимся ещё раз к иллюстрации 2-26. При вызове утилиты ls с опцией `-l` выводится таблица. В ней каждому файлу и каталогу соответствует строка. Значения столбцов в строке следующие:

1. Права доступа.
2. Число ссылок (hard link) на файл или каталог.
3. Владелец.
4. Группа владельца.
5. Размер объекта в байтах.
6. Дата и время последнего изменения.
7. Имя файла или каталога.

[204] https://ru.wikipedia.org/wiki/Права_доступа

Нас интересуют права доступа. Например, у файла `report.txt` они следующие: `-r--r--r--`. Разберёмся, что означает эта запись.

В Unix права доступа к файлу или каталогу хранятся в виде **битовой маски**[205]. Битовая маска — это положительное целое число. В памяти оно представляется в двоичном виде, как последовательность нулей и единиц. Значение каждого бита маски не зависит от других битов. Поэтому одна битовая маска хранит набор значений.

Какие значения можно упаковать в битовую маску? Например, признаки объекта. Каждый признак либо у объекта есть, либо нет. Если признак есть, его представляет бит со значением единица. Если признака нет, его бит равен нулю.

Вернёмся к правам доступа к файлу. Представим эти права как следующие признаки:

1. Разрешение на чтение.
2. Разрешение на запись.
3. Разрешение на исполнении.

Эти признаки помещаются в маску из трёх битов.

Исполнение файла означает его запуск как программы. Такие файлы называются исполняемыми.

Допустим, что к файлу есть полный доступ. Его содержимое можно прочитать или изменить. Сам файл можно скопировать, удалить или исполнить. Это значит, что пользователь имеет права на чтение, запись и исполнение файла. Запись разрешает не только изменение файла, но и его удаление. В этом случае маска с правами доступа к файлу выглядит так:

```
111
```

Все три бита в маске равны единицам.

Предположим, что читать и исполнять файл нельзя. Тогда первый бит маски (доступ на чтение) станет нулём. Третий бит (разрешение на исполнение) также станет нулём. В результате получим такую маску:

```
010
```

Для правильной работы с маской надо понимать, что означает каждый её бит. Сама маска не хранит этой информации.

Мы рассмотрели упрощённый пример прав доступа. Теперь разберёмся, как эти права устроены в Unix. Утилита ls выводит следующую строку с правами для файла `report.txt`:

[205] https://ru.wikipedia.org/wiki/Битовая_маска

```
-r--r--r--
```

Эта строка и есть битовая маска. В ней нулям соответствуют тире, а единицам — буквы латинского алфавита. Следуя этому правилу, строка `-r--r--r--` равна маске 0100100100. Если все биты маски равны единицам, утилита ls выведет строку `drwxrwxrwx`.

Строка прав доступа в Unix состоит из четырёх частей. Таблица 2-8 объясняет их значение.

Таблица 2-8. Части строки прав доступа в Unix

d	**rwx**	**rwx**	**rwx**
Признак каталога.	Права владельца файла или каталога. По умолчанию это тот, кто его создал.	Права группы пользователей, привязанной к файлу. По умолчанию это группа, к которой относится владелец.	Права всех остальных пользователей кроме владельца и группы, привязанной к файлу.

Для удобства каждую часть строки прав представляют отдельной битовой маской. На каждую из них отводится по 4 бита. Поэтому строка `-r--r--r--` представляется следующими четырьмя масками:

```
0000 0100 0100 0100
```

Что означают латинские буквы в строке прав доступа? Они соответствуют битам, установленным в единицу. Позиция каждого бита определяет действие пользователя над объектом: чтение, запись и исполнение. Буквы упрощают работу с битовой маской для человека. Согласитесь, что строка `-rw-r--r--` проще для чтения, чем двоичное число 0000011001000100.

Таблица 2-9 объясняет смысл каждой буквы в строке прав доступа.

Таблица 2-9. Буквы в строке прав доступа

Буква	**Значение для файла**	**Значение для каталога**
d	Если вместо буквы d первым символом стоит тире, это права для файла.	Права доступа соответствуют каталогу.
r	Чтение.	Вывод содержимого каталога.
w	Запись.	Создание, переименование или удаление файлов в каталоге.
x	Выполнение файла.	Переход в каталог, доступ к его файлам и подкаталогам.
—	Действие запрещено.	Действие запрещено.

Предположим, что все пользователи системы имеют полный доступ к файлу. Тогда строка его прав доступа выглядит так:

```
-rwxrwxrwx
```

Для каталога с теми же правами, первый символ тире поменяется на букву d. Его строка прав доступа выглядит так:

```
drwxrwxrwx
```

Утилита groups выводит список групп, к которым относится указанный пользователь. Вызовите её без параметров, чтобы получить группы текущего пользователя.

Теперь нам легко прочитать права доступа к файлам `report.txt` и `report1.txt` на иллюстрации 2-26. Первый файл могут читать все пользователи. Изменять и исполнять его не может никто. Файл `report1.txt` могут читать все. Изменять может только владелец. Исполнять не может никто.

Мы рассмотрели команды и утилиты для работы с файловой системой. Чтобы команда выполнилась успешно, запускающий её пользователь должен иметь доступ к целевому файлу или каталогу. Таблица 2-10 демонстрирует права, необходимые для каждой рассмотренной утилиты и команды.

Таблица 2-10. Права доступа для работы команд и утилит

Команда	Маска	Права доступа	Комментарий
`ls`	`r--`	Чтение	Только каталоги.
`cd`	`--x`	Выполнение	Только каталоги.
`mkdir`	`-wx`	Запись и выполнение.	Только каталоги.
`rm`	`-w-`	Запись	Для каталогов надо указывать опцию `-r`.
`cp`	`r--`	Чтение	Целевой каталог должен быть доступен на запись и исполнение.
`mv`	`r--`	Чтение	Целевой каталог должен быть доступен на запись и исполнение.
Исполнение	`r-x`	Чтение и выполнение.	Только для файлов.

Запуск файлов

В Windows есть строгие правила для определения какие файлы исполняемые, а какие нет. Расширение файла указывает его тип. Загрузчик Windows запускает только скомпилированные исполняемые файлы приложений. Они имеют расширение EXE или COM. Кроме них пользователь может запускать скрипты. Примеры их расширений: BAT, JS, PY и т.д. Каждый тип скрипта привязан к одному из установленных в системе интерпретаторов. Запустить скрипт нельзя, если для него нет подходящего интерпретатора.

Правила запуска файлов в Unix-окружении отличаются от Windows. В Unix можно запустить любой файл, если у пользователя есть права на его чтение и исполнение. В отличие от Windows расширение неважно. Например, файл с именем `report.txt` и расширением TXT можно исполнить, если назначить ему соответствующие права.

В Unix нет соглашения о расширении исполняемых файлов. Поэтому из имени файла нельзя однозначно определить его тип. Чтобы узнать тип, используйте утилиту file. Вот пример вызова утилиты для файла ls:

```
file /usr/bin/ls
```

В окружении MSYS2 она выведет на экран следующее:

```
/usr/bin/ls: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Wi\
ndows
```

Вывод означает, что файл имеет тип PE32[206]. Это исполняемый файл с машинным кодом. Он запускается загрузчиком Windows. Также в выводе указана разрядность файла: x86-64. Это значит, что утилита ls запустится только на 64-разрядных версиях Windows.

Окружение MSYS2 позволяет отбрасывать расширение EXE у исполняемых файлов. С точки зрения MSYS2 `ls` и `ls.exe` — это один и тот же файл. Поэтому загрузчик Windows корректно запускает утилиту ls.

Если запустить ту же команду file на ОС Linux, получится другой вывод. Например, такой:

```
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, in\
terpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=d0bc0fb9b\
3f60f72bbad3c5a1d24c9e2a1fde775, stripped
```

В Linux файл ls имеет тип ELF[207]. Это исполняемый файл с машинным кодом. Его запускает загрузчик Linux. Разрядность файла такая же как и в MSYS2: x86-64.

[206] https://ru.wikipedia.org/wiki/Portable_Executable
[207] https://ru.wikipedia.org/wiki/Executable_and_Linkable_Format

Мы научились отличать исполняемые файлы в Unix-окружении от неисполняемых. Теперь выясним, где их искать.

GNU-утилиты считаются частью ОС. Поэтому они доступны сразу после установки системы. Исполняемые файлы утилит находятся в каталогах `/bin` и `/usr/bin`. Пути к ним хранятся в переменной Bash с именем `PATH`. Вопрос в том, куда в Unix-окружении устанавливаются новые приложения?

В Windows на системном диске есть каталоги `Program Files` и `Program Files (x86)`. По умолчанию все приложения устанавливаются в них. Для каждого приложения создаётся новый подкаталог (например, `C:\Program Files (x86)\Notepad++`). В процессе установки в него копируются исполняемые файлы, библиотеки, файлы конфигурации и ресурсов. Без них приложение не запустится. Вместо каталогов `Program Files` и `Program Files (x86)` можно выбрать другой путь установки (например, `D:\Programs`). В этом случае в нём создаётся подкаталог приложения со всеми его файлами.

В Unix-окружении принято два варианта установки программ. Первый вариант напоминает подход Windows. В системном каталоге `/opt` создаётся подкаталог для приложения (например, `/opt/teamviewer`). В него копируются все файлы приложения. Этот вариант установки выбирают разработчики проприетарных программы с закрытым исходным кодом.

Утилиты и программы с открытым исходным кодом устанавливаются иначе. Приложению для работы нужны файлы различного типа. В Unix каждому типу файлов отводится свой системный каталог. Это значит, что исполняемые файлы всех приложений копируются в один каталог. Документация для них копируется в другой каталог. Файлы ресурсов всех приложений попадут в третий.

Таблица 2-11 объясняет назначение системных каталогов Unix.

Таблица 2-11. Назначение системных каталогов Unix

Каталог	Назначение
`/bin`	Исполняемые файлы системных утилит.
`/etc`	Конфигурационные файлы.
`/lib`	Библиотеки, необходимые для работы системных утилит.
`/usr/bin`	Исполняемые файлы приложений пользователя.
`/usr/lib`	Библиотеки, необходимые для приложений пользователя.
`/usr/local`	Приложения, скомпилированные пользователем самостоятельно.
`/usr/share`	Архитектурно-независимые файлы ресурсов приложений пользователя (например, иконки).

Таблица 2-11. Назначение системных каталогов Unix

Каталог	Назначение
`/var`	Файлы, создаваемые приложениями в процессе работы (например, лог-файлы).

Копирование файлов одного типа в специальные системные каталоги кажется неудачным решением. Его минус в сложности сопровождения. Например, приложение обновляется до следующей версии. Для этого надо найти и обновить все его файлы во всех системных каталогах. Если пропустить один из файлов, приложение перестанет работать.

Но у решения со специальными системными каталогами есть и сильная сторона. В Windows каждое приложение при установке копирует в свой каталог все необходимые ему файлы. Среди этих файлов есть и библиотеки с подпрограммами. Некоторым приложениям для работы нужны одни и те же библиотеки. В результате в файловой системе накапливаются десятки копий этих библиотек. У каждого приложения — своя собственная копия.

В Unix удаётся избежать копирование библиотек. Предположим, что все приложения соблюдают соглашение и устанавливают свои файлы в правильные каталоги. В этом случае каждой программе легко найти файлы другой программы. Благодаря этому, один и тот же файл используется всеми приложениями, которым он нужен. Поэтому в файловой системе достаточно хранить по одному экземпляру каждой библиотеки.

Предположим, что мы установили приложение (например, браузер). Согласно таблице 2-11 его исполняемый файл (например, `firefox`) копируется в каталог `/usr/bin`. Как запустить это приложение из командной оболочки Bash? Для этого есть несколько способов:

1. По имени исполняемого файла.
2. По абсолютному пути.
3. По относительному пути.

Рассмотрим каждый способ подробнее.

Первый вариант нам уже знаком. Именно так вызываются GNU-утилиты. Например, запустим find из каталога `/usr/bin`. Для этого выполним команду:

```
find --help
```

Аналогичная команда запускает и приложение, установленное в ОС Linux. Например, браузер firefox запускается так:

```
firefox
```

Почему эта команда сработала? Исполняемый файл `firefox` находится в системном каталоге `/usr/bin`. Его путь хранится в переменной окружения Bash с именем `PATH`. Bash получает команду `firefox`. Дальше он ищет исполняемый файл с таким именем по каждому из путей переменной `PATH`. Bash находит файл в каталоге `/usr/bin` и запускает.

Порядок путей в переменной `PATH` важен. В этом порядке Bash проходит по системным каталогам и ищет в них файл. Предположим, что файл `firefox` есть в обоих каталогах `/usr/local/bin` и `/usr/bin`. Если путь `/usr/local/bin` идёт первым в списке `PATH`, Bash запустит файл из него.

Второй способ запуска приложения похож на первый. Вместо имени исполняемого файла мы указываем абсолютный путь к нему. Например, запустим браузер следующей командой:

```
/usr/bin/firefox
```

Часто этим способом запускают проприетарные приложения. Они устанавливаются в системный каталог `/opt`. По умолчанию его нет в переменной `PATH`. Поэтому Bash не находит исполняемый файл самостоятельно. Чтобы команда сработала, вам нужно указать абсолютный путь до файла.

Третий вариант запуска приложения — это что-то среднее между первым и вторым способами. Если вы перейдёте в каталог `/usr`, то запустить браузер можно по относительному пути исполняемого файла:

```
1 cd /usr
2 bin/firefox
```

Теперь предположим, что исполняемый файл `firefox` находится в каталоге `/opt/firefox/bin`. Перейдём в этот каталог командой `cd` и запустим браузер так:

```
1 cd /opt/firefox/bin
2 firefox
```

В этом случае Bash сообщит, что файл `firefox` не найден. Дело в том, что мы пытаемся запустить приложение по имени исполняемого файла. Это первый способ запуска. Bash ищет файл `firefox` в путях переменной `PATH`. Но приложение установлено в `/opt`, которого в `PATH` нет.

Правильно указывать не имя файла, а относительный путь. Файл находится в текущем каталоге. Этот каталог обозначается точкой `.`. Учтём это и исправим команды запуска так:

```
1 cd /opt/firefox/bin
2 ./firefox
```

Теперь Bash поймёт, что исполняемый файл находится в текущем каталоге.

Рассмотрим, как добавить новый путь (например, `/opt/firefox/bin`) в переменную `PATH`. Для этого выполните следующие действия:

1. Перейдите в домашний каталог пользователя:

```
cd ~
```

2. Выведите соответствующий ему Windows путь (см. пример на иллюстрации 2-7):

```
pwd -W
```

3. В редакторе Блокнот или любом другом откройте файл `.bash_profile` из этого каталога.
4. В конец файла добавьте следующую строку:

```
PATH="/opt/firefox/bin:${PATH}"
```

Мы переобъявили переменную `PATH` и добавили в неё новый путь. Подробнее действия с переменными мы разберём в следующей главе.

Чтобы изменения вступили в силу, перезапустите терминал MSYS2. Теперь если вы наберёте команду `firefox`, Bash найдёт и запустит этот файл по пути `/opt/firefox/bin`.

Дополнительные возможности Bash

Мы познакомились со встроенными командами Bash и GNU-утилитами для работы с файловой системой. Теперь вы умеете запускать программы, копировать и удалять файлы из командной строки. То же самое легко сделать через графический интерфейс. Для простых задач выбор командного или графического интерфейса — это дело вкуса.

Интерпретатор Bash предлагает возможности, которых нет в графическом интерфейсе. Благодаря им, некоторые задачи выполняются быстрее и проще. Применяйте эти возможности, чтобы сэкономить время и избежать ошибок.

В этом разделе рассмотрим следующие механизмы Bash:

1. Перенаправление ввода-вывода.
2. Конвейеры.
3. Логические операторы.

Философия Unix

Дуглас Макилрой[208] — один из разработчиков Unix. Он обобщил философию этой ОС в трёх пунктах:

1. Пишите программы, которые делают что-то одно и делают это хорошо.
2. Пишите программы, которые бы работали вместе.
3. Пишите программы, которые бы поддерживали текстовые потоки, поскольку это универсальный интерфейс.

Текстовый формат данных — это основа философии Unix. На нём строятся первые два правила.

Благодаря текстовому формату, данные легко передавать между программами. Предположим, что два разработчика независимо друг от друга написали две утилиты. Обе утилиты принимают на вход и выводят результаты в виде текста. Теперь вы хотите скомбинировать эти утилиты для решения своей задачи. Сделать это просто. Не нужно конвертировать данные из одного формата в другой или расширять функциональность утилит. Вместо этого передайте вывод одной утилиты на вход другой.

Когда взаимодействие программ легко наладить, незачем перегружать каждую из них дополнительными возможностями. Например, вы пишете программу для копирования файлов. Она хорошо справляется со своей задачей. Но со временем вы понимаете, что ей нужна функция поиска. Тогда находить и копировать файлы было бы быстрее. Также было бы удобно создавать каталоги прямо в программе, чтобы переносить в них файлы. Этот пример показывает, что требования к приложению быстро растут. Если программа работает самостоятельно, придётся расширять её функции.

Когда приложения работают вместе, каждое из них решает только свою задачу. Если требуется дополнительная операция, проще обратиться к другой утилите. Не надо добавлять новую функцию в приложение. Она уже есть готовая и хорошо протестированная. Просто вызовите внешнюю утилиту, которая сделает то что нужно за вас. Именно такой подход поощряет философия Unix.

Перенаправление ввода-вывода

Утилиты командной строки появились в первых версиях Unix. Большая часть этих утилит попала в POSIX-стандарт. В современных Linux дистрибутивах их заменили на GNU-утилиты. Эти утилиты предлагают больше возможностей. В то же время они следуют стандарту POSIX и обеспечивают обратную совместимость со старыми программами.

GNU-утилиты следуют философии Unix. Они используют текстовый формат для ввода и вывода данных. Поэтому их так же легко комбинировать, как и Unix-утилиты.

[208] https://ru.wikipedia.org/wiki/Макилрой,_Дуглас

При комбинировании GNU-утилит возникает вопрос. Как правильно передавать текстовые данные между ними? У этой задачи есть несколько решений.

Предположим, что вывод утилиты помещается в одну строку. Вам нужно передать её на вход другой утилите. Для этого воспользуйтесь буфером обмена. Выделите мышью вывод утилиты. Затем наберите следующую команду и вставьте содержимое из буфера обмена в её конец. Это простой метод. Но он не сработает для копирования нескольких строк. При их вставке Bash интерпретирует разделители строк как нажатия клавиши Enter. По нажатию Enter команда начнёт исполняться. Из-за этого часть копируемых строк потеряется.

Другое решение — использовать файловую систему. Создайте временный файл, чтобы сохранить вывод одной утилиты. Затем передайте имя файла другой утилите. Она прочитает его содержимое и получит данные. Такой подход удобнее буфера обмена по двум причинам:

1. Нет ограничения на количество передаваемых строк.
2. Нет ручных операций с буфером обмена.

У Bash есть механизмы для перенаправления ввода и вывода команд в файлы. Это значит, что вашему приложению достаточно читать текстовые данные на входе и выводить результат на экран. Всю работу по перенаправлению этих данных возьмёт на себя Bash.

Рассмотрим пример. Предположим, что мы ищем файлы на диске. Результат поиска надо сохранить в файл. Для решения задачи воспользуемся утилитой find и оператором перенаправления `1>`. Тогда команда вызова утилиты выглядит так:

```
find / -path */doc/* -name README 1> readme_list.txt
```

После выполнения команды в текущем каталоге появится файл `readme_list.txt`. Утилита find запишет в него свой вывод. Формат вывода такой же, как если бы он печатался на экран. Если файл с именем `readme_list.txt` уже существует, find перезапишет его содержимое.

Что означает оператор `1>`? Это перенаправление стандартного потока вывода[209]. В Unix-окружении есть три стандартных потока[210]. Таблица 2-12 объясняет их назначение.

Таблица 2-12. Стандартные потоки POSIX

Номер	Название	Применение
0	Стандартный поток ввода (standard input или stdin).	Данные, которые передаются на вход программы. По умолчанию они поступают с устройства ввода типа клавиатуры.
1	Стандартный поток вывода (standard output или stdout).	Данные, которые выводит программа. По умолчанию они печатаются в окне терминала.

[209] https://ru.wikipedia.org/wiki/Перенаправление_ввода-вывода
[210] https://ru.wikipedia.org/wiki/Стандартные_потоки

Таблица 2-12. Стандартные потоки POSIX

Номер	Название	Применение
2	Стандартный поток ошибок (standard error или stderr).	Данные об ошибках, которые выводит программа. По умолчанию они печатаются в окне терминала.

Приложение работает в программном окружении операционной системы. Поток представляет собой канал связи между приложением и окружением. В ранних Unix-системах информация вводилась и выводилась через физические каналы. Ввод был привязан к клавиатуре, а вывод — к монитору. Потоки ввели как абстракцию[211] над этими каналами. Абстракция позволила работать с разными объектами по одному и тому же алгоритму. Так ввод с реального устройства можно заменить на ввод из файла. Аналогично печать на экран можно заменить на вывод в файл. При этом за ввод-вывод отвечает один и тот же код ОС.

Назначение потоков ввода и вывода понятно. Но зачем нужен поток ошибок? Представьте, что вы запустили утилиту find для поиска файлов. К некоторым каталогам у вас нет доступа. При попытке прочитать их содержимое выводится сообщение об ошибке. Если утилита нашла много файлов, сообщения об ошибках потеряются в её выводе. Разделение потоков вывода и ошибок поможет в этом случае. Перенаправьте в файл поток вывода. Тогда на экран напечатаются только сообщения об ошибках.

Чтобы перенаправить поток ошибок, используйте оператор `2>`. Например, добавим его в вызов утилиты find:

```
find / -path */doc/* -name README 2> errors.txt
```

Цифра перед знаком больше в операторах `2>` и `1>` означает номер потока.

Для перенаправления потока ввода используйте оператор `0<`. Вот простой, но не правильный пример поиска шаблона `Bash` в файле `README.txt`:

```
grep "Bash" 0< README.txt
```

Эта команда использует интерфейс утилиты grep для обработки потока ввода. Но grep умеет сама читать содержимое указанного файла. Поэтому лучше в неё всегда передавать имя файла. Например, так:

```
grep "Bash" README.txt
```

Рассмотрим пример посложнее. Некоторые руководства по Bash рекомендуют команду echo для вывода содержимого файла на экран. Например, с ней вывод файла `README.txt` выглядит так:

[211] https://ru.wikipedia.org/wiki/Уровень_абстракции_(программирование)

```
echo $( 0< README.txt )
```

Здесь echo получает на вход результат выполнения следующей команды:

```
0< README.txt
```

Замена вызова команды на её результат называется **подстановкой команды**. Когда Bash встречает символы `$(` и `)`, он выполняет заключённую между ними команду и подставляет её вывод.

Из-за подстановки команд наш вызов echo выполняется в два этапа:

1. Передать содержимое файла `README.txt` на стандартный поток ввода.
2. Вывести данные из стандартного потока ввода командой echo.

При подстановке команд учитывайте порядок исполнения. Сначала Bash выполняет все подстановки по порядку. Потом он выполняет получившуюся команду целиком.

Из-за нарушения порядка исполнения следующий вызов find завершится с ошибкой:

```
$ find / -path */doc/* -name README -exec echo $(0< {}) \;
```

Текст ошибки следующий:

```
bash: {}: No such file or directory
```

Команда должна вывести содержимое файлов, найденных утилитой find. Но вместо этого Bash выдал ошибку. Проблема в том, что команда `0< {}` выполнится перед вызовом утилиты find. В результате содержимое файла с именем `{}` перенаправляется на стандартный поток ввода. Но файла с таким именем не существует. Мы ожидали, что утилита find вместо символов `{}` подставит свой результат. Но она выполняется после подстановки команды `0< {}`, а не до.

Чтобы вызов find сработал, замените команду echo на утилиту cat. Получится следующее:

```
find / -path */doc/* -name README -exec cat {} \;
```

Команда выведет на экран содержимое найденных файлов.

Операторы перенаправления потоков ввода и вывода часто используют. Поэтому для них ввели краткую форму: ввод `<` и вывод `>`.

Вызов find с краткой формой перенаправления выглядит так:

```
find / -path */doc/* -name README > readme_list.txt
```

Вот вызов echo с краткой формой перенаправления:

```
echo $( < README.txt )
```

Предположим, что вы перенаправляете поток вывода в уже существующий файл. Его содержимое нельзя удалять. В этом случае допишите результат команды в конец файла. Для этого замените оператор `>` на `>>`.

Например, приложения на вашем компьютере установлены в каталоги `/usr` и `/opt`. Тогда следующие два вызова find найдут их README файлы:

```
find /usr -path */doc/* -name README > readme_list.txt
find /opt -name README >> readme_list.txt
```

Первый вызов find создаст файл `readme_list.txt` и запишет в него результат. Если файл уже существует, его содержимое перезапишется. Вторая команда find допишет свой результат в конец `readme_list.txt`. Если бы файла ещё не было, оператор `>>` его бы создал.

Полная форма оператора `>>` выглядит как `1>>`. Чтобы перенаправить поток ошибок без перезаписи файла, используйте оператор `2>>`.

Иногда нужно перенаправить и поток вывода, и поток ошибок в один файл. Для этого используйте операторы `&>` или `&>>`. Первый оператор перезапишет существующий файл. Второй — допишет данные в его конец. Например:

```
find / -path */doc/* -name README &> result_and_errors.txt
```

Эта команда работает в Bash, но не в Bourne shell. Дело в том, что операторы `&>` и `&>>` — это Bash расширения. Их нет в POSIX-стандарте. Если совместимость со стандартом важна, перенаправляйте поток ошибок в поток вывода с помощью оператора `2>&1`. Например, так:

```
find / -path */doc/* -name README > result_and_errors.txt 2>&1
```

Такое перенаправление называется **дублированием потоков** (duplicating). Применяйте его для записи двух потоков в файл или передаче их данных через конвейер другой команде.

Будьте осторожны с дублированием потоков. Легко допустить ошибку и перепутать порядок операторов в команде. Если вы работайте на Bash, всегда используйте операторы `&>` или `&>>`.

Вот пример ошибки с дублированием потоков:

```
find / -path */doc/* -name README 2>&1 > result_and_errors.txt
```

В результате поток ошибок выводится на экран, а не в файл `result_and_errors.txt`. Почему команда работает неправильно? Ведь мы перенаправили поток ошибок в поток вывода оператором `2>&1`. Рассмотрим эту ошибку подробнее.

Стандарт POSIX вводит понятие **файлового дескриптора**[212]. Дескриптор — это указатель на файл или канал коммуникации. Дескрипторы, как абстракция, нужны для эффективной работы с потоками. При запуске программы дескрипторы потоков вывода и ошибок указывают на окно терминала. Их можно связать с файлом. В этом случае дескрипторы потоков будут указывать на этот файл. Подробнее этот механизм описан в статье BashGuide[213].

Вернёмся к нашему вызову утилиты find. Bash выполняет перенаправления потоков слева направо. Их порядок представлен в таблице 2-13.

Таблица 2-13. Порядок перенаправления потоков

Порядок	Перенаправление	Результат
1	`2>&1`	Теперь поток ошибок указывает на то же, на что и поток вывода. В нашем случае это окно терминала.
2	`> result_and_errors.txt`	Теперь поток вывода указывает на файл `result_and_errors.txt`. При этом поток ошибок по-прежнему связан с окном терминала.

Исправим наш вызов find. Для этого изменим порядок операторов перенаправления. Сначала должен идти вывод в файл, а потом дублирование. Получим следующее:

```
find / -path */doc/* -name README > result_and_errors.txt 2>&1
```

Сначала поток вывода связывается с файлом. Затем поток ошибок перенаправляется в тот же файл.

Для записи потока вывода и потока ошибок в разные файлы укажите операторы перенаправления друг за другом. Например, так:

```
find / -path */doc/* -name README > result.txt 2> errors.txt
```

Конвейеры

Передавать данные между утилитами через временные файлы неудобно. Во-первых, надо запоминать их пути. Во-вторых, после передачи данных файлы надо удалять, чтобы зря не расходовать место на жёстком диске.

[212] https://ru.wikipedia.org/wiki/Файловый_дескриптор
[213] http://mywiki.wooledge.org/BashFAQ/055

Вместо временных файлов передавайте данные через **конвейеры**[214] (pipeline). Конвейер — это механизм взаимодействия процессов. Он передаёт сообщения без использования файловой системы.

Рассмотрим пример. Предположим, что мы ищем информацию о лицензии[215] в документации по Bash. Для этого применим утилиту grep. Выполним следующую команду в каталоге с документацией:

```
grep -R "GNU" /usr/share/doc/bash
```

Информацию о лицензии можно найти не только в каталоге с документацией, но и в info справке по Bash. Обработаем поток вывода info и найдём в нём нужное. Для этого применим конвейер и утилиту grep. Например, так:

```
info bash | grep -n "GNU"
```

Утилита info передаёт свои данные на поток вывода. Далее в команде идёт конвейер, обозначенный символом `|`. Он принимает вывод команды слева и передаёт его на ввод команде справа. Таким образом утилита grep получит текст справки по Bash. В этом тексте она ищет строки со словом “GNU”. Если Bash распространяется под GNU GPL[216] лицензией, вывод grep будет непустым.

В вызов утилиты grep мы добавили опцию `-n`. С ней grep выводит номера найденных строк. Это удобно для поиска конкретного места в файле.

du

Рассмотрим пример использования конвейеров посложнее. Утилита du оценивает использование дискового пространства. Запустите её без параметров в текущем каталоге. Она **рекурсивно** пройдёт по всем его подкаталогам и выведет их размер.

Рекурсивный обход означает повторение операции перехода в подкаталог. Алгоритм обхода выглядит так:

1. Проверить содержимое текущего каталога.
2. Если есть не посещённый подкаталог, перейти в него и начать с пункта 1 алгоритма.
3. Если все подкаталоги посещены, перейти на уровень выше и начать с пункта 1 алгоритма.
4. Если перейти на уровень выше нельзя, завершить алгоритм.

[214] https://ru.wikipedia.org/wiki/Конвейер_(Unix)
[215] https://ru.wikipedia.org/wiki/Лицензия_на_программное_обеспечение
[216] https://ru.wikipedia.org/wiki/GNU_General_Public_License

Следуя этому алгоритму, мы обойдём все подкаталоги от выбранной точки файловой системы. Это универсальный алгоритм обхода. К нему можно добавить действие над каждым найденным подкаталогом. В случае утилиты du действием будет расчёт занимаемого дискового пространства. В итоге алгоритм утилиты выглядит так:

1. Проверить содержимое текущего каталога.
2. Если есть не посещённый подкаталог, перейти в него и начать с пункта 1 алгоритма.
3. Если все подкаталоги посещены или их нет:

   3.1 Рассчитать и вывести на экран дисковое пространство, занимаемое текущим каталогом.

   3.2 Перейти на уровень выше.

   3.3 Начать с пункта 1 алгоритма.
4. Если перейти на уровень выше нельзя, завершить алгоритм.

Утилит du принимает на вход пути до каталогов и файлов. Каталоги она обходит рекурсивно, а для файлов выводит их размер.

Вызовем утилиту du для системного каталога `/usr/share` следующим образом:

```
du /usr/share
```

Вот сокращенный пример её вывода:

```
261     /usr/share/aclocal
58      /usr/share/awk
3623    /usr/share/bash-completion/completions
5       /usr/share/bash-completion/helpers
3700    /usr/share/bash-completion
2       /usr/share/cmake/bash-completion
2       /usr/share/cmake
8       /usr/share/cygwin
1692    /usr/share/doc/bash
85      /usr/share/doc/flex
...
```

Перед нами таблица. Она состоит из двух столбцов. В правом указан подкаталог, а в левом — сколько байтов он занимает на диске. В этот вывод можно добавить статистику по файлам в подкаталогах. Для этого укажите опцию `-a`. Получится следующая команда:

```
du /usr/share -a
```

Опция `-h` сделает вывод утилиты du нагляднее. С ней утилита будет переводить байты в килобайты, мегабайты и гигабайты.

Предположим, что мы оцениваем размер HTML файлов в каталоге `/usr/share`. Это сделает следующая команда:

```
du /usr/share -a -h | grep "\.html"
```

Здесь вывод утилиты du передаётся на вход grep через конвейер. Далее grep выводит строки, в которых встречается шаблон ".html".

Почему в шаблоне ".html" точка экранирована обратным слэшем? Дело в том, что точка означает однократное вхождение любого символа. Если указать шаблон ".html", в вывод утилиты grep попадут лишние файлы (например, `pod2html.1perl.gz`) и подкаталоги (`/usr/share/doc/pcre/html`). Если точку экранировать, утилита grep обработает её как символ точка.

В примере с утилитами du и grep конвейер объединяет только две команды. Однако, количество команд в конвейере не ограничено. Для примера отсортируем найденные HTML файлы в порядке убывания. Для этого добавим в конвейер утилиту sort. Получим такую команду:

```
du /usr/share -a -h | grep "\.html" | sort -h -r
```

По умолчанию утилита sort сортирует строки в порядке возрастания по алфавиту. Это называется **лексикографической сортировкой**[217]. Например, есть файл со следующими строками:

```
abc
aaaa
aaa
dca
bcd
dec
```

Если вызвать утилиту sort без опций, она отсортирует строки файла так:

```
aaa
aaaa
abc
bcd
dca
dec
```

Чтобы обратить порядок сортировки, укажите опцию sort `-r`. С ней результат будет такой:

[217] https://ru.wikipedia.org/wiki/Лексикографический_порядок

```
dec
dca
bcd
abc
aaaa
aaa
```

В выводе утилиты du в первом столбце идут размеры HTML файлов. Именно их будет сортировать sort. Размеры файлов — это числа. Лексикографическая сортировка не подходит для чисел. Чтобы понять почему, рассмотрим пример.

Есть следующий файл с числами:

```
3
100
2
```

Лексикографическая сортировка по возрастанию даст такой результат:

```
100
2
3
```

Число 100 оказалось меньше 2 и 3. Причина в том, что ASCII-код символа 1 меньше чем символов 2 и 3. Чтобы отсортировать целые числа по значению, используйте опцию sort `-n`.

В нашем конвейере из трёх команд утилита du вызывается с опцией `-h`. Это значит, что в её выводе байты будут переведены в крупные единицы измерения. Чтобы утилита sort обработала их корректно, вызовите её с той же опцией `-h`.

Конвейеры сочетаются с перенаправлением потоков. Например, следующая команда сохранит отфильтрованный и отсортированный вывод утилиты du в файл `result.txt`:

```
du /usr/share -a -h | grep "\.html" | sort -h -r > result.txt
```

Предположим, вы сочетаете конвейеры и перенаправление потоков. При этом поток выходных данных направляется и в файл, и на вход утилите. Как это сделать? У Bash нет встроенного механизма для этой задачи. Но её решает утилита tee. Рассмотрим пример.

Выполните следующую команду:

```
du /usr/share -a -h | tee result.txt
```

Команда выводит результат утилиты du на экран. Этот же результат она записывает в файл `result.txt`. Таким образом утилита tee продублировала свой поток ввода в указанный файл и в поток вывода. При этом содержимое уже существующего файла `result.txt` перезапишется. Чтобы дописать вывод tee в конец файла, используйте опцию `-a`.

Для тестирования бывает полезно прочитать поток данных между командами в конвейере. Утилита tee делает это. Вызовите её между командами. Например, так:

```
du /usr/share -a -h | tee du.txt | grep "\.html" | tee grep.txt | sort -h -r > resul\
t.txt
```

Вывод каждой команды конвейера попадёт в соответствующий файл. Эти промежуточные результаты пригодятся при поиске ошибок. Конечный результат работы конвейера по-прежнему записывается в файл `result.txt`.

xargs

Параметр `-exec` утилиты find выполняет указанную команду для каждого найденного объекта. Это напоминает конвейер тем, что find передаёт свой результат другой команде. Поведение конвейеров и опции find похоже внешне, но устроены они различно. Выбирайте подходящий механизм в зависимости от задачи.

Рассмотрим, как исполняется команда в опции `-exec` утилиты find. Её выполняет какая-то программа. Эту программу запускает встроенный интерпретатор find. Интерпретатор передаёт в неё найденный объект. Обратите внимание, что Bash не участвует в вызове `-exec`. Это значит, что в вызове нельзя использовать следующее:

- Встроенные команды Bash.
- Функции.
- Конвейеры.
- Перенаправление потоков.
- Условные операторы.
- Циклы.

Выполним следующую команду:

```
find ~ -type f -exec echo {} \;
```

В опции `-exec` утилита find вызывает встроенную команду Bash — echo. Это не приведёт к ошибке. Почему? На самом деле в нашем примере вызывается не команда Bash, а утилита. Некоторые встроенные команды Bash продублированы в виде отдельных утилит. Они хранятся в системном каталоге `/bin`. Среди них вы найдёте и файл `/bin/echo`. Именно он вызывается в нашем примере.

Предположим, что в действии `-exec` нельзя обойтись без Bash-команды или условной конструкции. В этом случае запустите интерпретатор явно и передайте в него команду. Например, так:

```
find ~ -type f -exec bash -c 'echo {}' \;
```

Результат этой команды такой же, как и у предыдущей. Она выведет результаты поиска find на экран.

Вывод find можно перенаправить другой утилите через конвейер. Но тогда конвейер передаст текст, а не отдельные имена объектов. Между текстом и именами есть разница. Например, выполните следующую команду:

```
find ~ -type f | grep "bash"
```

Её вывод выглядит например так:

```
1 /home/ilya.shpigor/.bashrc
2 /home/ilya.shpigor/.bash_history
3 /home/ilya.shpigor/.bash_logout
4 /home/ilya.shpigor/.bash_profile
```

Конвейер передал вывод find на поток ввода утилите grep. В результате grep отфильтровала текст. Она вывела только имена файлов, в которых встречается шаблон “bash”.

Когда find передаёт результаты в действие `-exec`, это не текст на потоке ввода. Вместо этого конструируется команда. Ей передаются имена найденных объектов как параметры. Можно ли добиться такого же поведения с помощью конвейера? Да. Для этого используйте утилиту xargs.

Применим утилиту xargs в нашем примере с find и grep. Будем фильтровать не имена файлов, а их содержимое. Для этого передадим утилите grep не текст на поток ввода, а имена файлов через параметры командной строки. Добавим вызов xargs так:

```
find ~ -type f | xargs grep "bash"
```

Эта команда не обработает файлы, имена которых содержат пробелы и символы перевода строки. Решение этой проблемы приводится в следующем разделе.

Вывод этой команды выглядит так:

```
/home/ilya.shpigor/.bashrc:# ~/.bashrc: executed by bash(1) for interactive shells.
/home/ilya.shpigor/.bashrc:# The copy in your home directory (~/.bashrc) is yours, p\
lease
/home/ilya.shpigor/.bashrc:# User dependent .bashrc file
/home/ilya.shpigor/.bashrc:# See man bash for more options...
/home/ilya.shpigor/.bashrc:# Make bash append rather than overwrite the history on d\
isk
/home/ilya.shpigor/.bashrc:# When changing directory small typos can be ignored by b\
ash
...
```

Утилита xargs конструирует команду из переданных в неё параметров и данных на потоке ввода. В этой команде сначала идут параметры xargs, а потом текст из потока ввода. Чтобы стало понятнее, обратимся к нашему примеру. Предположим, что первым файлом утилита find нашла `~/.bashrc`. Тогда вызов xargs после конвейера выглядит так:

```
xargs grep "bash"
```

В этот вызов передаётся два параметра: `grep` и “bash”. По этим параметрам xargs конструирует следующую команду:

```
grep "bash"
```

Дальше xargs добавляет к команде текст из потока ввода. В потоке ввода находится имя файла `~/.bashrc`. Поэтому у xargs получается такая команда:

```
grep "bash" ~/.bashrc
```

Утилита xargs выполняет сконструированную команду самостоятельно без вызова Bash. Поэтому на неё накладываются те же ограничения, что и на действие `-exec` утилиты find.

Данные из потока ввода утилита xargs добавляет в конец конструируемой команды. Место добавления этих данных можно указать. Для этого используйте параметр `-I`.

Рассмотрим пример. Предположим, что найденные файлы надо скопировать в домашний каталог пользователя. Это сделает следующая команда:

```
find /usr/share/doc/bash -type f -name "*.html" | xargs -I % cp % ~
```

Параметр `-I %` указывает утилите `xargs`, что текст из потока ввода подставляется вместо символа `%` в конструируемой команде. В нашем примере утилита cp вызывается для каждой строки, которую конвейер передаёт в xargs. Поэтому xargs сконструирует следующие две команды:

```
cp /usr/share/doc/bash/bash.html /home/ilya.shpigor
cp /usr/share/doc/bash/bashref.html /home/ilya.shpigor
```

Опция `-t` утилиты xargs выводит сконструированные команды на экран перед их исполнением. Использовать эту опцию можно так:

```
find /usr/share/doc/bash -type f -name "*.html" | xargs -t -I % cp % ~
```

Мы рассмотрели примеры комбинации утилиты find и конвейеров. Это учебные примеры для образовательных целей. Не делайте так в ваших Bash-скриптах! Вместо конвейеров обрабатывайте найденные файлы действием `-exec`. Так вы избежите ошибок обработки файлов с пробелами и переводом строки в именах.

Один из немногих случаев, когда оправдана комбинация утилиты find и конвейера — это параллельная обработка найденных файлов.

Рассмотрим пример. Предположим, что вы вызываете утилиту cp через действие `-exec`. Тогда файлы копируются по очереди друг за другом. Это неэффективно, если у вашего жёсткого диска высокая скорость доступа. Копирование файлов можно ускорить, если запустить одновременно несколько процессов. Параметр `-P` утилиты xargs указывает максимальное количество параллельно работающих процессов. Они будут выполнять сконструированные команды одновременно.

Предположим, что у процессора вашего компьютера четыре ядра. Тогда копирование можно выполнять в четыре параллельных потока. Команда для этого выглядит так:

```
find /usr/share/doc/bash -type f -name "*.html" | xargs -P 4 -I % cp % ~
```

В результате файлы копируются по четыре штуки за раз. Как только один из параллельных процессов завершается, начнётся копирование следующего файла. Это ускорит выполнение команды в несколько раз.

Для обработки данных на потоке ввода есть ряд утилит. Обычно их используют вместе с конвейерами для поиска и анализа текста. Таблица 2-14 приводит часто используемые из этих утилит.

Таблица 2-14. GNU-утилиты для обработки стандартного потока ввода

Утилита	Описание	Примеры
`xargs`	Конструирует команду по параметрам командной строки и тексту из стандартного потока ввода.	`find . -type f -print0 | xargs -0 cp -t ~`
`grep`	Ищет текст по указанному шаблону.	`grep -A 3 -B 3 "GNU" file.txt` `du /usr/share -a | grep "\.html"`
`tee`	Перенаправляет поток ввода одновременно в поток вывода и в файл.	`grep "GNU" file.txt | tee result.txt`
`sort`	Сортирует строки из потока ввода в прямом и обратном (параметр `-r`) порядке.	`sort file.txt` `du /usr/share | sort -n -r`
`wc`	Считает строки (параметр `-l`), слова (`-w`), буквы (`-m`) и байты (`-c`) в указанном файле или потоке ввода.	`wc -l file.txt` `info find | wc -m`
`head`	Выводит указанное число байтов (параметр `-c`) или строк (`-n`) файла или текста из начала потока ввода.	`head -n 10 file.txt` `du /usr/share | sort -n -r | head -10`
`tail`	Выводит указанное число байтов (параметр `-c`) или строк (`-n`) файла или текста из конца потока ввода.	`tail -n 10 file.txt` `du /usr/share | sort -n -r | tail -10`
`less`	Утилита для навигации по тексту из стандартного потока ввода. Для выхода из неё нажмите клавишу Q.	`less /usr/share/doc/bash/README` `du | less`

Проблемы передачи имён файлов через конвейер

Конвейеры часто используют неправильно, когда обрабатывают результаты утилит ls и find. Это может привести к ошибкам. Рассмотрим их на примерах.

Ожидается, что следующие две команды для поиска HTML файлов вернут одинаковый результат:

```
find /usr/share/doc/bash -name "*.html"
ls /usr/share/doc/bash | grep "\.html"
```

В некоторых случаях их результаты будут отличаться. Дело не в том, что утилиты find и grep по-разному обрабатывают шаблон поиска. Проблема в передаче имён файлов через конвейер.

Стандарт POSIX разрешает любые символы в именах файлов, в том числе пробелы и перевод строки. Единственный запрещённый символ — это **нуль-терминатор**[218] (NULL). Рассмотрим последствия этого правила.

Создайте в домашнем каталоге пользователя файл с символом перевода строки в имени. Этот символ обозначается как `\n`. Чтобы создать такой файл, вызовите утилиту touch:

```
touch ~/$'test\nfile.txt'
```

Утилита touch обновляет время последнего изменения указанного файла. После вызова touch это время будет равно текущему. Если файла не существует, утилита создаст его пустым.

Создайте ещё два файла: `test1.txt` и `file1.txt`. Для этого выполните следующую команду:

```
touch ~/test1.txt ~/file1.txt
```

Теперь вызовем утилиту ls в домашнем каталоге пользователя. Её вывод обработаем с помощью grep. Например, так:

```
1 ls ~ | grep test
2 ls ~ | grep file
```

Иллюстрация 2-27 приводит их вывод.

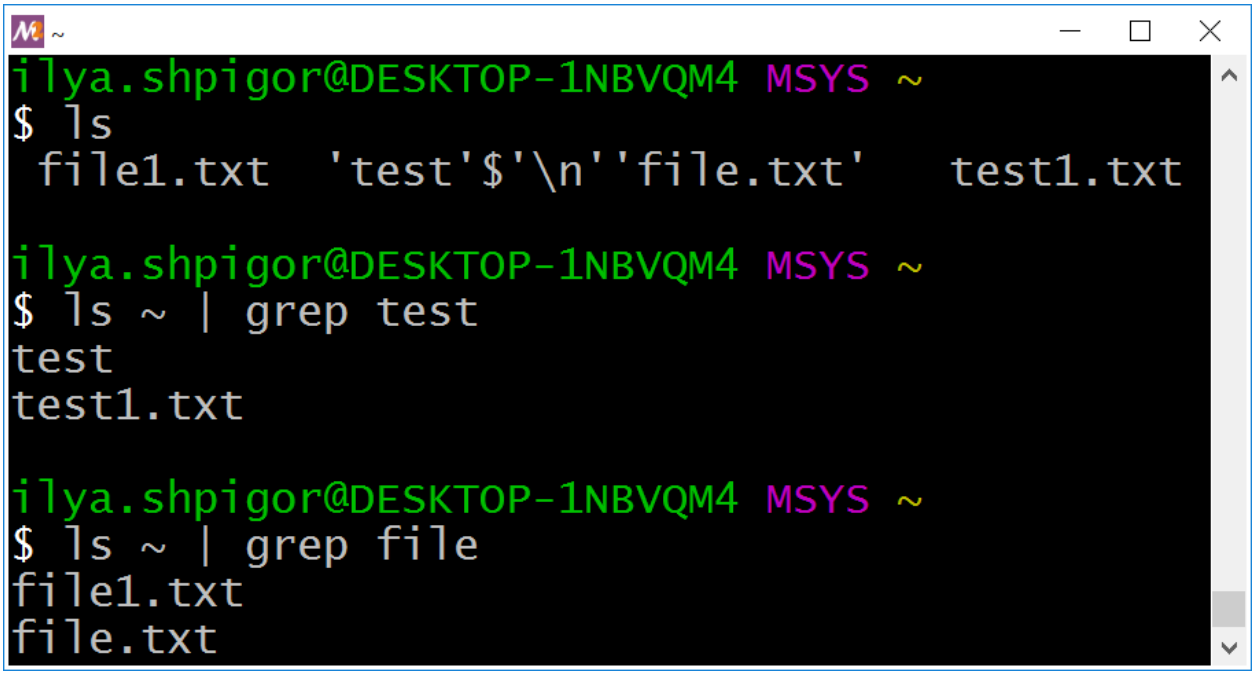


Иллюстрация 2-27. Результат обработки вывода ls с помощью grep

[218] https://ru.wikipedia.org/wiki/Нуль-терминированная_строка

Файл `test\nfile.txt` попал в вывод обеих команд, но его имя оказалось обрезано. Утилита ls передаёт его поток вывода в полной форме: ‘test’$’\n’‘file.txt’. Однако, конвейер заменяет символ `\n` на перевод строки. Поэтому имя файла делится на две части. Дальше grep обрабатывает обе части по отдельности так, как будто это два разных имени.

Это не единственная проблема. Предположим, что вы копируете файл с пробелом в имени (например, `test file.txt`). Из-за пробела следующая команда завершится с ошибкой:

```
ls ~ | xargs cp -t ~/tmp
```

В этом случае xargs сконструирует такой вызов утилиты cp:

```
cp -t ~/tmp test file.txt
```

Эта команда копирует файлы `test` и `file.txt` в каталог `~/tmp`. Но в нашем случае ни одного из этих файлов не существует. Причиной ошибки стал механизм word splitting. Он разделил имя файла с пробелом на два отдельных имени. Ошибка решается двойными кавычками. Например, так:

```
ls ~ | xargs -I % cp -t ~/tmp "%"
```

Но если в имени файла стоит не пробел, а перевод строки, добавление кавычек не поможет. Как тогда решить проблему? Ответ — не используйте ls. Утилита find с действием `-exec` справится с вашей задачей лучше. Вот пример копирования файлов:

```
find . -name "*.txt" -exec cp -t tmp {} \;
```

Иногда без конвейера и xargs не обойтись. В этом случае используйте опцию `-print0` утилиты find. Например, так:

```
find . -type f -print0 | xargs -0 -I % bsdtar -cf %.tar %
```

Опция `-print0` меняет формат вывода find. По умолчанию утилита использует перевод строки в качестве разделителя между путями до найденных объектов. С `-print0` этот разделитель заменяется на нуль-терминатор.

Если формат вывода find изменился, надо также поменять формат ввода xargs. По умолчанию утилита xargs ожидает на вход строки с разделителем `\n`. Если передать ей опцию `-0`, она станет ожидать разделитель нуль-терминатор. Таким образом мы согласовали формат вывода одной утилиты и ввода другой.

Формат вывода утилиты grep тоже настраивается для передачи через конвейер. Используйте опцию `-Z`. Она разделяет файлы в выводе grep нуль-терминатором. Эта опция аналогична `-print0` у find. Вот пример её использования:

```
grep -RlZ "GNU" . | xargs -0 -I % bsdtar -cf %.tar %
```

Эта команда ищет файлы в которых встречается шаблон “GNU”. Конвейер передаёт их имена утилите xargs. Она конструирует вызов утилиты bsdtar для архивации найденных файлов.

Из рассмотренных примеров следует следующее:

1. При передаче имён файлов через конвейер помните о пробелах и переводах строк.
2. Никогда не обрабатывайте вывод утилиты ls. Вместо неё используйте find с действием `-exec`.
3. Если без xargs не обойтись, всегда используйте опцию `-0`. На вход утилите передавайте только имена файлов, разделённые нуль-терминатором.

Упражнение 2-7. Конвейеры и перенаправление потоков ввода-вывода

Напишите команду для архивирования фотографий с помощью утилиты bsdtar.
Если вы пользователь Linux или macOS, используйте утилиту tar.
Фотографии хранятся в структуре каталогов из упражнения 2-6:

```
~/
  photo/
        2019/
             11/
             12/
        2020/
             01/
```

Фотографии одного месяца попадают в один архив.
После выполнения команды получится следующее:

```
~/
  photo/
        2019/
             11.tar
             12.tar
        2020/
             01.tar
```

Логические операторы

Конвейеры позволяют сочетать несколько команд. В результате получается **линейный алгоритм**[219]. В таком алгоритме действия выполняются последовательно друг за другом без каких-либо условных операторов.

[219] https://ru.wikipedia.org/wiki/Алгоритм#Виды_алгоритмов

Предположим, что нам нужен более сложный алгоритм. В нём результат выполнения первой команды определяет следующий шаг. Если команда выполнилась успешно, требуется одно действие. В противном случае — другое. Про такой алгоритм говорят, что он содержит **ветвление**[220]. Сам алгоритм называется **разветвляющимся**.

Рассмотрим разветвляющийся алгоритм на примере. Напишем команду для копирования каталога. При успешном выполнении она записывает строку “OK” в лог-файл. В противном случае пишется строка “Error”.

Применим конвейер и составим такую команду:

```
cp -R ~/docs ~/docs-backup | echo "OK" > result.log
```

К сожалению, команда не сработает. Она запишет строку “OK” в файл `result.log` независимо от результата копирования. Даже если каталога `docs` не существует, в лог-файле будет сообщение об успешном копировании.

Чтобы вывод echo зависел от результата утилиты cp, применим оператор *&&*. Получим такую команду:

```
cp -R ~/docs ~/docs-backup && echo "OK" > result.log
```

Теперь команда печатает строку “OK” в файл, только если утилита cp успешно скопирует каталог.

Что такое оператор *&&*? На самом деле это операция логического И. Но в данном случае она выполняется не над выражениями (условиями), а над командами Bash (действиями). Возникает вопрос: какой смысл выполнять логическую операцию над двумя действиями? Давайте разберёмся.

Стандарт POSIX требует, чтобы каждая запущенная программа при завершении выдавала **код возврата**[221] (exit status). При успешном завершении этот код равен нулю. В противном случае он принимает любое значение от 1 до 255.

Когда логический оператор применяется к команде, он работает с её кодом возврата. Поэтому сначала команда исполняется, а потом её код возврата используется в логическом выражении.

Вернёмся к нашему примеру:

```
cp -R ~/docs ~/docs-backup && echo "OK" > result.log
```

Предположим, что утилита cp завершилась успешно. В этом случае она вернёт ноль. В Bash ноль соответствует значению “истина”. Поэтому левая часть нашего оператора *&&*

[220] https://ru.wikipedia.org/wiki/Ветвление_(программирование)
[221] https://ru.wikipedia.org/wiki/Код_возврата

будет истинной. Этой информации ещё не достаточно, чтобы вычислить значение всего выражения. Оно может быть истинным или ложным в зависимости от правого операнда. Чтобы узнать его значение, оператор && должен выполнить команду echo. Она всегда завершается успешно и возвращает код ноль. Таким образом результатом работы оператора && будет “истина”.

Возникает следующий вопрос: как мы используем результат оператора && в нашем примере? Ответ — никак. Логические операторы нужны для вычисления выражений. Но в Bash их часто применяют ради побочного эффекта, а именно — порядка вычисления операндов.

Рассмотрим случай, когда утилита cp в нашей команде завершилась с ошибкой. Тогда её код возврата не ноль. Для Bash это эквивалентно значению “ложь”. В этом случае оператор && уже может вычислить значение всего выражения. Ему не нужно вычислять правый операнд. Вспомните: если хотя бы один операнд логического И ложный, всё выражение будет ложным. Таким образом код возврата команды echo не важен. Поэтому она не выполняется и записи в файл `result.log` не происходит.

Вычисление только тех операндов, которые достаточны для вывода значения всего выражения, называется **коротким замыканием**[222] (short-circuit).

Код возврата последней выполненной команды всегда сохраняется в переменной окружения Bash с именем `?`. Для вывода её значения используйте команду echo. Например:

```
echo $?
```

В условии задачи говорится, что при ошибке в лог-файл выводится строка “Error”. Дополним нашу команду оператором логического ИЛИ. В Bash он обозначается как ||. Получим следующее:

```
cp -R ~/docs ~/docs-backup && echo "OK" > result.log || echo "Error" > result.log
```

Мы получили разветвляющийся алгоритм. Если утилита cp выполнится успешно, в лог-файл запишется “OK”. В противном случае запишется “Error”. Почему это так? Чтобы ответить на этот вопрос, давайте разберёмся в приоритете операций.

Для простоты обозначим все коды возврата команды буквами латинского алфавита. Утилита cp возвращает код A. Первая команда echo вернёт B, а вторая — C. Тогда команду можно записать в виде следующего выражения:

```
A && B || C
```

В Bash приоритет операторов && и || одинаковый. Выражение вычисляется слева направо. В таком случае говорят, что операторы **левоассоциативны**[223]. Учитывая это, перепишем наше выражение в следующем виде:

[222] https://en.wikipedia.org/wiki/Short-circuit_evaluation
[223] https://ru.wikipedia.org/wiki/Очерёдность_операций

```
(A && B) || C
```

Добавление скобок ничего не меняет. По-прежнему сначала вычисляется выражение (A && B). Затем при необходимости вычисляется операнд C.

Итак, что произойдёт, если A равно истине? В этом случае оператор && вычислит свой правый операнд B. Тогда в лог-файл будет записана строка “OK”. Далее Bash обработает оператор ||. В этот момент уже известно значение его левой части (A && B). Оно равно истине. Поэтому на значение всего выражения правый операнд не повлияет. Его вычислять не нужно. Для логического ИЛИ всё выражение истинно, если хотя бы один из операндов истинен. Поэтому строка “Error” не запишется в лог-файл.

Если значение A ложь, выражение (A && B) также будет ложным. При этом вычислять операнд B не нужно. Поэтому вывода “OK” в лог-файле не будет. Bash перейдёт к следующему оператору ||. Интерпретатору уже известно, что левый операнд ложный. Поэтому для вычисления всего выражения надо узнать значение его правой части. В результате выполнится вторая команда echo и строка “Error” запишется в лог-файл.

Принцип работы логических операторов и короткого замыкания неочевиден. Возможно, вам понадобится время, чтобы в нём разобраться. Это важно. В каждом современном языке программирования встречаются логические выражения. Поэтому понимание правил их вычисления пригодится вам в дальнейшем.

Команды в языке Bash можно сочетать не только конвейерами и логическими операторами. Они объединяются и точкой с запятой. Команды следующие друг за другом через ; выполняются по порядку без каких-либо условий.

Рассмотрим пример. Предположим, что вы копируете два каталога по разны целевым путям. Одним вызовом утилиты cp этого не сделать. Но можно объединить два вызова в одну команду. Например, так:

```
cp -R ~/docs ~/docs-backup ; cp -R ~/photo ~/photo-backup
```

В результате утилита cp вызывается дважды независимо от результата копирования каталога `docs`. Даже если произойдёт ошибка, каталог `photo` всё равно скопируется.

Отличается ли поведение команд следующих через ; от конвейера? Ошибка выполнения первой части команды не повлияет на работу второй. То есть в обоих случаях получается линейный алгоритм. В примере с копированием каталогов разницы никакой нет. Утилита cp игнорирует входные данные на потоке ввода. Следующая команда с конвейером выполняет тот же самый алгоритм копирования:

```
cp -R ~/docs ~/docs-backup | cp -R ~/photo ~/photo-backup
```

Однако, в общем случае различие есть. Команды, следующие через ; выполняются независимо друг от друга. Если используется конвейер, зависимость возникает. Стандартный поток вывода первой команды передаётся на вход второй. Это может изменить её поведение.

Сравните следующие две команды:

```
ls /usr/share/doc/bash | grep "README" * -
ls /usr/share/doc/bash ; grep "README" * -
```

Параметр - утилиты grep добавляет данные со стандартного потока ввода в конец команды.

Иллюстрация 2-28 демонстрирует результаты обеих команд.

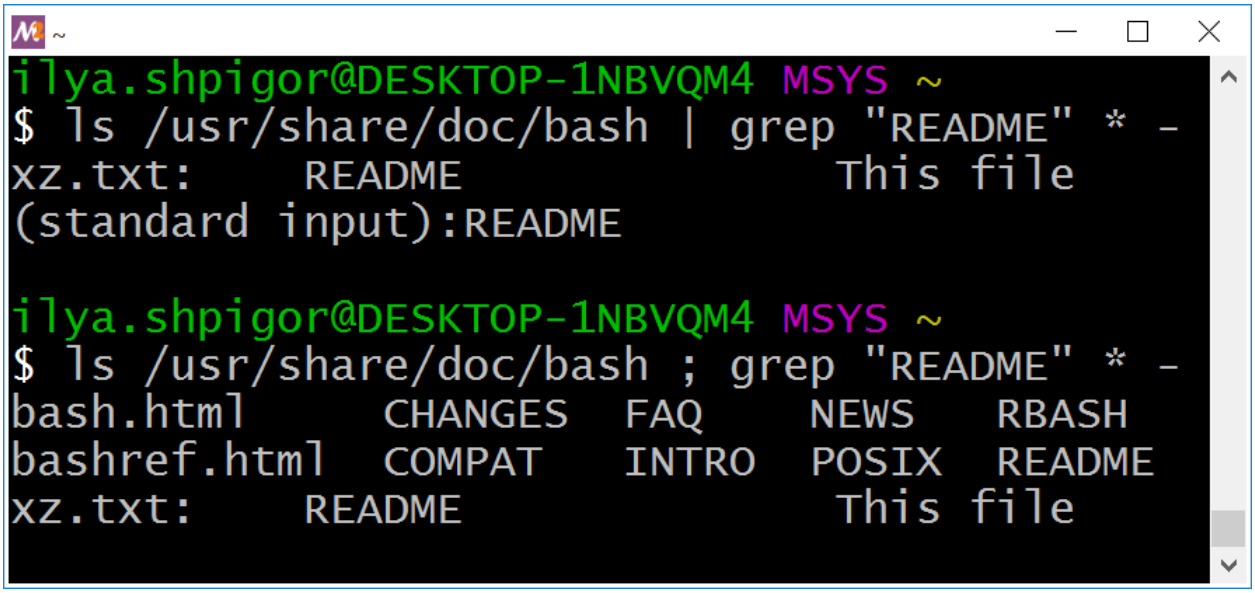


Иллюстрация 2-28. Результаты выполнения команд

Отличается даже поведение утилиты ls. При использовании конвейера её результат не выводится на экран, а перенаправляется на вход grep.

Разберёмся с выводом команд. Второй параметр grep — это шаблон “*”. Поэтому сначала утилита обработает все файлы в текущем каталоге. Слово “README” встречается в файле `xz.txt`. Поэтому на экран выводится следующая строка:

```
xz.txt:    README             This file
```

Далее grep обрабатывает вывод ls, полученный из потока ввода. В этих данных тоже встречается слово “README”. Об этом сообщает следующий вывод:

```
(standard input):README
```

Таким образом утилита grep обработала и файлы текущего каталога, и вывод ls.

В варианте команды с ; утилита ls выводит свой результат на экран. После этого вызывается утилита grep. Она обработает все файлы текущего каталога и поток ввода. Но теперь данных на потоке ввода нет. Поэтому grep найдёт слово “README” только в файле `xz.txt`.

Упражнение 2-8. Логические операторы

Напишите команду, работающую по такому алгоритму:

1. Скопировать файл README с документацией по Bash в домашний каталог пользователя.

2. Архивировать скопированный файл ~/README.

3. Удалить скопированный файл ~/README.

Каждый шаг выполняется только, если предыдущий завершился успешно.
Результат каждого шага записывается в лог-файл result.txt.

Разработка Bash-скриптов

Мы изучили основные приёмы для работы с файловой системой на Bash. Пришло время перейти от составления отдельных команд к программам. Программы, написанные на языке Bash, называют сценариями оболочки (shell scripts) или **скриптами**.

Инструменты для разработки

В прошлой главе мы вводили Bash-команды через эмулятор терминала. При этом набранный текст сохранялся в памяти эмулятора. Физически она находится в оперативной памяти компьютера. RAM — это место для временного хранения информации. После выключения компьютера, она очищается.

Из-за временного характера RAM-памяти терминала будет недостаточно для разработки программ. Вам понадобится удобный редактор[224]. С ним вы сможете создавать файлы с исходным кодом и сохранять их на жёстком диске. Жёсткий диск в отличие от RAM предназначен для долгосрочного хранения информации.

Редактор исходного кода

Писать Bash-скрипты можно в любом текстовом редакторе. Подойдёт даже стандартное приложение Windows — Блокнот[225] (Notepad). Но работать с ним будет неудобно. У Блокнота нет дополнительных возможностей для редактирования исходного кода. Эти возможности увеличат вашу продуктивность. Попробуйте несколько специальных редакторов и выберите из них понравившийся.

Мы рассмотрим три популярных редактора исходного кода. На самом деле их намного больше. Если ни один из трёх вам не подошёл, поищите альтернативы в интернете.

Обратите внимание, что офисный пакет MS Office и его открытый аналог LibreOffice[226] не подходят для написания кода. Они сохраняют файлы в бинарных или XML-подобных[227] форматах. Интерпретаторы и компиляторы не смогут их прочитать. Им нужны текстовые файлы. Именно в таком формате сохраняют файлы редакторы исходного кода.

[224] https://ru.wikipedia.org/wiki/Редактор_исходного_кода
[225] https://ru.wikipedia.org/wiki/Блокнот_(программа)
[226] https://ru.wikipedia.org/wiki/LibreOffice
[227] https://ru.wikipedia.org/wiki/XML

Notepad++[228] — это быстрый и минималистичный свободный редактор с открытым исходным кодом. Он запускается только на ОС Windows. Поэтому для macOS или Linux, лучше рассмотреть другие варианты. Последнюю версию Notepad++ можно загрузить с официального сайта[229].

Sublime Text[230] — проприетарный **кроссплатформенный**[231] редактор. Кроссплатформенность означает, что программа запускается на разных ОС и **аппаратных платформах**[232]. Sublime Text можно использовать бесплатно без активации и лицензии. Но в этом режиме редактор регулярно выводит диалоговое окно с предложением купить лицензию. Загрузить программу можно с официального сайта[233].

Visual Studio Code[234] — свободный кроссплатформенный редактор от компании Microsoft. Его исходный код открыт. Это означает, что программа доступна бесплатно и без лицензии. Редактор можно загрузить с официального сайта[235].

Для работы с исходным кодом у всех трёх редакторов есть следующие возможности:

- Подсветка синтаксиса[236].
- Автодополнение[237].
- Поддержка и конвертирование широко распространённых кодировок[238].

Редактировать исходный код можно и без этих возможностей. Но они ускоряют работу, облегчают редактирование программы и поиск в ней ошибок. Кроме того они помогут вам быстрее освоится с синтаксисом Bash.

Запуск редактора из Bash

Как и другие программы редактор исходного кода запускается через графический интерфейс ОС. Например, в случае Windows — через меню Пуск или иконку на рабочем столе. Кроме этого редактор можно запустить через интерфейс командной строки. Так удобнее редактировать файлы, имена которых вернула какая-то Bash-команда или утилита.

В прошлой главе мы запускали приложения тремя способами:

1. По абсолютному пути.
2. По относительному пути.

[228] https://ru.wikipedia.org/wiki/Notepad++#cite_note-8
[229] https://notepad-plus-plus.org/downloads/
[230] https://ru.wikipedia.org/wiki/Sublime_Text#cite_note-Features-4
[231] https://ru.wikipedia.org/wiki/Кроссплатформенность
[232] https://ru.wikipedia.org/wiki/Аппаратная_платформа_компьютера
[233] https://www.sublimetext.com/
[234] https://ru.wikipedia.org/wiki/Visual_Studio_Code
[235] https://code.visualstudio.com/
[236] https://ru.wikipedia.org/wiki/Подсветка_синтаксиса
[237] https://ru.wikipedia.org/wiki/Автодополнение
[238] https://ru.wikipedia.org/wiki/Набор_символов

3. По имени исполняемого файла.

Для последнего способа каталог установки программы должен быть в переменной `PATH`.

В качестве примера запустим редактор Notepad++ из командной строки.

По умолчанию путь установки редактора следующий:

```
C:\Program Files (86)\Notepad++
```

Если вы не знаете путь установки, прочитайте его в свойствах ярлыка Notepad++ на рабочем столе или в меню Пуск.

В Unix-окружении тот же путь установки выглядит так:

```
/c/Program Files (x86)/Notepad++
```

Если запустить Notepad++ по этому абсолютному пути, будет выведено сообщение об ошибке как на иллюстрации 3-1.

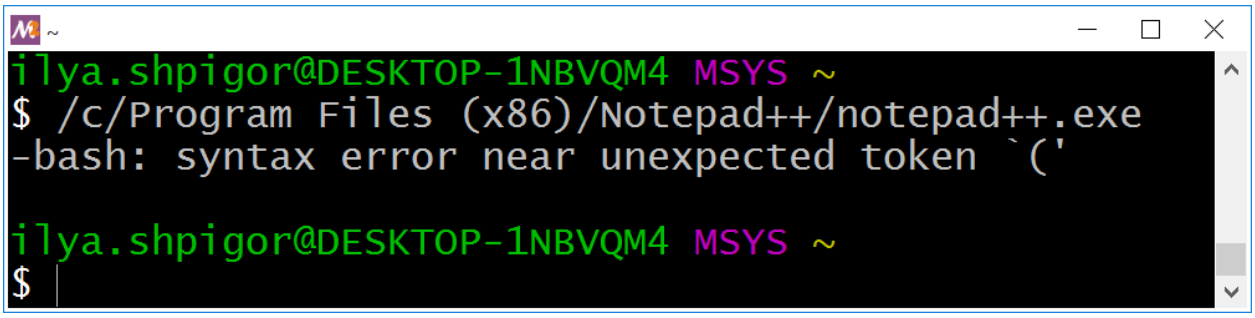


Иллюстрация 3-1. Результат запуска Notepad++

В чём ошибка? На самом деле в этой команде несколько проблем. Рассмотрим их по порядку. Для начала попробуем выполнить следующую команду cd:

```
cd /c/Program Files
```

Её результат на иллюстрации 3-2.

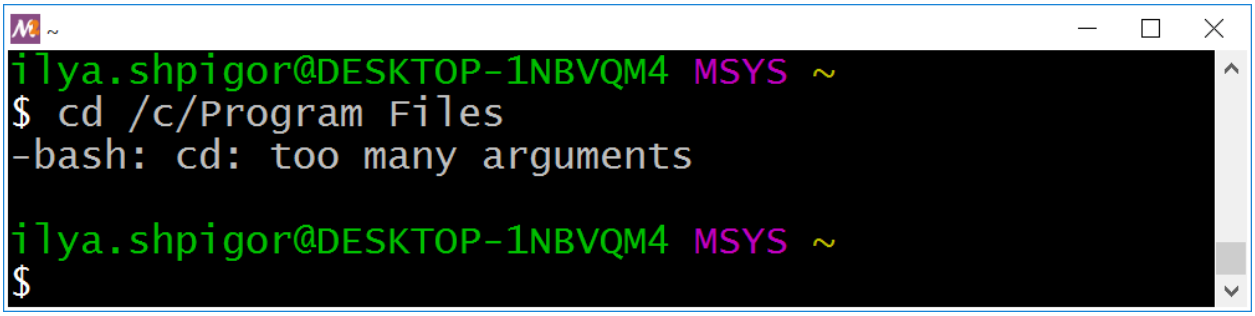


Иллюстрация 3-2. Результат выполнения cd

Bash пишет: в команду было передано больше параметров, чем нужно. Команда cd принимает на вход один путь к целевому каталогу. Ошибка возникла из-за word splitting. Bash интерпретирует пробел не как часть пути, а как разделитель между двумя параметрами. Таким образом в команду cd передаются два пути: `/c/Program` и `Files`.

Подобные ошибки решаются двумя способами:

1. Заключить путь в двойные кавычки:

```
cd "/c/Program Files"
```

2. Экранировать все пробелы с помощью обратного слэша:

```
cd /c/Program\ Files
```

Каждая из этих команд выполнится корректно.

Теперь попробуем перейти по пути `/c/Program Files (x86)`:

```
cd /c/Program Files (x86)
```

Как мы выяснили, Bash обрабатывает пробелы по своему усмотрению. Поэтом экранируем их с помощью обратного слэша. Получим следующую команду:

```
cd /c/Program\ Files\ (x86)
```

Эта команда всё равно завершится с ошибкой, как на иллюстрации 3-3.

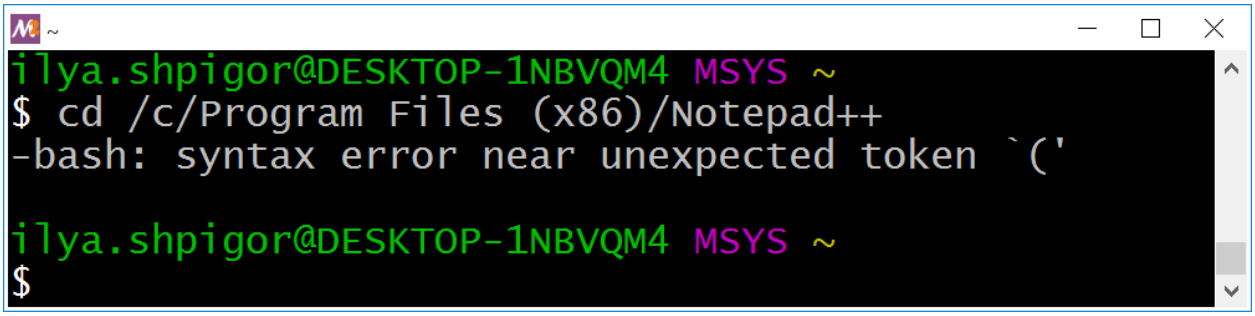


Иллюстрация 3-3. Результат выполнения cd

При запуске Notepad++ выводилось такое же сообщение об ошибке (см. иллюстрацию 3-1). Проблема в том, что кавычки (и) являются частью синтаксиса Bash. Поэтому интерпретатор пытается обработать их как конструкцию языка. Мы уже сталкивались с этой проблемой, когда группировали выражения утилиты find. Экранирование или двойные кавычки решат проблему. Например:

```
cd /c/Program\ Files\ \(x86\)
cd "/c/Program Files (x86)"
```

Применив кавычки, мы получим такую команду запуска Notepad++ по абсолютному пути:

```
"/c/Program Files (x86)/Notepad++/notepad++.exe"
```

Каждый раз набирать полный путь установки редактора неудобно. Намного лучше было бы запускать его по имени исполняемого файла. Чтобы этот способ заработал, добавьте путь установки Notepad++ в переменную `PATH`. Для этого в конец файла `~/.bash_profile` добавьте следующую строку:

```
PATH="/c/Program Files (x86)/Notepad++:${PATH}"
```

Закройте и снова откройте терминал MSYS2. Теперь Notepad++ запускается следующей командой:

```
notepad++.exe
```

Вместо редактирования переменной `PATH` можно использовать псевдонимом (alias). Этот механизм заменяет введённую команду на другую. Таким образом в окне терминала можно сократить набор длинных строк.

Объявим псевдоним `notepad++` для следующей команды:

```
"/c/Program Files (x86)/Notepad++/notepad++.exe"
```

Для этого выполните такую команду alias:

```
alias notepad++="/c/Program\ Files\ \(x86\)/Notepad++/notepad++.exe"
```

Теперь встретив команду `notepad++`, Bash будет запускать редактор по указанному нами абсолютному пути. У этого решения есть одна проблема: псевдоним придётся объявлять каждый раз после перезапуска терминала MSYS2. Чтобы это происходило автоматически, добавьте команду alias в файл `~/.bashrc`. Bash исполняет этот файл перед каждым запуском терминала.

После интеграции Notepad++ в Bash, файлы в редакторе открываются из командной строки. Для этого передайте имя файла первым параметром программе notepad++. Например:

```
notepad++ test.txt
```

Если файла `test.txt` не существует, откроется диалог с предложением его создать.

Допустим, вы запустили графическое приложение в окне терминала. После этого в терминал нельзя вводить команды. Его контролирует запущенное приложение. Так оно выводит на экран свои диагностические сообщения. Только после завершения программы, терминал заработает в обычном режиме.

Чтобы продолжить работу с терминалом, запускайте графические приложения в **фоновом режиме** (background). Для этого добавьте амперсанд `&` в конце команды. Например:

```
notepad++ test.txt &
```

В результате приложение по-прежнему будет выводить сообщения в окно терминала, но при этом вы сможете вводить Bash-команды. Чтобы полностью разделить окно терминал и запущенное приложение, выполните команду disown с опцией `-a`:

```
disown -a
```

В результате вывод сообщений в окно терминала прекратится. После закрытия этого окна, приложение продолжит свою работу.

Команды запуска Notepad++ и disown можно объединить в одну. Например, так:

```
notepad++ test.txt & disown -a
```

Параметр `-a` команды disown отделяет все приложения запущенные в фоновом режиме.

Переменная окружения Bash `$!` хранит идентификатор PID[239] последнего запущенного приложения. С её помощью в команду disown можно передать только PID редактора. В этом случае команда запуска Notepad++ будет выглядеть так:

[239] https://ru.wikipedia.org/wiki/Идентификатор_процесса

```
notepad++ test.txt & disown $!
```

Чтобы вывести список всех приложений, запущенных в фоновом режиме, наберите команду jobs с опцией `-l`:

```
jobs -l
```

Так вы узнаете их идентификаторы PID.

Зачем нужны скрипты?

В прошлой главе мы научились писать сложные Bash команды с использованием конвейеров и логических операторов. Конвейеры объединяют несколько команд в одну. Так получается линейный алгоритм. Логические операторы добавляют в него ветвление. В результате получается настоящая программа.

Почему для программирования на Bash средств командного интерпретатора оказывается недостаточно? Bash-скрипты — это программы, хранящиеся на жёстком диске. Разберёмся, зачем они нужны.

Команда резервного копирования

Для примера напишем команду резервного копирования фотографий на внешний жёсткий диск[240]. Команда будет состоять из двух действий: архивирования и копирования. Предположим, что фотографии хранятся в каталоге `~/photo`, а `/d` — это точка монтирования внешнего диска. Тогда команда может быть такой:

```
bsdtar -cjf ~/photo.tar.bz2 ~/photo && cp -f ~/photo.tar.bz2 /d
```

Благодаря логическому И (&&), копирование выполняется только после успешного архивирования. Если утилита bsdtar вернула ошибку, копирования не будет.

В нашем примере резервное копирование выполняется в два этапа: архивирование и копирование. Такое разделение действий нужно для демонстрации. То же самое поведение даст один вызов утилиты bsdtar. Укажите в нём путь до создаваемого архива на диске D. Например, так:

[240] https://ru.wikipedia.org/wiki/Внешний_жёсткий_диск

```
bsdtar -cjf /d/photo.tar.bz2 ~/photo
```

Предположим, что наша команда резервного копирования будет запускаться автоматически (например, по расписанию). Тогда вы не сможете прочитать сообщение об ошибке, если что-то пойдёт не так. В таких случаях поможет вывод в лог-файл. Добавим этот вывод для вызова утилиты bsdtar. Получим:

```
bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
echo "bsdtar - OK" > results.txt ||
echo "bsdtar - FAILS" > results.txt
```

Bash-команду можно разбить на несколько строк. Есть два способа переноса строк:

1. Перенос строки сразу после логического оператора (&& или ||).
2. Перенос строки после обратного слеша .

Второй вариант выглядит так:

```
bsdtar -cjf ~/photo.tar.bz2 ~/photo \
&& echo "bsdtar - OK" > results.txt \
|| echo "bsdtar - FAILS" > results.txt
```

Теперь выведем в лог-файл результат утилиты cp. Получим:

```
cp -f ~/photo.tar.bz2 /d &&
echo "cp - OK" >> results.txt ||
echo "cp - FAILS" >> results.txt
```

Резервное копирование должно выполняться одной командой. Поэтому попробуем объединить вызовы bsdtar и cp логическим И. Получится следующее:

```
bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
  echo "bsdtar - OK" > results.txt ||
  echo "bsdtar - FAILS" > results.txt &&
cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  echo "cp - FAILS" >> results.txt
```

Что будет делать эта команда? Для удобства перепишем её в виде логического выражения. Заменим каждый вызов команды или утилиты на букву латинского алфавита. Получится следующее:

```
B && O1 || F1 && C && O2 || F2
```

Буквы B и C обозначают вызовы утилит bsdtar и cp. O1 и F1 — это вывод в лог-файл строк “bsdtar - OK” и “bsdtar - FAIL”. Аналогично, O2 и F2 — это вывод результата cp.

Если B истинно, порядок исполнения команд очевиден. Последовательность действий будет такой:

1. B
2. O1
3. C
4. O2 или F2

Если же bsdtar вернёт ошибку, значение B будет ложь. Тогда выполнятся такие действия:

1. B
2. F1
3. C
4. O2 или F2

Операция копирования не имеет смысла, если архивирование завершилось с ошибкой. Лишние проблемы создаёт поведение утилиты bsdtar. Если указанного каталога или файла не существует, утилита создаст пустой архив. В этом случае cp успешно его скопирует. После этого в лог-файл запишется строка “cp - OK”. Тогда лог-файл будет таким:

```
1 bsdtar - FAILS
2 cp - OK
```

Такой вывод только запутает пользователя.

Вернёмся к нашему выражению:

```
B && O1 || F1 && C && O2 || F2
```

Почему утилита cp вызывается после ошибки в bsdtar? Дело в том, что команда echo всегда выполняется успешно. Её код возврата всегда истинен. Это значит, что значения O1, F1, O2 и F2 — истина.

Рассмотрим только команду вызова bsdtar и вывод её результата в лог-файл. Ей соответствует следующая часть логического выражения:

```
B && O1 || F1
```

Заключим левую часть выражения в скобки:

```
(B && O1) || F1
```

Теперь мы получили логическое ИЛИ для операндов (B && O1) и F1. F1 — всегда истина. Поэтому и всё выражение всегда истинно.

Проблему можно решить, если инвертировать результат вызова F1 с помощью логического НЕ. Оно обозначается как восклицательный знак !. Получим такое выражение:

```
B && O1 || ! F1 && C && O2 || F2
```

Теперь в случае ошибки утилиты bsdtar в лог-файл будет выведено “bsdtar - FAIL”. Но оставшаяся часть выражения всё равно будет обработана. Операции C и O2 не будут выполнены. Они связаны логическим И с результатом F1, который всегда ложен. Но после них идёт действие F2. Оно будет выполнено.

Для удобства добавим к нашему выражению скобки. Получим:

```
(B && O1 || ! F1 && C && O2) || F2
```

Теперь очевидно, что если выражение в скобках ложно, Bash выполнит действие F2. Иначе ему не вывести значение всего выражения.

В результате выполнения всей команды в лог-файл будет выведено:

```
1 bsdtar - FAILS
2 cp - FAILS
```

Такой вывод лучше предыдущего. Теперь утилита cp не вызывается и пустой архив не копируется. Но представьте, что в нашей команде резервного копирования 100 действий. Если ошибка произойдёт на 50-ом действии, результаты всех оставшихся всё равно попадут в лог-файл. Этот вывод только помешает найти проблему. Лучшим решением было бы прекратить выполнение команды после первой же ошибки. Для этого сгруппируем вызовы утилит и выводы их результатов в лог-файл. Получим:

```
(B && O1 || ! F1) && (C && O2 || F2)
```

Проверим, что теперь произойдёт если B ложно. В этом случае выполнится действие F1. Его результат инвертируется. Поэтому вся левая часть выражения будет ложной:

```
(B && O1 || ! F1)
```

Дальше из-за короткого замыкания правый операнд логического И не вычисляется. Это значит, что все действия в правой части выражения не выполнятся:

```
(C && O2 || F2)
```

Мы получили нужное нам поведение.

Добавим последний штрих. Результат действия F2 нужно инвертировать. Тогда всё выражение будет ложным, если C ложно. Это значит, что команда резервного копирования завершилась ошибкой, если утилита cp не смогла отработать. Звучит логично. Кроме того, это полезно при интеграции нашей команды с другими командами.

Конечный вариант нашего выражения будет таким:

```
(B && O1 || ! F1) && (C && O2 || ! F2)
```

Теперь вернёмся к реальному коду на Bash. Наша команда резервного копирования стала такой:

```
(bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
  echo "bsdtar - OK" > results.txt ||
  ! echo "bsdtar - FAILS" > results.txt) &&
(cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt)
```

Как это часто бывает в программировании, такую команду несложно написать, но трудно прочитать и понять.

Плохое техническое решение

Мы написали длинную и сложную Bash-команду резервного копирования. Если она выполняется регулярно, её надо где-то сохранить. Иначе каждый раз придётся набирать команду вручную в окне терминала.

Все выполненные в терминале команды автоматически сохраняются в файле истории. У каждого пользователя он свой по пути `~/.bash_history`. По комбинации клавиш Ctrl+R в этом файле можно быстро найти нужную команду.

Что если мы просто сохраним команду резервного копирования в файле истории? Там её можно будет быстро найти и исполнить. Это решение кажется надёжным и удобным. Но не торопитесь с выводами. Давайте рассмотрим возможные проблемы.

Прежде всего размер файла истории ограничен. По умолчанию сохраняются только 500 последних выполненных команд. Если превысить это число, то каждая новая команда будет записана вместо самой старой. Из-за этого команда резервного копирования может быть случайно удалена из истории.

Максимальный размер файла истории можно увеличить. Но сразу возникает вопрос: увеличить на сколько? Какой размер не выберем, есть риск его переполнения. Можно вообще снять ограничение на размер. Тогда будут сохраняться все введённые команды, а старые никогда не будут удаляться.

Кажется, нам удалось найти решение задачи: файл истории с неограниченным размером. Могут ли с ним возникнуть какие-то проблемы? Давайте подумает. Предположим, что вы используйте Bash год или два. Все введённые за это время команды попадут в файл `.bash_history`. Учтите, что одни и те же команды в нём дублируются. Например, каждая команда `cd ~` будет записана в этот файл, даже если она там уже есть. Скорее всего за год размер файла достигнет нескольких сотен мегабайт. При этом большая часть информации в нём не нужна. Нас интересует небольшой набор команд, которые мы используем регулярно. В результате возникает первая проблема: нерациональное использование места на жёстком диске.

Вы можете возразить, что хранить лишние две-три сотни мегабайт — не проблема для современных компьютеров. Да, это так. Но не забывайте, что по нажатию Ctrl+R Bash ищет нужную команду по всему файлу `.bash_history`. Чем он больше, тем дольше длится поиск. Со временем вы станете ждать десятки секунд даже на мощном компьютере. Дальше с ростом файла время ожидания станет только больше.

Разрастание файла истории увеличивает время поиска. Не только потому, что Bash приходится перебирать больше строк в нём. По нажатию Ctrl+R надо ввести начало искомой команды. Представьте, что история огромна. Тогда в ней много команд, которые начинаются одинаково. Это значит, что вам придётся набрать больше символов, чтобы найти из них нужную. Неудобство вызова команды — это вторая проблема нашего решения.

Предположим, у вас появились новые альбомы фотографий. Они хранятся не в каталоге `~/photo`, а например в `~/Documents/official_photo`. Наша команда резервного копирования работает только с путём `~/photo`. Чтобы скопировать фотографии из другого пути, команду надо переписать. Выполните новую команду. Теперь она тоже сохранилась в файле истории `.bash_history`. Это опять увеличит время её поиска. Итак, сложность расширения функций — третья проблема.

Возможно у вас уже есть несколько команд для резервного копирования. Одна копирует фотографии, а другая документы. Объединить их будет проблематично. Вам придётся написать новую команду, в которую войдут действия из уже существующих.

Какой можно сделать вывод? Файл истории не подходит для долговременного хранения команд. Причина всех возникших проблем одна. Мы пытаемся использовать механизм файла истории не по назначению. Он создавался не для этого. В результате мы пришли к плохому техническому решению.

От плохих решений не застрахован никто. Профессионалы с большим опытом тоже нередко к ним приходят. Почему? Причины бывают разные. В нашем случае сыграл роль недостаток знаний. Мы освоились с работой Bash в режиме командного интерпретатора. Эти знания мы применили для новой задачи. Но всех её требований не учли. Как оказалось, просто заархивировать и скопировать файлы недостаточно.

Полный список требований выглядит так:

1. Команда должна храниться неограниченно долго.
2. Команда должна быстро вызываться.
3. Нужна возможность для её расширения.
4. Нужна возможность для сочетания её с другими командами.

Для начала оценим свои знания Bash. Их просто недостаточно, чтобы удовлетворить всем этим требованиям. Все известные нам механизмы не подходят. Может быть нам бы помог Bash-скрипт? Предлагаю изучить его возможности. Затем проверим, подходит ли он для нашей задачи.

Запуск скрипта

Создадим Bash-скрипт с нашей командой для резервного копирования. Для этого сделайте следующее:

1. Откройте редактор исходного кода и создайте в нём новый файл. Если вы интегрировали Notepad++ в Bash, выполните команду:

```
notepad++ ~/photo-backup.sh
```

2. Скопируйте команду резервного копирования в файл:

```
(bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
  echo "bsdtar - OK" > results.txt ||
  ! echo "bsdtar - FAILS" > results.txt) &&
(cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt)
```

3. Сохраните файл с именем `photo-backup.sh` в домашнем каталоге пользователя.
4. Закройте редактор.

Мы получили файл Bash-скрипта. Чтобы его исполнить, запустите интерпретатор и передайте скрипт первым параметром. Запустить интерпретатор можно по имени его исполняемого файла — `bash`:

```
bash photo-backup.sh
```

Мы только что написали и запустили наш первый Bash-скрипт. Он представляет собой последовательность команд, записанных в файл. Команды исполняются в том же порядке, как если бы вы читали их из скрипта и вводили вручную.

Запускать скрипты с явным вызовом интерпретатора Bash неудобно. Есть способ запускать их так же как и любую GNU-утилиту: по относительному или абсолютному пути. Для этого скрипт придётся изменить. Вот порядок действий:

1. В окне терминала выполните команду:

```
chmod +x ~/photo-backup.sh
```

2. Откройте файл скрипта в редакторе и добавьте в начало следующую строку:

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

3. Сохраните изменённый файл и закройте редактор.

Теперь скрипт запускается по относительному или абсолютному пути.

Разберём наши действия. Первое, что мешает запустить скрипт после его создания — это права доступа. По умолчанию все новые файлы получают следующие права:

```
-rw-rw-r--
```

Это значит, что владелец и его группа могут читать и изменять файл. Все остальные могут только читать. Запускать файл не может никто.

Утилита chmod меняет права указанного файла. Мы передали ей опцию `+x`. В результате все пользователи смогут запускать файл. Его битовая маска прав стала такой:

```
-rwxrwxr-x
```

Теперь файл можно запустить по относительному или абсолютному пути. В этом случае ваш командный интерпретатор попробует его исполнить. Если вы используете Bash, скрипт выполнится корректно.

Если ваш командный интерпретатор не Bash (например, C shell[241]), скрипт вероятно завершится с ошибкой. Проблема в том, что скрипт написан на языке одного интерпретатора, а исполняется другим.

В скрипте можно явно указать интерпретатор, который должен его исполнять. Для этого в начале файла напечатайте **шебанг**[242]. Шебангом называются символы решётка и восклицательный знак `#!`. После них идёт абсолютный путь до файла интерпретатора. В нашем случае получится такая строка:

[241] https://ru.wikipedia.org/wiki/Csh
[242] https://ru.wikipedia.org/wiki/Шебанг_(Unix)

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

Теперь Bash будет исполнять скрипт, независимо от выбранного пользователем командного интерпретатора.

Если в скрипте не указан интерпретатор для запуска, утилита file определит его как обычный текстовый файл:

```
~/photo-backup.sh: ASCII text
```

После добавления строки `#!`, тот же файл определится как Bash-скрипт:

```
~/photo-backup.sh: Bourne-Again shell script, ASCII text executable
```

В некоторых системах Unix (например, FreeBSD) путь к Bash отличается от стандартного `/bin/bash`. Если вам важна переносимость скриптов, то вместо абсолютного пути до интерпретатора указывайте следующее:

```
#!/usr/bin/env bash
```

С помощью утилиты env исполняемый файл Bash будет найден по одному из путей переменной `PATH`.

Последовательность команд

Листинг 3-1 демонстрирует наш скрипт.

Листинг 3-1. Скрипт для резервного копирования

```
#!/bin/bash
(bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
  echo "bsdtar - OK" > results.txt ||
  ! echo "bsdtar - FAILS" > results.txt) &&
(cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt)
```

Команда резервного копирования слишком длинная. Из-за этого её трудно читать и изменять. Попробуем разбить её на две отдельные команды. Результат приведён в листинге 3-2.

Листинг 3-2. Разделение команд bsdtar и cp

```
#!/bin/bash

bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
  echo "bsdtar - OK" > results.txt ||
  ! echo "bsdtar - FAILS" > results.txt

cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt
```

Поведение скрипта изменилось. Теперь команды не связаны логическим И. Поэтому утилита cp будет вызываться независимо от результата bsdtar. Такое поведение неправильно.

Исправим скрипт. Он должен завершаться при ошибке утилиты bsdtar. Чтобы завершить скрипт, воспользуемся командой exit. В качестве параметра она принимает код возврата. Скрипт вернёт этот код после своего завершения.

Листинг 3-3 демонстрирует скрипт с вызовом exit.

Листинг 3-3. Добавление команды exit

```
#!/bin/bash

bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
  echo "bsdtar - OK" > results.txt ||
  (echo "bsdtar - FAILS" > results.txt ; exit 1)

cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt
```

Мы внесли два изменения в команду вызова утилиты bsdtar. Сначала она соответствовала такому выражению:

```
B && O1 || ! F1
```

После добавления exit выражение стало выглядеть так:

```
B && O1 || (F1 ; E)
```

Команда exit обозначена как E. Теперь если bsdtar вернёт ошибку, будет вычислен правый операнд логического ИЛИ. Он равен `(F1 ; E)`. Мы удалили отрицание результата команды

echo. Этот результат больше не важен. Не зависимо от него после echo будет вызван exit. Команды, разделённые точкой с запятой, выполняются друг за другом без каких-либо условий.

С нашим решением есть одна проблема. Когда интерпретатор встречает круглые скобки в скрипте или команде, он запускает сам себя в дочернем процессе[243]. Такой дочерний процесс называется **subshell**. Он исполняет указанные в скобках команды. После этого управление передаётся обратно родительскому процессу Bash, породившему subshell. Родительский процесс продолжает исполнение скрипта или команды.

В нашем случае команда exit означает выход из subshell. Выполняющий скрипт родительский процесс Bash продолжит работу. Чтобы решить эту проблему, надо заменить круглые скобки на фигурные. Указанные в них команды будут выполняться в текущем процессе Bash без создания subshell. Исправленная версия скрипта приведена в листинге 3-4.

Листинг 3-4. Вызов exit в том же процессе Bash

```
#!/bin/bash

bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
  echo "bsdtar - OK" > results.txt ||
  { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }

cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt
```

Обратите внимание на обязательную точку с запятой перед закрывающей фигурной скобкой }. Также обязательны пробелы после открывающей скобки.

Есть решение изящнее, чем вызов команды exit. Предположим, что скрипт нужно завершить после первой команды, вернувшей ненулевой код возврата. Для этого используйте встроенную команду set. Она изменяет параметры работы интерпретатора. В нашем случае команду надо вызвать с опцией `-e`:

```
set -e
```

Эту же опцию `-e` можно указать при явном запуске Bash. Например:

```
bash -e
```

У опции `-e` есть несколько проблем[244]. Опция меняет поведение только текущего процесса Bash. Порождённые им subshell работают как обычно.

[243] https://en.wikipedia.org/wiki/Child_process
[244] http://mywiki.wooledge.org/BashFAQ/105

Каждая команда в конвейере или логическом операторе выполняется в отдельном subshell. Поэтому опция `-e` никак не повлияет на поведение этих команд. В нашем случае такое решение не подойдёт.

Параметризация

Предположим, что вы перенесли фотографии из каталога `~/photo` в `~/Documents/Photo`. Тогда в нашем скрипте резервного копирования тоже придётся поменять путь. После изменения мы получим код как на листинге 3-5.

Листинг 3-5. Новый каталог фотографий

```
#!/bin/bash

bsdtar -cjf ~/photo.tar.bz2 ~/Documents/Photo &&
  echo "bsdtar - OK" > results.txt ||
  { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }

cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt
```

Каждый раз при смене каталога фотографий придётся редактировать скрипт. Это неудобно. Лучше сделать скрипт универсальным. Для этого он должен принимать путь к каталогу с фотографиями в качестве параметра.

При запуске любого Bash-скрипта в него можно передать параметры командной строки. Это работает точно так же как и для любой GNU-утилиты. Просто укажите параметры через пробел после имени скрипта. Например:

```
./photo-backup.sh ~/Documents/Photo
```

Запустите наш скрипт этой командой. Интерпретатор Bash передаст в него путь к фотографиям `~/Documents/Photo`. Этот путь будет доступен в коде скрипта через переменную `$1`. Если передать больше параметров, они будут доступны через переменные `$2`, `$3`, `$4` и т. д. в зависимости от их количества. Эти параметры называются позиционными (**positional parameters**).

В переменную `$0` запишется относительный путь к скрипту `./photo-backup.sh`.

Перепишем наш скрипт. Пусть путь до каталога фотографий читается из первого параметра. Получим код как в листинге 3-6.

Листинг 3-6. Чтение пути из первого параметра

```
#!/bin/bash

bsdtar -cjf ~/photo.tar.bz2 "$1" &&
  echo "bsdtar - OK" > results.txt ||
  { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }

cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt
```

Путь до фотографий хранится в переменной `$1`. Мы подставляем её значение в вызов утилиты bsdtar. При этом обращение к переменной заключается в кавычки. Если их не поставить, сработает механизм word splitting. Тогда путь содержащий пробелы будет разделён на несколько параметров.

Предположим, что фотографии хранятся в каталоге `~/photo album`. Тогда команда запуска скрипта будет такой:

```
./photo-backup.sh "~/photo album"
```

Если передать параметр `$1` без кавычек в утилиту bsdtar, её вызов будет таким:

```
bsdtar -cjf ~/photo.tar.bz2 ~/photo album &&
  echo "bsdtar - OK" > results.txt ||
  { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
```

В этом случае утилита bsdtar получит строку “/photo album” по частям. Вместо одного параметра будет два: ”/photo” и “album”. Таких каталогов не существует. Поэтому скрипт завершится с ошибкой.

Заключать пути в кавычки только при вызове скрипта недостаточно. Кавычки надо применять во всех местах подстановки переменной `$1` . При вызове скрипта они обрабатываются и отбрасываются командным интерпретатором Bash. Наш скрипт выполняет не этот процесс. Вместо этого он запускает дочерний процесс Bash, который читает и исполняет скрипт. Дочерний процесс Bash не знает про кавычки в команде вызова скрипта.

Итак, что нам дала параметризация скрипта? Вместо решения для резервного копирования фотографий мы сделали универсальную программу. Она работает с любыми входными файлами: документами, медиафайлами, исходным кодом программ и т.д.

У нашего скрипта есть ещё одна проблема. Она связана с именем архива. Предположим, что наш скрипт вызывается дважды для копирования фотографий и документов:

```
./photo-backup.sh ~/photo
./photo-backup.sh ~/Documents
```

Обе эти команды создают архив с именем `photo.tar.bz2` в домашнем каталоге пользователя. Обе копируют архив на диск D. Очевидно, что результат второй команды полностью перезапишет результат первой. Это не то, что нам нужно.

Попробуем исправить ошибку. Для этого подставим первый параметр скрипта не только в качестве пути к архивируемым данным, но и вместо имени архива. Отредактированный скрипт приведён в листинге 3-7.

Листинг 3-7. Чтение имени архива из первого параметра

```
#!/bin/bash

bsdtar -cjf "$1".tar.bz2 "$1" &&
  echo "bsdtar - OK" > results.txt ||
  { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }

cp -f "$1".tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt
```

Теперь имя архива будет соответствовать каталогу с архивируемыми данными. Допустим, вы вызовите скрипт так:

```
./photo-backup.sh ~/Documents
```

Тогда будет создан архив с именем `Documents.tar.bz2`. Он будет скопирован на диск D. При этом его имя не конфликтует с именем архива фотографий `photo.tar.bz2`.

Исправим последний недочёт скрипта. Заменим копирование архива на переименование. Тогда ненужный промежуточный архив в каталоге пользователя будет удалён. Результат приведён в листинге 3-8.

Листинг 3-8. Удаление временного архива

```
#!/bin/bash

bsdtar -cjf "$1".tar.bz2 "$1" &&
  echo "bsdtar - OK" > results.txt ||
  { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }

mv -f "$1".tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt
```

Теперь у нас есть универсальный скрипт для резервного копирования. Его старое имя `photo-backup.sh` больше не подходит. Ведь скрипт умеет копировать любые данные. Переименуем его на `make-backup.sh`.

Сочетание с утилитами и командами

Наш универсальный скрипт для резервного копирования можно сочетать с GNU-утилитами, Bash-командами и другими скриптами.

Сейчас скрипт вызывается только по абсолютному или относительному пути. Если интегрировать его в Bash, вы сможете вызывать скрипт по имени. Тогда его станет удобнее сочетать с другими программами.

Нам знакомы два способа интеграции приложения с Bash по опыту настройки Notepad++. Кроме них есть ещё третий способ. Вот полный список вариантов:

1. Добавить путь до скрипта в переменную `PATH`. Для этого отредактируйте файл `~/.bash_profile`.
2. Определить псевдоним alias с абсолютным путём до скрипта. Это можно сделать в файле `~/.bashrc`.
3. Скопировать скрипт в каталог `/usr/local/bin`. Путь до него по умолчанию добавляется в переменную `PATH`. Если в вашем окружении MSYS2 этого каталога нет — создайте его.

Чтобы удалить объявленный псевдоним, вызовите Bash-команду `unalias`. Пример её использования для нашего скрипта:

```
unalias make-backup.sh
```

После интеграции с Bash скрипт запускается по имени. Например, так:

```
make-backup.sh ~/photo
```

Скрипт можно использовать в конструкциях с конвейерами и логическими операторами также, как любую встроенную команду Bash или GNU-утилиту.

Рассмотрим пример. Предположим, что нужно создать резервную копию всех PDF документов из каталога `~/Documents`. Эти документы можно найти с помощью утилиты find. Например, так:

```
find ~/Documents -type f -name "*.pdf"
```

Заархивируем и скопируем каждый найденный файл с помощью нашего скрипта. Команда для этого выглядит так:

```
find ~/Documents -type f -name "*.pdf" -exec make-backup.sh {} \;
```

Как мы уже знаем, экранированная точка с запятой `\;` означает выполнение действия над каждым из указанных файлов.

В результате на диск D будут скопированы архивы с каждым найденным PDF файлом. То есть каждый файл окажется в отдельном архиве. Это неудобно. Будет лучше собрать все PDF файлы в один архив.

Попробуем обработать нашим скриптом все найденные файлы за раз. Получится такая команда:

```
find ~/Documents -type f -name *.pdf -exec make-backup.sh {} +
```

В результате на диске D мы получим архив только первого найденного PDF файла. Куда делись остальные документы? Рассмотрим вызов утилиты bsdtar в нашем скрипте. Для простоты опустим выводы echo в лог-файл. Вызов выглядит так:

```
bsdtar -cjf "$1".tar.bz2 "$1"
```

Проблема в том, что мы обрабатываем только первый позиционный параметр, переданный на вход скрипта. Он сохраняется в переменной `$1`. При этом игнорируются все дальнейшие параметры в переменных `$2`, `$3` и т.д. Но именно в них передаются результаты поиска утилиты find, когда после действия `-exec` идёт знак `+`.

Чтобы решить проблему, воспользуемся переменной `$@`. Интерпретатор сохраняет в неё все параметры, переданные в скрипт. Перепишем вызов bsdtar следующим образом:

```
bsdtar -cjf "$1".tar.bz2 "$@"
```

Теперь вместо первого параметра `$1` мы передаём утилите bsdtar все входные параметры скрипта `$@`. Обратите внимание, что в качестве имени архива по-прежнему подставляется первый параметр `$1`.

Листинг 3-9 демонстрирует исправленный скрипт. Он обрабатывает произвольное число входных параметров.

Листинг 3-9. Обработка произвольного числа входных параметров

```
#!/bin/bash

bsdtar -cjf "$1".tar.bz2 "$@" &&
  echo "bsdtar - OK" > results.txt ||
  { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }

mv -f "$1".tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  ! echo "cp - FAILS" >> results.txt
```

В Bash есть переменная $*. Она очень похожа на $@. Если строку из переменной $* заключить в двойные кавычки при подстановке, Bash интерпретирует её как одно слово. В этом же случае строка в переменной $@ интерпретируется как набор слов.

Рассмотрим пример. Предположим, наш скрипт вызывается так:

```
make-backup.sh "one two three"
```

Тогда при подстановке “$*” в скрипте мы получим:

```
"one two three"
```

Подстановка же “$@” даст следующее:

```
"one" "two" "three"
```

Всегда предпочитайте использовать $@ вместо $*. Единственное исключение это случай, когда все входные параметры надо представить одним словом.

Возможности скриптов

На примере задачи резервного копирования мы рассмотрели возможности Bash-скриптов.

Напомним требования к задаче:

1. Команда должна храниться неограниченно долго.
2. Команда должна быстро вызываться.
3. Нужна возможность для её расширения.
4. Нужна возможность для сочетания её с другими командами.

Наш финальный скрипт `make-backup.sh` удовлетворяет всем этим требованиям. Проверим каждое из них:

1. Скрипт хранится на жёстком диске. Это долговременная память.
2. Скрипт легко интегрировать с Bash. Благодаря этому, его так же удобно вызывать, как и любую GNU-утилиту.
3. Скрипт представляет собой последовательность команд. Каждая из них начинается с новой строки. Его удобно читать и редактировать. Благодаря параметризации, его легко обобщить для решения однотипных задач.
4. За счёт интеграции с Bash скрипт удобно сочетать с другими командами, в том числе с помощью конвейеров и логических операторов.

Если ваша задача требует любую из перечисленных возможностей — пишите Bash-скрипт.

Переменные и параметры

В этой книге не раз упоминались переменные в Bash. Нам уже знаком список системных путей в переменной `PATH`. Мы использовали позиционные параметры в скрипте для резервного копирования. Настало время хорошо разобраться в этой теме.

Сначала выясним, что называется переменной в программировании. Это область памяти (обычно оперативной), в которой хранится значение. В первых языках программирования (например, **ассемблере**[245]) обращение к переменным происходит по их адресу. То есть, чтобы записать новое значение или прочитать текущее, надо указать адрес памяти. В 32-разрядных процессорах длина адреса 4 байта (число от 0 до 4294967295), а в 64-разрядных она в два раза больше. Запоминать и оперировать такими большими числами неудобно. Поэтому современные языки программирования позволяют заменять адреса переменных на их имена. Эти имена в процессе компиляции или интерпретации программы транслируются в адреса памяти. Таким образом всю работу по “запоминанию” больших чисел берёт на себя компилятор или интерпретатор.

Зачем нужны переменные? Наш опыт работы с `PATH` и позиционными параметрами показал, что переменные хранят какие-то данные. Это нужно для одной из следующих целей:

1. Передать информацию из одной части программы или системы в другую.
2. Сохранить промежуточный результат вычислений для дальнейшего использования.
3. Сохранить текущее состояние программы или системы. Это состояние может определять дальнейшее поведение.
4. Задать константное значение, которое позже будет многократно использоваться.

Для каждой цели в языках программирования вводится специальный тип переменной. Язык Bash не исключение.

[245] https://ru.wikipedia.org/wiki/Язык_ассемблера

Классификация переменных

У интерпретатора Bash есть два режима работы: интерактивный (командная оболочка) и неинтерактивном (исполнение скриптов). В каждом режиме переменные решают сходные задачи. Но контексты этих задач различаются. Поэтому признаков для классификации переменных в Bash больше, чем в других интерпретируемых языках.

Упростим терминологию для удобства. Это не совсем правильно, но позволит избежать путаницы. Когда говорим о скриптах, будем использовать термин “переменная” (variable). Когда речь о командной оболочке и аргументах командной строки, будем применять термин “параметр” (parameter). В английской литературе эти термины часто используют как синонимы.

Для классификации переменных в Bash есть четыре признака. Они приведены в таблице 3-1.

Таблица 3-1. Классификация переменных в Bash

Признак классификации	Типы	Определение	Примеры
Механизм установки	Пользовательские переменные	Устанавливаются пользователем.	`filename="README.txt" ; echo "$filename"`
	Зарезервированные (системные) переменные	Устанавливаются интерпретатором и нужны для его корректной работы.	`echo "$PATH"`
	Специальные параметры	Устанавливаются интерпретатором и доступны только для чтения.	`echo "$?"`
Область видимости[246]	Переменные окружения (или глобальные)	Доступны в любом экземпляре интерпретатора. Выводятся утилитой `env`, запущенной без параметров.	`echo "$PATH"`
	Локальные переменные	Доступны только в конкретном экземпляре интерпретатора.	`filename="README.txt" ; echo "$filename"`
Содержимое	Строка	Хранит строку.	`filename="README.txt"`

[246] https://ru.wikipedia.org/wiki/Область_видимости

Таблица 3-1. Классификация переменных в Bash

Признак классификации	Типы	Определение	Примеры
	Число	Хранит целое число.	`declare -i number=10/2 ; echo "$number"`
	Индексируемый массив	Хранит нумерованный список строк.	`cities=("London" "New York" "Berlin") ; echo "${cities[1]}" cities[0]="London" ; cities[1]="New York" ; cities[2]="Berlin" ; echo "${cities[1]}"`
	Ассоциативный массив[247]	Структура данных, каждый элемент которой — это пара ключ-значение. Ключом и значением являются строки.	`declare -A cities=( ["Alice"]="London" ["Bob"]="New York" ["Eve"]="Berlin" ) ; echo "${cities[Bob]}"`

Возможность изменения	Константны	Не могут быть удалены. Хранят значения, которые нельзя переопределить.	`readonly CONSTANT="ABC" ; echo "$CONSTANT"`
			
			`declare -r CONSTANT="ABC" ; echo "$CONSTANT"`

	Переменные	Могут быть удалены. Их значения можно переопределить.	`filename="README.txt"`

Рассмотрим каждый тип переменных.

Механизм установки

Пользовательские переменные

Назначение пользовательских переменных очевидно из названия. Их объявляет пользователь для своих целей. Такие переменные обычно хранят промежуточные результаты работы скрипта, его состояние и часто используемые константы.

[247] https://ru.wikipedia.org/wiki/Ассоциативный_массив

Чтобы объявить пользовательскую переменную, укажите её имя, поставьте знак равно и наберите значение переменной.

Рассмотрим пример. Объявим переменную с именем `filename`. В ней хранится имя файла `README.txt`. Объявление переменной выглядит так:

```
filename="README.txt"
```

Пробелы до и после знака равно не ставятся. Другие языки программирования это допускают, но не Bash. Это значит, что интерпретатор не сможет обработать следующее объявление:

```
filename = "README.txt"
```

Bash интерпретируют эту строку как вызов команды `filename` с двумя параметрами `=` и `"README.txt"`

В именах переменных допустимы только символы латинского алфавита, числа и знак подчёркивания _. Имя не должно начинаться с числа. Регистр букв важен. Это значит, что `filename` и `FILENAME` — две разные переменные.

Предположим, что мы объявили переменную `filename`. В результате для неё выделилась область в памяти процесса интерпретатора. В этой области сохранилась строка `README.txt`. Чтобы прочитать строку из памяти, к переменной надо обратиться по имени. При этом интерпретатор Bash должен понять, что вы имеете в виду. Если поставить знак доллара $ перед словом `filename`, Bash обработает его как имя переменной.

Обращение к переменной в команде или скрипте должно выглядеть так:

```
$filename
```

Bash обрабатывает слова со знаком доллара по-особенному. Встретив такое слово в команде, интерпретатор запускает механизм подстановки переменных (**parameter expansion**). Этот механизм заменяет все вхождения имени переменной на её значение. Рассмотрим следующую команду:

```
cp $filename ~
```

После подстановки переменных она будет выглядеть так:

```
cp README.txt ~
```

Всего интерпретатор совершает девять видов подстановок. Порядок их выполнения важен. Если его не учесть, могут возникнуть ошибки. Рассмотрим пример такой ошибки. Предположим, что в скрипте мы работаем с файлом “my file.txt”. Для удобства поместим его имя в переменную. Её объявление выглядит так:

```
filename="my file.txt"
```

Далее переменная используется в вызове утилиты cp. Команда её вызова выглядит так:

```
cp $filename ~
```

После подстановки переменных Bash выполняет word splitting. Это другой механизм подстановки. После него вызов утилиты cp станет таким:

```
cp my file.txt ~
```

Эта команда завершится с ошибкой. Вместо одного параметра с именем файла, в утилиту cp передаются два: `my` и `file.txt`. Таких файлов не существует.

Если в значении переменной встречается специальный символ, опять возникнет проблема. Например:

```
filename="*file.txt"
rm $filename
```

В результате вызова утилиты rm будут удалены все файлы, заканчивающиеся на `file.txt`. В этом виноват механизм globbing. Он тоже выполняется после подстановки переменных. После globbing в утилиту rm будут переданы все файлы из текущего каталога, имена которых соответствуют шаблону поиска `*file.txt`. Это может привести к неожиданному результату. Например, такому:

```
rm report_file.txt myfile.txt msg_file.txt
```

Чтобы избежать нежелательных подстановок Bash, заключайте все обращения к переменным в двойные кавычки “. Например, так:

```
filename1="my file.txt"
cp "$filename1" ~

filename2="*file.txt"
rm "$filename2"
```

Благодаря кавычкам, значение переменной будет подставлено без дальнейших изменений:

```
cp "my file.txt" ~
rm "*file.txt"
```

Мы уже знаем несколько подстановок[248], которые выполняет Bash. Таблица 3-2 приводит их полный список и порядок выполнения.

Таблица 3-2. Подстановки Bash

Порядок выполнения	Подстановка	Комментарий	Пример
1	Brace Expansion	Подстановка фигурных скобок.	`echo a{d,c,b}e`
2	Tilde Expansion	Подстановка символа тильда ~.	`cd ~`
3	Parameter Expansion	Подстановка параметров и переменных.	`echo "$PATH"`
4	Arithmetic Expansion	Подстановка вместо арифметических выражений их результатов.	`echo $((4+3))`
5	Command Substitution	Подстановка вместо команды её вывода.	`echo $(< README.txt)`
6	Process Substitution	Подстановка вместо команды её вывода. В отличие от Command Substitution эта подстановка выполняется асинхронно[249]. Ввод и вывод команды привязаны к временному файлу.	`diff <(sort file1.txt) <(sort file2.txt)`
7	Word Splitting	Разделение аргументов командной строки на слова и передача их в качестве отдельных параметров.	`cp file1.txt file2.txt ~`

[248] http://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Shell-Expansions
[249] https://ru.wikipedia.org/wiki/Асинхронность#Асинхронность_в_информатике

Таблица 3-2. Подстановки Bash

Порядок выполнения	Подстановка	Комментарий	Пример
8	Filename Expansion (globbing)	Подстановка имён файлов вместо шаблонов.	`rm ~/delete/*`
9	Quote Removal	Удаление всех неэкранированных символов \, ‘ и “, которые не были получены в результате одной из предыдущих подстановок.	`cp "my file.txt" ~`

Упражнение 3-1. Тестирование подстановок Bash

Выполните в терминале пример каждой подстановки Bash из таблицы 3-2.
Разберитесь, как получилась конечная команда.
Придумайте свои примеры.

Знак $ перед именем переменной — это сокращенная форма подстановки переменных. В полном виде она выглядит так:

```
${filename}
```

Используйте полную форму, чтобы избежать неоднозначности. Например, если сразу за именем переменной следует текст:

```
prefix="my"
name="file.txt"
cp "$prefix_$name" ~
```

Тогда интерпретатор ищет переменную с именем `prefix_`. То есть он приклеит символ подчёркивания к имени переменной. Полная форма записи подстановки переменных решит эту проблему:

```
cp "${prefix}_${name}" ~
```

Альтернативное решение — заключить каждое имя переменной в кавычки. Например, так:

```
cp "$prefix"_"$name" ~
```

Полная форма записи подстановки переменных устраняет неоднозначности. Кроме этого она поможет, когда переменная не была определена. В этом случае можно подставить некоторое значение по умолчанию. Например:

```
cp file.txt "${directory:-~}"
```

При обработке этой команды Bash проверит, определена ли переменная `directory` и имеет ли она непустое значение. Если это так, выполнится обычная подстановка. В противном случае Bash подставит значение, следующее за символом минус -. В нашем примере — это домашний каталог пользователя ~.

Задать значение по умолчанию можно несколькими способами. Все они приведены в таблице 3-3.

Таблица 3-3. Задание значения по умолчанию при подстановке

Форма записи	Описание
`${parameter:-word}`	Если переменная `parameter` не объявлена или имеет пустое значение, будет подставлено значение по умолчанию `word`. В противном случае подставляется значение переменной.
`${parameter:=word}`	Если переменная не объявлена или имеет пустое значение, ей будет присвоено значение по умолчанию. Затем она будет подставлена. В противном случае подставляется значение переменной. Переопределение позиционных и специальных параметров таким способом недопустимо.
`${parameter:?word}`	Если переменная не объявлена или имеет пустое значение, значение по умолчанию будет выведено в стандартный поток ошибок. После этого выполнение скрипта будет завершено с кодом возврата отличным от 0. В противном случае подставляется значение переменной.
`${parameter:+word}`	Если переменная не объявлена или имеет пустое значение, подстановки не будет. В противном случае подставляется значение по умолчанию.

Упражнение 3-2. Полная форма подстановки параметров

Напишите скрипт, который ищет файлы с расширением TXT в текущем каталоге.
Скрипт игнорирует подкаталоги.
Все найденные файлы копируются или перемещаются в домашний каталог пользователя.
При вызове скрипта можно выбрать действие: копировать или перемещать файлы.
Если действие не указано, выполняется копирование.

Зарезервированные переменные

Переменные может объявлять не только пользователь, но и сам интерпретатор. В этом случае они называются **зарезервированными** (reserved) или **переменными оболочки** (shell variables). Интерпретатор присваивает им значение по умолчанию. Значение некоторых переменных оболочки можно изменить.

Зарезервированные переменные исполняют две функции:

1. Передача информации от командного интерпретатора в запускаемое им приложение.
2. Хранение текущего состояния самого интерпретатора.

Переменные оболочки делятся на две группы:

1. Переменные Bourne Shell.
2. Переменные Bash.

Первая группа унаследована из Bourne Shell и нужна Bash для POSIX-совместимости. Часто используемые из этих переменных представлены в таблице 3-4.

Таблица 3-4. Зарезервированные переменные Bourne Shell

Имя	Значение
`HOME`	Домашний каталог текущего пользователя. Значение переменной используется при вызове встроенной команды `cd` без параметров и подстановке символа тильда ~.
`IFS`[250]	Список следующих друг за другом символов-разделителей. Вводимые строки будут разделены этими символами на слова (например, при word splitting). По умолчанию разделители такие: пробел, табуляция[251], перевод строки.
`PATH`	Список путей, по которым интерпретатор ищет вызываемые утилиты и программы. Пути в списке разделены двоеточиями.
`PS1`	Приглашение командной строки. Может включать управляющие символы[252]. Перед выводом на экран они заменятся на конкретные значения (например, имя текущего пользователя).

[250] http://mywiki.wooledge.org/IFS
[251] https://ru.wikipedia.org/wiki/Табуляция
[252] https://www.gnu.org/software/bash/manual/html_node/Controlling-the-Prompt.html#Controlling-the-Prompt

Кроме унаследованных переменных оболочки Bourne Shell в Bash появились новые. Они приведены в таблице 3-5. Кроме перечисленных есть и другие переменные Bash, но они используются редко.

Таблица 3-5. Зарезервированные переменные Bash

Имя	Значение
`BASH`	Полный путь до исполняемого файла Bash. Этот файл соответствует текущему процессу Bash.
`BASHOPTS`	Список дополнительных опций[253] текущего процесса Bash. Опции в списке разделены двоеточиями.
`BASH_VERSION`	Версия запущенного Bash интерпретатора.
`GROUPS`	Список групп, к которым относится текущий пользователь.
`HISTCMD`	Номер текущей команды в истории команд.
`HISTFILE`	Файл, в котором сохраняется история команд. По умолчанию это `~/.bash_history`.
`HISTFILESIZE`	Максимально допустимое число строк в файле истории команд. Значение по умолчанию 500.
`HISTSIZE`	Максимально допустимое число команд в файле истории команд. Значение по умолчанию 500.
`HOSTNAME`	Имя текущего компьютера как узла вычислительной сети.
`HOSTTYPE`	Строка с описанием аппаратной платформы, на которой запущен Bash.
`LANG`	Региональные настройки[254] пользовательского интерфейса. Некоторые из них переопределяются переменными `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_NUMERIC`, `LC_TYPE`.
`MACHTYPE`	Строка с описанием системы, на которой запущен Bash. Включает в себя информацию из переменных `HOSTTYPE` и `OSTYPE`.
`OLDPWD`	Предыдущий рабочий каталог, который устанавливала встроенная команда `cd`.
`OSTYPE`	Строка с описанием ОС, на которой запущен Bash.
`POSIXLY_CORRECT`	Если эта переменная определена, Bash работает в режиме POSIX-совместимости[255].

[253] https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html#The-Shopt-Builtin
[254] https://ru.wikipedia.org/wiki/Региональные_настройки_(программирование)
[255] https://www.gnu.org/software/bash/manual/html_node/Bash-POSIX-Mode.html#Bash-POSIX-Mode

Таблица 3-5. Зарезервированные переменные Bash

Имя	Значение
`PWD`	Текущий каталог, который установила встроенная команда `cd`.
`RANDOM`	Каждый раз при чтении этой переменной возвращается случайное число от 0 до 32767. При записи переменной назначается инициализирующее число (seed[256]) для генератора псевдослучайных чисел[257].
`SECONDS`	Количество секунд, прошедших с момента запуска текущего процесса Bash.
`SHELL`	Полный путь к исполняемому файлу командного интерпретатора для текущего пользователя.
`SHELLOPTS`	Список дополнительных опций[258] комндного интерпретатора. Опции в списке разделены двоеточиями.
`SHLVL`	Уровень вложенности текущего экземпляра Bash. Эта переменная увеличивается на единицу каждый раз при запуске Bash из командного интерпретатора.
`UID`	Идентификатор текущего пользователя.

Зарезервированные переменные делятся на три группы в зависимости от допустимых над ними действий:

1. При запуске интерпретатор назначает переменной значение. В течении всей сессии оно остаётся неизменным. Пользователь может его прочитать, но не изменить. Примеры: `BASHOPTS`, `GROUPS`, `SHELLOPTS`, `UID`.
2. При запуске интерпретатор назначает переменной значение по умолчанию. Оно меняется в результате выполнения пользователем команд или иных событий. Значение некоторых переменных можно переобъявить явно, но это может нарушить работу интерпретатора. Примеры: `HISTCMD`, `OLDPWD`, `PWD`, `SECONDS`, `SHLVL`.
3. При запуске интерпретатор назначает переменной значение по умолчанию. Единственный способ его изменить — это переобъявить. Примеры: `HISTFILESIZE`, `HISTSIZE`

Специальные параметры

Специальные параметры назначаются интерпретатором, как и переменные оболочки. Некоторые параметры хранят состояние запущенного экземпляра Bash (например, PID). Другие нужны для передачи параметров командной строки в вызываемые приложения и чтения их кода возврата. Все позиционные параметры относятся к специальным.

Часто используемые специальные параметры приведены в таблице 3-6.

[256] https://en.wikipedia.org/wiki/Random_seed
[257] https://ru.wikipedia.org/wiki/Генератор_псевдослучайных_чисел
[258] https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html#The-Shopt-Builtin

Таблица 3-6. Специальные параметры Bash

Имя	Значение
`$*`	Содержит все позиционные параметры, переданные в скрипт. Параметры начинаются не с нулевого (`$0`), а с первого (`$1`). Без двойных кавычек (`$*`), каждый позиционный параметр подставляется как отдельное слово. С двойными кавычками (`"$*"`), Bash подставляет одно слово, содержащее все параметры. Они разделяются первым символом зарезервированной переменной `IFS`.
`$@`	Массив со всеми позиционными параметрами, переданными в скрипт. Параметры начинаются с первого (`$1`). Без двойных кавычек (`$@`), Bash обрабатывает каждый элемент массива как строку без кавычек. В этом случае выполняется word splitting. С Двойными кавычками (“$@”), Bash обрабатывает каждый элемент массива как строку с кавычками. Wird splitting не происходит.
`$#`	Число позиционных параметров, переданных в скрипт.
`$1`, `$2`...	Содержит значение соответствующего позиционного параметра. `$1` соответствует первому параметру, `$2` — второму и т.д. Номера указываются в десятичной системе.
`$?`	Код возврата последней выполненной команды в активном режиме (foreground). Если команды выполнялись в конвейере, параметр хранит код возврата последней из них.
`$-`	Содержит опции текущего экземпляра интерпретатора.
`$$`	Идентификатор процесса текущего экземпляра интерпретатора. При подстановке в subshell, возвращает PID родительского процесса.
`$!`	Идентификатор процесса PID последней команды, запущенной в фоновом режиме.
`$0`	Имя текущего командного интерпретатора или выполняемого в данный момент скрипта.

Специальные параметры нельзя менять непосредственно. Например, следующее перезапись параметра `$1` не сработает:

```
1="new value"
```

Позиционные параметры можно изменять командой set. Она перезаписывает не один параметр, а сразу все. Форма вызова set в этом случае следующая:

```
set -- НОВОЕ_ЗНАЧЕНИЕ_$1 НОВОЕ_ЗНАЧЕНИЕ_$2 НОВОЕ_ЗНАЧЕНИЕ_$3...
```

Что делать, если нужно изменить только один параметр? Предположим, ваш скрипт вызывается с четырьмя параметрами. Например, так:

```
./my_script.sh arg1 arg2 arg3 arg4
```

Заменим третий параметр `arg3` на значение `new`. Это сделает такой вызов set:

```
set -- "${@:1:2}" "new" "${@:4}"
```

Первый аргумент set — подстановка первых двух элементов из массива `$@`. Второй аргумент — новое значение третьего параметра. Дальше подставляются все параметры, начиная с четвертого.

Все специальные параметры из таблицы 3-6 доступны в режиме POSIX-совместимости.

Область видимости

Переменные окружения

В любой программе и программной системе переменные разделены по **областям видимости** (scope). Область видимости — это часть программы или системы, в которой имя переменной остаётся связанным с её значением. Другими словами конвертировать имя переменной в её адрес можно только в области видимости этой переменной. За пределами области видимости то же самое имя может быть связано с другой переменной.

Область видимости называется **глобальной** (global scope), если распространяется на всю систему. То есть переменные этой области видимости доступны из любой части программы или системы.

Все зарезервированные переменные Bash находятся в глобальной области видимости. Переменные в этой области видимости называются **переменными окружения** (environment variables). Получается, что все зарезервированные переменные являются переменными окружения. Пользовательские переменные также можно объявлять в глобальной области видимости. Тогда они станут переменными окружения.

Зачем интерпретатор хранит переменные в глобальной области видимости? Дело в том, что в Unix есть специальный набор настроек. Они влияют на поведение запускаемых пользователем приложений. Например, региональные настройки. Согласно им каждое запущенное приложение адаптирует свой интерфейс. Именно такие настройки передаются через переменные окружения.

Предположим, что один процесс порождает дочерний процесс. В этом случае дочерний процесс копирует все переменные окружения родителя. Таким образом все утилиты и приложения, запущенные из командного интерпретатора, наследуют его переменные окружения. Так глобальные настройки передаются во все запускаемые пользователем программы.

Дочерние процессы могут изменять свои переменные окружения. В результате порождённые ими процессы унаследуют эти изменения. Однако, это никак не отразится на соответствующих переменных родительского процесса.

Чтобы объявить переменную окружения, используйте встроенную команду export. Например:

```
export BROWSER_PATH="/opt/firefox/bin"
```

Переменную можно сначала объявить, а потом поместить в глобальную область видимости. Например:

```
1 BROWSER_PATH="/opt/firefox/bin"
2 export BROWSER_PATH
```

Переменные окружения можно объявлять и переопределять для каждого запускаемого приложения отдельно. Для этого в команде вызова программы перечислите их имена и значения через пробел. Например, следующим образом:

```
MOZ_WEBRENDER=1 LANG="en_US.UTF-8" /opt/firefox/bin/firefox
```

Такое решение работает для интерпретатора Bash. Для других интерпретаторов (например, Bourne Shell) придётся использовать утилиту env. Вызовите утилиту, перечислите через пробел переменные окружения и добавьте команду запуска приложения. Например, так:

```
env MOZ_WEBRENDER=1 LANG="en_US.UTF-8" /opt/firefox/bin/firefox
```

Вызовите утилиту env без параметров. Она выведет все объявленные переменные окружения для текущего экземпляра интерпретатора. Попробуйте получить этот вывод в своём терминале:

```
env
```

Команда export и утилита env выводят одно и то же, если вызвать их без параметров. Предпочтительней использовать export. Во-первых, вывод команды отсортирован. Во-вторых, все значения переменных заключены в двойные кавычки. Это убережёт вас от ошибки, если в значении переменной встретится перевод строки.

Исторически сложилось так, что имена переменных окружения пишутся буквами в верхнем регистре. Поэтому давать имена локальным переменным в нижнем регистре считается хорошей практикой. Таким образом вы предотвратите случайное использование одной переменной вместо другой.

Локальные переменные

Мы познакомились с пользовательскими переменными. В зависимости от способа их объявления они могут быть **локальными** или переменными окружения (глобальными).

Добавить переменную в глобальную область видимости можно одним из следующих способов:

1. Добавить команду export в объявление переменной.
2. Передать переменную при запуске программы. Это можно сделать как с помощью утилиты env, так и без неё.

Если вы не сделали ничего из перечисленного, переменная будет локальной. Она будет доступна только в текущем экземпляре интерпретатора. Говорят, что локальная переменная имеет **ограниченную область видимости** (local scope). При этом никакие дочерние процессы (кроме subshell) её не наследуют.

Рассмотрим пример. Предположим, что вы объявили переменную в окне терминала MSYS2 следующим образом:

```
filename="README.txt"
```

Теперь в этом же окне терминала вы можете вывести её значение:

```
echo "$filename"
```

Та же самая команда отработает корректно, если выполнить её в subshell:

```
(echo "$filename")
```

Однако, если прочитать значение переменной из дочернего процесса, получится пустое значение. Чтобы запустить дочерний процесс, вызовите новый экземпляр интерпретатора в окне терминала. Например, так:

```
bash -c 'echo "$filename"'
```

В опции `-c` передаётся команда, которая выполнится дочерним процессом Bash. Аналогичный вызов Bash происходит неявно при запуске скрипта из командного интерпретатора.

Обратите внимание на одинарные кавычки ‘, в которые мы поместили вызов echo. Они отключают все подстановки для строки в них. У двойных кавычек поведение отличается. Они разрешают только подстановку команд и параметров. Если в нашем вызове `bash` использовать двойные кавычки, то произойдёт подстановка параметров. В результате команда запуска дочернего процесса Bash станет такой:

```
bash -c "echo README.txt"
```

Это совсем не то, что нам нужно. Мы проверяем, как дочерний процесс прочитает значение локальной переменной. В данном случае родительский процесс уже подставил это значение в вызов bash.

При изменении локальной переменной в subshell, её значение в родительском процессе не изменится. Например, в результате следующих команд напечатается строка “README.txt”:

```
1 filename="README.txt"
2 (filename="CHANGELOG.txt")
3 echo "$filename"
```

То есть присвоение переменной `filename` нового значения в subshell никак не отразилось на родительском процессе.

После объявления локальной переменной она попадает в список **переменных оболочки** (shell variables). К ним относятся все локальные переменные и переменные окружения, доступные в текущем экземпляре интерпретатора. Их можно вывести встроенной командой set, если запустить её без параметров. Попробуйте найти нашу переменную `filename` следующим образом:

```
set | grep filename=
```

В выводе этой команды вы увидите следующую строку:

```
filename=README.txt
```

Это значит, что переменная `filename` попала в список переменных оболочки.

Содержимое переменной

Типы переменных

В компилируемых языках программирования (например, C) принято использовать **статическую типизацию**[259]. Это означает, что при объявлении переменной указывается, как хранить её значение в памяти. Рассмотрим пример, чтобы лучше понять о чём речь.

Предположим, что мы объявляем переменную с именем `number`. В объявлении обязательно надо указать её тип. Например, целое беззнаковое (положительное) число размером два байта. В результате на эту переменную в памяти будет отведено ровно два байта. Далее переменной присваиваем значение 203 или 0xCB в шестнадцатеричной системе. В памяти это значение сохранится в следующем виде:

[259] https://ru.wikipedia.org/wiki/Статическая_типизация

```
00 CB
```

На самом деле в памяти современного компьютера вся информация хранится в двоичном виде. Вместо двоичного мы используем шестнадцатеричный формат для наглядности.

Чтобы хранить значение 203, достаточно и одного байта. Но при объявлении переменной мы зарезервировали два. Неиспользуемый байт в нашем случае останется равным нулю. Во всей области видимости переменной `number` никто не сможет использовать этот байт. Если переменная находится в глобальной области видимости, на протяжении работы программы этот байт будет зарезервирован и не использован.

Если переменной присвоить значение 14037 или 0x36D5, в отведённую ей область памяти запишутся следующие два байта:

```
36 D5
```

Порядок байтов[260] (endianness) при хранении чисел в памяти определяется свойствами CPU. В нашем примере порядок байтов от старшего к младшему (big-endian). Альтернативный порядок — от младшего к старшему (little-endian).

Теперь предположим, что в переменной нужно сохранить значение 107981 или 0x1A5CD. Это число не помещается в два байта. Размер переменной определён при её объявлении и не может быть автоматически расширен. Поэтому записываемое значение будет обрезано до двух байтов. В результате в памяти окажется следующее:

```
A5 CD
```

Старшая единица была отброшена. Теперь если вы прочитаете значение переменной `number`, то получите 42445 или 0xA5CD. Это значит, что записанное в `number` число 107981 потеряно и его невозможно восстановить. Такая проблема называется **переполнением**[261].

Рассмотрим другой пример статической типизации. Предположим, что нам нужно сохранить имя пользователя в переменной `username`. Для этого объявляем переменную строкового типа. Во многих компилируемых языках программирования при объявлении строки надо указать её допустимую длину. Для примера длина составляет десять символов. После объявления переменной присваиваем ей значение “Alice” в ASCII-кодировке. Если использовать компилятор языка C, строка в памяти будет выглядеть так:

[260] https://ru.wikipedia.org/wiki/Порядок_байтов
[261] https://ru.wikipedia.org/wiki/Целочисленное_переполнение

```
41 6C 69 63 65 00 00 00 00 00
```

ASCII-коды букв в шестнадцатеричной системе можно проверить по таблице[262].

Для хранения строки “Alice” достаточно шести байтов: пять для каждой буквы плюс один для нуль-терминатора (00) на конце. Однако, мы зарезервировали десять байтов, поэтому неиспользуемая память будет заполнена нулями или случайными значениями.

Динамическая типизация[263] — это альтернатива статической типизации. При динамической типизации способ хранения переменной выбирается иначе. Это происходит не в момент объявления переменной, а в момент присваивания ей нового значения. При присваивании переменной назначается **метаинформация**[264] о её текущем типе. Пока программа исполняется, значение переменной и соответствующая метаинформация могут меняться. Таким образом представление переменной в памяти также меняется. Динамическая типизация обычно применяется в интерпретируемых языках программирования (например, Python).

Метаинформация — это дополнительная информация о каком-либо объекте или данных. Библиотечный каталог[265] — хороший пример метаинформации. В нём для каждой книги заведена карточка. В карточке указывается автор, название произведения, издательство, год издания и количество страниц. Карточка содержит метаинформацию о книге.

Строго говоря, в языке Bash нет системы типов. Его нельзя считать языком со статической или динамической типизацией. В Bash все **скалярные переменные**[266] являются строками.

Скалярной называется переменная, которая хранит данные **примитивного типа**[267]. Это минимальные строительные блоки из которых собираются данные более сложных **составных типов**[268]. Как правило, скалярная переменная — это просто имя для адреса памяти, по которому хранится её значение.

Рассмотрим, как Bash представляет свои скалярные переменные в памяти. Есть следующее объявление:

```
declare -i number=42
```

В памяти переменная `number` будет сохранена как строка:

[262] https://ru.wikipedia.org/wiki/ASCII
[263] https://ru.wikipedia.org/wiki/Динамическая_типизация
[264] https://ru.wikipedia.org/wiki/Метаданные
[265] https://ru.wikipedia.org/wiki/Библиотечный_каталог
[266] https://en.wikipedia.org/wiki/Variable_(computer_science)
[267] https://ru.wikipedia.org/wiki/Примитивный_тип
[268] https://en.wikipedia.org/wiki/Composite_data_type

```
34 32 00
```

В языке с системой типов достаточно одного байта для хранения этого числа. Но в Bash нам потребовалось три: по байту для каждого символа (4 и 2) плюс нуль-терминатор на конце.

В Bourne Shell есть только скалярные переменные. В Bash появились два новых составных типа: индексируемый **массив**[269] и ассоциативный массив.

Индексируемый массив представляет собой пронумерованный набор строк. В нём каждой строке соответствует порядковый номер. Массивы этого типа хранятся в памяти в виде **связанного списка**[270]. Связанный список — это структура данных, состоящая из узлов. Каждый узел содержит данные (в нашем случае строку) и адрес в памяти следующего узла.

Ассоциативный массив устроен сложнее. Он представляет собой набор элементов. Каждый элемент состоит из двух строк. Первая из них называется ключом, а вторая — значением. Чтобы прочитать или записать строку в массив, нужно указать соответствующий ей ключ. Точно так же в индексируемом массиве для доступа к строке указывается её порядковый номер. Очевидно, что под одним номером может храниться только одна строка. Аналогично один ключ в ассоциативном массиве соответствует только одной строке. В памяти такой массив хранится в виде **хеш-таблицы**[271].

Почему “массивы” в Bash называются массивами? Ведь фактически они представляют собой связанные списки и хэш-таблицы. Настоящий массив — это структура данных, элементы которой хранятся в памяти последовательно друг за другом. У каждого элемента есть порядковый номер, называемый **индексом** (index) или идентификатором. Элементы “массивов” в Bash хранятся в памяти не последовательно. Значит согласно определению, это не массивы.

Рассмотрим, как элементы настоящего массива хранятся в памяти. Предположим, у нас есть массив с числами от пяти до девяти. Каждый его элемент занимает один байт. Тогда размер массива равен пяти байтам. В памяти он будет выглядеть следующим образом:

```
05 06 07 08 09
```

Индексы начинаются с нуля. Тогда элемент с порядковым номером ноль равен пяти, а с номером три — восьми. Элементы в памяти следуют друг за другом. Индекс представляет собой смещение в памяти относительно начала массива.

Вернёмся к вопросу о названиях структур данных в Bash. Ответ на него знают только авторы языка. Однако, мы можем предположить. Название “массив” даёт пользователю подсказку о том, как следует работать с этой структурой. Имея опыт в других языках (например, C) пользователь знает, что читать и писать отдельные элементы массива надо по индексу. Поэтому он сможет использовать уже знакомый ему синтаксис языка C в Bash. При этом пользователю совсем необязательно знать, как на самом деле “массив” Bash хранится в памяти.

[269] https://ru.wikipedia.org/wiki/Массив_(тип_данных)
[270] https://ru.wikipedia.org/wiki/Связный_список
[271] https://ru.wikipedia.org/wiki/Хеш-таблица

Атрибуты

У языка Bash нет системы типов. В нём все скалярные переменные хранятся в памяти как строки. Но в Bash есть составные типы — массивы. Они представляют собой комбинации строк.

Тип переменной (скалярная или составная) выбирается при её определении. Для этого надо указать метаинформацию, которая в Bash называется **атрибутами**. Кроме типа атрибуты определяют константность и область видимости переменной.

Чтобы указать атрибуты переменной, используйте встроенную команду declare. Если вызвать её без параметров, она выведет имена и значения всех объявленных в данный момент переменных: локальных и окружения. Эту же информацию выводит команда set.

У команды declare есть опция `-p`. Она добавляет в вывод атрибуты переменных.

Если вам нужна информация по конкретной переменной, передайте команде declare её имя. Например, так:

```
declare -p PATH
```

Команда declare без параметров выводит информацию не только об объявленных переменных, но и о доступных **подпрограммах**[272]. В Bash они называются **функциями**. Функция — это фрагмент программы или самостоятельный блок кода, который выполняет определённую задачу.

Чтобы команда declare вывела только информацию о функциях, используйте опцию `-f`. Например, так:

```
declare -f
```

Если вас интересует конкретная функция, укажите её имя после опции `-f`. Пример для функции `quote`:

```
declare -f quote
```

Эта команда выведет на экран определение функции.

Функция `quote` заключает переданную ей строку в одинарные кавычки. Если одинарные кавычки уже есть в строке, они будут экранированы. Функция вызывается точно так же, как и любая встроенная команда Bash. Например:

[272] https://ru.wikipedia.org/wiki/Функция_(программирование)

```
quote "this is a 'test' string"
```

Без опции `-f` declare не сможет вывести определение конкретной функции. То есть следующая команда не сработает:

```
declare quote
```

Команда declare не только выводит информацию об уже объявленных переменных и функциях. Она также устанавливает атрибуты при объявлении новой переменной.

Часто используемые опции команды declare приведены в таблице 3-7.

Таблица 3-7. Опции команды declare и соответствующие атрибуты переменных

Опция	Значение
`-a`	Объявленная переменная является индексируемым массивом. Элементы такого массива доступны по целочисленным номерам.
`-A`	Объявленная переменная является ассоциативным массивом. Каждому элементу такого массива соответствует ключ-строка.
`-g`	Объявление переменной в глобальной области видимости скрипта. При этом переменная не попадает в окружение.
`-i`	Объявление целочисленной переменной. Присваиваемое ей значение обрабатывается как арифметическое выражение.
`-r`	Объявление константы. После объявления ей нельзя присвоить другое значение.
`-x`	Объявление переменной окружения.

Рассмотрим примеры объявлений с атрибутами. Начнём с целочисленных и строковых переменных. Выполните в окне терминала следующее:

```
1 declare -i sum=11+2
2 text=11+2
```

Мы объявили две переменные с именами `sum` и `text`. Первая из них объявлена как целочисленная. Её значение равно 13 (сумма чисел 11 и 2). Значение второй переменной `text` равно строке “11+2”.

Обратите внимание, что обе переменные хранятся в памяти в виде строк. Опция `-i` не задаёт тип переменной, а ограничивает её допустимые значения.

Попробуйте присвоить переменной `sum` строку. Например, одним из следующих способов:

```
declare -i sum="test"
sum="test"
```

В результате переменная `sum` станет равна нулю.

Предположим, вы объявили переменную как целочисленную. Тогда для арифметических операции над ней не нужны дополнительные подстановки Bash. Например, следующие команды выполнятся корректно:

```
sum=sum+1       # 13 + 1 = 14
sum+=1          # 14 + 1 = 15
sum+=sum+1      # 15 + 15 + 1 = 31
```

В комментариях к командам приводятся их результаты.

Выполним те же самые операции со строковой переменной. Результаты будут отличаться:

```
text=text+1     # "text+1"
text+=1         # "text+1" + "1" = "text+11"
text+=text+1    # "text+11" + "text" + "1" = "text+11text+1"
```

Вместо арифметических операций над числами произошло склеивание строк. Чтобы выполнить эти операции над текстовой переменной, нужна арифметическая подстановка. Например:

```
text=11
text=$(($text + 2))  # 11 + 2 = 13
```

Опция `-r` команды declare объявляет переменную константой. Например, так:

```
declare -r filename="README.txt"
```

Теперь при каждой попытке изменить значение переменной `filename` или удалить её, Bash выводит сообщение об ошибке. Поэтому следующие команды завершатся с ошибкой:

```
filename="123.txt"
unset filename
```

Для удаления переменной любого типа, кроме константы, используйте встроенную команду unset.

Команда declare с опцией `-x` объявляет переменную окружения. То же самое объявление делает команда export. Следующие два объявления переменной `BROWSER_PATH` эквивалентны:

```
export BROWSER_PATH="/opt/firefox/bin"
declare -x BROWSER_PATH="/opt/firefox/bin"
```

Хорошей практикой считается использовать команду export вместо declare с `-x`. Это улучшает читаемость кода. Вам не нужно вспоминать, что значит опция `-x`. По этой же причине предпочитайте использовать команду readonly вместо declare с `-r`. Она тоже объявляет константу и её проще запомнить.

Обратите внимание, что команда readonly объявляет переменную в глобальной области видимости скрипта. Команда declare с `-r` даст другой результат. Если использовать declare в теле функции, объявленная переменная будет локальной. Вне функции она недоступна. Чтобы переменная стала глобальной (как с readonly), используйте опцию declare `-g`. Например, так:

```
declare -gr filename="README.txt"
```

Индексируемые массивы

В Bourne Shell есть только скалярные переменные (строки). В Bash по просьбам пользователей разработчики добавили массивы. Когда они могут понадобиться?

У строкового типа есть серьёзное ограничение. При записи в скалярную переменную какого-то значения, логически получается один элемент. Например, вы сохраняете в переменную с именем `files` список файлов. Элементы списка разделены пробелами. В результате `files` хранит одну строку с точки зрения Bash. Это может привести к ошибкам.

Как мы выяснили, стандарт POSIX разрешает любые символы в именах файлов, кроме нуль-терминатора (NUL). NUL означает конец имени файла. Этот же самый символ в Bash означает конец строки. Поэтому строковая переменная может содержать NUL не в произвольном месте, а только в конце. Получается, у вас нет надёжного способа разделить имена файлов в списке. NUL использовать нельзя. Любой другой символ-разделитель может встретиться в этих именах.

Именно проблема разделителя мешает надёжно обработать вывод утилиты ls. Утилита find позволяет разделять элементы своего вывода с помощью NUL, а ls - нет. Никакой символ кроме NUL не будет надёжным разделителем. Поэтому не объявляйте переменные так:

```
files=$(ls Documents/*.txt)
```

В результате в переменную `files` будет записана строка со всеми TXT файлами каталога `Documents`. Если в именах файлов встречаются пробелы или символы переноса строки, восстановить исходную информацию будет проблематично.

Массивы добавлены в Bash для решения этой проблемы. Массив хранит список отдельных элементов. Прочитать их в исходном виде не составляет труда. Поэтому вместо присваивания переменной вывода утилиты ls, используйте массив. Например:

```
declare -a files=(Documents/*.txt)
```

Инициализацией массива называется определение его элементов. Массив можно инициализировать при объявлении или после. В примере выше инициализация `files` происходит при объявлении.

Bash способен вывести тип переменной самостоятельно. Этот механизм работает, когда вы присваиваете значение переменной при объявлении. В зависимости от значения Bash добавляет соответствующий атрибут. В таком случае команду declare можно опустить. Например, наш массив `files` можно объявить без declare:

```
files=(Documents/*.txt)
```

Предположим, что элементы массива получаются не в результате подстановки, а известны заранее. В этом случае их можно задать явно при объявлении. Это будет выглядеть так:

```
files=("/usr/share/doc/bash/README" "/usr/share/doc/flex/README.md" "/usr/share/doc/\
xz/README")
```

Элементы массива можно читать из значений других переменных. Например:

```
1 bash_doc="/usr/share/doc/bash/README"
2 flex_doc="/usr/share/doc/flex/README.md"
3 xz_doc="/usr/share/doc/xz/README"
4 files=("$bash_doc" "$flex_doc" "$xz_doc")
```

Элементами массива `files` станут текущие значения переменных `bash_doc`, `flex_doc` и `xz_doc`. Изменение этих переменных после объявления массива, никак не отразится на его содержимом.

При объявлении массива для каждого его элемента можно явно указать индекс. Например:

```
1 bash_doc="/usr/share/doc/bash/README"
2 flex_doc="/usr/share/doc/flex/README.md"
3 xz_doc="/usr/share/doc/xz/README"
4 files=([0]="$bash_doc" [1]="$flex_doc" [5]="/usr/share/doc/xz/README")
```

Обратите внимание на отсутствие пробелов до и после каждого знака равно. Запомните простое правило: при объявлении переменных в Bash пробелы до и после знака равно не ставятся.

Вместо инициализации всего массива за раз, можно определять его элементы по отдельности. Например, так:

```
files[0]="$bash_doc"
files[1]="$flex_doc"
files[5]="/usr/share/doc/xz/README"
```

В последних двух объявлениях массива `files` нумерация индексов идёт не по порядку. Это не ошибка. Bash допускает **массивы с пропусками** (sparse arrays).

Вывести все элементы массива можно с помощью следующей подстановки:

```
$ echo "${files[@]}"
/usr/share/doc/bash/README /usr/share/doc/flex/README.md /usr/share/doc/xz/README
```

В этой команде символ $ указывает на приглашение командной строки. После него через пробел идёт команда. На следующей строке — вывод её результата.

Иногда бывает полезно вывести только индексы элементов. Для этого в подстановке добавьте восклицательный знак перед именем массива. Например:

```
$ echo "${!files[@]}"
0 1 5
```

При подстановке индекс элемента можно рассчитать по формуле. Просто укажите в квадратных скобках арифметическое выражение для его вычисления. Например, так:

```
echo "${files[4+1]}"
files[4+1]="/usr/share/doc/xz/README"
```

В арифметическом выражении можно использовать переменные. Причём они могут быть объявлены и как целочисленные, и как строковые. Например:

```
i=4
echo "${files[i+1]}"
files[i+1]="/usr/share/doc/xz/README"
```

Следующие подряд элементы можно подставить одной командой. Для этого после двоеточия укажите стартовый индекс и число элементов. Например, так:

```
$ echo "${files[@]:1:2}"
/usr/share/doc/flex/README.md /usr/share/doc/xz/README
```

Эта команда выведет два элемента начиная с первого. Обратите внимание, что индексы элементов в этом случае не важны. Мы прочитали имена файлов под номерами 1 и 5.

Bash, начиная с версии 4, предоставляет встроенную команду readarray (также известную как mapfile). Она читает содержимое текстового файла в массив. Рассмотрим, как её использовать.

Предположим, что у нас есть файл с именем `names.txt`. Его содержимое такое:

```
Alice
Bob
Eve
Mallory
```

Создадим массив со строками из этого файла. Для этого достаточно выполнить следующую команду:

```
readarray -t names_array < names.txt
```

В результате содержимое файла будет записано в массив с именем `names_array`.

Упражнение 3-3. Объявление массивов

Выполните самостоятельно все рассмотренные способы объявления массивов:

1. С помощью команды declare.

2. Без использования команды declare.

3. Все элементы массива подставляются в результате globbing.

4. Все элементы массива указываются явно при объявлении.

5. Каждый элемент массива задаётся отдельно.

6. В качестве элементов массива используются значения объявленных ранее переменных.

7. Элементы массива читаются из текстового файла.

Выведите содержимое массива с помощью команды echo для каждого случая.
Убедитесь, что объявления выполнились корректно.

Мы научились объявлять и инициализировать индексируемые массивы. Теперь разберёмся, как их использовать. Предположим, что массив `files` содержит список имён файлов. Вам нужно скопировать первый файл в списке. Для этого воспользуемся утилитой cp:

```
cp "${files[0]}" ~/Documents
```

В большинстве языков программирования принято нумеровать элементы массивов и строк с нуля, а не с единицы. Это правило справедливо и для Bash.

Для чтения элемента массива нужна полная форма подстановки параметров с фигурными скобками. После имени переменной в квадратных скобках указывается индекс нужного элемента. Для подстановки всех элементов используйте символ @ вместо индекса. Например, так:

```
cp "${files[@]}" ~/Documents
```

Чтобы получить размер массива, поставьте символ решётка # перед его именем. Например:

```
echo "${#files[@]}"
```

При подстановке элементов массива всегда используйте двойные кавычки, чтобы предотвратить word splitting.

Чтобы удалить элемент массива, используйте встроенную команду unset. Например, удаление четвёртого элемента (не забывайте про нумерацию с нуля) выглядит так:

```
unset 'files[3]'
```

Обратите внимание на обязательные одинарные кавычки. Они выключат все возможные подстановки интерпретатора.

С помощью команды unset можно также очистить весь массив:

```
unset files
```

Ассоциативные массивы

Мы рассмотрели индексируемые массивы. В них элементами являются строки, а индексами — целые положительные числа. Массивы этого типа по указанному индексу возвращают соответствующую ему строку.

В Bash версии 4 добавили ассоциативные массивы. В них индексы — это не числа, а строки. Такая строка-индекс называется **ключом** (key). Ассоциативный массив по указанной строке-индексу возвращает соответствующую ей строку-значение. Когда это может быть полезно? Рассмотрим пример.

Предположим, нам нужен скрипт для хранения контактов. Скрипт позволяет добавить в список имя человека, его email или номер телефона. Для простоты можно опустить фамилию и хранить только имя. По запросу скрипт выводит контактные данные человека.

Если для нашей задачи применить индексируемый массив, поиск контактов будет неэффективным. Придётся перебирать элементы массива друг за другом и сравнивать имя человека с искомым. Если нужное имя найдено, скрипт выведет на экран соответствующие контактные данные.

Ассоциативный массив сделает поиск по контактам быстрее и проще. В этом случае перебирать элементы массива не нужно. Достаточно указать ключ и получить соответствующий элемент массива. Рассмотрим это решение подробнее.

Объявление и инициализация ассоциативного массива с контактами выглядит так:

```
declare -A contacts=(["Alice"]="alice@gmail.com" ["Bob"]="(697) 955-5984" ["Eve"]="(\
245) 317-0117" ["Mallory"]="mallory@hotmail.com")
```

Ассоциативный массив всегда объявляется с командой declare и её опцией `-A`. Bash не сможет правильно вывести тип переменной, даже если указать в качестве индексов строки. Поэтому в результате следующего объявления вы получите индексируемый массив, а не ассоциативный:

```
contacts=(["Alice"]="alice@gmail.com" ["Bob"]="(697) 955-5984" ["Eve"]="(245) 317-01\
17" ["Mallory"]="mallory@hotmail.com")
```

Проверим, чему равна переменная `contacts` в этом случае:

```
$ declare -p contacts
declare -a contacts='([0]="mallory@hotmail.com")'
```

Мы получили индексируемый массив с одним элементом. Bash сконвертировал все строки-ключи в индекс ноль. Поэтому в нулевой элемент записался только контакт последнего человека в списке инициализации.

Элементы массива можно задавать по отдельности. Например, так:

```
declare -A contacts
contacts["Alice"]="alice@gmail.com"
contacts["Bob"]="(697) 955-5984"
contacts["Eve"]="(245) 317-0117"
contacts["Mallory"]="mallory@hotmail.com"
```

Итак, мы объявили ассоциативный массив. Его элементы доступны по ключам. В нашем случае ключ — это имя человека.

Чтобы прочитать элемент массива по его ключу, выполните такую подстановку:

```
$ echo "${contacts["Bob"]}"
(697) 955-5984
```

Для вывода всех элементов массива просто укажите в качестве ключа символ @:

```
$ echo "${contacts[@]}"
(697) 955-5984 mallory@hotmail.com alice@gmail.com (245) 317-0117
```

Чтобы вывести список всех ключей, поставьте восклицательный знак ! перед именем массива. Например:

```
$ echo "${!contacts[@]}"
Bob Mallory Alice Eve
```

Размер массива выводится с помощью символа решётка #. Например, так:

```
$ echo "${#contacts[@]}"
4
```

Поместим ассоциативный массив с контактами в скрипт. Тогда имя интересующего человека можно передать через параметр командной строки. В ответ скрипт выведет соответствующий email или телефон.

Листинг 3-10 демонстрирует скрипт с контактами.

Листинг 3-10. Скрипт для хранения контактов

```
#!/bin/bash

declare -A contacts=(
  ["Alice"]="alice@gmail.com"
  ["Bob"]="(697) 955-5984"
  ["Eve"]="(245) 317-0117"
  ["Mallory"]="mallory@hotmail.com")

echo "${contacts["$1"]}"
```

Для редактирования контактов измените инициализацию массива в скрипте.

Удалить ассоциативный массив или его элемент можно командой unset:

```
unset contacts
unset 'contacts[Bob]'
```

Подстановка нескольких элементов ассоциативного массива работает так же, как и для индексируемого массива. Например:

```
$ echo "${contacts[@]:Bob:2}"
(697) 955-5984 mallory@hotmail.com
```

В этом случае Bash подставит два элемента: соответствующий ключу `Bob` и следующий за ним в памяти. Проблема в том, что порядок следования элементов в памяти не соответствует порядку их инициализации. Индекс каждого элемента рассчитывается **хеш-функцией**[273]. На вход она принимает строку-ключ, а на выходе возвращает уникальное целое число. Из-за этой особенности подстановка нескольких элементов ассоциативного массива мало полезна на практике.

Условные операторы

Работая с утилитой find, мы впервые познакомились с условными конструкциями. Затем мы выяснили, что у Bash есть собственные логические операторы И (&&) и ИЛИ (||). Это не единственные формы ветвления в языке Bash.

В этом разделе мы рассмотрим операторы if и case. Они часто используются в скриптах. Эти операторы взаимозаменяемы. Но каждый из них лучше справляется с определёнными задачами.

Оператор if

Представьте, что вы пишете однострочную команду. При этом вы стараетесь сделать её как можно компактнее. Короткая команда удобнее длинной. Её проще набрать и меньше вероятность ошибиться.

Теперь представьте, что вы пишете скрипт. Он хранится на жёстком диске. При этом вы вызываете его регулярно и иногда изменяете. Здесь компактность не так важна. В первую очередь скрипт должен быть удобен для чтения и редактирования.

Операторы && и || хорошо подходят для однострочных команд. Но для скриптов есть альтернативы получше. На самом деле всё зависит от конкретного случая. Иногда операторы && и || вписываются в код скрипта без проблем. Но зачастую они приводят к трудночитаемому коду. Поэтому лучше заменять их на операторы if или case. Рассмотрим эти случаи подробнее.

Ещё раз обратимся к скрипту для резервного копирования из листинга 3-9. Вызов утилиты bsdtar в этом скрипте выглядит так:

[273] https://ru.wikipedia.org/wiki/Хеш-функция

```
bsdtar -cjf "$1".tar.bz2 "$@" &&
  echo "bsdtar - OK" > results.txt ||
  { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
```

Чтобы улучшить читаемость скрипта, мы разбили вызовы утилит bsdtar и mv на отдельные команды. Это помогло, но лишь отчасти. Вызов bsdtar всё ещё слишком длинный. При его изменении легко допустить ошибку. Такой подверженный ошибкам код называется **хрупким** (fragile). Это верный признак плохого технического решения, принятого при его разработке.

Распишем алгоритм вызова bsdtar по шагам:

1. Прочитать из переменной $@ список файлов и каталогов. Архивировать и сжать их.
2. Если архивирование и сжатие прошло успешно, записать в лог-файл строку “bsdtar - OK”.
3. Если произошла ошибка, записать в лог-файл строку “bsdtar - FAILS” и завершить работу скрипта.

Вопросы вызывает третий пункт. При успешном завершении bsdtar выполняется только одно действие. В случае же ошибки — действий два и они объединены в **блок команд**[274] с помощью фигурных скобок[275] .

Конструкция if введена в язык Bash как раз для удобства работы с блоками команд. В общем случае она выглядит так:

```
if УСЛОВИЕ
then
  ДЕЙСТВИЕ
fi
```

Эту конструкцию можно записать и в одну строку. Для этого перед then и fi добавьте по точке с запятой:

```
if УСЛОВИЕ; then ДЕЙСТВИЕ; fi
```

УСЛОВИЕ и ДЕЙСТВИЕ в операторе if представляют собой команду или блок команд. Если УСЛОВИЕ завершилось успешно с кодом возврата 0, будут выполнены команды, соответствующие ДЕЙСТВИЮ.

Рассмотрим следующий пример конструкции if:

[274] https://ru.wikipedia.org/wiki/Блок_(программирование)
[275] https://www.gnu.org/software/bash/manual/html_node/Command-Grouping.html

```
if cmp file1.txt file2.txt &> /dev/null
then
  echo "Файлы file1.txt и file2.txt идентичны"
fi
```

Здесь в качестве УСЛОВИЯ вызывается утилита cmp. Она побайтово сравнивает содержимое двух файлов. Если они отличаются, cmp напечатает в стандартный поток вывода позицию первого различающегося символа. При этом код возврата утилиты будет отличным от нуля. Если содержимое файлов совпадает — утилита вернёт ноль.

В конструкции if нас интересует только код возврата утилиты cmp. Поэтому мы перенаправляем её вывод в файл `/dev/null`[276]. Это специальный системный файл. Запись в него всегда проходит успешно, а все записанные данные удаляются.

Итак, если содержимое файлов `file1.txt` и `file2.txt` совпадает, утилита cmp вернёт код ноль. Тогда условие конструкции if будет истинно. В этом случае команда echo выведет сообщение на экран.

Мы рассмотрели пример, когда действие совершается при выполнении условия. Но бывают случаи, когда с помощью условия выбирается одно из двух действий. Именно так работает конструкция if-else. В общем виде она выглядит так:

```
if УСЛОВИЕ
then
  ДЕЙСТВИЕ_1
else
  ДЕЙСТВИЕ_2
fi
```

Запись if-else в одну строку выглядит так:

```
if УСЛОВИЕ; then ДЕЙСТВИЕ_1; else ДЕЙСТВИЕ_2; fi
```

В этой конструкции блок команд ДЕЙСТВИЕ_2 выполнится, если УСЛОВИЕ вернёт код ошибки отличный от нуля. В противном случае выполнится блок ДЕЙСТВИЕ_1.

Конструкцию if-else можно дополнить условиями и действиями с помощью блоков elif. Рассмотрим пример. Предположим, в зависимости от значения переменной вы выбираете одно из трёх действий. Следующая конструкция if даст такое поведение:

[276] https://ru.wikipedia.org/wiki//dev/null

```
if УСЛОВИЕ_1
then
  ДЕЙСТВИЕ_1
elif УСЛОВИЕ_2
then
  ДЕЙСТВИЕ_2
else
  ДЕЙСТВИЕ_3
fi
```

Количество блоков elif неограниченно. Добавляйте их в конструкцию if-else столько, сколько вам нужно.

Дополним наш пример сравнения двух файлов. Будем выводить сообщение не только при их совпадении, но и при их различии. Для этого воспользуемся конструкцией if-else. Получится следующее:

```
if cmp file1.txt file2.txt &> /dev/null
then
  echo "Файлы file1.txt и file2.txt идентичны"
else
  echo "Файлы file1.txt и file2.txt различаются"
fi
```

Вернёмся к нашему скрипту резервного копирования. В нём в зависимости от результата утилиты bsdtar выполняется блок команд. Поэтому операторы && и || стоит заменить на конструкцию if.

Перепишем вызов и обработку результата bsdtar. Для этого применим конструкцию if-else. Получится следующее:

```
if bsdtar -cjf "$1".tar.bz2 "$@"
then
  echo "bsdtar - OK" > results.txt
else
  echo "bsdtar - FAILS" > results.txt
  exit 1
fi
```

Согласитесь, что теперь читать и редактировать код стало проще. Его можно упростить ещё. Применим технику **раннего возврата**[277] и заменим конструкцию if-else на if:

[277] https://habr.com/ru/post/348074/

```
if ! bsdtar -cjf "$1".tar.bz2 "$@"
then
  echo "bsdtar - FAILS" > results.txt
  exit 1
fi

echo "bsdtar - OK" > results.txt
```

Поведение кода осталось таким же. С помощью логического отрицания ! мы инвертировали результат утилиты bsdtar. Теперь если она завершится с ошибкой, условие оператора if станет истинным. В этом случае выводится сообщение “bsdtar - FAILS” и вызывается команда exit. Если утилита bsdtar отработает корректно, блок команд конструкции if не выполнится. В результате в лог-файл напечатается строка “bsdtar - OK”.

Рассмотрим технику раннего возврата. Это полезный приём, который сделает ваш код проще и понятнее для чтения. Его идея в том, чтобы в случае ошибки завершить программу как можно раньше. Если этого не сделать, вам не избежать вложенных конструкций if.

Рассмотрим пример. Представьте, что некоторый алгоритм состоит из пяти действий. Каждое последующее действие выполняется только при успешном завершении предыдущего. Этот алгоритм можно реализовать с помощью вложенных конструкций if. Например, так:

```
if ДЕЙСТВИЕ_1
then
  if ДЕЙСТВИЕ_2
  then
    if ДЕЙСТВИЕ_3
    then
      if ДЕЙСТВИЕ_4
      then
        ДЕЙСТВИЕ_5
      fi
    fi
  fi
fi
```

Такое вложение выглядит запутанным. Добавьте в него блоки else с обработкой ошибок и читать код станет ещё сложнее.

Вложенные операторы if — это серьёзная проблема для читаемости кода. Она решается техникой раннего возврата. Применим её для нашего алгоритма. Получим следующее:

```
if ! ДЕЙСТВИЕ_1
then
  # обработка ошибки
fi

if ! ДЕЙСТВИЕ_2
then
  # обработка ошибки
fi

if ! ДЕЙСТВИЕ_3
then
  # обработка ошибки
fi

if ! ДЕЙСТВИЕ_4
then
  # обработка ошибки
fi

ДЕЙСТВИЕ_5
```

Поведение программы не изменилось. Алгоритм по-прежнему состоит из пяти действий. Ошибка при выполнении любого из них прерывает работу программы. Но благодаря раннему возврату, код стал проще и понятнее.

В последнем примере мы используем **комментарии**[278]. Они выглядят так: “# обработка ошибки”. Комментарий — это строка или её часть, которую игнорирует интерпретатор. В Bash комментарием является всё, что идёт после символа решётка #.

Польза комментариев — это предмет бесконечных споров в сообществе программистов. Они нужны для пояснений к коду. Однако, некоторые считают, что наличие комментариев — это признак непонятного, плохо написанного кода. Если вы только начинаете изучать программирование, обязательно используйте их. Комментируйте сложные конструкции в своих скриптах, смысл которых вы можете забыть. В будущем это поможет вспомнить, как эти конструкции работают.

Предположим, что каждому действию алгоритма соответствует одна короткая команда. Все ошибки обрабатываются командой exit без вывода в лог-файл. В этом случае конструкции if можно заменить на оператор ||. При этом код останется простым и понятным. Он будет выглядеть, например, так:

[278] https://ru.wikipedia.org/wiki/Комментарии_(программирование)

```
ДЕЙСТВИЕ_1 || exit 1
ДЕЙСТВИЕ_2 || exit 1
ДЕЙСТВИЕ_3 || exit 1
ДЕЙСТВИЕ_4 || exit 1
ДЕЙСТВИЕ_5
```

Операторы && и || выразительнее чем if только тогда, когда действия и обработка ошибок выполняются короткими командами.

Перепишем скрипт резервного копирования с использованием конструкции if. Листинг 3-11 демонстрирует результат.

Листинг 3-11. Скрипт с ранним возвратом

```
#!/bin/bash

if ! bsdtar -cjf "$1".tar.bz2 "$@"
then
  echo "bsdtar - FAILS" > results.txt
  exit 1
fi

echo "bsdtar - OK" > results.txt

mv -f "$1".tar.bz2 /d && echo "cp - OK" >> results.txt || ! echo "cp - FAILS" >> res\
ults.txt
```

В скрипте мы заменили операторы && и || в вызове bsdtar на конструкцию if. Поведение скрипта при этом не изменилось.

В общем случае логические операторы и конструкция if не эквивалентны. Рассмотрим пример. Предположим, есть выражение из трёх команд A, B и C:

```
A && B || C
```

Может показаться, что следующая конструкция if-else даст такое же поведение:

```
if A
then
  B
else
  C
fi
```

В этой конструкции если A истинно, то выполняется B. Иначе выполняется C. Но в выражении с операторами && и || поведение иное! В нём если A истинно, выполняется B. Далее выполнение C зависит от результата B. Если B истинно, C выполняться не будет. Если же B ложно, C исполнится. Таким образом исполнение C зависит и от результата A, и от результата B. В конструкции if-else такой зависимости нет.

Упражнение 3-4. Оператор if

```
Дана Bash команда. Она ищет строку "123" в файлах каталога с именем target.
Если в файле встречается строка, он копируется в текущий каталога.
Если строки в файле нет, он удаляется из каталога target.

Команда выглядит так:
( grep -RlZ "123" target | xargs -0 cp -t . && echo "cp - OK" || ! echo "cp - FAILS"\
 ) && ( grep -RLZ "123" target | xargs -0 rm && echo "rm - OK" || echo "rm - FAILS" \
)

Сделайте из этой команды скрипт. Замените операторы && и || на конструкции if-else.
```

Оператор [[

Мы познакомились с оператором if. В качестве условия в нём вызывается встроенная команда Bash или сторонняя утилита.

Например, вызовем утилиту grep и в зависимости от её результата выберем действие. Если использовать grep в условии оператора if, нам пригодится опция утилиты `-q`. С ней grep не станет выводить результат на стандартный поток вывода. Вместо этого при первом вхождении искомой строки или шаблона вернётся код ноль. Условие if с вызовом grep может выглядеть так:

```
1 if grep -q -R "General Public License" /usr/share/doc/bash
2 then
3   echo "Bash распространяется под лицензией GPL"
4 fi
```

Теперь предположим, что в условии if сравниваются две строки или числа. Для этой цели в Bash есть специальный оператор [[. Двойные квадратные скобки являются **зарезервированным словом**[279] интерпретатора. Это значит, что интерпретатор обрабатывает его самостоятельно.

В Bourne shell оператора [[нет. Он также не попал в POSIX-стандарт. Поэтому если важна совместимость со стандартом, используйте устаревший оператор test[280] или его синонимом [. Никогда не используйте test в Bash. Его возможности по сравнению с оператором [[ограничены, а правильные способы применения неочевидны.

[279] https://ru.wikipedia.org/wiki/Зарезервированное_слово
[280] http://mywiki.wooledge.org/BashFAQ/031

Начнём с простого примера использования оператора [[. Надо сравнить две строки. В этом случае условие if выглядит так:

```
if [[ "abc" = "abc" ]]
then
  echo "Строки равны"
fi
```

Выполните этот код. На экран будет выведено сообщение, что строки равны. Подобная проверка не слишком полезна. Чаще значение какой-то переменной сравнивается со строкой. В этом случае оператор [[выглядит так:

```
if [[ "$var" = "abc" ]]
then
  echo "Переменная равна строке \"abc\""
fi
```

В этом условии двойные кавычки необязательны. Globbing и word splitting не выполняются при подстановке переменной в операторе [[. То есть интерпретатор никак не обрабатывает значение переменной var, а использует его как есть. Проблема возникнет, только если пробелы встречаются не в значении переменной, а в строке справа. Например:

```
if [[ "$var" = abc def ]]
then
  echo "Переменная равна строке \"abc def\""
fi
```

Выполнение такого условия завершится с ошибкой.

Чтобы избежать подобных проблем, всегда используйте кавычки при работе со строками. Последуем этому правилу и перепишем прошлый пример так:

```
if [[ "$var" = "abc def" ]]
then
  echo "Переменная равна строке abc def"
fi
```

В операторе [[можно сравнить значения двух переменных друг с другом. Например, так:

```
if [[ "$var" = "$filename" ]]
then
  echo "Переменные равны"
fi
```

В таблице 3-8 приведены все операции сравнения строк, допустимые в операторе [[.

Таблица 3-8. Операции сравнения строк в операторе [[

Операция	Описание	Пример
>	Строка слева больше строки справа в порядке **лексикографической сортировки**[281].	[[“bb” > “aa”]] && echo “Строка bb больше чем aa”
<	Строка слева меньше строки справа в порядке лексикографической сортировки.	[[“ab” < “ac”]] && echo “Строка ab меньше чем ac”
= или ==	Строки равны.	[[“abc” = “abc”]] && echo “Строки равны”
!=	Строки не равны.	[[“abc” != “ab”]] && echo “Строки не равны”
-z	Строка пустая.	[[-z “$var”]] && echo “Строка пустая”
-n	Строка не пустая.	[[-n “$var”]] && echo “Строка не пустая”
-v	Переменная объявлена с любым значением.	[[-v var]] && echo “Переменная объявлена”
= или ==	Поиск в строке слева подстроки по шаблону справа. В этом случае шаблон не заключается в кавычки.	[[“$filename” = READ*]] && echo “Имя файла начинается с READ”
!=	Проверка, что шаблон справа не встречается в строке слева. В этом случае шаблон не заключается в кавычки.	[[“$filename” != READ*]] && echo “Имя файла не начинается с READ”
=~	Поиск в строке слева подстроки по **регулярному выражению**[282] справа.	[[“$filename” =~ ^READ.*]] && echo “Имя файла начинается с READ”

В операторе [[можно использовать логические операции И, ИЛИ и НЕ. Они комбинируют несколько выражений в одно условие. Таблица 3-9 приводит примеры таких условий.

[281] https://ru.wikipedia.org/wiki/Лексикографический_порядок
[282] https://www.opennet.ru/docs/RUS/bash_scripting_guide/c11895.html

Таблица 3-9. Логические операции в операторе [[

Операция	Описание	Пример
&&	Логическое И.	[[-n “$var” && “$var” < “abc”]] && echo “Строка не пустая и меньше чем abc”
||	Логическое ИЛИ.	[[“abc” < “$var” \|\| -z “$var”]] && echo “Строка больше чем abc или пустая”
!	Логическое НЕ.	[[! “abc” < “$var”]] && echo “Строка не больше чем abc”

Выражения в операторе [[можно группировать с помощью круглых скобок. Например, так:

```
[[ (-n "$var" && "$var" < "abc") || -z "$var" ]] && echo "Строка не пустая и меньше \
чем abc или строка пустая"
```

В операторе [[можно сравнивать не только строки. У него есть операции для проверки файлов и каталогов на различные условия. Эти операции приведены в таблице 3-10.

Таблица 3-10. Операции проверки файлов в операторе [[

Операция	Описание	Пример
-e	Файл существует.	[[-e “$filename”]] && echo “Файл $filename существует”
-f	Указанный объект является обычным файлом (не каталогом и не **файлом устройства**[283]).	[[-f “~/README.txt”]] && echo “README.txt - это обычный файл”
-d	Указанный объект является каталогом.	[[-f “/usr/bin”]] && echo “/usr/bin - это каталог”
-s	Файл не пустой.	[[-s “$filename”]] && echo “Файл $filename не пустой”
-r	Файл существует и доступен для чтения пользователю, запустившему скрипт.	[[-r “$filename”]] && echo “Файл $filename существует и доступен для чтения”
-w	Файл существует и доступен для записи пользователю, запустившему скрипт.	[[-w “$filename”]] && echo “Файл $filename существует и доступен для записи”
-x	Файл существует и доступен для исполнения пользователю, запустившему скрипт.	[[-x “$filename”]] && echo “Файл $filename существует и доступен для исполнения”

[283] https://ru.wikipedia.org/wiki/Специальный_файл_устройства

Таблица 3-10. Операции проверки файлов в операторе [[

Операция	Описание	Пример
-N	Файл существует и был модифицирован с момента последнего чтения.	[[-N “$filename”]] && echo “Файл $filename существует и был модифицирован”
-nt	Файл слева от оператора новее, чем файл справа. Либо файл слева существует, а справа - нет.	[[“$file1” -nt “$file2”]] && echo “Файл $file1 новее чем $file2”
-ot	Файл слева от оператора старее, чем файл справа. Либо файл справа существует, а слева — нет.	[[“$file1” -ot “$file2”]] && echo “Файл $file1 старее чем $file2”
-ef	Слева и справа от оператора указан путь до одного и того же существующего файла. Если ваша система поддерживает **жёсткие ссылки**, то ссылки слева и справа от оператора указывают на один и тот же файл.	[[“$file1” -ef “$file2”]] && echo “Файлы $file1 и $file2 совпадают”

Кроме строк оператор [[может сравнивать целые числа. Соответствующие операции приведены в таблице 3-11.

Таблица 3-11. Операции сравнения целых чисел в операторе [[

Операция	Описание	Пример
-eq	Число слева равно числу справа.	[[“$var” -eq 5]] && echo “Переменная равна 5”
-ne	Не равно.	[[“$var” -ne 5]] && echo “Переменная не равна 5”
-gt	Больше (>).	[[“$var” -gt 5]] && echo “Переменная больше 5”
-ge	Больше или равно.	[[“$var” -ge 5]] && echo “Переменная больше или равна 5”
-lt	Меньше (<).	[[“$var” -lt 5]] && echo “Переменная меньше 5”
-le	Меньше или равно.	[[“$var” -le 5]] && echo “Переменная меньше или равна 5”

Таблица 3-11 вызывает вопросы. Эти операции сложнее запомнить чем привычные знаки сравнения чисел (<, > и =). Почему в операторе [[не используются знаки сравнения? Чтобы ответить на этот вопрос, обратимся к истории оператора [[.

Оператор [[пришёл в Bash на замену устаревшего test. В первой версии Bourne shell 1979 года test был сторонней утилитой. Только начиная с версии System III shell 1981 года, он стал встроенной командой интерпретатора. Но это изменение не затронуло синтаксис test. Дело в том, что к этому времени было написано много кода на старом синтаксисе. Поэтому новая версия интерпретатора вынуждена была его поддерживать.

Рассмотрим синтаксис оператора test. Когда он был сторонней утилитой, формат его входных параметров подчинялся правилам Bourne shell. Например, вот типичный вызов test для сравнения значения переменной `var` и числа пять:

```
test "$var" -eq 5
```

Эта команда не вызывает вопросов. В утилиту test передаются три параметра: значение переменной `var`, опция `-eq` и число 5. Если этот вызов использовать как условие конструкции if, получим следующее:

```
1  if test "$var" -eq 5
2  then
3    echo "Переменная равна 5"
4  fi
```

В Bourne shell для оператора test добавили синоним [. Единственное отличие между ними — это наличие закрывающей скобки]. Для test она не нужна. С помощью синонима перепишем условие конструкции if так:

```
1  if [ "$var" -eq 5 ]
2  then
3    echo "Переменная равна 5"
4  fi
```

Синоним [добавили для лучшей читаемости кода. Благодаря ему, конструкция if в Bourne shell стала больше походить на if в других языках программирования (например, C). Проблема в том, что операторы [и test эквивалентны. Этот факт легко упустить из виду, особенно имея опыт программирования на других языках. Такое несоответствие ожидаемого и реального поведения приводит к ошибкам.

Например, программисты часто забывают пробел между скобкой [и следующим далее символом. То есть получается подобное условие:

```
if ["$var" -eq 5]
then
  echo "Переменная равна 5"
fi
```

Просто замените в условии скобку [на test и ошибки станет очевидна:

```
if test"$var" -eq 5
then
  echo "Переменная равна 5"
fi
```

Между именем команды и её параметрами всегда должен стоять пробел.

Вернёмся к нашему вопросу о знаках сравнения для чисел. Представьте себе следующий вызов test:

```
test "$var" > 5
```

Как вы помните, символ > является сокращением для перенаправления стандартного потока вывода `1>`. Поэтому наш вызов test выполнит следующее:

1. Вызовет встроенную команду test и передаст ей на вход переменную `var`.
2. Перенаправит вывод test в файл с именем `5` в текущем каталоге.

Мы ожидаем совсем другое поведение. Подобную ошибку легко допустить и сложно обнаружить. Чтобы её избежать и были введены двухбуквенные операции для сравнения чисел. Эти операции перекочевали в новый Bash-оператор [[. По идее, ничто не мешало заменить их на знаки сравнения. Но такое решение усложнило бы портирование старого кода с Bourne shell на Bash. Рассмотрим пример.

Представьте, что в вашем старом коде есть следующая конструкция if:

```
if [ "$var1" -gt 5 -o 4 -lt "$var2" ]
then
  echo "Переменная var1 больше 5 или var2 больше 4"
fi
```

Намного безопаснее поставить по дополнительной скобке в начале и в конце выражения, чем менять `-gt` на >, а `-lt` на <. При таких заменах легко допустить ошибку.

В операторе [[знаки сравнения можно использовать только для строк. Почему? Для сравнения строк не было задачи обеспечить обратную совместимость. Первая версия утилиты test вообще не поддерживала лексикографического сравнения строк. То есть знаков сравнения < и > не было. Они появились только в расширении POSIX-стандарта и только для строк. Для чисел добавлять их было уже поздно. Стандарт говорит, что знаки сравнения должны быть экранированы `/<` и `/>`. Из стандарта они попали в оператор [[, но уже без экранирования.

Упражнение 3-5. Оператор [[

Напишите скрипт для сравнения двух каталогов с именами dir1 и dir2.
На экран должны выводиться все файлы, которые есть в одном каталоге,
но отсутствуют в другом.

Оператор case

В программах выполняемые действия часто зависят от каких-то значений. Если значение одно, выбирается первое действие. Если значение другое, то — второе действие. Именно так работают условные операторы. Мы уже познакомились с конструкцией if. Кроме неё в Bash есть конструкция case. В некоторых случаях она удобнее чем if.

Рассмотрим пример. Предположим, что вы пишете скрипт для архивации документов. У скрипта есть три режима работы: архивация со сжатием, архивация без сжатия и разархивация. Нужное действие выбирается с помощью опции скрипта. Один из вариантов опций предлагает таблица 3-12.

Таблица 3-12. Опции скрипта архивации

Опция	Режим работы
-a	Архивация без сжатия
-c	Архивация со сжатием
-x	Разархивация

Выбирая формат опций и параметров скриптов, всегда следуйте POSIX-соглашению[284] и GNU-расширению[285] к нему.

Проверить опцию скрипта можно в конструкции if. Например, как в листинге 3-12.

Листинг 3-12. Скрипт архивации документов

```
#!/bin/bash

operation="$1"

if [[ "$operation" == "-a" ]]
then
  bsdtar -c -f documents.tar ~/Documents
elif [[ "$operation" == "-c" ]]
then
  bsdtar -c -j -f documents.tar.bz2 ~/Documents
```

[284] https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html
[285] https://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html

```
elif [[ "$operation" == "-x" ]]
then
  bsdtar -x -f documents.tar*
else
  echo "Указана недопустимая опция"
  exit 1
fi
```

Опция скрипта передаётся в позиционном параметре `$1`. Он сохраняется в переменной `operation` для удобства. Дальше в зависимости от её значения вызывается утилита bsdtar с теми или иными параметрами. Значение переменной `operation` проверяется в конструкции if. Попробуем заменить её на конструкцию case. Листинг 3-13 демонстрирует результат.

Листинг 3-13. Скрипт архивации документов

```
#!/bin/bash

operation="$1"

case "$operation" in
  "-a")
    bsdtar -c -f documents.tar ~/Documents
    ;;

  "-c")
    bsdtar -c -j -f documents.tar.bz2 ~/Documents
    ;;

  "-x")
    bsdtar -x -f documents.tar*
    ;;

  *)
    echo "Указана недопустимая опция"
    exit 1
    ;;
esac
```

Назовём наш скрипт `archiving-case.sh`. Тогда его можно запустить одним из следующих способов:

```
./archiving-case.sh -a
./archiving-case.sh -c
./archiving-case.sh -x
```

Если передать в скрипт любые другие параметры, он завершится с ошибкой. Скрипт выведет сообщение: “Указана недопустимая опция”.

При обработке ошибок в скриптах никогда не забывайте о команде exit. Код возврата после ошибки должен быть ненулевым.

В общем случае конструкция case сравнивает переданную в неё строку со списком шаблонов. В зависимости от совпадения с шаблоном выполняется один из блоков case.

Каждый блок case состоит из следующих элементов:

1. Шаблон или список шаблонов, разделённых символом |.
2. Правая круглая скобка).
3. Набор команд, которые выполняются при совпадении шаблона и переданной в case строки.
4. Два знака точка с запятой ;;. Они означают окончание набора команд.

Проверка шаблонов происходит последовательно. Сначала проверяется первый, потом — второй и так далее. Если строка совпала с первым шаблоном, будет выполнен соответствующий ему блок. После этого остальные шаблоны и их блоки игнорируются. Bash продолжит исполнение скрипта с команды, следующей за конструкцией case.

Шаблон * без кавычек соответствует любой строке. Обычно он идёт в конце списка. В его блоке обрабатываются случаи, когда ни один из шаблонов не подошёл. Как правило, это означает ошибку.

На первый взгляд может показаться, что конструкции if и case эквивалентны. Это не так. Они лишь позволяют добиться одинакового поведения.

Для удобства запишем конструкции if и case из нашего примера в общем виде. Вариант с if выглядит так:

```
if УСЛОВИЕ_1
then
  ДЕЙСТВИЕ_1
elif УСЛОВИЕ_2
then
  ДЕЙСТВИЕ_2
elif УСЛОВИЕ_3
then
  ДЕЙСТВИЕ_3
else
  ДЕЙСТВИЕ_4
fi
```

Вариант с case выглядит так:

```
case СТРОКА in
  ШАБЛОН_1)
    ДЕЙСТВИЕ_1
     ;;

  ШАБЛОН_2)
    ДЕЙСТВИЕ_2
    ;;

  ШАБЛОН_3)
    ДЕЙСТВИЕ_3
    ;;

  ШАБЛОН_4)
    ДЕЙСТВИЕ_4
    ;;
esac
```

Теперь различия между конструкциями стали очевиднее. Прежде всего, if проверяет результаты логических выражений. Конструкция case проверяет совпадение строки с шаблонами. Это значит, что нет смысла передавать в case логическое выражение. Так вы обработаете только два случая: когда выражение истинно и когда — ложно. Конструкция if намного удобнее для подобной проверки.

Второе различие if и case заключается в количестве условий. В if каждая ветвь конструкции (if, elif и else) проверяет новое логическое выражение. В общем случае эти выражения никак не связаны. В нашем примере они проверяют значения одной и той же переменной, но это частный случай. Конструкция case работает с одной-единственной переданной в неё строкой.

Операторы if и case принципиально отличаются. Они не взаимозаменяемы. В каждом конкретном случае используйте конструкцию в зависимости от характера проверки. Следующие вопросы помогут вам сделать правильный выбор:

- Сколько условий надо проверить?
- Требуются ли составные логические выражения или достаточно сравнения одной строки?

Блоки case можно отделять друг от друга двумя знаками точка с запятой `;;` или точкой с запятой и амперсандом `;&`. Синтаксис с амперсандом допустим в Bash, но не является частью POSIX-стандарта. Он означает выполнение следующего блока case без проверки его шаблона. Это может быть полезно, если требуется начать выполнение алгоритма с определённого шага в зависимости от какого-то условия. Также синтаксис с амперсандом позволяет избежать дублирования кода.

Рассмотрим пример проблемы дублирования кода. Напишем скрипт, который архивирует PDF документы и копирует результат в специальный каталог. Для выбора действия в скрипт передаётся опция. Например, `-a` для архивации и `-c` для копирования. Допустим, что после архивации всегда надо выполнять копирование. В этом случае возникнет дублирование кода.

Листинг 3-14 демонстрирует конструкцию case, в которой команда копирования архива дублируется.

Листинг 3-14. Скрипт архивации и копирования PDF документов

```
#!/bin/bash

operation="$1"

case "$operation" in
  "-a")
    find Documents -name "*.pdf" -type f -print0 | xargs -0 bsdtar -c -j -f document\
s.tar.bz2
    cp documents.tar.bz2 ~/backup
    ;;

  "-c")
    cp documents.tar.bz2 ~/backup
    ;;

  *)
    echo "Указана недопустимая опция"
    exit 1
```

```
    ;;
esac
```

Дублирование кода можно избежать, если поставить разделитель ;& между блоками обработки -a и -c. Исправленный скрипт приведён в листинге 3-15.

Листинг 3-15. Скрипт копирования и архивации PDF документов

```
#!/bin/bash

operation="$1"

case "$operation" in
  "-a")
    find Documents -name "*.pdf" -type f -print0 | xargs -0 bsdtar -c -j -f document\
s.tar.bz2
    ;&

  "-c")
    cp documents.tar.bz2 ~/backup
    ;;

  *)
    echo "Указана недопустимая опция"
    exit 1
    ;;
esac
```

Разделитель ;& может быть полезным. Но используйте его только в случае крайней необходимости. Проблема в том, что визуально его легко спутать с разделителем ;; и неправильно прочитать и понять код.

Альтернатива оператору case

Конструкция case и ассоциативный массив решают сходные задачи. Массив даёт соотношение между данными (ключ-значение). Конструкция case — между данными и командами (значение-действие).

Обычно работать с данными удобнее, чем с кодом. Их проще изменять и проверять на корректность. Поэтому в некоторых случаях конструкцию case стоит заменить на ассоциативный массив. По сравнению с другими языками программирования в Bash легко конвертировать данные в команды.

Рассмотрим пример. Напишем скрипт-обёртку для утилит архивации. В зависимости от переданной в скрипт опции вызывается либо программа bsdtar, либо tar. Листинг 3-16 демонстрирует такой скрипт. В нём опция обрабатывается с помощью конструкции case.

Листинг 3-16. Скрипт-обёртка для утилит bsdtar и tar

```
#!/bin/bash

utility="$1"

case "$utility" in
  "-b"|"--bsdtar")
    bsdtar "${@:2}"
    ;;

  "-t"|"--tar")
    tar "${@:2}"
    ;;

  *)
    echo "Указана недопустимая опция"
    exit 1
    ;;
esac
```

Здесь для первых двух блоков case мы используем список шаблонов. Команда первого блока выполняется при совпадении переменной `utility` со строкой `-b` или `--bsdtar`. Аналогично второй блок выполнится при совпадении переменной с `-t` или `--tar`.

Вот пример запуска скрипта:

```
./tar-wrapper.sh --tar -cvf documents.tar.bz2 Documents
```

В этом случае скрипт вызовет утилиту tar для архивации каталога `Documents`. Чтобы вызвать bsdtar, замените опцию `--tar` на `-b` или на `--bsdtar`. Например:

```
./tar-wrapper.sh -b -cvf documents.tar.bz2 Documents
```

Первый параметр скрипт обрабатывает самостоятельно. Все последующие параметры передаются в утилиту архивации без изменений. Для такой передачи мы используем параметр `$@`. Это не массив. Но он поддерживает синтаксис для подстановки следующих подряд элементов массива. В скрипте мы подставляем в вызов утилиты архивации все элементы `$@` начиная со второго.

Перепишем скрипт-обёртку с помощью ассоциативного массива.

Прежде всего разберёмся в механизмах Bash для конвертации данных в команды. Команду и её параметры надо сохранить в качестве значения какой-то переменной. Чтобы вызвать эту

команду, Bash должен подставить переменную в каком-то месте скрипта. При этом важно, чтобы Bash после подстановки правильно выполнил получившуюся команду.

Мы рассмотрели алгоритм для конвертирования данных в команду. Выполним его по шагам в командном интерпретаторе Bash. Сначала объявим переменную. Для примера её значение соответствует вызову утилиты ls:

```
ls_command="ls"
```

Теперь подставим значение этой переменной. Bash выполнит его как команду:

```
$ls_command
```

В результате выполнится команда ls. Она выведет на экран содержимое текущего каталога. Что произошло? Bash подставил значение переменной `ls_command`. После этого команда стала выглядеть так:

```
ls
```

После подстановки Bash просто исполнил получившуюся команду.

Почему мы не используем двойные кавычки " при подстановке переменной `ls_command`? Чтобы ответить на этот вопрос, сделаем небольшое изменение. Добавим опцию в вызов утилиты ls. Например, объявим переменную `ls_command` так:

```
ls_command="ls -l"
```

В этом случае подстановка с двойными кавычками приведёт к ошибке:

```
$ "$ls_command"
ls -l: command not found
```

Проблема в том, что двойные кавычки предотвращают word splitting. Из-за этого после подстановки получится такая команда:

```
"ls -l"
```

Другими словами Bash должен выполнить команду или утилиту с именем "ls -l", вызванную без параметров. Как вы помните, POSIX-стандарт допускает пробелы в именах файлов. Поэтому "ls -l" является корректным именем исполняемого файла. Мы столкнулись с одним из редких случаев, когда при подстановке переменной двойные кавычки не нужны.

Если двойные кавычки при подстановке всё-таки нужны, эту проблему можно решить. Используйте встроенную команду интерпретатора eval. Она принимает на вход параметры и формирует из них команду для исполнения. При этом для полученной команды выполняется word splitting независимо от двойных кавычек.

Выполним значение переменной `ls_command` с помощью eval:

```
eval "$ls_command"
```

Многие руководства по Bash утверждают, что использовать eval и хранить команды в переменных — плохая практика. Она может привести к серьёзным ошибкам и уязвимостям. В общем случае это справедливо. Будьте осторожны и никогда не передавайте в eval введённые пользователем данные.

Перепишем наш скрипт-обёртку с использованием ассоциативного массива. Листинг 3-17 демонстрирует результат.

Листинг 3-17. Скрипт-обёртка для утилит bsdtar и tar

```
#!/bin/bash

option="$1"

declare -A utils=(
  ["-b"]="bsdtar"
  ["--bsdtar"]="bsdtar"
  ["-t"]="tar"
  ["--tar"]="tar")

if [[ -z "$option" || ! -v utils["$option"] ]]
then
  echo "Указана недопустимая опция"
  exit 1
fi

${utils["$option"]} "${@:2}"
```

Здесь массив `utils` хранит допустимые опции скрипта и соответствующие им команды вызова утилит. С помощью массива можно по опции легко найти команду.

Рассмотрим команду вызова утилиты:

```
${utils["$option"]} "${@:2}"
```

В ней Bash подставляет вызов утилиты из массива `utils`. В качестве ключа элемента выступает опция скрипта `option`. Если указанного ключа нет, произойдёт ошибка. Вместо элемента массива Bash подставит пустую строку. Чтобы это избежать, мы проверяем переданную в скрипт опцию в конструкции if.

В конструкции if вычисляются два логических выражения:

1. Переменная `option` со значением параметра `$1` не пустая.
2. В массиве `utils` есть элемент, соответствующий значению `option`.

Во втором выражении используется опция `-v` оператора [[. Она проверяет, была ли переменная объявлена. Если при объявлении переменной присвоили пустую строку, проверка всё равно пройдёт.

Наш скрипт-обёртка показал, что массив в некоторых случаях даёт более компактный и удобный для чтения код. Всегда рассматривайте возможность заменить конструкцию case на массив в своих программах.

Упражнение 3-6. Оператор case

В домашнем каталоге пользователя есть два конфигурационных файла:
`.bashrc-home` и `.bashrc-work`.
Напишите скрипт для переключения между ними.
Для этого скопируйте один из файлов по пути `~/.bashrc` или создайте символьную ссылку.
Решив задачу с помощью оператора case, замените его на ассоциативный массив.

Арифметические выражения

Интерпретатор Bash может выполнять математические операции над целыми числами. К таким операциям относятся простые арифметические действия: сложение, вычитание, умножение и деление. Кроме них есть битовые и логические операции. Они часто применяются в программировании. Поэтому в этом разделе мы рассмотрим их подробнее.

В Bash для арифметики с плавающей точкой используйте калькулятор bc[286] или dc[287].

Представление целых чисел

Рассмотрим способы представления целых чисел в памяти компьютера. Это поможет лучше понять, как работают математические операции в Bash.

Целые числа могут быть положительными и отрицательными. Соответствующий им тип данных называется **целое**[288] (integer[289]).

Если переменная целого типа принимает только положительные значения, она называется **беззнаковой** (unsigned). Если допустимы как положительные, так и отрицательные значения — это переменная **со знаком** (signed).

Наиболее распространены три способа представления целых в памяти компьютера:

[286] https://ru.wikipedia.org/wiki/Bc
[287] https://ru.wikipedia.org/wiki/Dc
[288] https://ru.wikipedia.org/wiki/Целое_(тип_данных)
[289] https://en.wikipedia.org/wiki/Integer_(computer_science)

- **Прямой код**[290] (signed magnitude representation или SMR).
- **Обратный код**[291] (ones' complement).
- **Дополнительный код**[292] (two's complement).

Прямой код

Все числа в памяти компьютера представляются в двоичном виде. То есть любое число — это последовательность нулей и единиц. Что означают эти нули и единицы, зависит от способа представления числа.

Начнём с самого простого представления чисел — прямого кода. Его можно использовать двумя способами:

1. Для записи только положительных целых (беззнаковых).
2. Для записи как положительных, так и отрицательных целых (со знаком).

Под любое число отводится фиксированный блок памяти. В первом варианте прямого кода все биты этой памяти используются одинаково. В них хранится значение числа. Таблица 3-13 приводит примеры такого способа хранения.

Таблица 3-13. Представление беззнаковых целых в прямом коде

Десятичное число	Шестнадцатеричный формат	Прямой код
0	0	0000 0000
5	5	0000 0101
60	3C	0011 1100
110	6E	0110 1110
255	FF	1111 1111

Предположим, что на число выделен один байт памяти. Тогда в прямом коде можно сохранить целые беззнаковые числа от 0 до 255.

Прямой код можно использовать вторым способом. Он позволяет хранить целые числа со знаком. Для этого старший бит числа резервируется для знака. Поэтому на значение числа остаётся меньше битов. Например, отведём для хранения числа один байт памяти. Один бит уйдёт на знак. Останется только семь битов на значение числа.

Таблица 3-14 демонстрирует представление целых со знаком в прямом коде.

[290] https://ru.wikipedia.org/wiki/Прямой_код
[291] https://ru.wikipedia.org/wiki/Обратный_код
[292] https://ru.wikipedia.org/wiki/Дополнительный_код

Таблица 3-14. Представление целых со знаком в прямом коде

Десятичное число	Шестнадцатеричный формат	Прямой код
-127	FF	1111 1111
-110	EE	1110 1110
-60	BC	1011 1100
-5	85	1000 0101
-0	80	1000 0000
0	0	0000 0000
5	5	0000 0101
60	3C	0011 1100
110	6E	0110 1110
127	7F	0111 1111

Обратите внимание, что старший (первый) бит всех отрицательных чисел равен единице, а положительных — нулю. Из-за знака теперь нельзя сохранить числа больше 127 в одном байте. По этой же причине минимальное отрицательное число равно -127.

Прямой код не получил широкого распространения в компьютерной технике по двум причинам:

1. Арифметические операции над отрицательными числами требуют усложнения архитектуры процессора. Модуль процессора для суммирования положительных чисел не подходит для отрицательных.
2. Существует два представления нуля: положительное (0000 0000) и отрицательное (1000 0000). Это осложняет операцию сравнения, так как в памяти эти значения не равны.

Постарайтесь разобраться в принципе работы прямого кода. Без этого вы не поймёте другие два способа представления целых.

Обратный код

У прямого кода есть два недостатка. Они привели к техническим проблемам при использовании кода в компьютерах. Это заставило инженеров искать альтернативное представление чисел в памяти. Так появился обратный код.

Первая проблема прямого кода связана с операциями над отрицательными числами. Прямой код решает именно её. Разберёмся, почему вообще возникла эта сложность.

Для примера сложим числа 10 и -5. Представим их в прямом коде. Предположим, что на каждое число отводится один байт в памяти компьютера. Тогда получим следующий результат:

```
10 = 0000 1010
-5 = 1000 0101
```

Теперь возникает вопрос — как процессору сложить эти числа? У любого современного процессора есть стандартный модуль под названием **сумматор**[293]. Он побитово складывает два числа. Если применить его для нашей задачи, получим следующее:

```
10 + (-5) = 0000 1010 + 1000 0101 = 1000 1111 = -15
```

Результат неверный. Это означает, что сумматор не подходит для сложения целых в прямом коде. Проблема в том, что при сложении не учитывается старший бит числа.

Проблема решается двумя способами:

1. Добавить в процессор специальный модуль для операций над отрицательными числами.
2. Изменить способ представления отрицательных целых так, чтобы сумматор смог их складывать.

Развитие компьютерной техники пошло по второму пути. Он дешевле, чем усложнение процессора.

Принцип работы обратного кода очень похож на прямой код. Старший бит отводится под знак. Остальные биты хранят значение числа. Отличие в том, что для отрицательных чисел все биты значения инвертируются. То есть нули становятся единицами, а единицы — нулями. Биты значения положительных чисел не инвертируются.

Таблица 3-15 демонстрирует представление чисел в обратном коде.

Таблица 3-15. Представление целых со знаком в обратном коде

Десятичное число	Шестнадцатеричный формат	Обратный код
-127	80	1000 0000
-110	91	1001 0001
-60	C3	1100 0011
-5	FA	1111 1010
-0	FF	1111 1111
0	0	0000 0000
5	5	0000 0101
60	3C	0011 1100
110	6E	0110 1110
127	7F	0111 1111

[293] https://ru.wikipedia.org/wiki/Сумматор

Вместимость памяти при использовании прямого и обратного кодов одинакова. В одном байте по-прежнему можно сохранить числа от -127 до 127.

Что дало инвертирование битов значения для отрицательных чисел? Проверим, как теперь будет работать сложение чисел. Представим 10 и -5 в обратном коде. Затем сложим их с помощью сумматора.

Числа в обратном коде выглядят так:

```
10 = 0000 1010
-5 = 1111 1010
```

Сложение даст следующее:

```
10 + (-5) = 0000 1010 + 1111 1010 = 1 0000 0100
```

Обратите внимание, что в результате сложения произошло переполнение. Старшая единица не поместилась в один байт, отведённый под число. В этом случае она отбрасывается. Тогда результат сложения станет таким:

```
0000 0100
```

Отброшенная единица влияет на конечный результат. Нужен второй этап вычисления, чтобы её учесть. На этом этапе просто добавим единицу к результату:

```
0000 0100 + 0000 0001 = 0000 0101 = 5
```

Мы получили правильный результат сложения чисел 10 и -5.

Если в результате сложения получилось отрицательное число, второй этап вычисления не нужен. Для примера сложим числа -7 и 2. Сначала представим их в обратном коде:

```
-7 = 1111 1000
2 = 0000 0010
```

Выполним первый этап сложения:

```
-7 + 2 = 1111 1000 + 0000 0010 = 1111 1010
```

Старший бит равен единице. Это значит, что мы получили отрицательное число. В этом случае второй этап сложения не нужен.

Проверим корректность результата. Для удобства переведём число из обратного кода в прямой. Чтобы это сделать, инвертируем все биты значения числа. Знаковый бит оставляем без изменений. В результате получим следующее:

```
1111 1010 -> 1000 0101 = -5
```

Мы снова получили верный результат.

Обратный код решил одну проблему. Если представить числа в нём, стандартный сумматор сможет их сложить независимо от знака. Недостаток такого решения в том, что сложение происходит в два этапа. Это замедляет работу компьютера.

У прямого кода есть вторая проблема: представление нуля двумя способами. Её обратный код решить не смог.

Дополнительный код

Дополнительный код решает обе проблемы прямого кода. Во-первых, он позволяет стандартному сумматору складывать отрицательные числа. В обратном коде это действие выполняется в два этапа. В дополнительном коде достаточно одного. Во-вторых, ноль представляется одним единственным способом.

Положительные числа в дополнительном коде выглядят так же, как и в прямом. Старший знаковый бит равен нулю. Остальные биты хранят значение числа. У отрицательных чисел старший бит равен единице. Биты значения инвертируются, как в обратном коде. Затем к результату прибавляется единица.

Представление чисел в дополнительном коде приведено в таблице 3-16.

Таблица 3-16. Представление целых со знаком в дополнительном коде

Десятичное число	Шестнадцатеричный формат	Дополнительный код
-127	81	1000 0001
-110	92	1001 0010
-60	C4	1100 0100
-5	FB	1111 1011
0	0	0000 0000
5	5	0000 0101
60	3C	0011 1100
110	6E	0110 1110
127	7F	0111 1111

Вместимость памяти при использовании дополнительного кода не меняется. По-прежнему в одном байте можно сохранить числа от -127 до 127.

Рассмотрим сложение отрицательных чисел в обратном коде. Для примера сложим 14 и -8. Сначала представим каждое число в дополнительном коде. Получим:

```
14 = 0000 1110
-8 = 1111 1000
```

Теперь выполним сложение:

```
14 + (-8) = 0000 1110 + 1111 1000 = 1 0000 0110
```

В результате сложения произошло переполнение. Старшая единица не поместилась в один байт. Её надо отбросить. Тогда конечный результат будет таким:

```
0000 0110 = 6
```

Если результат сложения отрицательный, то отбрасывать старший бит не нужно. Для примера сложим числа -25 и 10. В дополнительном коде они выглядят так:

```
-25 = 1110 0111
10 = 0000 1010
```

Сложение чисел даст:

```
-25 + 10 = 1110 0111 0000 1010 = 1111 0001
```

Переведём результат из дополнительного кода в обратный, а потом в прямой. Для этого выполним следующие преобразования:

```
1111 0001 - 1 = 1111 0000 -> 1000 1111 = -15
```

При переводе из обратного кода в прямой мы инвертируем все разряды кроме старшего со знаком. В итоге мы получим правильный результат сложения чисел -25 и 10.

Дополнительный код позволил стандартному сумматору складывать отрицательные числа. Результат сложения вычисляется за один этап. Поэтому в отличие от обратного кода нет потери производительности.

Дополнительный код решил проблему представления нуля. Все биты этого числа — нули. Других вариантов нет. Поэтому сравнивать числа стало проще.

Во всех современных компьютерах целые представляются в дополнительном коде.

Упражнение 3-7. Арифметические действия в дополнительном коде

```
Выполните сложение однобайтовых целых в дополнительном коде:

* 79 и -46
* -97 и 96

Выполните сложение двухбайтовых целых в дополнительном коде:

* 12868 и -1219
```

Конвертирование чисел

Мы узнали, как числа представляются в памяти компьютера. Когда это может пригодиться вам на практике?

Современные языки программирования берут на себя конвертирование чисел в правильный формат. Например, вы объявляете целую знаковую переменную в десятичной системе счисления. Вам не надо заботиться о том, в каком виде она хранится в памяти компьютера. Если значение переменной станет отрицательным, она сохранится в дополнительном коде без вашего участия.

В некоторых случаях с переменной надо работать как с набором битов. Тогда объявите её положительным целым. Все операции над ней выполняйте в шестнадцатеричной системе счисления. Главное — не переводите её в десятичную систему. Так вы обойдёте задачу конвертирования чисел.

Проблема возникает, когда необходимо интерпретировать данные с устройства. Такая задача часто возникает в **системном программировании**[294]. К нему относится разработка драйверов устройств, ядер и модулей ОС, системных библиотек и стеков сетевых протоколов.

Рассмотрим пример. Предположим, что вы пишете драйвер для периферийного устройства. Устройство периодически отправляет на CPU данные (например, результаты измерений). Возникает задача интерпретировать их правильно. Представление чисел на устройстве и компьютере может отличаться (например, порядком байтов). В этом случае вам понадобятся знания о представлении чисел в памяти.

Ещё одна задача, с которой сталкивается каждый программист, — это **отладка**[295]. Отладкой программы называется поиск и устранение в ней ошибок. Для примера в арифметическом выражении происходит переполнение. Зная как числа представляются в памяти, вам будет легче обнаружить проблему.

[294] https://ru.wikipedia.org/wiki/Системное_программное_обеспечение
[295] https://ru.wikipedia.org/wiki/Отладка_программы

Оператор ((

Bash выполняет целочисленную арифметику в **математическом контексте** (math context). Его синтаксис напоминает язык C.

Предположим, что результат сложения двух чисел надо сохранить в переменной. Объявим её с целочисленным атрибутом `-i`. Затем сразу присвоим ей значение. Например, так:

```
declare -i var=12+7
```

В результате переменная будет равна числу 19, а не строке "12+7". Если объявить переменную с атрибутом `-i`, присваиваемое ей значение всегда будет вычисляться в математическом контексте. Это и произошло в нашем примере.

Математический контекст можно объявить явно. Это делает встроенная Bash-команда let.

Предположим, что переменная объявлена без атрибута `-i`. Тогда команда let позволит присвоить ей значение арифметического выражения. Например, так:

```
let text=5*7
```

В результате переменная `text` будет равна 35.

Если переменная объявлялась с атрибутом `-i`, команда let не нужна. Например:

```
declare -i var
var=5*7
```

Значением переменной `var` будет 35.

Объявление переменной с атрибутом `-i` неявно создаёт математический контекст. Это может стать источником ошибок. Поэтому старайтесь не использовать атрибут `-i`. Независимо от него, значение переменной хранится в памяти в виде строки. Конвертирование строки в число и обратно происходит каждый раз при присвоении.

Команда let позволяет работать со строковой переменной как с целочисленной. Например, так:

```
let var=12+7
let var="12 + 7"
let "var = 12 + 7"
let 'var = 12 + 7'
```

Результат всех четырёх команд одинаков. Переменной `var` будет присвоено значение 19.

Команда let принимает на вход параметры. Каждый из них должен быть корректным арифметическим выражением. Если в выражении встречаются пробелы, оно будет разделено на части из-за word splitting. В этом случае let вычислит каждую часть выражения по отдельности. Это может привести к ошибке.

Для примера рассмотрим такую команду:

```
let var=12 + 7
```

Здесь в результате word splitting команда let получит на вход три выражения: “var=12”, “+” и “7”. Вычисление второго из них “+” приведёт к ошибке. Плюс означает арифметическую операцию сложения. Она требует двух операндов. Но в нашем случае операндов нет.

Предположим, что все переданные в команду let выражения корректны. Тогда они вычисляются друг за другом. Например:

```
1 let a=1+1 b=5+1
2 let "a = 1 + 1" "b = 5 + 1"
3 let 'a = 1 + 1' 'b = 5 + 1'
```

В результате всех трёх команд переменной `a` будет присвоено значение 2, а переменной `b` — 6.

Чтобы предотвратить word splitting в параметрах команды let, заключайте их в одинарные или двойные кавычки.

У команды let есть синоним — оператор ((. В нём word splitting не выполняется. Поэтому выражения в операторе не требуют кавычек. Всегда используйте оператор ((вместо let. Это поможет избежать ошибки.

Отношения оператора ((и команды let напоминают отношения test и [[. В обоих случаях используйте операторы, а не команды.

Оператор ((имеет две формы. Первая форма называется **арифметической оценкой** (arithmetic evaluation[296]). Это синоним команды let. Арифметическая оценка выглядит так:

```
((var = 12 + 7))
```

Здесь вместо команды let ставятся открывающие скобки ((. В конце добавляются закрывающие скобки)). Эта форма оператора ((возвращает код ноль при успешном выполнении и единицу в случае ошибки. Вычислив выражение, Bash подставит вместо него код возврата.

Вторая форма оператора ((называется **арифметической подстановкой** (arithmetic expansion[297]). Она выглядит так:

[296] https://wiki.bash-hackers.org/syntax/ccmd/arithmetic_eval
[297] https://wiki.bash-hackers.org/syntax/expansion/arith

```
var=$((12 + 7))
```

Здесь перед оператором ((ставится знак доллара $. В этом случае Bash вычислит значение выражения. Затем он подставит это значение вместо выражения. Это отличается от поведения первой формы оператора ((, при которой подставляется код возврата.

Вторая форма оператора ((является частью POSIX-стандарта. Используйте её для переносимого кода. Первая форма оператора ((доступна только в интерпретаторах Bash, ksh и zsh.

В операторе ((имена переменных можно указывать без знака доллар $. Bash всё равно правильно подставит их значения. Например, следующие два выражения для вычисления `result` эквивалентны:

```
1  a=5 b=10
2  result=$(($a + $b))
3  result=$((a + b))
```

В обоих случаях переменная `result` станет равна 15.

Не используйте знак доллара в операторе ((. Это сделает ваш код чище и понятнее.

В Bash есть устаревшая форма арифметической подстановки — оператор “$[ ]”. Никогда не используйте её. Для расчёта арифметических выражений есть GNU-утилита expr. Она нужна для совместимости со старыми скриптами, написанными на Bourne Shell. Никогда не используйте expr при разработке новых скриптов.

Таблица 3-17 демонстрирует операции, допустимые в арифметических выражениях.

Таблица 3-17. Операции в арифметических выражениях

Операция	Описание	Пример
	Вычисления	
`*`	Умножение	`echo "$((2 * 9)) = 18"`
`/`	Деление	`echo "$((25 / 5)) = 5"`
`%`	Остаток от деления	`echo "$((8 % 3)) = 2"`
`+`	Сложение	`echo "$((7 + 3)) = 10"`
`-`	Вычитание	`echo "$((8 - 5)) = 3"`

Таблица 3-17. Операции в арифметических выражениях

Операция	Описание	Пример
`**`	Возведение в степень	`echo "$((4**3)) = 64"`
	Битовые операции	
`~`	Побитовое НЕ (NOT)	`echo "$((~5)) = -6"`
`<<`	Битовый сдвиг влево	`echo "$((5 << 1)) = 10"`
`>>`	Битовый сдвиг вправо	`echo "$((5 >> 1)) = 2"`
`&`	Побитовое И (AND)	`echo "$((5 & 4)) = 4"`
`|`	Побитовое ИЛИ (OR)	`echo "$((5 | 2)) = 7"`
`^`	Побитовое исключающее ИЛИ[298] (XOR)	`echo "$((5 ^ 4)) = 1"`
	Присваивания	
`=`	Обычное присваивание	`echo "$((num = 5)) = 5"`
`*=`	Умножение и присваивание результата	`echo "$((num *= 2)) = 10"`
`/=`	Деление и присваивание результата	`echo "$((num /= 2)) = 5"`
`%=`	Остаток от деления и присваивание результата	`echo "$((num %= 2)) = 1"`
`+=`	Сложение и присваивание результата	`echo "$((num += 7)) = 8"`
`-=`	Вычитание и присваивание результата	`echo "$((num -= 3)) = 5"`
`<<=`	Битовый сдвиг влево и присваивание результата	`echo "$((num <<= 1)) = 10`

[298] https://ru.wikipedia.org/wiki/Исключающее_«или»

Таблица 3-17. Операции в арифметических выражениях

Операция	Описание	Пример
`>>=`	Битовый сдвиг вправо и присваивание результата	`echo "$((num >>= 2)) = 2"`
`&=`	Побитовое И (AND), затем присваивание результата	`echo "$((num &= 3)) = 2"`
`^=`	Побитовое исключающее ИЛИ (XOR), затем присваивание результата	`echo "$((num^=7)) = 5"`
`|=`	Побитовое ИЛИ (OR), затем присваивание результата	`echo "$((num |= 7)) = 7"`
	Сравнения	
`<`	Меньше	`((num < 5)) && echo "переменная num меньше 5"`
`>`	Больше	`((num > 5)) && echo "переменная num больше 3"`
`<=`	Меньше или равно	`((num <= 20)) && echo "переменная num меньше или равна 20"`
`>=`	Больше или равно	`((num >= 15)) && echo "переменная num больше или равна 15"`
`==`	Равно	`((num == 3)) && echo "переменная num равна 3"`
`!=`	Не равно	`((num != 3)) && echo "переменная num не равна 3"`
	Логические операции	
`!`	Логическое НЕ (NOT)	`(( ! num )) && echo "переменная num имеет значение ЛОЖЬ"`

Таблица 3-17. Операции в арифметических выражениях

Операция	Описание	Пример
`&&`	Логическое И (AND)	`(( 3 < num && num < 5 )) && echo "переменная num больше 3, но меньше 5"`
`||`	Логическое ИЛИ (OR)	`(( num < 3 || 5 < num )) && echo "переменная num меньше 3 или больше 5"`
	Другие операции	
`num++`	Постфикс-инкремент	`echo "$((num++))"`
`num--`	Постфикс-декремент	`echo "$((num--))"`
`++num`	Префикс-инкремент	`echo "$((++num))"`
`--num`	Префикс-декремент	`echo "$((--num))"`
`+num`	Унарный плюс или умножение числа на 1	`a=$((+num))"`
`-num`	Унарный минус или умножение числа на -1	`a=$((-num))"`
`УСЛОВИЕ ? ДЕЙСТВИЕ1 : ДЕЙСТВИЕ2`	Тернарная условная операция[299]	`a=$(( b < c ? b : c ))`
`ДЕЙСТВИЕ1, ДЕЙСТВИЕ2`	Список выражений	`((a = 4 + 5, b = 16 - 7))`
`( ДЕЙСТВИЕ1 )`	Группирование выражений (подвыражение)	`a=$(( (4 + 5) * 2 ))`

Все операции выполняются в порядке их приоритета. Операции с большим приоритетом исполняются первыми.

Таблица 3-18 демонстрирует порядок операций.

[299] https://ru.wikipedia.org/wiki/Тернарная_условная_операция

Таблица 3-18. Порядок выполнения математических операций

Порядок выполнения	Операция	Описание
1	`( ДЕЙСТВИЕ1 )`	Группирование выражений
2	`num++, num--`	Постфиксный инкремент и декремент
3	`++num, --num`	Префиксный инкремент и декремент
4	`+num, -num`	Унарный плюс и минус
5	`~, !`	Побитовое и логическое отрицание
6	`**`	Возведение в степень
7	`*, /, %`	Умножение, деление, нахождение остатка
8	`+, -`	Сложение и вычитание
9	`<<, >>`	Битовые сдвиги
10	`<, <=, >, >=`	Сравнения
11	`==, !=`	Равенство и неравенство
12	`&`	Побитовое И
13	`^`	Побитовое исключающее ИЛИ
14	`|`	Побитовое ИЛИ
15	`&&`	Логическое И
16	`||`	Логическое ИЛИ
17	`УСЛОВИЕ ? ДЕЙСТВИЕ1 : ДЕЙСТВИЕ2`	Тернарная условная операция
18	`=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`	Присваивания
19	`ДЕЙСТВИЕ1, ДЕЙСТВИЕ2`	Список выражений

Порядок выполнения можно изменить с помощью круглых скобок “( )”. Их содержимое называется **подвыражением** (subexpression). Bash вычисляет значения подвыражений в

первую очередь. Если подвыражений несколько, они вычисляются по порядку.

Предположим, в вашем коде используется числовая константа. Её значение можно указать в произвольной системе счисления. Для выбора системы счисления используйте префикс. Список допустимых префиксов приведён в таблице 3-19.

Таблица 3-19. Префиксы для указания системы счисления константы

Префикс	Система счисления	Пример
`0`	Восьмеричная	`echo "$((071)) = 57"`
`0x`	Шестнадцатеричная	`echo "$((0xFF)) = 255"`
`0X`	Шестнадцатеричная	`echo "$((0XFF)) = 255"`
`<основание>#`	Система с указанным основанием от 2 до 64	`echo "$((16#FF)) = 255"`
		`echo "$((2#101)) = 5"`

При выводе на экран или в файл Bash всегда переводит значения чисел в десятичную систему. Встроенная команда printf меняет формат вывода чисел. Её можно вызвать, например, так:

```
printf "%x\n" 250
```

Эта команда выведет на экран число 250 в шестнадцатеричной системе.

Аналогично можно вывести и значение переменной:

```
printf "%x\n" $var
```

Арифметические действия

Начнём с самых простых математических операций — арифметических. В языках программирования они обозначаются привычными символами:

- `+` сложение
- `-` вычитание
- `/` деление
- `*` умножение

Кроме них в программировании часто встречаются ещё два действия: возведение в степень и вычисление остатка от деления[300].

Возведение в степень принято записывать в виде a^b. Здесь a является основанием, а b — показателем степени. Например, два в степени семь записывается как 2^7. В Bash это арифметическое действие обозначается двумя звёздочками:

[300] https://ru.wikipedia.org/wiki/Деление_с_остатком

```
2**7
```

Вычисление остатка от деления — это сложная, но важная в программировании операция. Рассмотрим её подробнее. Предположим, что мы разделили одно целое число на другое. В результате получилось дробное число. Тогда говорят, что при делении появился остаток.

Например, разделим 10 (делимое) на 3 (делитель). Если округлить результат, получим 3,33333 (частное). В этом случае остаток от деления равен 1. Чтобы его найти, умножим делитель 3 на целую часть частного 3 (неполное частное). Результат вычтем из делимого 10. Получим остаток 1.

Запишем наши вычисления в виде формул. Для этого введём следующие обозначения:

- a — делимое
- b — делитель
- q — неполное частное
- r — остаток

Тогда делимое вычисляется по формуле:

```
a = b * q + r
```

Отсюда выведем формулу для нахождения остатка:

```
r = a - b * q
```

Выбор неполного частного q вызывает вопросы. Иногда на его роль подходят несколько чисел. Выбрать из них правильное помогает ограничение. Частное q должно быть таким, чтобы остаток от деления r по абсолютной величине оказался меньше делителя b. Другими словами должно выполняться неравенство:

```
|r| < |b|
```

Операция нахождения остатка в Bash обозначается знаком процент %. В некоторых языках этим же символом обозначается операция **modulo**[301]. Это два разных действия[302]. Когда знаки делимого и делителя совпадают, они дают одинаковый результат.

Для примера вычислим остаток и modulo при делении 19 на 12 и -19 на -12. Получим:

[301] https://en.wikipedia.org/wiki/Modulo_operation
[302] https://habr.com/ru/post/421071

```
19 % 12 = 19 - 12 * 1 = 7
19 modulo 12 = 19 - 12 * 1 = 7

-19 % -12 = -19 - (-12) * 1 = -7
-19 modulo -12 = -19 - (-12) * 1 = -7
```

Теперь рассмотрим случаи, когда знаки делимого и делителя различаются:

```
19 % -12 = 19 - (-12) * (-1) = 7
19 modulo -12 = 19 - (-12) * (-2) = -5

-19 % 12 = -19 - 12 * (-1) = -7
-19 modulo 12 = -19 - 12 * (-2) = 5
```

Остаток и modulo различаются.

Для расчёта modulo применяется та же формула, что и для остатка. Отличается только выбор неполного частного q. Для нахождения остатка, частное вычисляется по формуле:

```
q = a / b
```

Результат округляется к меньшему по модулю числу. То есть все знаки после запятой отбрасываются.

Неполное частное для modulo считается по-разному в зависимости от знаков a и b. Если знаки совпадают, формула для частного та же:

```
q = a / b
```

Если знаки разные, формула другая:

```
q = (a / b) + 1
```

В обоих случаях результат округляется к меньшему по модулю числу.

Когда говорят об остатке от деления r, обычно предполагают, что и делимое a и делитель b положительны. Поэтому в справочниках часто встречается такое условие для остатка:

```
0 ≤ r < |b|
```

Однако при делении чисел с разными знаками остаток может быть отрицательным. Запомните простое правило: у остатка r всегда такой же знак, что и у делимого a. Если знаки различаются, значит вы нашли modulo.

Всегда помните о различии остатка от деления и modulo. Одни языки программирования вычисляют остаток в операторе %, другие — modulo. Это приводит к путанице.

Если сомневаетесь в своих вычислениях, проверьте их. Bash в операторе % всегда считает остаток от деления. Предположим, что нужно найти остаток деления 32 на -7. Следующая команда выведет результат:

```
echo $((32 % -7))
```

Остаток от деления равен четырём.

Теперь найдём modulo для этой же пары чисел. Воспользуйтесь онлайн-калькулятором[303]. В поле “Expression” введите 32, в поле “Modulus” — 7. Нажмите кнопку “CALCULATE”. Вы получите два результата:

- “Result” равен 4.
- “Symmetric representation” равно -3.

Второй ответ -3 и есть modulo.

Для чего в программировании используют остаток от деления? Самая распространённая задача — это проверка числа на чётность. С её помощью контролируется целостность переданных данных в компьютерных сетях. Такая проверка называется **бит контроля чётности**[304].

Чтобы проверить число на чётность, достаточно найти остаток его деления на 2. Если остаток равен нулю, значит число чётное. В противном случае — нечётное.

Другая задача, в которой не обойтись без вычисления остатка, — это преобразование единиц времени. Рассмотрим пример. Предположим, что 128 секунд надо перевести в минуты. Для этого подсчитаем целое число минут в 128 секундах. Затем добавим к результату остаток.

Чтобы найти целое число минут, разделим 128 на 60. Получим неполное частное 2. То есть в 128 секундах — 2 минуты. Чтобы найти оставшиеся секунды, вычислим остаток от деления 128 на 60:

```
r = 128 - 60 * 2 = 8
```

Остаток равен 8. Получается, что 128 секунд равны двум минутам и восьми секундам.

Вычисление остатка будет полезно и при работе с циклами. Например, если нужно выполнять действие на каждой N-ой итерации цикла. Предположим, что нас интересует каждая 10 итерация. Тогда надо проверять остаток от деления счётчика цикла на 10. Если остаток равен нулю, значит текущая итерация кратна 10.

Операция modulo широко применяется в **криптографии**[305].

[303] https://planetcalc.com/8326/
[304] https://ru.wikipedia.org/wiki/Бит_чётности
[305] https://en.wikipedia.org/wiki/Modulo_operation#Properties_(identities)

Упражнение 3-8. Modulo и остаток от деления

Вычислите остаток от деления и modulo:

* 1697 % 13
* 1697 modulo 13

* 772 % -45
* 772 modulo -45

* -568 % 12
* -568 modulo 12

* -5437 % -17
* -5437 modulo -17

Битовые операции

Битовые операции[306] — это ещё один тип математических действий. Эти операции активно используется в программировании. Своё название они получили потому, что выполняются над каждым битом числа по отдельности.

Побитовое отрицание

Начнём с самой простой битовой операции — отрицания. В компьютерной литературе она иногда обозначается как НЕ или NOT. В Bash отрицание обозначается знаком тильда ~.

Чтобы выполнить побитовое отрицание, замените значение каждого бита числа на противоположное. То есть каждая единица заменяется на ноль и наоборот.

Например, выполним побитовое отрицание числа 5. Получим:

```
5 = 101
~5 = 010
```

Если ограничиться чистой математикой, побитовое отрицание — это очень простая операция. В программировании же с ней возникают сложности. Прежде всего встаёт вопрос: сколько байтов отводится под число? Предположим, что в нашем примере число 5 хранится в двухбайтовой переменной. Тогда в памяти в двоичном виде оно выглядит так:

```
00000000 00000101
```

После побитового отрицания значение переменной станет таким:

[306] https://ru.wikipedia.org/wiki/Битовая_операция

```
11111111 11111010
```

Как интерпретировать полученный результат? Если переменная объявлена как беззнаковое целое, результатом будет число 65530 в прямом коде. Если же переменная знаковая, её значение хранится в дополнительном коде. В этом случае результатом будет -6.

Команды и операторы Bash представляют целые по-разному. Например, echo всегда выводит числа как знаковые. Команда printf позволяет указать формат вывода: знаковое или беззнаковое целое.

В языке Bash нет типов. Все скалярные переменные хранятся в виде строк. Поэтому интерпретация целых чисел происходит в момент их подстановки в арифметические выражения. В зависимости от контекста они подставляются как знаковые или как беззнаковые.

В Bash под целые числа отводится 64 бита независимо от наличия знака. Таблица 3-20 демонстрирует их максимальные и минимальные допустимые значения.

Таблица 3-20. Максимальные и минимальные целые в Bash

Целое число	Шестнадцатеричная система	Десятичная система
Максимальное положительное знаковое	7FFFFFFFFFFFFFFF	9223372036854775807
Минимальное отрицательное знаковое	8000000000000000	-9223372036854775808
Максимальное беззнаковое	FFFFFFFFFFFFFFFF	18446744073709551615

Следующие примеры демонстрируют интерпретацию целых в командах echo, printf и операторе ((:

```
$ echo $((16#FFFFFFFFFFFFFFFF))
-1

$ printf "%llu\n" $((16#FFFFFFFFFFFFFFFF))
18446744073709551615

$ if ((18446744073709551615 == 16#FFFFFFFFFFFFFFFF)); then echo "ok"; fi
ok

$ if ((-1 == 16#FFFFFFFFFFFFFFFF)); then echo "ok"; fi
ok

$ if ((18446744073709551615 == -1)); then echo "ok"; fi
ok
```

Последний пример со сравнением чисел 18446744073709551615 и -1 показывает, что знаковые и беззнаковые целые хранятся в памяти одинаково. Но в зависимости от контекста они интерпретируются по-разному.

Вернёмся к примеру с побитовым отрицанием числа 5. В Bash его результатом будет 64 битное число 0xFFFFFFFFFFFFFFFA в шестнадцатеричной системе. Число можно вывести как положительное или как отрицательное целое:

```
$ echo $((~5))
-6

$ printf "%llu\n" $((~5))
18446744073709551610
```

Числа 18446744073709551610 и -6 равны с точки зрения Bash. Потому что все их биты в памяти совпадают.

Упражнение 3-9. Побитовое отрицание

Выполните побитовое отрицание следующих беззнаковых двухбайтовых целых:

* 56
* 1018
* 58362

Повторите вычисления для случая, когда эти целые являются знаковыми.

Побитовое И, ИЛИ, исключающее ИЛИ

Операция побитового И также известна как AND. Она напоминает логическое И.

В логическом И результат выражения истинен, когда оба операнда истинны. Для остальных значений операндов результатом будет ложь.

Побитовое И выполняется над двумя числами. Они представляются в двоичном виде. Затем над каждой соответствующей парой битов двух чисел выполняется логическое И.

Запишем подробнее алгоритм для выполнения побитового И:

1. Представить оба операнда в двоичном виде.
2. Если число битов в одном операнде меньше чем в другом, дополнить его слева нулями.
3. Применить логическое И к каждой паре битов, которые стоят на одинаковых позициях. То есть выполнить логическое И с первым битом первого числа и с первым битом второго числа. Затем перейти ко второму биту и т.д.

Рассмотрим пример. Вычислим побитовое И для чисел 5 и 3. В двоичном виде они выглядят так:

```
5 = 101
3 = 11
```

У числа 3 оказалось меньше битов, чем у 5. Поэтому дополним его двоичное представление одним нулём слева. Получим:

```
3 = 011
```

Теперь выполним операцию логического И для каждой пары битов чисел 5 и 3. Для удобства запишем двоичное представление чисел в столбик. Получим следующее:

```
101
011
---
001
```

Переведём результат в десятичную систему:

```
001 = 1
```

Это значит, что результат побитового И для чисел 5 и 3 равен 1.

В Bash операция побитового И обозначается знаком амперсанд &. Выполним наше вычисление и выведем результат на экран:

```
echo $((5 & 3))
```

Операция побитового ИЛИ (OR) выполняется аналогично побитовому И. Только вместо логического И над каждой парой битов чисел выполняется логическое ИЛИ.

Вычислим побитовое ИЛИ для чисел 10 и 6. В двоичном виде они выглядят так:

```
10 = 1010
6 = 110
```

Число 6 надо дополнить нулём до четырёх битов:

```
6 = 0110
```

Теперь выполним логическое ИЛИ над каждой парой битов чисел 10 и 6:

```
1010
0110
----
1110
```

Переведём результат в десятичную систему:

```
1110 = 14
```

В Bash побитовое ИЛИ обозначается знаком |. Выведем результат для нашего примера:

```
echo $((10 | 6))
```

Операция побитового исключающего ИЛИ (XOR) похожа на побитовое ИЛИ. В ней над каждой парой битов операндов выполняется **логическое исключающее ИЛИ**. Исключающее ИЛИ возвращает ложь если операнды не равны. В противном случае результат операции — истина.

Вычислим исключающее ИЛИ для чисел 12 и 5. Переведём числа в двоичный вид:

```
12 = 1100
5 = 101
```

Дополним число 5 до четырёх битов:

```
5 = 0101
```

Выполним побитовое исключающее ИЛИ для каждой пары битов:

```
1100
0101
----
1001
```

Переведём результат в десятичную систему:

```
1001 = 9
```

В Bash исключающее ИЛИ обозначается символом ^. Расчёт нашего примера будет выглядеть так:

```
echo $((12 ^ 5))
```

Упражнение 3-10. Побитовые И, ИЛИ, исключающее ИЛИ

Выполните побитовое И, ИЛИ, исключающее ИЛИ для беззнаковых двухбайтовых целых:

* 1122 и 908
* 49608 и 33036

Битовые сдвиги

Битовым сдвигом называется смена позиций битов числа.

Есть три типа сдвигов:

1. Логический
2. Арифметический
3. Циклический

Самый простой из них — это логический. Рассмотрим сначала его.

Операция битового сдвига принимает два операнда. Первый — это число, над которым выполняется операция. Второй — количество битов, на которое происходит сдвиг.

Чтобы выполнить логический сдвиг, представьте исходное число в двоичном виде. Предположим, что выполняется сдвиг вправо на два бита. Тогда два крайние бита числа справа отбрасываются. Вместо них слева добавляются нули. Аналогично выполняется сдвиг влево. Два бита слева отбрасываются. Два нуля справа добавляются.

Рассмотрим пример. Выполним логический сдвиг беззнакового однобайтового целого 58 вправо на три бита. Сначала представим число в двоичном виде:

```
58 = 0011 1010
```

Теперь отбросим три бита справа:

```
0011 1010 >> 3 = 0011 1
```

Затем дополним результат нулями слева:

```
0011 1 = 0000 0111 = 7
```

Результат сдвига — число 7.

Попробуем сдвинуть число 58 на три бита влево. Получим следующее:

```
0011 1010 << 3 = 1 1010 = 1101 0000 = 208
```

Алгоритм аналогичен сдвигу вправо. Сначала отбрасываем крайние биты слева, а затем добавляем нули справа.

Рассмотрим второй тип сдвига — арифметический. Влево он выполняется точно так же, как и логический сдвиг.

Арифметический сдвиг вправо отличается от логического. Чтобы его выполнить, отбросьте нужное количество битов справа. Затем дополните результат битами слева. Их значение должно совпадать со старшим битом числа. Если он равен единице, добавляем справа единицы. В противном случае добавляем нули. Благодаря этому, после сдвига знак числа не меняется.

Для примера выполним арифметический сдвиг знакового однобайтового целого -105 вправо на два бита.

Сначала представим число в дополнительном коде:

```
-105 = 1001 0111
```

Теперь выполним арифметический сдвиг вправо на два бита. Получим:

```
1001 0111 >> 2 -> 1001 01 -> 1110 0101
```

В нашем случае старший бит равен единице. Поэтому мы дополняем результат слева двумя единицами.

Мы получили отрицательное число в дополнительном коде. Переведём его в прямой:

```
1110 0101 = 1001 1011 = -27
```

Результат сдвига — число -27.

Операции << и >> интерпретатора Bash выполняют арифметические сдвиги. Рассмотренные нами примеры можно вычислить с помощью следующих команд:

```
$ echo $((58 >> 3))
7

$ echo $((58 << 3))
464

$ echo $((-105 >> 2))
-27
```

Результат сдвига 58 влево на три бита отличается от нашего, потому что Bash оперирует восьмибайтовыми целыми.

Циклический сдвиг редко применяется в программировании. Поэтому большинство языков не имеет для него встроенного оператора.

В циклическом сдвиге отброшенные биты появляются на освободившемся месте с другого конца числа.

Например, выполним циклический сдвиг числа 58 вправо на три бита. Результат будет следующим:

```
0011 1010 >> 3 = 010 0011 1 = 0100 0111 = 71
```

Отброшенные справа биты 010 оказались в левой части результата.

Упражнение 3-11. Битовые сдвиги

Выполните арифметические битовые сдвиги знаковых двухбайтовых целых:

* 25649 >> 3
* 25649 << 2
* -9154 >> 4
* -9154 << 3

Применение битовых операций

Битовые операции широко применяются в системном программировании. Часто при работе с компьютерной сетью и периферийными устройствами приходится переводить данные из одного формата в другой.

Рассмотрим пример. Предположим, вы работаете с периферийным устройством. На устройстве порядок байтов от старшего к младшему (big-endian). Ваш компьютер использует другой порядок — от младшего к старшему (little-endian).

Устройство посылает на компьютер целое беззнаковое число. В шестнадцатеричной системе оно равно 0xAABB. Чтобы компьютер правильно прочитал это число, надо изменить порядок байтов в нём. После преобразования число 0xAABB станет равно 0xBBAA.

Для изменения порядка байтов сделаем следующее:

1. Прочитаем младший байт числа (крайний справа) и сдвинем его влево на восемь битов, т.е. на один байт. Это делает следующая Bash-команда:

```
little=$(((0xAABB & 0x00FF) << 8))
```

2. Прочитаем старший байт числа (крайний слева) и сдвинем его вправо на восемь битов. Команда:

```
big=$(((0xAABB & 0xFF00) >> 8))
```

3. Соединим старший и младший байты с помощью побитового ИЛИ:

```
result=$((little | big))
```

В результате в переменную `result` запишется число 0xBBAA.

Все шаги нашего вычисления можно выполнить одной командой:

```
value=0xAABB
result=$(( ((value & 0x00FF) << 8) | ((value & 0xFF00) >> 8) ))
```

Другой пример применения битовых операций. Они незаменимы для вычисления масок. Нам уже знакомы маски с правами доступа к файлам в Unix-окружении. Предположим, что файл имеет права `-rw-r--r--`. В двоичном виде эта маска выглядит так:

```
0000 0110 0100 0100
```

Проверим, имеет ли владелец файла право на его исполнение. Для этого вычислим побитовое И с маской 0000 0001 0000 0000. Получим:

```
0000 0110 0100 0100 & 0000 0001 0000 0000 = 0000 0000 0000 0000 = 0
```

Результат равен нулю. Это значит, что владелец не может исполнять файл.

Для добавления битов в маску применяется побитовое ИЛИ. Добавим владельцу файла право на исполнение. Вычисление выглядит так:

```
0000 0110 0100 0100 | 0000 0001 0000 0000 = 0000 0111 0100 0100 = -rwxr--r--
```

Нумерация битов в числе начинается с нуля. Обычно она идёт справа налево.

Мы выполнили побитовое ИЛИ маски с числом 0000 0001 0000 0000. В нём восьмой бит равен единице. С его помощью мы меняем восьмой бит маски. При этом значение бита маски неважно. Он будет выставлен в единицу не зависимо от текущего значения. Все биты числа кроме восьмого равны нулям. Благодаря этому, биты маски в тех же позиция не изменятся.

Для удаления битов из маски применяется побитовое И. Например, удалим право владельца ƒайла на запись. Получим следующее:

```
0000 0111 0100 0100 & 1111 1101 1111 1111 = 0000 0101 0100 0100 = -r-xr--r--
```

Чтобы выставить девятый бит маски в ноль, мы выполнили побитовое И с числом 1111 1101 1111 1111. В нём девятый бит равен нулю, а все остальные — единицам. Поэтому в результате побитового И изменится только девятый бит маски. Все остальные сохранят свои значения.

Операционная система выполняет операции с масками каждый раз, когда вы обращаетесь к ƒайлу. Так она проверяет ваши права на доступ.

Рассмотрим последний пример использования битовых операций. До недавнего времени битовые сдвиги широко применялись как альтернатива умножения и деления на степень двойки[307]. Например, сдвиг влево на два бита соответствует умножению на 2^2 (т.е. четыре). Проверим это утверждение такой Bash-командой:

```
1 $ echo $((3 << 2))
2 12
```

Результат правильный. Умножение 3 на 4 даст 12.

Для примера сдвиг вправо на три бита соответствует делению на 2^3 (т.е. восемь). Проверим:

```
1 $ echo $((16 >> 3))
2 2
```

Подобные трюки сокращают число тактов процессора на выполнение операций умножения и деления. Сейчас эти оптимизации стали ненужны из-за развития компиляторов и процессоров. Компиляторы при генерации кода автоматически выбирают самые “дешевые” по числу тактов ассемблерные инструкции. Процессоры выполняют эти инструкции параллельно в несколько потоков[308]. Поэтому сегодня программисты склонны писать более удобный

[307] https://habr.com/ru/company/pvs-studio/blog/141880
[308] https://ru.wikipedia.org/wiki/Hyper-threading

для чтения и понимания код, а не более оптимальный. Операции умножения и деления с этой точки зрения лучше, чем битовые сдвиги.

Битовые операции также активно применяются в криптографии и компьютерной графике.

Логические операции

Для сравнения целых чисел в конструкции if оператор [[неудобен. В нём отношения между числами обозначают двухбуквенные сокращения. Например, `-gt` для отношения больше. Удобнее использовать оператор ((в форме арифметической оценки. Тогда сокращения заменяются на привычные символы сравнения чисел (>, <, =).

Рассмотрим пример. Предположим, что значение переменной надо сравнить с константой 5. Это сделает следующая конструкция if:

```
if ((var < 5))
then
  echo "Значение var меньше 5"
fi
```

Оператор ((можно заменить на команду let. В результате получим то же поведение:

```
if let "var < 5"
then
  echo "Значение var меньше 5"
fi
```

Однако оператор ((использовать всегда предпочтительнее.

Обратите внимание на важное отличие арифметической оценки и подстановки. Согласно POSIX-стандарту, любая программа или команда при успешном выполнении возвращает ноль. При ошибке возвращается код возврата от 1 до 255. Этот код интерпретируется так: ноль означает истину, а не ноль — ложь. В этом смысле результат арифметической подстановки инвертирован, а оценки нет.

Арифметическая оценка — это синоним команды let. Значит она подчиняется требованиям POSIX-стандарта, как и любая другая команда. Арифметическая подстановка выполняется в контексте другой команды. Поэтому результат её работы зависит от реализации интерпретатора. В Bash если условие в операторе ((в форме подстановки истинно, будет возвращена единица. В противном случае оператор возвращает ноль. Такое поведение соответствует правилам вывода логических выражений языка C.

Рассмотрим пример. Предположим, есть команда для вывода результата сравнения переменной с числом. Она выглядит так:

```
((var < 5)) && echo "Значение var меньше 5"
```

Здесь используется арифметическая оценка. Поэтому если значение переменной меньше 5, оператор ((выполнится успешно. Тогда, согласно стандарту POSIX, он вернёт код ноль.

Если использовать оператором ((в форме арифметической подстановки, результат будет отличаться. Например:

```
echo "$((var < 5))"
```

Если условие истинно, команда echo выведет число 1. Такой результат согласуется с правилами вывода языка C.

Логические операции обычно применяют в форме арифметической оценки оператора ((. Они работают так же, как логические операторы Bash.

Для примера сравним значение переменной с двумя константами:

```
1 if ((1 < var && var < 5))
2 then
3   echo "Значение var меньше 5, но больше 1"
4 fi
```

В этом случае условие истинно, когда выполняются оба неравенства.

Аналогично работает логическое ИЛИ:

```
1 if ((var < 1 || 5 < var))
2 then
3   echo "Значение var меньше 1 или больше 5"
4 fi
```

Выражение истинно, если хотя бы одно из неравенств выполняется.

Логическое НЕ редко применяется к самим числам. Чаще его используют для отрицания выражения. Если применить НЕ к числу, вывод результата соответствует POSIX-стандарту. Другими словами ноль означает истинна, а не ноль — ложь. Например:

```
1 if ((! var))
2 then
3   echo "Значение var равно истина или ноль"
4 fi
```

Это условие выполнится только, если переменная равна нулю.

Инкремент и декремент

Операции инкремента и декремента впервые появились в языке программирования B[309]. Кен Томпсон и Денис Ритчи разработали его в 1969 году, работая в Bell Labs. Позднее Денис Ритчи перенёс эти операции в свой новый язык C. Оттуда их скопировали в Bash.

Начнём с операций присваивания. Тогда смысл инкремента и декремента станет понятнее.

Обычное присваивание в арифметической оценке выглядит так:

```
((var = 5))
```

В результате значение переменной `var` станет равно 5.

Bash позволяет объединить присваивание с арифметическим действием или битовой операцией. Например, одновременное сложение и присваивание выглядит так:

```
((var += 5))
```

Здесь выполняются два действия:

1. К текущему значению переменной `var` прибавляется число 5.
2. Результат сложения записывается в переменную `var`.

Аналогично работают остальные операции присваивания. Сначала выполняется действие, затем результат записывается в переменную. Такой синтаксис позволяет сократить код и сделать его нагляднее.

Теперь рассмотрим операции инкремента и декремент. У них есть две формы: постфиксная и префиксная. Они записываются по-разному. В постфиксной форме знаки ++ и – идут после имени переменной, а в префиксной — до.

Рассмотрим префиксный инкремент:

```
((++var))
```

Результат этой команды такой же, как у следующей операции присваивания:

```
((var+=1))
```

[309] https://ru.wikipedia.org/wiki/Би_(язык_программирования)

Инкремент увеличивает значение переменной на единицу. Декремент — уменьшает на единицу.

Зачем вводить отдельные операции для прибавления и вычитания единицы? Ведь есть достаточно компактные операции сложения и вычитания, совмещённые с присваиванием (+= и -=).

Скорее всего дело в **счётчике цикла**[310]. Он часто используется в программировании. Счётчик отсчитывать номер итерации цикла. Это нужно, чтобы вовремя прервать его выполнение. Инкремент и декремент упрощают работу со счётчиком. Кроме того современные процессоры выполняют эти операции на аппаратном уровне. Поэтому они работают быстрее, чем сложение и вычитание с присваиванием.

Чем отличаются префиксный и постфиксный инкременты? Если выражение состоит только из операции инкремента, то результат будет одинаковым для обеих форм.

Например, следующие команды увеличат значение переменной на единицу:

```
((++var))
((var++))
```

Разница между формами инкремента появляется, при присваивании результата переменной.

Рассмотрим следующий пример:

```
var=1
((result = ++var))
```

В результате значения обеих переменных `result` и `var` станут двум. Это означает, что префиксный инкремент сначала прибавляет единицу, а потом возвращает результат сложения.

Если расписать префиксный инкремент по отдельным командам, получится следующее:

```
var=1
((var = var + 1))
((result = var))
```

Поведение постфиксного инкремента отличается. Заменим форму инкремента в нашем примере:

```
var=1
((result = var++))
```

После выполнения команд в переменную `result` запишется единица, а в `var` — двойка. Постфиксный инкремент сначала возвращает значение, а потом прибавляет единицу.

Распишем постфиксный инкремент по отдельным командам:

[310] https://ru.wikipedia.org/wiki/Счётчик_цикла

```
var=1
((tmp = var))
((var = var + 1))
((result = tmp))
```

Обратите внимание на порядок выполнения постфиксного инкремента. Сначала `var` увеличивается на единицу. Только после этого её прошлое значение возвращается в качестве результата. Поэтому прошлое значение `var` пришлось сохранить во временную переменную `tmp`.

Постфиксная и префиксная формы декремента работают аналогично инкременту.

Всегда используйте префиксную форму инкремента и декремента вместо постфиксной. Во-первых, она быстрее выполняется процессором. Потому что не надо сохранять текущее значение переменной. Во-вторых, с постфиксной формой легче допустить ошибку из-за неочевидного порядка присваивания.

Тернарная условная операция

Тернарная условная операция также известна как тернарный оператор. Она впервые появилась в языке Алгол[311]. Операция оказалась удобной и востребованной программистами. Поэтому её добавили в языки следующего поколения: BCPL[312] и C. Дальше её переняли почти все современные языки: C++, C#, Java, Python, PHP и т.д.

Тернарный оператор представляет собой компактную форму конструкции if.

Для примера рассмотрим такой оператор if:

```
if ((var < 10))
then
  ((result = 0))
else
  ((result = var))
fi
```

Здесь переменной `result` присваивается ноль, если `var` меньше 10. В противном случае `result` присваивается значение `var`.

Такое же поведение даст тернарный оператор. Он выглядит так:

```
((result = var < 10 ? 0 : var))
```

Одна строка заменила шесть строк конструкции if.

Тернарный оператор состоит из условного выражения и двух действий. В общем случае он выглядит так:

[311] https://ru.wikipedia.org/wiki/Алгол
[312] https://ru.wikipedia.org/wiki/BCPL

```
(( УСЛОВИЕ ? ДЕЙСТВИЕ 1 : ДЕЙСТВИЕ 2 ))
```

Если УСЛОВИЕ истинно, выполняется ДЕЙСТВИЕ 1. Иначе — ДЕЙСТВИЕ 2. Такое поведение полностью совпадает с условным оператором if. Запишем его тоже в общем виде:

```
if УСЛОВИЕ
then
  ДЕЙСТВИЕ 1
else
  ДЕЙСТВИЕ 2
fi
```

Сравните тернарный оператор и конструкцию if.

К сожалению, Bash допускает тернарный оператор только в арифметической оценке и подстановке. Это означает, что в качестве условия и действий можно указать только арифметические выражения. Вызов команд Bash или внешних утилит из тернарного оператора невозможен. Такого ограничения нет в других языках программирования.

Используйте тернарный оператор как можно чаще. Это считается хорошей практикой. С ним код станет компактнее и удобнее для чтения. Также считается, что в меньшем объёме кода меньше места для возможной ошибки.

Операторы цикла

Условные операторы управляют **порядком выполнения**[313] программы. Порядок выполнения — это последовательность исполнения операторов, команд и инструкций программы.

Условный оператор выбирает ветвь исполнения в зависимости от логического выражения. Иногда этого недостаточно. Нужны дополнительные средства для управления порядком выполнения. **Операторы цикла**[314] решают задачи, с которыми не справляются условные операторы.

Оператор цикла многократно повторяет один и тот же блок команд. Однократное выполнение этого блока называется **итерацией цикла**. На каждой итерации проверяется условие цикла. В зависимости от результата, цикл продолжается или прекращается.

Повторение команд

Зачем в программе повторять один и тот же блок команд? Чтобы ответить на этот вопрос, рассмотрим несколько примеров.

[313] https://ru.wikipedia.org/wiki/Порядок_выполнения
[314] https://ru.wikipedia.org/wiki/Цикл_(программирование)

Утилита find нам уже знакома. Она ищет файлы и каталоги на жёстком диске. Если в вызов утилиты добавить опцию `-exec`, можно указать действие. Оно выполнится над каждым найденным объектом.

Например, следующая команда удалит все PDF документы пользователя в каталоге `~/Documents`:

```
find ~/Documents -name "*.pdf" -exec rm {} \;
```

В этом случае find несколько раз вызовет утилиту rm. На каждом вызове ей передаётся очередной результат поиска find. Получается, что утилита find выполняет оператор цикла. Цикл завершится после обработки всех найденных файлов.

Утилита du — это ещё один пример повторения действий. Утилита оценивает объём использованного дискового пространства на дисках. У du есть необязательный параметр. Это путь, с которого начинается оценка.

Вот пример вызова утилиты:

```
du ~/Documents
```

Для выполнения этой команды утилита рекурсивно обойдёт все подкаталоги `~/Documents`. Размер каждого найденного файла добавится к конечному результату. Это означает, что инкремент результата оценки повторяется снова и снова.

Утилита du выполняет цикл для прохода по всем файлам в каждом найденном ею каталоге. Размер каждого файла читается и прибавляется к конечному результату.

Повторение операций часто встречается в математических расчётах. Каноничный пример — это вычисление **факториала**[315]. Факториалом числа N называется произведение последовательных натуральных чисел от 1 до N включительно.

Например, факториал числа 4 вычисляется так:

```
4! = 1 * 2 * 3 * 4 = 24
```

Факториал легко вычислить с помощью оператора цикла. Для этого цикл должен последовательно перебрать целые числа от 1 до N. Каждое число умножается на конечный результат. В этом случае повторяется операция умножения.

В качестве последнего примера повторения действий рассмотрим события в компьютерной системе.

Представьте, что вы пишете программу. Она загружает на компьютер файлы из интернета. Для начала программа устанавливает соединение с сервером. Если сервер не отвечает, у программы есть два варианта действий. Первый — завершить выполнение с ненулевым

[315] https://ru.wikipedia.org/wiki/Факториал

кодом возврата. Второй — ожидать ответа. Второй вариант предпочтительнее. Есть много причин, по которым ответ от сервера задерживается. Например, перегружена сеть или сам сервер. Двух-трёхсекундного ожидания будет достаточно, чтобы получить ответ. Тогда наша программа продолжит работу.

Возникает вопрос: как в программе ожидать наступление события? Самый простой способ — использовать оператор цикла. Условием выхода из него будет наступление ожидаемого события. В нашем примере цикл завершится при получении ответа от сервера. Цикл продолжается пока событие не наступило. При этом его блок команд пустой. Такая техника называется **активным ожиданием** событий или **busy-waiting**[316].

Вместо пустого блока команд в цикле ожидания можно останавливать программу на короткое время. Тогда ОС сможет работать над другой задачей, пока ваша программа остановлена.

Мы рассмотрели несколько примеров, когда программа повторяет одни и те же действия. Запишем задачи, решаемые в каждом примере:

1. Однообразная обработка нескольких сущностей. Например, результатов поиска утилиты find.
2. Накопление конечного результата из промежуточных данных. Например, сбор статистики утилитой du.
3. Математические расчёты. Например, вычисление факториала.
4. Ожидание наступления какого-либо события. Например, получение ответа от сервера по сети.

Список далеко не полный. Это наиболее часто встречающиеся в программировании задачи, для решения которых требуются оператор цикла.

Оператор while

В Bash есть два оператора цикла: while и for. Сначала познакомимся с оператором while. Он проще чем for.

Синтаксис while напоминает условный оператор if. В общем виде он выглядит так:

```
while УСЛОВИЕ
do
  ДЕЙСТВИЕ
done
```

Оператор можно записать в одну строку:

[316] https://en.wikipedia.org/wiki/Busy_waiting

```
while УСЛОВИЕ; do ДЕЙСТВИЕ; done
```

В конструкции while УСЛОВИЕМ и ДЕЙСТВИЕМ может быть одна команда или блок команд. Точно так же как в операторе if. ДЕЙСТВИЕ называется **телом цикла** (loop body).

Выполнение while начинается с проверки УСЛОВИЯ. Если команда УСЛОВИЯ вернула нулевой код, оно считается истинным. В этом случае выполняется ДЕЙСТВИЕ. Дальше опять проверяется УСЛОВИЕ. Если оно по-прежнему истинно, снова выполняется ДЕЙСТВИЕ. Цикл прервётся тогда, когда УСЛОВИЕ станет ложным.

Используйте цикл while, когда количество итераций заранее неизвестно. Например, при активном ожидании какого-то события.

Для примера напишем скрипт. Он проверит доступность сервера в интернете. Для такой проверки отправим серверу запрос. Как только сервер пришлёт ответ, наш скрипт выведет сообщение и завершится.

Чтобы отправить серверу запрос, вызовем утилиту ping[317]. Утилита использует ICMP[318] **протокол**. Протокол — это соглашение о формате сообщений между компьютерами в сети. ICMP протокол описывает формат сообщений для обслуживания сети. Они нужны, например, чтобы проверить доступность какого-то компьютера.

В качестве входного параметра утилита ping принимает URL[319] или IP-адрес[320] целевого **хоста**. Хостом называется любой подключённый к сети компьютер или устройство.

Команда для вызова утилиты ping выглядит так:

```
ping google.com
```

В качестве целевого хоста мы указали сервер Google. Утилита будет отправлять ему ICMP-сообщения. Сервер будет отвечать на каждое из них. Вывод утилиты выглядит так:

```
1 PING google.com (172.217.21.238) 56(84) bytes of data.
2 64 bytes from fra16s13-in-f14.1e100.net (172.217.21.238): icmp_seq=1 ttl=51 time=17.\
3 8 ms
4 64 bytes from fra16s13-in-f14.1e100.net (172.217.21.238): icmp_seq=2 ttl=51 time=18.\
5 5 ms
```

Это информация о каждом отправленном ICMP-сообщении и ответе на него. Сейчас утилита работает в бесконечном цикле. Чтобы её остановить, нажмите комбинацию клавиш Ctrl+C.

Чтобы проверить доступность сервера, достаточно отправить ему одно ICMP-сообщение. Укажем это с помощью опции `-c` утилиты ping. Команда станет выглядеть так:

[317] https://ru.wikipedia.org/wiki/Ping
[318] https://ru.wikipedia.org/wiki/ICMP
[319] https://ru.wikipedia.org/wiki/URL
[320] https://ru.wikipedia.org/wiki/IP-адрес

```
ping -c 1 google.com
```

Если сервер `google.com` доступен, утилита вернёт код ноль. В противном случае код будет ненулевым.

Утилита ping ожидает ответ от сервера, пока её не прервёт пользователь. С помощью опции `-W` ограничим время ожидания одной секундой. Получится такая команда:

```
ping -c 1 -W 1 google.com
```

У нас готово условие для конструкции while. Запишем конструкцию целиком:

```
while ! ping -c 1 -W 1 google.com &> /dev/null
do
  sleep 1
done
```

Нас не интересует вывод утилиты ping. Поэтому перенаправим его в файл `/dev/null`.

В условии цикла результат вызова ping инвертирован. Поэтому тело цикла выполняется до тех пор, пока утилита возвращает отличный от нуля код. Другими словами цикл выполняется, пока сервер недоступен.

В теле цикла вызывается утилита sleep.
Она останавливает выполнение скрипта на указанное количество секунд. В нашем примере остановка длится одну секунду.

Для параметра утилиты sleep можно указать суффикс. Секундам соответствует суффикс s (например, 5s), минутам — m, часам — h и дням — d.

Листинг 3-18 демонстрирует полный скрипт для проверки доступности сервера.

Листинг 3-18. Скрипт для проверки доступности сервера

```
#!/bin/bash

while ! ping -c 1 -W 1 google.com &> /dev/null
do
  sleep 1
done

echo "Сервер google.com доступен"
```

У конструкции while есть альтернативная форма until. В ней ДЕЙСТВИЕ выполняется до тех пор, пока УСЛОВИЕ ложно. То есть цикл выполняется, пока УСЛОВИЕ возвращает отличный от нуля код. С помощью формы until можно инвертировать условие while.

В общем виде конструкция until выглядит так:

```
until УСЛОВИЕ
do
  ДЕЙСТВИЕ
done
```

Запись unitl в одну строку похожа на while:

```
until УСЛОВИЕ; do ДЕЙСТВИЕ; done
```

Заменим конструкцию while на until в листинге 3-18. Для этого удалим отрицание результата утилиты ping. Получится скрипт, приведённый в листинге 3-19.

Листинг 3-19. Скрипт для проверки доступности сервера

```
#!/bin/bash

until ping -c 1 -W 1 google.com &> /dev/null
do
  sleep 1
done

echo "Сервер google.com доступен"
```

Поведение скриптов в листингах 3-18 и 3-19 полностью совпадает.

Выбирайте форму while или until в зависимости от условия цикла. Старайтесь составлять условия без отрицаний. Отрицания усложняют чтение кода.

Бесконечный цикл

Конструкция while часто применяется в **бесконечных циклах**[321]. Такие циклы выполняются всё время, пока работает программа.

Бесконечные циклы встречаются в системном ПО, которое работает до отключения питания компьютера. Например, в ОС или прошивках микроконтроллеров. Такие циклы также применяются в компьютерных играх и программах-мониторах для сбора статистики.

Цикл while станет бесконечным, если его условие всегда истинно. Самый простой способ задать такое условие — вызвать встроенную команду интерпретатора true. Например, так:

[321] https://ru.wikipedia.org/wiki/Бесконечный_цикл

```
while true
do
  sleep 1
done
```

Команда true всегда возвращает истину. То есть её код возврата ноль. У true есть симметричная команда false. Она всегда возвращает единицу, то есть ложь.

В большинстве языков программирования true и false являются **литералами**[322]. Литералы — это зарезервированными слова. В случае true и false они обозначают значения истина и ложь.

Команду true в условии while можно заменить на двоеточие. Тогда получим следующее:

```
while :
do
  sleep 1
done
```

Команда двоеточие — это синонимом true. Она нужна для совместимости[323] с Bourne shell. В нём команды true и false отсутствуют. В POSIX-стандарт включены все три команды: двоеточие, true и false.

Рассмотрим пример бесконечного цикла. Напишем скрипт для вывода статистики об использовании дискового пространства. Для этого воспользуемся утилитой df. При вызове без параметров она выведет следующее:

```
$ df
Filesystem     1K-blocks      Used Available Use% Mounted on
C:/msys64       41940988  24666880  17274108  59% /
Z:             195059116 110151748  84907368  57% /z
```

Занятое (Used) и свободное (Available) дисковое пространство указано в байтах. Добавим в вызов утилиты опцию `-h`. Тогда вместо байтов получим килобайты, мегабайты, гигабайты и терабайты. Также добавим опцию `-T`. Она покажет тип файловой системы для каждого диска. Вывод утилиты станет таким:

[322] https://ru.wikipedia.org/wiki/Литерал_(информатика)
[323] https://stackoverflow.com/questions/3224878/what-is-the-purpose-of-the-colon-gnu-bash-builtin

```
$ df -hT
Filesystem     Type  Size  Used Avail Use% Mounted on
C:/msys64      ntfs   40G   24G   17G  59% /
Z:             hgfs  187G  106G   81G  57% /z
```

Чтобы вывести информацию обо всех точках монтирования, добавьте опцию `-a`.

Напишем бесконечный цикл, в теле которого вызывается утилита df. Получится простейший скрипт для наблюдения за файловой системой. Скрипт приведён в листинге 3-20.

Листинг 3-20. Скрипт для наблюдения за файловой системой

```
#!/bin/bash

while :
do
  clear
  df -hT
  sleep 2
done
```

В начале цикла вызывается утилита clear. Она очищает окно терминала от текста. Благодаря этому, в окне останется вывод нашего скрипта без лишней информации.

При работе с Bash часто возникает задача циклического выполнения команды. Для этого есть специальная утилита watch. Она входит в состав пакета procps. Чтобы установить этот пакет в окружение MSYS2, выполните следующую команду:

```
pacman -S procps
```

Теперь скрипт из листинга 3-20 можно заменить одной командой:

```
watch -n 2 "df -hT"
```

Опция `-n` утилиты watch задаёт интервал между вызовами команды. Команда для исполнения указывается после всех опций.

Опция watch `-d` подсвечивает разницу в выводе команды, выполненной на текущей итерации и на прошлой. Благодаря этому, легче отследить произошедшие изменения.

Чтение стандартного потока ввода

Цикл while хорошо подходит для обработки потока ввода. Рассмотрим пример такой задачи. Напишем скрипт, который прочитает ассоциативный массив из текстового файла.

Листинг 3-10 демонстрирует скрипт для работы с контактами. Они хранятся в коде скрипта. Из-за этого контакты неудобно редактировать. Пользователь должен знать синтаксис Bash. Иначе он допустит ошибку при инициализации элемента массива, и скрипт перестанет работать.

Проблему редактирования можно решить. Поместим контакты в отдельный текстовый файл. При старте скрипт будет загружать из него все контакты. Так мы разделим данные и код.

Листинг 3-21 демонстрирует один из вариантов формата файла с контактами.

Листинг 3-21. Файл с контактами `contacts.txt`

```
Alice=alice@gmail.com
Bob=(697) 955-5984
Eve=(245) 317-0117
Mallory=mallory@hotmail.com
```

Напишем скрипт для чтения этого файла. Удобнее читать контакты сразу в ассоциативный массив. Тогда механизм поиска контакта по имени останется таким же эффективным, как и раньше.

Для чтения файла нам понадобится цикл. Перед его выполнением мы не знаем, сколько итераций понадобиться. Значит, нам нужен цикл while.

Почему число итераций цикла неизвестно? Файл контактов надо читать построчно. Каждая его строка хранит одну запись. Скрипт читает запись, добавляет её в ассоциативный массив и переходит к следующей строке файла. Получается, что число итераций цикла равно числу строк. Но размер файла нам неизвестен, пока мы не прочитаем его полностью. Поэтому число итераций также неизвестно.

Для чтения строк файла применим встроенную команду интерпретатора read. Она читает строку из стандартного потока ввода. Затем сохраняет строку в переменную. Имя переменной передаётся в команду как параметр. Например:

```
read var
```

После запуска этой команды пользователь должен ввести строку и нажать Enter. Она сохранится в переменной `var`. Если вызвать read без параметров, введённая строка сохранится в зарезервированной переменной `REPLY`.

Команда read читает введённую пользователем строку. При этом read удаляет из строки символы обратного слэша . Они экранируют специальные символы. Поэтому read считает слэши ненужными. Чтобы отключить эту функцию, используйте опцию `-r`. В противном случае некоторые символы из ввода могут потеряться.

Команде read можно передать на вход несколько имён переменных. В этом случае введённый пользователем текст разделится на части. Разделителями будут символы из зарезервированной переменной `IFS`. По умолчанию это пробел, знак табуляции и перевод строки.

Рассмотрим пример. Предположим, что вводимые пользователем строки сохраняются в двух переменных с именами `path` и `file`. Вызов read в этом случае выглядит так:

```
read -r path file
```

Дальше пользователь вводит следующий текст:

```
~/Documents report.txt
```

Тогда путь `~/Documents` попадёт в переменную `path`, а имя файла `report.txt` в `file`.

Если путь содержит пробелы, произойдёт ошибка. Предположим, пользователь ввёл следующее:

```
~/My Documents report.txt
```

Тогда в переменную `path` попадёт строка `~/My`. В `file` запишется всё остальное: `Documents report.txt`. Не забывайте про такое поведение команды read.

Проблему разделения строки можно решить. Для этого переопределим зарезервированную переменную `IFS`. В качестве разделителя укажем только запятую:

```
IFS=$',' read -r path file
```

В этом примере мы применили специфичный для Bash вид кавычек[324] `$'...'`. В них не выполняются никакие подстановки. Но некоторые управляющие последовательности разрешены: `\n` (новая строка), `\\` (экранированный обратный слэш), `\t` (табуляция) и `\xnn` (байты в шестнадцатеричной системе).

Теперь следующий ввод пользователя обработается корректно:

```
1 ~/My Documents,report.txt
```

Путь и имя файла разделены запятой. При этом она не встречается ни в пути, ни в имени. Поэтому ввод пользователя обработается корректно. Строка `~/My Documents` попадёт в переменную `path`, а `report.txt` — в `file`.

Команда read читает данные со стандартного потока ввода. Это значит, что ей на вход можно перенаправить содержимое файла.

Для примера прочитаем первую строку файла `contacts.txt` из листинга 3-21. Это сделает следующая команда:

[324] http://mywiki.wooledge.org/Quotes

```
read -r contact < contacts.txt
```

После выполнения этой команды в переменную `contact` запишется строка “Alice=alice@gmail.com”.

Имя и контактные данные можно записать в разные переменные. Для этого в качестве разделителя укажем знак равно =. Получим такую команду read:

```
IFS='=' read -r name contact < contacts.txt
```

Теперь имя `Alice` запишется в переменную `name`, а адрес электронной почты в `contact`.

Чтобы прочитать весь файл `contacts.txt`, напишем такой цикл while:

```
while IFS='=' read -r name contact < "contacts.txt"
do
  echo "$name = $contact"
done
```

К сожалению, это не сработает. Произойдёт **зацикливание**. Зацикливанием называется бесконечное повторение тела цикла из-за ошибки. Причина проблемы в том, что read всегда читает только первую строку файла и возвращает нулевой код возврата. Нулевой код в условии цикла приведёт к повторному выполнению его тела снова и снова.

Чтобы цикл while последовательно прошёл по всем строкам файла, запишем его в следующей форме:

```
while УСЛОВИЕ
do
  ДЕЙСТВИЕ
done < ФАЙЛ
```

Чтобы обработать ввод пользователя с клавиатуры, в качестве файла укажите `/dev/tty`. Тогда цикл будет обрабатывать ввод до тех пор, пока пользователь не нажмёт сочетание клавиш Ctrl+D.

Правильный вариант цикла while для чтения файла `contacts.txt` выглядит так:

```
while IFS='=' read -r name contact
do
  echo "$name = $contact"
done < "contacts.txt"
```

Этот цикл выведет на экран всё содержимое файла контактов.

Нам остался последний шаг. На каждой итерации цикла будем добавлять в массив элемент, соответствующий значениям переменных `name` и `contact`.

Конечный вариант скрипта для работы с файлом контактов приведён в листинге 3-22

Листинг 3-22. Скрипт для работы с файлом контактов

```
#!/bin/bash

declare -A array

while IFS=$'=' read -r name contact
do
  array[$name]=$contact
done < "contacts.txt"

echo "${array["$1"]}"
```

Мы получили такое же поведение, как у скрипта из листинга 3-10.

Оператор for

Кроме while в Bash есть оператор цикла for. Используйте его, когда количество итераций известно заранее.

У оператора for есть две формы. Первая нужна для последовательной обработки слов в строке. Во второй форме условием цикла выступает арифметическое выражение.

Первая форма for

Начнём с первой формы for. В общем виде она выглядит так:

```
for ПЕРЕМЕННАЯ in СТРОКА
do
  ДЕЙСТВИЕ
done
```

В однострочном виде эта же конструкция записывается так:

```
for ПЕРЕМЕННАЯ in СТРОКА; do ДЕЙСТВИЕ; done
```

ДЕЙСТВИЕМ в конструкции for может быть одна команда или блок команд. Точно так же как в операторе while.

Перед первой итерацией цикла Bash выполнит все подстановки в условии конструкции for. Что это значит? Предположим, что вместо СТРОКИ вы указали команду. Тогда перед началом цикла команда выполнится и её вывод заменит СТРОКУ. Если указать шаблон — он будет развёрнут.

Дальше СТРОКА разделяется на слова. Разделители читаются из переменной `IFS`. Затем выполняется первая итерация цикла. Во время итерации первое слово из СТРОКИ будет доступно в теле цикла как значение ПЕРЕМЕННОЙ. На второй итерации в ПЕРЕМЕННУЮ запишется второе слово СТРОКИ и т.д. Цикл завершится после прохода по всем словам СТРОКИ.

Рассмотрим пример цикла for. Напишем скрипт для обработки слов в строке. Строка передаётся в скрипт первым параметром.

Листинг 3-23 демонстрирует код скрипта.

Листинг 3-23. Скрипт для обработки слов в строке

```
#!/bin/bash

for word in $1
do
  echo "$word"
done
```

Обратите внимание, что позиционный параметр `$1` не надо заключать в кавычки. Если это сделать, не сработает word splitting. Входная строка не разделится на слова. Тогда тело цикла выполнится один раз. При этом в переменную `word` запишется вся входная строка. Это не то, что нам нужно. Скрипт должен обработать каждое слово строки по отдельности.

Передаваемую в скрипт строку надо заключить в кавычки. Тогда она целиком попадёт в позиционный параметр `$1`. Например:

```
./for-string.sh "this is a string"
```

Проблему кавычек при передаче строки в скрипт можно решить. Замените в условии цикла позиционный параметр `$1` на `$@`. Получится такая конструкция for:

```
for word in $@
do
  echo "$word"
done
```

Теперь сработают оба варианта вызова скрипта:

```
./for-string.sh this is a string
./for-string.sh "this is a string"
```

У условия цикла for есть краткая форма. Она перебирает все входные параметры скрипта. Мы записали условие цикла так:

```
for word in $@
```

Тот же самый результат даст следующее условие:

```
for word
do
  echo "$word"
done
```

Мы просто отбросили “in $@” в условии. Поведение цикла от этого не изменилось.

Немного усложним задачу. Предположим, что скрипт получает на вход список путей. Их разделяют запятые. В самих путях могут встречаться пробелы. Чтобы правильно обработать такой ввод, переопределим переменную `IFS`.

Листинг 3-24 демонстрирует цикл for для обработки списка путей.

Листинг 3-24. Скрипт для обработки списка путей

```
#!/bin/bash

IFS=$','
for path in $1
do
  echo "$path"
done
```

Через переменную `IFS` мы указали единственный разделитель слов — запятую. Поэтому цикл for при разделении строки `$1` будет ориентироваться на запятые, а не на пробелы.

Скрипт можно вызвать например так:

```
./for-path.sh "~/My Documents/file1.pdf,~/My Documents/report2.txt"
```

В этом случае кавычки для строки с путями обязательны. Если их опустить и заменить в скрипте параметр `$1` на `$@`, возникнет ошибка. Во время вызова скрипта произойдёт word splitting. При этом разделители прочитаются из переменной `IFS` окружения. То есть до нашего переопределения `IFS` в скрипте. Поэтому строка с путями разделится пробелами.

Если в одном из путей встретится запятая, опять же возникнет ошибка.

Цикл for позволяет пройти по элементам индексируемого массива. Это работает так же, как перебор слов в строке. Листинг 3-25 демонстрирует пример.

Листинг 3-25. Скрипт для обработки всех элементов массива

```
#!/bin/bash

array=(Alice Bob Eve Mallory)

for element in "${array[@]}"
do
  echo "$element"
done
```

Предположим, вам нужны только первые три элемента. Тогда в условии цикла можно подставить не весь массив, а только нужные элементы. Например, как в листинге 3-26.

Листинг 3-26. Скрипт для обработки первых трёх элементов массива

```
#!/bin/bash

array=(Alice Bob Eve Mallory)

for element in "${array[@]:0:2}"
do
  echo "$element"
done
```

Другой вариант — перебирать в цикле не сами элементы, а их индексы. Запишем индексы нужных элементов через пробел. Затем пройдём по ним в цикле. Получится следующее:

```
array=(Alice Bob Eve Mallory)

for i in 0 1 2
do
  echo "${array[i]}"
done
```

Цикл пройдёт только по элементам с индексами 0, 1 и 2.

Нужные индексы элементов можно указать через подстановку ϑигурных скобок. Например, так:

```
array=(Alice Bob Eve Mallory)

for i in {0..2}
do
  echo "${array[i]}"
done
```

Результат будет тем же — скрипт выведет первые три элемента массива.

Не используйте индексы элементов при обработке массивов с пропусками. Вместо этого подставляйте нужные элементы массива в условии цикла, как в листингах 3-25 и 3-26.

Обработка списка файлов

Цикл for подходит для обработки списка файлов. Для решения этой задачи важно правильно составить условие цикла. При этом часто совершают ряд типичных ошибок. Рассмотрим их на примерах.

Напишем скрипт для вывода типов файлов в текущем каталоге. Вывести тип файла можно утилитой file.

Главная ошибка при составлении условия цикла for — пренебрежение шаблонами (globbing). Вместо шаблона в качестве СТРОКИ часто подставляют вывод утилит ls или find. Например, так:

```
for filename in $(ls)
for filename in $(find . - type f)
```

Это неправильно. Такое решение приведёт к следующим проблемам:

1. Word splitting разделит на части имена файлов и каталогов с пробелами.
2. Если имя файла содержит символ звёздочка *, перед итерацией цикла выполнится подстановка шаблонов. Результат подстановки запишется в переменную `filename` вместо настоящего имени файла.
3. Вывод утилиты ls зависит от региональных настроек. Из-за этого некоторые символы национального алфавита в именах файлов могут поменяться на знаки вопроса. Тогда цикл for не сможет обработать эти файлы.

Всегда используйте шаблоны в цикле for для перебора имён файлов. Это единственное правильное решение задачи.

Для нашей задачи условие цикла выглядит так:

```
for filename in *
```

Листинг 3-27 демонстрирует полную версию скрипта.

Листинг 3-27. Скрипт для вывода типов файлов

```
#!/bin/bash

for filename in *
do
  file "$filename"
done
```

Не забывайте про двойные кавычки при подстановке переменной `filename`. Это предотвратит word splitting в именах файлов с пробелами.

Шаблон в условии цикла for сработает, если надо пройти по файлам из конкретного каталога. Такой шаблон выглядит, например, так:

```
for filename in /usr/share/doc/bash/*
```

Шаблон может отфильтровать файлы с определённым расширением или именем. Например:

```
for filename in ~/Documents/*.pdf
```

В Bash начиная с версии 4 шаблоны позволяют обходить каталоги рекурсивно. Например:

```
shopt -s globstar

for filename in **
```

Чтобы это сработало, включите опцию интерпретатора `globstar` с помощью команды shopt.

Вместо шаблона ** Bash подставит список всех подкаталогов и файлов в них, начиная с текущего каталога. Этот механизм можно совмещать с обычными шаблонами.

Например, пройдём по всем файлам с расширением PDF из домашнего каталога пользователя. Условие цикла for для этого выглядит так:

```
shopt -s globstar

for filename in ~/**/*.pdf
```

Скрипт из листинга 3-27 можно заменить следующим вызовом утилиты find:

```
find . -maxdepth 1 -exec file {} \;
```

Такое решение эффективнее, чем цикл for. Оно компактнее и работает быстрее из-за меньшего числа операций.

Когда стоит обрабатывать файлы в цикле for, а когда утилитой find? Используйте find, когда файлы можно обработать одной короткой командой. Если для обработки нужны условные операторы или блок команд, вызов find становится громоздким. В этом случае цикл for предпочтительнее.

В скрипте из листинга 3-27 конструкцию for можно заменить на while. Чтобы получить список файлов для обработки, вызовем утилиту find. При этом важно использовать её опцию `-print0`. Листинг 3-28 демонстрирует результат.

Листинг 3-28. Скрипт для вывода типов файлов

```
#!/bin/bash

while IFS= read -r -d '' filename
do
  file "$filename"
done < <(find . -maxdepth 1 -print0)
```

В этом скрипте есть несколько важных решений. Рассмотрим их подробнее. Первый вопрос: зачем переменной `IFS` присваивать пустое значение? Без этого word splitting разделит вывод команды find пробелами, табуляцией и переводом строк. Тогда имена файлов с этими символами обработаются неправильно.

Второе важное решение — опция `-d` команды read. Она определяет символ для разделения текста на входе команды. В переменную `filename` запишется часть текста до очередного разделителя.

В нашем примере разделитель для команды read пустой. Это означает NUL-символ. Его можно указать и явно. Например, так:

```
while IFS= read -r -d $'\0' filename
```

Благодаря опции `-d`, команда read правильно обработает вывод утилиты find. Утилита вызвана с опцией `-print0`. Это значит, что найденные файлы в выводе разделит NUL-символ.

Обратите внимание, что указать NUL-символ в качестве разделителя через переменную `IFS` нельзя. Другими словами следующий вариант не сработает:

```
while IFS=$'\0' read -r filename
```

Проблема в особенности интерпретации[325] переменной `IFS`. Если её значение пустое, Bash вообще не выполняет word splitting.

В скрипте из листинга 3-28 осталось ещё одно неочевидное решение. Вывод утилиты find передаётся в цикл while через подстановку процесса. Почему не подходит подстановка команды? Например, такая:

```
while IFS= read -r -d '' filename
do
  file "$filename"
done < $(find . -maxdepth 1 -print0)
```

Так перенаправить результат выполнения команды нельзя. Оператор < связывает поток ввода с указанным файловым дескриптором. Но при подстановке команды никакого дескриптора нет. Bash вызывает утилиту find и подставляет её вывод вместо `$(...)`. При подстановке процессов вывод find запишется во временный файл. У него есть дескриптор. Поэтому перенаправление потоков сработает.

У подстановки процессов есть одна проблема. Эта подстановка не входит в POSIX-стандарт. Если вам важно следовать стандарту, используйте конвейер. Листинг 3-29 демонстрирует такое решение.

Листинг 3-29. Скрипт для вывода типов файлов

```
#!/bin/bash

find . -maxdepth 1 -print0 |
while IFS= read -r -d '' filename
do
  file "$filename"
done
```

Комбинация цикла while и утилиты find предпочтительнее for в одном случае: если вы обрабатываете файлы и условие их поиска сложное.

При комбинации while и find всегда используйте NUL-символ в качестве разделителя. Так вы избежите проблем обработки имён файлов с пробелами.

Вторая форма for

Во второй форме оператора for условием цикла выступает арифметическое выражение. Разберёмся, в каких случаях оно понадобится. Рассмотрим примеры.

Предположим, нам нужен скрипт для расчёта факториала. Решение задачи зависит от способа ввода данных. Первый вариант — число для расчёта известно заранее. Тогда подойдёт первая форма цикла for. Листинг 3-30 демонстрирует такой вариант скрипта.

[325] https://mywiki.wooledge.org/IFS

Листинг 3-30. Скрипт для расчёта факториала числа 5

```
#!/bin/bash

result=1

for i in {1..5}
do
  ((result *= $i))
done

echo "Факториал числа 5 равен $result"
```

Второй вариант — пользователь передаёт число для расчёта через входной параметром скрипта. Для решения такой задачи попробуем следующий вариант условия цикла for:

```
for i in {1..$1}
```

Ожидается, что Bash выполнит подстановку фигурных скобок для целых чисел от одного до значения параметра `$1`. Это не сработает.

Согласно таблице 3-2, подстановка фигурных скобок выполняется до подстановки параметров. Поэтому в условии цикла вместо строки “1 2 3 4 5” получится строка “”. Bash не распознал подстановку фигурных скобок, потому что верхняя граница диапазона — не число. Дальше строка “” запишется в переменную `i`. Из-за этого оператор ((не сможет обработать её корректно.

Утилита seq решит нашу проблему. Она генерирует последовательность целых или дробных чисел.

Таблица 3-21 демонстрирует способы вызова утилиты seq.

Таблица 3-21. Способы вызова утилиты seq

Число параметров	Описание параметров	Пример команды	Результат
1	Последнее число в генерируемой последовательности. Последовательность начинается с единицы.	`seq 5`	1 2 3 4 5
2	Первое и последнее число в последовательности.	`seq -3 3`	-2 -1 0 1 2
3	Первое число, шаг и последнее число в последовательности.	`seq 1 2 5`	1 3 5

Числа в выводе утилиты seq разделяются переводом строки `\n`. Опция `-s` позволяет указать

другой разделитель. Перевод строки входит в список стандартных разделителей переменной `IFS`. Поэтому в конструкции for опция `-s` для seq не нужна.

В таблице 3-21 перевод строки заменён на пробел в столбце “Результат” для удобства.

Воспользуемся утилитой seq, чтобы написать параметризуемый скрипт для расчёта факториала. Он приведён в листинге 3-31.

Листинг 3-31. Скрипт для расчёта факториала

```
#!/bin/bash

result=1

for i in $(seq $1)
do
  ((result *= $i))
done

echo "Факториал числа $1 равен $result"
```

Это решение работает. Однако, его нельзя назвать эффективным. В условии цикла for вызывается внешняя утилита. Такой вызов сравним с запуском обычной программы. Например, калькулятора. Для создания нового процесса ядро ОС выполняет несколько сложных операций. По меркам процессора они занимают значительное время. Поэтому старайтесь обходиться встроенными средствами Bash везде, где это возможно.

Для решения задачи нам пригодится вторая форма оператора for. В общем виде она выглядит так:

```
for (( ВЫРАЖЕНИЕ_1; ВЫРАЖЕНИЕ_2; ВЫРАЖЕНИЕ_3 ))
do
  ДЕЙСТВИЕ
done
```

В однострочном виде эта конструкция записывается так:

```
for (( ВЫРАЖЕНИЕ_1; ВЫРАЖЕНИЕ_2; ВЫРАЖЕНИЕ_3 )); do ДЕЙСТВИЕ; done
```

Цикл for с арифметическим условием работает так:

1. ВЫРАЖЕНИЕ_1 выполняется однократно перед первой итерацией цикла.
2. Цикл выполняется до тех пор, пока ВЫРАЖЕНИЕ_2 остаётся истинным. Как только оно вернуло ложь в качестве результата, цикл завершается.
3. В конце каждой итерации выполняется ВЫРАЖЕНИЕ_3.

Заменим вызов утилиты seq на арифметическое выражение в листинге 3-31. Результат приведён в листинге 3-32.

Листинг 3-32. Скрипт для расчёта факториала

```
#!/bin/bash

result=1

for (( i = 1; i <= $1; ++i ))
do
  ((result *= i))
done

echo "Факториал числа $1 равен $result"
```

Скрипт стал работать быстрее. Теперь он использует только встроенные операторы Bash. Для их исполнения не нужно создавать новые процессы.

Рассмотрим алгоритм конструкции for в скрипте:

1. Перед первой итерацией цикла объявляется переменная `i`. Это счётчик цикла. Ему присваивается единица.
2. Счётчик цикла сравнивается с входным параметром: “i <= $1”. Если условие выполняется, возвращается нулевой код возврата.
3. Если условие вернуло ноль, выполняется первая итерация цикла. В противном случае цикл завершается.
4. В теле цикла вычисляется арифметическое выражение “result *= i”. В результате значение переменной `result` будет умножено на `i`.
5. После выполнения первой итерации, вычисляется третье выражение `++i` в условии цикла. В результате значение переменной `i` станет равно двум.
6. Переход ко второму шагу алгоритма с проверкой условия “i <= $1”. Если условие по-прежнему истинно, выполняется следующая итерация цикла.

В общем случае для переменных в операторе ((и команде let знака доллара $ не нужен. Однако, в нашем условии цикла он необходим. Без него Bash не поймёт, что имеется ввиду позиционный параметр `$1`, а не целое число единица.

В цикле мы используем префиксную форму инкремента. Она выполняется быстрее, чем постфиксная.

Используйте вторую форму оператора for, если счётчик цикла рассчитывается по формуле. Других эффективных решений в этом случае нет.

Управление циклом

Цикл завершается согласно своему условию. Кроме условия есть дополнительные средства для управления циклом. Они позволяют прервать его выполнение или пропустить текущую итерацию. Рассмотрим их подробнее.

break

Встроенная команда break немедленно прекращает выполнение цикла. Она полезна для обработки ошибок или выхода из бесконечного цикла.

Для примера напишем скрипт. Он ищет элемент индексируемого массива с определённым значением. Как только элемент найден, нет смысла продолжать цикл. Можно сразу выйти из него. Листинг 3-33 демонстрирует такой скрипт.

Листинг 3-33. Скрипт поиска элемента в массиве

```
#!/bin/bash

array=(Alice Bob Eve Mallory)
is_found="0"

for element in "${array[@]}"
do
  if [[ "$element" == "$1" ]]
  then
    is_found="1"
    break
  fi
done

if [[ "$is_found" -ne "0" ]]
then
  echo "Элемент со значением $1 есть в массиве"
else
  echo "Элемента со значением $1 нет в массиве"
fi
```

Искомый элемент массива передаётся в скрипт в параметре `$1`.

Результат поиска хранится в переменной `is_found`. В конструкции if сравнивается текущий элемент массива и искомый. Если они равны, переменной `is_found` присваивается единица. Затем выполнение цикла прерывается командой break.

После цикла в операторе if проверяется значение `is_found`. В зависимости от результата выводится сообщение.

Используйте команду break, чтобы вынести из тела цикла всё лишнее. Чем меньше тело цикла, тем проще его прочитать и понять. Например, в листинге 3-33 можно вывести результат поиска прямо в цикле. Тогда переменная `is_found` не нужна. С другой стороны обработка найденного элемента может быть сложной. Помещать такую обработку в тело цикла — плохая идея.

Если не имеет смысла выполнять скрипт после завершения цикла, команда break не подойдёт. Вместо неё используйте команду exit. Например, если во входных данных скрипта обнаружилась ошибка. Также exit подойдёт, если результат работы цикла обрабатывается в его теле.

Заменим команду break на exit в листинге 3-33. Результат приведён в листинге 3-34.

Листинг 3-34. Скрипт поиска элемента в массиве

```
#!/bin/bash

array=(Alice Bob Eve Mallory)

for element in "${array[@]}"
do
  if [[ "$element" == "$1" ]]
  then
    echo "Элемент со значением $1 есть в массиве"
    exit 0
  fi
done

echo "Элемента со значением $1 нет в массиве"
```

С помощью команды exit мы обработали результат поиска в теле цикла. В данном случае это сократило наш код и сделало его проще. Но при сложной обработке результата эффект будет обратный.

Скрипты из листингов 3-33 и 3-34 дают одинаковый результат.

continue

Встроенная команда continue прекращает исполнение текущей итерации цикла. При этом цикл не завершится. Он продолжит выполняться со следующей итерации.

Рассмотрим пример. Предположим, что надо рассчитать сумму положительных чисел в массиве. Отрицательные числа нас не интересуют. С помощью конструкции if проверим знак в теле цикла. Если знак положительный — добавим число к результату. Получим скрипт, как в листинге 3-35.

Листинг 3-35. Скрипт для расчёта суммы положительных чисел в массиве

```
#!/bin/bash

array=(1 25 -5 4 -9 3)
sum=0

for element in "${array[@]}"
do
  if (( 0 < element ))
  then
    ((sum += element))
  fi
done

echo "Сумма положительных чисел равна $sum"
```

Если `element` больше нуля, его значение добавляется к результату `sum`.

Воспользуемся командой continue, чтобы получить такое же поведение. Результат приведён в листинге 3-36.

Листинг 3-36. Скрипт для расчёта суммы положительных чисел в массиве

```
#!/bin/bash

array=(1 25 -5 4 -9 3)
sum=0

for element in "${array[@]}"
do
  if (( element < 0))
  then
    continue
  fi

  ((sum += element))
done

echo "Сумма положительных чисел равна $sum"
```

Мы инвертировали условие конструкции if. Теперь оно истинно для отрицательных чисел. В этом случае вызовется команда continue. Она прервёт текущую итерацию цикла. Операция

сложения после if не выполнится. Вместо этого начнётся следующая итерация со следующим элементом массива.

На самом деле мы применили технику раннего возврата в контексте цикла. Используйте команду continue, чтобы обработать ошибки. Также она пригодится для условий, когда выполнять тело цикла до конца не имеет смысла. Так вы избежите вложенных конструкций if. В результате код станет понятнее и чище.

Упражнение 3-12. Операторы цикла

```
Напишите игру "Больше-Меньше". В ней один участник загадывает число от 1 до 100.
Второй участник пытается его отгадать за семь попыток.

Ваш скрипт загадывает число. Пользователь вводит вариант ответа.
Скрипт отвечает больше или меньше ответ чем загаданное число.
Затем пользователь пытается отгадать число ещё шесть раз.
```

Функции

Bash относится к **процедурным языкам программирования**[326]. Процедурные языки позволяют разделить программу на логические части — **подпрограммы**. Подпрограмма — это самостоятельный блок кода, который решает конкретную задачу. Подпрограммы вызываются из основной программы.

В современных языках подпрограммы называются **функциями**. Мы уже сталкивались с ними, когда знакомились с командой declare. Теперь рассмотрим подробнее, как функции устроены и для чего нужны.

Парадигмы программирования

Для начала разберёмся с терминологией. Она поможет понять, зачем вообще нужны функции.

Что такое процедурное программирование? Это одна из **парадигм**[327] разработки ПО. Парадигма — это набор идей, методов и принципов, которые определяют способ написания программ.

Современные языки следуют одной из двух доминирующих сегодня парадигм:

1. Императивное программирование. Разработчик явно указывает машине, как ей изменять своё состояние. Другими словами он задаёт полный алгоритм вычисления результата.

[326] https://ru.wikipedia.org/wiki/Процедурное_программирование
[327] https://ru.wikipedia.org/wiki/Парадигма_программирования

2. Декларативное программирование. Разработчик указывает свойства желаемого результата, но не алгоритм его вычисления.

Bash следует первой парадигме. Это императивный язык.

Императивная и декларативная парадигмы определяют общие принципы написания программы. В рамках одной парадигмы есть различные методологии (подходы). Методология предлагает конкретные приёмы программирования. Так у императивной парадигмы есть две основных методологии:

1. Процедурное программирование.
2. Объектно-ориентированное программирование.

Эти методологии предлагают по-разному структурировать исходный код программы. Bash следует первой методологии.

Рассмотрим процедурное программирование подробнее. Процедурный язык предоставляет средства для объединения наборов команд в независимые блоки кода. Эти блоки кода называются процедурами или функциями. Функцию можно вызвать из любого места программы. На вход она принимает параметры. Этот механизм похож на передачу параметров командной строки в скрипт. Поэтому функцию иногда называют программой в программе или подпрограммой.

Основная задача функций — управление сложностью программ. Чем больше объём исходного кода, тем сложнее его сопровождать и поддерживать в рабочем состоянии. Ситуацию усугубляют повторяющиеся фрагменты кода. Они разбросаны по всей программе и могут содержать ошибки. После исправления ошибки в одном таком фрагменте, надо найти и исправить все остальные. Если фрагмент вынести в функцию, то достаточно исправить ошибку только в ней.

Рассмотрим пример повторяющегося фрагмента кода. Представьте, что вы пишете большую программу. Чтобы обработать ошибки, программа выводит в поток ошибок текстовые сообщения. Тогда в исходном коде появится много мест с вызовом команды echo. Например, таких:

```
>&2 echo "Произошла ошибка N"
```

В какой-то момент вы решаете, что лучше записывать все ошибки в файл. Тогда анализировать их станет легче. Пользователи вашей программы могут перенаправить поток ошибок в лог-файл сами. Но, предположим, что не все умеют пользоваться перенаправлением. Поэтому программа должна сама писать сообщения в лог-файл.

Внесём изменение в программу. Для этого нужно пройти по всем местам обработки ошибок. Каждый вызов команды echo надо заменить на следующий:

```
echo "Произошла ошибка N" >> debug.log
```

Если по невнимательности пропустить и не исправить какой-то вызов echo, его вывод не попадёт в лог-файл. Этот вывод может оказаться важным. Без него вы не поймёте, почему программа не работает у пользователя.

Мы рассмотрели одну из сложностей сопровождения программ. Она часто встречается при изменении кода, написанного ранее. В нашем примере проблема возникла из-за нарушения принципа разработки “не повторяйся” (**don’t repeat yourself**[328] или DRY). Один и тот же код вывода ошибок копировался снова и снова в разные места программы. Так делать нельзя.

Функции решают проблему дублирования кода. Чем-то это решение напоминает циклы. Отличие в том, что цикл многократно исполняет набор команд в одном месте программы. В отличие от цикла функция позволяет исполнять одни и те же команды в разных местах программы.

Функция улучшит читаемость кода программы. Она объединяет набор команд в единый блок. Если дать блоку говорящее имя, станет очевидна решаемая им задача. В программе функция вызывается по своему имени. Благодаря этому, программу станет легче читать. Вместо десятка строк тела функции, будет стоять её имя. Оно объяснит читателю, что происходит в функции.

Функции в командном интерпретаторе

Функции доступны в обоих режимах Bash: командный интерпретатор и исполнение скриптов. Начнём с командного интерпретатора.

В общем виде функция объявляется так:

```
1 ИМЯ_ФУНКЦИИ()
2 {
3   ДЕЙСТВИЕ
4 }
```

В одну строку функцию можно объявить так:

```
ИМЯ_ФУНКЦИИ() { ДЕЙСТВИЕ ; }
```

Обратите внимание на обязательную точку с запятой перед закрывающей фигурной скобкой }.

Тело функции ДЕЙСТВИЕ может быть одной командой или блоком команд.

[328] https://ru.wikipedia.org/wiki/Don’t_repeat_yourself

На имена функций в Bash накладываются те же ограничения, что и на имена переменных. В них допустимы только символы латинского алфавита, числа и знак подчёркивания _. Имя не должно начинаться с числа.

Рассмотрим, как объявлять и использовать функции в командном интерпретаторе. Предположим, вам нужна статистика использования оперативной памяти. Эта информация доступна через файловую систему **proc**[329] или procfs. Через proc можно узнать список работающих процессов, состояние ОС и оборудования компьютера. Эта информация доступна через файлы, находящихся по системному пути `/proc`.

Статистика использования оперативной памяти доступна в файле `/proc/meminfo`. Прочитаем его с помощью утилиты cat:

```
cat /proc/meminfo
```

Вывод команды зависит от вашей системы. Для окружения MSYS2 он даст меньше информации, для Linux-системы — больше.

Для MSYS2 содержимое файла `meminfo` будет примерно таким:

```
MemTotal:        6811124 kB
MemFree:         3550692 kB
HighTotal:             0 kB
HighFree:              0 kB
LowTotal:        6811124 kB
LowFree:         3550692 kB
SwapTotal:       1769472 kB
SwapFree:        1636168 kB
```

Таблица 3-22 объясняет значение каждого поля.

Таблица 3-22. Поля в файле `meminfo`

Поле	Значение
MemTotal	Объём доступной в системе RAM.
MemFree	Объём не используемой в данный момент RAM. Считается как LowFree + HighFree.
HighTotal	Объём доступной памяти в области RAM выше 860 мегабайтов.
HighFree	Объём не используемой памяти в области RAM выше 860 мегабайтов.
LowTotal	Объём доступной памяти в области RAM ниже 860 мегабайтов.

[329] https://ru.wikipedia.org/wiki/Procfs

Таблица 3-22. Поля в файле `meminfo`

Поле	Значение
LowFree	Объём не используемой памяти в области RAM ниже 860 мегабайтов.
SwapTotal	Объём доступной памяти в **области подкачки**[330] на жёстком диске.
SwapFree	Объём не используемой памяти в области подкачки.

Подробнее значения полей файла `meminfo` рассматриваются в статье[331].

Чтобы не набирать команду чтения файла `meminfo` каждый раз, объявим функцию с коротким именем. Например, так:

```
mem() { cat /proc/meminfo; }
```

Это однострочное объявление функции с именем `mem`. Её можно вызвать так же, как любую Bash-команду. Например:

```
mem
```

Функция выведет статистику использования памяти.

Команда unset удаляет объявленную ранее функцию. Удалим нашу функцию `mem` следующей командой:

```
unset mem
```

Предположим, что переменная и функция объявлены с одинаковыми именами. Чтобы удалить именно функцию, используйте опцию `-f` команды unset. Например, так:

```
unset -f mem
```

Объявление функции можно добавить в файл `~/.bashrc`. Тогда функция будет доступна при каждом запуске командной оболочки.

В командной строке мы объявили функцию `mem` в однострочном формате. Его удобнее и быстрее набирать. В файле `~/.bashrc` важна наглядность. Там функцию `mem` лучше объявить в стандартном виде. Например, так:

[330] https://ru.wikipedia.org/wiki/Подкачка_страниц
[331] http://markelov.blogspot.com/2009/01/linux-procmeminfo.html

```
mem()
{
  cat /proc/meminfo
}
```

Отличие функций от псевдонимов

Мы объявили функцию `mem` для вывода статистики использования оперативной памяти. То же поведение даст следующий псевдоним:

```
alias mem="cat /proc/meminfo"
```

Если функции и псевдонимы работают одинаково, что выбрать?

Функции и псевдонимы похожи в одном — это встроенные механизмы Bash. С точки зрения пользователя они сокращают ввод длинных команд. Но принцип работы этих механизмов принципиально различается.

Псевдоним заменяет один текст на другой во введённой пользователем команде. Другими словами Bash находит в команде текст, который совпадает с именем alias. Затем заменяет этот текст на значение псевдонима и исполняет получившуюся команду.

Предположим, вы определили псевдоним для утилиты cat. Он добавляет опцию `-n` в вызов утилиты. Благодаря опции, в вывод добавляются номера строк. Псевдоним выглядит так:

```
alias cat="cat -n"
```

Теперь каждый раз когда команда начинается со слова “cat”, Bash подставит вместо него “cat -n”. Например, вы вводите команду:

```
cat ~/.bashrc
```

После подстановки псевдонима она выглядит так:

```
cat -n ~/.bashrc
```

Подстановка заменила только слово “cat” на “cat -n”. Следующий далее путь до файла не изменился.

Чтобы подставить значение alias до исполнения команды, введите её и нажмите Ctrl+Alt+E.

Теперь рассмотрим, как работают функции. В отличие от псевдонима тело функции не подставляется в команду. Когда Bash встречает имя функции в команде, он исполняет её тело.

Пример. Попробуем с помощью функции получить то же поведение, как у псевдонима для утилиты cat. Если бы функции работали как alias, такое определение решило бы задачу:

```
cat() { cat -n; }
```

Мы ожидаем, что в следующей команде Bash просто добавит опцию `-n`:

```
cat ~/.bashrc
```

Но это не сработает. Bash не подставляет тело функции в команду. Bash его исполняет и подставляет в команду результат.

В нашем случае утилита cat будет вызвана с опцией `-n`, но без параметра `~/.bashrc`. Это совершенно не то что нужно.

Чтобы решить задачу, передадим имя файла в функцию в качестве параметра. Это работает так же, как передача параметра в команду или скрипт. После вызова функции укажите список параметров, разделённых пробелами.

В общем виде вызов функции и передача в неё параметров выглядит так:

```
ИМЯ_ФУНКЦИИ ПАРАМЕТР1 ПАРАМЕТР2 ПАРАМЕТР3
```

Чтобы прочитать параметры в теле функции, используйте переменные `$1`, `$2`, `$3` и т.д. Прочитать сразу все параметры можно через переменную `$@`.

Исправим объявление функции `cat`. Все её входные параметры передадим в утилиту cat:

```
cat() { cat -n $@; }
```

Такая функция тоже не заработает. Дело в том, что при её выполнении произойдёт **рекурсия**. Рекурсией называется вызов функции из неё же самой.

Перед выполнением команды “cat -n $@” Bash проверит список объявленных функций. В списке будет функция с именем `cat`. Её тело выполняется в данный момент, но это не важно. Поэтому вместо вызова утилиты Bash вызовет функцию `cat`. Этот вызов повторится снова и снова. Возникнет бесконечная рекурсия, которая похожа на бесконечный цикл.

Рекурсия — вовсе не ошибка в поведении интерпретатора. Это мощный механизм, который значительно упрощает сложные алгоритмы (например, обход **графа**[332] или **дерева**[333]).

Ошибка в нашем объявлении функции `cat`. Рекурсивный вызов произошел случайно и привел к зацикливанию. Решить эту проблему можно двумя способами:

1. Использовать встроенную команду command.
2. Переименовать функцию так, чтобы её имя отличалось от имени утилиты.

Рассмотрим первое решение. В качестве параметров command получает команду. Если в команде встречаются имена псевдонимов и функций, Bash не станет их обрабатывать. Тело псевдонима не подставится. Функция не вызовется.

Применим команду command в объявлении функции `cat`. Получим следующее:

[332] https://ru.wikipedia.org/wiki/Граф_(математика)
[333] https://ru.wikipedia.org/wiki/Дерево_(структура_данных)

```
cat() { command cat -n "$@"; }
```

Второе решение — просто переименовать функцию. Такой вариант сработает:

```
cat_func() { cat -n "$@"; }
```

Всегда помните о проблеме случайной рекурсии. Не давайте функциям имена, совпадающие с именами команд интерпретатора и GNU-утилит.

Подведём итоги сравнения функций и псевдонимов в командном интерпретаторе. Если нужно просто сократить длинную команду, используйте alias.

Функция нужна только в следующих случаях:

1. Для выполнения действия нужны условные операторы, циклы или блок команд.
2. Параметры команды находятся не в конце.

Рассмотрим пример второго случая — команду, которую нельзя заменить псевдонимом. Сократим вызов утилиты find для поиска файлов в указанном каталоге. Поиск в домашнем каталоге выглядит так:

```
find ~ -type f
```

С помощью псевдонима для этой команды параметризовать путь не получится. Следующий вариант не заработает:

```
alias="find -type f"
```

Проблема в том, что путь должен идти до опции `-type`.

Заменим псевдоним на функцию. В её теле можно выбрать позицию для подстановки параметра в вызов find. Например, так:

```
find_func() { find $1 -type f; }
```

Функции в скриптах

В скриптах функции объявляются точно так же, как в командном интерпретаторе. Допускаются оба варианта объявления: стандартный и однострочный.

Для примера вернёмся к проблеме обработки ошибок в большой программе. Объявим следующую функцию для вывода сообщений об ошибках:

```
print_error()
{
  >&2 echo "Произошла ошибка: $@"
}
```

Текст, объясняющий причину ошибки, передаётся в функцию через параметр. Допустим, наша программа читает файл на диске. Но файл оказался недоступен. Тогда сообщить о проблеме можно так:

```
print_error "файл readme.txt не найден"
```

Предположим, что требования к программе изменились. Теперь сообщения об ошибках нужно выводить в лог-файл. Для этого достаточно исправить объявление функции `print_error`. Команда echo изменится на следующую:

```
print_error()
{
  echo "Произошла ошибка: $@" >> debug.log
}
```

После изменения функции все сообщения об ошибках выводятся в файл `debug.log`. Менять что-либо в местах вызова функции не нужно.

Встречаются ситуация, когда одна функция должна вызвать другую. Это допустимо в Bash. В общем случае функцию можно вызвать из любого места программы.

Рассмотрим пример. Предположим, интерфейс программы надо перевести на другой язык. Такая процедура называется **локализацией**[334]. Сообщения об ошибках лучше выводить на понятном пользователю языке. Для этого продублируем текст всех сообщений на всех языках, поддерживаемых программой. Как это сделать?

Самое простое решение — присвоить каждой ошибке уникальный код. Такая практика часто встречается в системном программировании. Применим этот подход в нашей программе. Тогда функция `print_error` в качестве параметра будет принимать код ошибки.

Код ошибки можно выводить прямо в лог-файл. Но тогда пользователю понадобится информация о значениях кодов. Удобнее выводить текст сообщения, как и раньше. Для этого код ошибки надо конвертировать в текст на нужном языке. Для этой задачи объявим специальную функцию.

Напишем функцию для конвертирования кода ошибки в сообщение. Для конвертирования применим конструкцию case. Каждый блок case соответствует определённому коду ошибки. Объявление функции выглядит так:

[334] https://ru.wikipedia.org/wiki/Локализация_программного_обеспечения

```
code_to_error()
{
  case $1 in
    1)
      echo "Не найден файл"
      ;;
    2)
      echo "Нет прав для чтения файла"
      ;;
  esac
}
```

Теперь перепишем объявление функции `print_error` так:

```
print_error()
{
  echo "$(code_to_error $1) $2" >> debug.log
}
```

Вызов функции `print_error` выглядит, например, так:

```
print_error 1 "readme.txt"
```

В результате вызова в лог-файл запишется строка:

```
Не найден файл readme.txt
```

Первым параметром в функцию передаётся код ошибки. Вторым параметром — имя файла, который привёл к проблеме.

Сопровождать механизм вывода сообщений об ошибках в нашей программе стало проще. Предположим, надо добавить вывод ошибок на другом языке. Для этого достаточно объявить две функции:

- `code_to_error_ru` для сообщений на русском.
- `code_to_error_en` для сообщений на английском.

Чтобы выбрать правильную функцию, можно проверить значение переменной `LANGUAGE` в функции `print_error`.

Если переменная `LANGUAGE` недоступна в вашей системе, используйте переменную `LANG`.

Наше решение с конвертированием кода ошибок — это учебный пример. Для локализации скриптов у Bash есть специальный механизм. В нём используются PO-файлы с текстами на разных языках. Подробнее об этом механизме читайте в статье BashFAQ[335].

Упражнение 3-13. Функции

Напишите следующие функции, чтобы выводить сообщения об ошибках на русском и английском языках:

* print_error
* code_to_error_ru
* code_to_error_en

Напишите два варианта функции code_to_error:

* с конструкцией case.
* с ассоциативным массивом.

Возврат результата функции

Чтобы вернуть результат функции, процедурные языки имеют встроенную команду. Обычно она называется return. В Bash эта команда тоже есть. Но её поведение отличается. Команда return в Bash не возвращает значение. Она передаёт код возврата, то есть целое число от 0 до 255.

Полный алгоритм вызова и выполнения функции выглядит так:

1. При выполнении команды встречается имя функции.
2. Интерпретатор переходит в тело функции и исполняет его с первой команды.
3. Если в теле функции встречается команда return, выполнение функции прекращается. Bash переходит в место её вызова. В специальный параметр `$?` записывается код возврата функции. Это параметр команды return.
4. Если в теле функции нет return, Bash выполняет его до последней команды. После этого интерпретатор переходит в место вызова функции.

В других процедурных языках команда return возвращает переменную любого типа: число, строку или массив. Такое же поведение можно получить и в Bash. Для этого есть три способа:

1. Подстановка команд.
2. Глобальная переменная.
3. Вызывающая сторона указывает глобальную переменную.

Рассмотрим пример каждого из трёх способов.

В прошлом разделе мы написали функции `code_to_error` и `print_error` для вывода сообщений об ошибках. Они выглядят так:

[335] https://mywiki.wooledge.org/BashFAQ/098

```
code_to_error()
{
  case $1 in
    1)
      echo "Не найден файл"
      ;;
    2)
      echo "Нет прав для чтения файла"
      ;;
  esac
}

print_error()
{
  echo "$(code_to_error $1) $2" >> debug.log
}
```

Здесь работает первый способ возврата значения. Вызов функции `code_to_error` помещается в подстановку команды. Благодаря этому, Bash подставит в место вызова функции всё, что она выведет на консоль.

В нашем примере функция `code_to_error` выводит сообщение об ошибке через команду echo. Далее Bash подставляет этот вывод в тело функции `print_error`. В результате получается команда echo, состоящая из двух частей:

1. Вывод функции `code_to_error`. Это сообщение об ошибке.
2. Входной параметр `$2` функции `print_error`. Это имя файла, доступ к которому вызвал ошибку.

Составная команда в функции `print_error` выводит полное сообщение об ошибке в лог-файл.

Второй способ вернуть значение из функции — записать его в глобальную переменную. Такая переменная доступна в любом месте скрипта. То есть и в теле функции, и в месте её вызова.

Все объявленные в скрипте переменные являются глобальными по умолчанию. У этого правила есть одно исключение. Его мы рассмотрим далее.

Перепишем функции `code_to_error` и `print_error`. Сохраним результат `code_to_error` в глобальной переменной. Затем прочитаем эту переменную в функции `print_error`. Получится следующее:

```
code_to_error()
{
  case $1 in
    1)
      error_text="Не найден файл"
      ;;
    2)
      error_text="Нет прав для чтения файла"
      ;;
  esac
}

print_error()
{
  code_to_error $1
  echo "$error_text $2" >> debug.log
}
```

Результат функции `code_to_error` записывается в переменную `error_text`. Затем значения параметра `$2` и `error_text` подставляются в команду echo в функции `print_error`. Так получается сообщение для вывода в лог-файл.

Возвращать значение из функции через глобальную переменную опасно. Это чревато конфликтом имён. Для примера предположим, что в скрипте есть другая переменная `error_text`. Она никак не связана с выводом в лог-файл. Тогда любой вызов функции `code_to_error` перезапишет значение этой переменной. Это приведёт к ошибкам во всех местах использования `error_text` вне функции.

Решить проблему конфликта имён поможет **соглашение об именовании переменных**. Такое соглашение — это один из пунктов **стандарта оформления кода**[336] (coding style). Любой крупный программный проект должен иметь такой стандарт.

Вот пример соглашения об именовании переменных:

> Все глобальные переменные, через которые функции возвращают значения, имеют префикс знак подчёркивания _.

Будем следовать этому соглашению. Тогда переменная для возврата значения из функции `code_to_error` должна называться `_error_text`. Проблема решится, но лишь отчасти. Предположим, одна функция вызывает другую (вложенный вызов). Случайно они возвращают свои значения через переменные с одинаковыми именами. Это приведёт к ошибке.

Третий способ возврата значения из функции решает проблему конфликта имён. Вызывающая сторона задаёт имя глобальной переменной. Функция записывает свой результат в переменную с этим именем.

[336] https://ru.wikipedia.org/wiki/Стандарт_оформления_кода

Как работает передача имени переменной в функцию? Имя передаётся через параметр функции, как и любое другое значение. Дальше, функция использует команду eval. Эта команда конвертирует текст в Bash-команду. Имя переменной хранится в виде текста. Поэтому без eval обратиться к переменной не получится.

Перепишем функцию `code_to_error`. Вместо одного параметра будем передавать в неё два:

1. Код ошибки в `$1`.
2. Имя глобальной переменной для возврата значения в `$2`.

Получится такой код:

```
code_to_error()
{
  local _result_variable=$2

  case $1 in
    1)
      eval $_result_variable="'Не найден файл'"
      ;;
    2)
      eval $_result_variable="'Нет прав для чтения файла'"
      ;;
  esac
}

print_error()
{
  code_to_error $1 "error_text"
  echo "$error_text $2" >> debug.log
}
```

На первый взгляд код мало отличается от предыдущего варианта. Но это не так. Мы получили дополнительную гибкость поведения. Теперь вызывающая сторона выбирает имя глобальной переменной для возврата значения. Это имя явно указывается в коде вызова. Поэтому обнаружить конфликт и решить его стало проще.

Область видимости переменных

Конфликт имён — это серьёзная проблема. Она возникает в Bash, когда функции объявляют переменные в глобальном пространстве имён. В результате имена двух переменных могу совпасть. Тогда к ним обращаются разные функции в разные моменты времени. Это приводит к путанице и потере данных.

Чтобы решить конфликт имён в Bash, ограничивайте **область видимости**[337] переменных.

[337] https://ru.wikipedia.org/wiki/Пространство_имён_(программирование)

Рассмотрим этот механизм подробнее.

Если объявить переменную с ключевым словом local, её область видимости ограничится телом функции. Другими словами, переменная будет доступна только в теле функции.

Наш последний вариант функции `code_to_error` выглядит так:

```
code_to_error()
{
  local _result_variable=$2

  case $1 in
    1)
      eval $_result_variable="'Не найден файл'"
      ;;
    2)
      eval $_result_variable="'Нет прав для чтения файла'"
      ;;
  esac
}
```

Здесь переменная `_result_variable` объявлена как локальная. Это значит, что она доступна для чтения и записи только в теле `code_to_error` и любых вызываемых ею функциях.

В Bash область видимости локальной переменной ограничена временем исполнения функции, в которой она объявлена. Такая область видимости называется **динамической**[338]. В современных языках чаще встречается **лексическая** область видимости. При этом подходе переменная доступна только в теле функции, но не за его пределами (например, в вызываемых функциях).

Локальные переменные не попадают в глобальную область видимости. Это гарантирует, что никакая функция не перезапишет их случайно.

Упражнение 3-14. Область видимости переменных

Какой текст выведет на консоль скрипт из листинга 3-37 после выполнения?

[338] https://ru.wikipedia.org/wiki/Область_видимости#Лексические_vs._динамические_области_видимости

Листинг 3-37. Скрипт для тестирования области видимости переменной

```
#!/bin/bash

bar()
{
  echo "bar1: var = $var"
  var="bar_value"
  echo "bar2: var = $var"
}

foo()
{
  local var="foo_value"

  echo "foo1: var = $var"
  bar
  echo "foo2: var = $var"
}

echo "main1: var = $var"
foo
echo "main2: var = $var"
```

Неосторожное обращение с локальными переменными приводит к ошибкам. Проблема в том, что локальная переменная скрывает глобальную с тем же именем. Рассмотрим пример.

Предположим, вы пишете функцию для обработки файла. Например, она с помощью утилиты grep ищет шаблон в файле. Функция выглядит так:

```
check_license()
{
  local filename="$1"
  grep "General Public License" "$filename"
}
```

Теперь допустим, что в начале скрипта объявлена глобальная переменная с именем `filename`. Например:

```
#!/bin/bash

filename="$1"
```

Выполнится ли функция `check_license` корректно? Да выполнится, благодаря **сокрытию глобальной переменной**. Сокрытие работает так. При обращении к имени `filename` в теле функции Bash подставит локальную переменную, а не глобальную. Это происходит потому, что локальная переменная объявлена позже глобальной. Из-за сокрытия в теле функции нельзя получить доступ к глобальной переменной `filename`.

Случайное сокрытие переменных приводит к ошибкам. Старайтесь исключить саму возможность такой ситуации. Для этого добавляйте префикс или постфикс для имён локальных переменных. Например, символ подчёркивания в конец имени.

Глобальная переменная становится недоступна в теле функции только после объявления локальной переменной с тем же именем. Рассмотрим следующий вариант функции `check_license`:

```
#!/bin/bash

filename="$1"

check_license()
{
  local filename="$filename"
  grep "General Public License" "$filename"
}
```

Здесь локальная переменная `filename` инициализируется значением глобальной переменной с тем же именем. Причина в том, что подстановка переменных выполняется до операции присваивания. То есть в момент присваивания подставляется значение параметра скрипта `$1`. Например, если в скрипт передать имя файла `README`, то присваивание выглядит так:

```
  local filename="README"
```

В Bash начиная с версии 4.2 изменилось ограничение области видимости массивов. Если объявить индексируемый или ассоциативный массив в функции, он по умолчанию попадёт в локальную область видимости. Чтобы объявить массив глобальным, используйте опцию `-g` команды declare.

Вот пример объявления локального массива `files`:

```
check_license()
{
  declare files=(Documents/*.txt)
  grep "General Public License" "$files"
}
```

В следующем примере массив `files` попадёт в глобальную область видимости:

```
check_license()
{
  declare -g files=(Documents/*.txt)
  grep "General Public License" "$files"
}
```

Мы познакомились с функциями в Bash. Вот общие рекомендации по их использованию:

1. Тщательно выбирайте имена для функций. Каждое имя сообщает читателю кода, что делает функция.
2. В функциях объявляйте только локальные переменные. Используйте соглашение об их именовании. Так вы решите конфликт имён локальных и глобальных переменных.
3. Не используйте глобальные переменные в функциях. Вместо этого передавайте значение глобальной переменной в функцию через параметр.
4. Не используйте ключевое слово function при объявлении функций. Оно есть в Bash, но отсутствует в POSIX-стандарте.

Рассмотрим подробнее последний совет. Следующий вариант объявления функции не рекомендуется:

```
function check_license()
{
  declare files=(Documents/*.txt)
  grep "General Public License" "$files"
}
```

Ключевое слово function полезно только в одном случае. Оно решает конфликт между именем функции и псевдонимом (alias).

Например, следующее объявление функции не заработает без слова function:

```
alias check_license="grep 'General Public License'"

function check_license()
{
  declare files=(Documents/*.txt)
  grep "General Public License" "$files"
}
```

После такого объявления функцию можно вызвать, если поставить перед ней слэш . Например, так:

```
\check_license
```

Без слэша Bash подставит значение псевдонима:

```
check_license
```

В скриптах имена псевдонимов и функций конфликтуют редко. Каждый скрипт запускается в новом процессе Bash. В нём нет пользовательских alias из файла `.bashrc`. Конфликт имён может произойти по ошибке в режиме командного интерпретатора.

Пакетный менеджер

Мы познакомились со встроенными командами Bash и со стандартным набором GNU-утилит. Эти утилиты устанавливаются в Unix-окружение по умолчанию. Их возможностей может оказаться недостаточно для решения вашей задачи. Эта проблема решается установкой дополнительных программ и утилит.

Подходы к установке программ в Windows и Unix-окружение отличаются. Рассмотрим, как правильно устанавливать и обновлять ПО в любом Unix-окружении или Linux дистрибутиве.

Репозиторий

ПО устанавливается в Unix-окружение из **репозитория**[339] (software repository). Репозиторием называется сервер-хранилище всех доступных приложений. Эти приложения собираются **мейнтейнерами** из исходного кода открытого ПО. Большинство мейнтейнеров — это добровольцы и энтузиасты свободного ПО.

Каждое приложение в репозитории хранится в виде файла. Эти файлы имеют специальный формат. Например, deb, RPM, zst и т.д. Разные Linux-дистрибутивы используют разные форматы. Файл с приложением называется **пакетом** (package). Пакет — это модуль для установки ПО в систему.

Репозиторий хранит пакеты с приложениями и библиотеками. Кроме этого в репозитории есть метаинформация обо всех пакетах. Она хранится в одном или нескольких файлах. Эта метаинформация называется **индексом пакетов**.

Устанавливать пакеты в Unix-окружение можно сразу из нескольких репозиториев. Например, один репозиторий предлагает новые версии пакетов, а другой их специальные сборки. В зависимости от требований можно выбрать из какого репозитория устанавливать пакет.

Работа с пакетами

Для работы с репозиторием Unix-окружение предлагает специальную программу. Она называется **пакетным менеджером**[340] (package manager).

Зачем нужен пакетный менеджер? Для примера в Windows его нет. Пользователи этой ОС вручную загружают программы из интернета и устанавливают их.

[339] https://help.ubuntu.ru/wiki/репозиторий
[340] https://ru.wikipedia.org/wiki/Система_управления_пакетами

Пакетный менеджер устанавливает и удаляет пакеты в Unix-окружении. При этом его главная задача — отслеживать **зависимости пакетов**. Предположим, что программа из одного пакета использует возможности программы или библиотеки из другого. Тогда говорят, что первый пакет зависит от второго.

Зависимость пакетов предотвращает многократную установку одного и того же приложения или библиотеки в систему. Вместо этого нужные пакеты устанавливаются однократно. Все зависящие от них программы знают место их установки на диске и используют их совместно.

Устанавливайте приложения в Unix-окружение или ОС Linux только через пакетный менеджер. Единственное исключение из этого правила — проприетарные программы. Их приходится устанавливать вручную. Обычно такие программы распространяются в одном пакете. Он включает все необходимые зависимости (библиотеки и приложения). В таких случаях отслеживать зависимости не нужно и без пакетного менеджера можно обойтись.

Алгоритм установки пакета из репозитория выглядит так:

1. Загрузить из репозитория индекс пакетов.
2. Найти нужное приложение или библиотеку в индексе пакетов.
3. Загрузить на локальный компьютер пакет с приложением или библиотекой из репозитория.
4. Установить загруженный пакет.

Пакетный менеджер выполняет все эти шаги. Вам нужно только знать его интерфейс и вызывать с правильными параметрами.

Окружение MSYS2 использует пакетный менеджер pacman[341]. Он создавался для дистрибутива Arch Linux. Менеджер pacman работает с пакетами простого формата. Чтобы собирать приложения и библиотеки в такие пакеты, специальные навыки и опыт не нужны.

Рассмотрим команды для работы с репозиторием на примере менеджера pacman.

Чтобы загрузить из репозитория индекс пакетов, выполните команду:

```
pacman -Syy
```

Следующая команда найдёт пакет в загруженном индексе по ключевому слову:

```
pacman -Ss КЛЮЧЕВОЕ_СЛОВО
```

Предположим, вы ищете утилиту для работы с документами MS Word. Тогда следующая команда найдёт подходящий для этого пакет:

[341] https://wiki.archlinux.org/index.php/Pacman_(Русский)

```
pacman -Ss word
```

В списке результатов будут два пакета:

- mingw-w64-i686-antiword
- mingw-w64-x86_64-antiword

Это сборки утилиты antiword для 32-разрядных и 64-разрядных систем. Утилита antiword конвертирует документы MS Word в текстовый формат.

Для установки пакета выполните команду:

```
pacman -S ИМЯ_ПАКЕТА
```

Чтобы установить утилиту antiword, выполните следующее:

```
pacman -S mingw-w64-x86_64-antiword
```

В результате pacman установит antiword и все пакеты, необходимые для его работы.

Теперь утилита запускается следующей командой:

```
antiword
```

Вы установили пакет в систему. Если он стал ненужным, удалите его. При этом все зависимости пакета тоже будут удалены, если ими не пользуются другие приложения. Для удаления пакета выполните команду:

```
pacman -Rs ИМЯ_ПАКЕТА
```

Чтобы удалить утилиту antiword, выполните команду:

```
pacman -Rs mingw-w64-x86_64-antiword
```

Предположи, вы установили в систему несколько пакетов. Через некоторое время в репозитории появятся их новые версии. Вы решаете обновить свои пакеты до новых версий. Для этого выполните команду:

```
pacman -Syu
```

В результате все установленные в систему пакеты обновятся до текущих версий в репозитории.

Мы рассмотрели основные команды pacman. Другие пакетные менеджеры работают по такому же принципу. Для установки и удаления пакетов они выполняют те же действия, что и pacman. Единственное их отличие в названии и параметрах командной строки.

Таблица 4-1 демонстрирует команды для работы с пакетами в различных дистрибутивах Linux.

Таблица 4-1. Команды для работы с пакетами

Команда	MSYS2 и Arch Linux	Ubuntu	CentOS	Fedora
Получить индекс пакетов	`pacman -Syy`	`apt-get update`	`yum check-update`	`dnf check-update`
Поиск пакета по ключевому слову	`pacman -Ss КЛЮЧЕВОЕ_СЛОВО`	`apt-cache search КЛЮЧЕВОЕ_СЛОВО`	`yum search КЛЮЧЕВОЕ_СЛОВО`	`dnf search КЛЮЧЕВОЕ_СЛОВО`
Установить пакет из репозитория	`pacman -S ИМЯ_ПАКЕТА`	`apt-get install ИМЯ_ПАКЕТА`	`yum install ИМЯ_ПАКЕТА`	`dnf install ИМЯ_ПАКЕТА`
Установить пакета из локального файла	`pacman -U ИМЯ_ФАЙЛА`	`dpkg -i ИМЯ_ФАЙЛА`	`yum install ИМЯ_ФАЙЛА`	`dnf install ИМЯ_ФАЙЛА`
Удалить установленный пакет	`pacman -Rs ИМЯ_ПАКЕТА`	`apt-get remove ИМЯ_ПАКЕТА`	`yum remove ИМЯ_ПАКЕТА`	`dnf erase ИМЯ_ПАКЕТА`
Обновить все установленные пакеты	`pacman -Syu`	`apt-get upgrade`	`yum update`	`dnf upgrade`

Заключение

На этом завершается наше знакомство с Bash. Мы рассмотрели только азы языка. Многие темы остались за рамками этой книги. Среди них:

- Работа со строками[342].
- Регулярные выражения[343].
- Потоковый редактор sed[344].
- Язык обработки входного потока awk[345].

Эти темы важны. Они — материал для углубленного изучения. Знать их необязательно, если вы применяете Bash для решения простых задач и элементарной автоматизации. Изучите эти темы, если собираетесь писать на Bash сложные приложения.

Возможно, вам понравилось программировать. Вы нашли это занятие полезным. Что делать дальше после прочтения этой книги?

Прежде всего признаем, что Bash — это не **язык программирования общего назначения**. Под этим термином понимают языки для разработки приложений в различных прикладных областях. В таких языках нет конструкций, подходящих только для одной области и бесполезных в другой.

Bash считается **предметно-ориентированным**[346] языком. Значит ли это, что он бесполезен? Вовсе нет. Bash — это мощный вспомогательный инструмент разработчика ПО. Сегодня он применяется для интеграции больших программных проектов, задач тестирования, сборки программ и автоматизации. При этом вы не найдёте коммерческий проект, написанный только на Bash. Этот язык хорошо справляется с задачами, для которых он создавался. Но во многих прикладных областях он уступает современным языкам общего назначения.

Языки программирования не создаются просто так для развлечения. Автор каждого из них сталкивался с прикладными задачами. Для их решения существующие на тот момент языки не подходили. Поэтому новый язык ориентировался на конкретный тип задач. Это привело к тому, что сильные стороны современных языков общего назначения проявляются только в одной или нескольких прикладных областях. Не удивительно, что прикладная область определяет подходяще ей языки.

[342] https://www.opennet.ru/docs/RUS/bash_scripting_guide/x4171.html
[343] https://www.opennet.ru/docs/RUS/bash_scripting_guide/c11895.html
[344] https://www.opennet.ru/docs/RUS/bash_scripting_guide/a14586.html#AEN14605
[345] https://www.opennet.ru/docs/RUS/bash_scripting_guide/x14802.html
[346] https://ru.wikipedia.org/wiki/Предметно-ориентированный_язык

Итак, вы прочитали эту книгу. Теперь пришло время выбрать интересную вам прикладную область. Почитайте в интернете статьи. Подумайте: какая областью разработки ПО вам подходит? Только ответив на этот вопрос, выбирайте язык программирования для дальнейшего изучения.

Таблица 5-1 послужит отправной точкой для знакомства с прикладными областями. Помимо областей в ней перечислены используемые в них языки.

Таблица 5-1. Прикладные области разработки ПО

Прикладная область	Языки программирования
Приложения для мобильных устройств[347]	Java, C, C++, HTML5, JavaScript
Web-разработка[348] (фронтэнд[349])	JavaScript, PHP, HTML5, CSS, SQL
Web-разработка[350] (бэкэнд[351])	JavaScript, PHP, Ryby, Perl, C#, Java, Go
Высоконагруженное серверное ПО[352]	C++, Python, Ruby, SQL
Системное администрирование[353]	Bash, Python, Perl, Ruby
Встраиваемые системы[354]	C, C++, Ассемблер
Машинное обучение[355] и анализ данных[356]	Python, Java, C++
Информационная безопасность[357]	C, C++, Python, Bash
Корпоративное ПО[358]	Java, C#, C++, SQL
Компьютерные игры[359]	C++

Чтобы стать высококлассным специалистом, недостаточно знать язык программирования. Обязательно владеть технологиями, которые применяются в этой области. Например, специалист по информационной безопасности должен разбираться в устройстве сетей и операционных систем. С профессиональным ростом к вам придёт понимание, какие технологии надо изучать.

Предположим, вы определились с прикладной областью и языком программирования. Те-

[347] https://ru.wikipedia.org/wiki/Мобильное_приложение
[348] https://ru.wikipedia.org/wiki/Веб-приложение
[349] https://ru.wikipedia.org/wiki/Фронтенд_и_бэкенд
[350] https://ru.wikipedia.org/wiki/Веб-приложение
[351] https://ru.wikipedia.org/wiki/Фронтенд_и_бэкенд
[352] https://ru.wikipedia.org/wiki/Сервер_(программное_обеспечение)
[353] https://ru.wikipedia.org/wiki/Системный_администратор
[354] https://ru.wikipedia.org/wiki/Встраиваемая_система
[355] https://ru.wikipedia.org/wiki/Машинное_обучение
[356] https://ru.wikipedia.org/wiki/Анализ_данных
[357] https://ru.wikipedia.org/wiki/Информационная_безопасность
[358] https://ru.qwe.wiki/wiki/Enterprise_software
[359] https://ru.wikipedia.org/wiki/Компьютерная_игра

перь пришло время записаться на бесплатный онлайн-курс. Эта книга познакомила вас с основами программирования. Поэтому изучение нового языка пойдёт быстрее. Конструкции Python или C++ покажутся вам знакомыми по Bash. Однако, у этих языков есть концепции, которые придётся изучить с нуля. Не теряйте мотивации, применяйте новые знания на практике и учитесь на своих ошибках. Только так вы добьётесь результата.

Надеюсь, что из этой книги вы узнали нечто новое и приятно провели время за чтением. Если книга вам понравилась, поделитесь ею с друзьями. Также я буду благодарен, если вы уделите мне несколько минут и напишете отзыв на Goodreads[360].

Если у вас остались вопросы или появились замечания по книге, пишите мне на почту petrsum@gmail.com[361]. Также задавайте вопросы в разделе “Issues” GitHub репозитория книги[362].

Спасибо вам за то, что прочитали “Программирование на Bash с нуля”!

[360] https://www.goodreads.com/book/show/53883360-bash
[361] mailto:petrsum@gmail.com
[362] https://github.com/ellysh/bash-programming-from-scratch-ru/issues

Благодарности

Ни одна книга не пишется в одиночку. Эту книгу мне помогли написать несколько людей. Начиная с общей идеи и заканчивая конкретными замечаниями. Этих людей я хочу поблагодарить.

Спасибо Софии Каюновой за пожелание научиться программировать. Думаю, тогда у меня впервые появилась идея написать самоучитель для друзей.

Спасибо Виталию Липатову за то, что познакомил меня с Linux и Bash. Именно он заложил фундамент моих профессиональных навыков.

Спасибо Руслану Пясецкому за консультации по Bash. Он открыл для меня идиомы и подводные камни языка.

Также спасибо всем, кто поддерживал меня и мотивировал довести эту работу до конца.

Список терминов

А

Абстракция[363] (abstraction[364]) — программный модуль, приложение или библиотека, которые повторяют основные свойства реального объекта. Абстракции помогают управлять сложностью программных систем. Они скрывают несущественные детали. Также они позволяют работать с разными объектами по одному алгоритму.

Алгоритм[365] (algorithm[366]) — это конечная последовательность инструкций, понятных исполнителю. Задача алгоритма — что-то вычислить или решить задачу.

Аргумент (argument[367]) — слово или строка, которые передаются в программу через интерфейс командной строки. Пример аргументов:

```
grep "GNU" README.txt
```

Асинхронность (asynchrony[368]) — означает события, происходящие независимо от основного потока выполнения программы. Так же под асинхронностью понимаются методы обработки таких событий.

Б

Библиотека[369] (library[370]) — набор подпрограмм и объектов, собранных в самостоятельный модуль или файл. Приложения используют ресурсы библиотек как составные части.

В

Встроенные команды (builtin commands[371]) — команды, которые интерпретатор исполняет самостоятельно. Для их выполнения не нужны сторонние утилиты. Пример встроенной команды - pwd.

[363] https://ru.wikipedia.org/wiki/Уровень_абстракции_(программирование)
[364] https://en.wikipedia.org/wiki/Abstraction_(computer_science)
[365] https://ru.wikipedia.org/wiki/Алгоритм
[366] https://en.wikipedia.org/wiki/Algorithm
[367] http://linuxcommand.org/lc3_wss0120.php
[368] https://en.wikipedia.org/wiki/Asynchrony_(computer_programming)
[369] https://ru.wikipedia.org/wiki/Библиотека_(программирование)
[370] https://en.wikipedia.org/wiki/Library_(computing)
[371] https://www.gnu.org/software/bash/manual/html_node/Shell-Builtin-Commands.html

Вычислительный процесс[372] (process[373]) — экземпляр компьютерной программы, который исполняется процессором.

Д

Дистрибутив Linux[374] (**Linux distribution**[375]) — операционная система, основанная на ядре Linux и наборе приложений GNU[376]. ОС собирается из пакетов с помощью пакетного менеджера. Она представляет собой набор готовых для работы программ и библиотек.

Дочерний процесс (child process[377]) — процесс, порождённый другим процессом (родительским).

З

Зарезервированные переменные (reserved variable) — то же что и переменные оболочки (shell variables).

И

Идиома[378] (idiom[379]) — способ выражения типовой конструкции в языке программирования. Идиома представляет собой шаблон реализации алгоритма или структуры данных на конкретном языке программирования. Вот идиома в Bash для обработки списка файлов в цикле for:

```
for file in ./*.txt
do
  cp "$file" ~/Documents
done
```

Интерпретатор[380] (interpreter[381]) — программа, которая исполняет инструкции. Инструкции пишутся на языке программирования. Предварительная компиляция исходного кода для этого не требуется.

Итерация[382] (iteration) — однократное исполнение команд в теле цикла.

[372] https://ru.wikipedia.org/wiki/Процесс_(информатика)
[373] https://en.wikipedia.org/wiki/Process_(computing)
[374] https://ru.wikipedia.org/wiki/Дистрибутив_Linux
[375] https://en.wikipedia.org/wiki/Linux_distribution
[376] https://ru.wikipedia.org/wiki/Список_пакетов_GNU
[377] https://en.wikipedia.org/wiki/Child_process
[378] https://ru.wikipedia.org/wiki/Идиома_(программирование)
[379] https://en.wikipedia.org/wiki/Programming_idiom
[380] https://ru.wikipedia.org/wiki/Интерпретатор
[381] https://en.wikipedia.org/wiki/Interpreter_(computing)
[382] https://ru.wikipedia.org/wiki/Итерация_(программирование)

К

Команда (command) — текст, введённый после приглашения интерпретатора. Этот текст соответствует действию, которое интерпретатор выполняет самостоятельно или с помощью другого приложения.

Компилятор[383] (compiler[384]) — программа для перевода текста программы с языка программирования в машинный код.

Компьютерная программа[385] (computer program[386]) — набор инструкций, которые может исполнить компьютер. Каждая программа решает прикладную задачу.

Конвейер[387] (pipeline[388]) — механизм взаимодействия процессов в Unix-подобных ОС. Он строится на передаче сообщений. Также конвейером называют два и более процесса со связанными потоками ввода-вывода. Поток вывода одного процесса передаётся напрямую в поток ввода другого и так далее.

Л

Литерал[389] (literal[390]) — условное обозначение в исходном коде программы. Оно представляет собой фиксированное значение. В зависимости от типа данных литералы записываются по-разному. Большинство языков программирования поддерживают литералы для целых чисел, чисел с плавающей точкой[391] и строк. Пример строкового литерала (`~/Documents`) в Bash:

```
1 var="~/Documents"
```

Логическое выражение[392] (Boolean expression[393]) — конструкция языка программирования. В результате её вычисления получается либо значение “истина”, либо “ложь”.

Логический оператор[394] (logical operator[395]) — операция над логическими выражениями. Она комбинирует их в одно выражение. Результат операции зависит от значений исходных выражений.

[383] https://ru.wikipedia.org/wiki/Компилятор
[384] https://en.wikipedia.org/wiki/Compiler
[385] https://ru.wikipedia.org/wiki/Компьютерная_программа
[386] https://en.wikipedia.org/wiki/Computer_program
[387] https://ru.wikipedia.org/wiki/Конвейер_(Unix)
[388] https://en.wikipedia.org/wiki/Pipeline_(Unix)
[389] https://ru.wikipedia.org/wiki/Литерал_(информатика)
[390] https://en.wikipedia.org/wiki/Literal_(computer_programming)
[391] https://ru.wikipedia.org/wiki/Число_с_плавающей_запятой
[392] https://ru.wikipedia.org/wiki/Логическое_выражение
[393] https://en.wikipedia.org/wiki/Boolean_expression
[394] https://ru.wikipedia.org/wiki/Логическая_операция
[395] https://en.wikipedia.org/wiki/Logical_connective

М

Массив[396] (array[397]) — структура данных, состоящая из набора элементов. Расположение каждого элемента определяет порядковый номер. В памяти компьютера элементы массива хранятся последовательно друг за другом.

Многозадачность[398] (multitasking[399]) — параллельное выполнение нескольких задач (процессов) за определённый отрезок времени. Это достигается за счёт переключения компьютера (переключение контекста[400]) между задачами и выполнения их по частям.

Мультипрограммирование[401] (multiprogramming[402]) — распределение нагрузки компьютера между несколькими программами. Например, компьютер выполняет программу до тех пор, пока ей не потребуется некоторый ресурс. Если ресурс занят, программа останавливается. Компьютер переключается на другую программу. Он вернётся к выполнению первой программы, когда нужный ей ресурс освободится.

О

Область видимости[403] (scope block[404]) — часть программы или системы, в которой имя переменной остаётся связанным с её значением. Другими словами имя переменной корректно конвертируется в адрес памяти, по которому хранится её значение. За пределами области видимости то же имя может указывать на другую область памяти.

Операнд[405] (operand[406]) — аргумент математической операции или команды. Он представляет собой данные для обработки. Например, в следующей операции сложения операнды — это числа 1 и 4:

```
1 + 4
```

Оператор перенаправления (redirection operator[407]) — специальная конструкция языков Bash и Bourne shell, которая перенаправляет потоки ввода-вывода для встроенных команд, утилит и приложений. Как источник и цель перенаправления указываются файловые дескрипторы. Они связаны с файлами или стандартными потоками. Пример перенаправления вывода утилиты find в файл `result.txt`:

[396] https://ru.wikipedia.org/wiki/Массив_(тип_данных)
[397] https://en.wikipedia.org/wiki/Array_data_structure
[398] https://ru.wikipedia.org/wiki/Многозадачность
[399] https://en.wikipedia.org/wiki/Computer_multitasking
[400] https://ru.wikipedia.org/wiki/Переключение_контекста
[401] https://ru.wikipedia.org/wiki/Мультипрограммирование
[402] https://www.geeksforgeeks.org/difference-between-multitasking-multithreading-and-multiprocessing/
[403] https://ru.wikipedia.org/wiki/Область_видимости
[404] https://en.wikipedia.org/wiki/Scope_(computer_science)
[405] https://ru.wikipedia.org/wiki/Операнд
[406] https://en.wikipedia.org/wiki/Operand
[407] https://www.gnu.org/software/bash/manual/html_node/Redirections.html#Redirections

```
find / -path */doc/* -name README 1> result.txt
```

Опции интерпретатора (shell options[408]) — настройки, которые меняют поведение интерпретатора в режимах оболочки и исполнения скриптов. Настройки задаются встроенной командой set. Например, вот команда включения отладочного вывода интерпретатора:

```
set -x
```

Опция (**option**[409]) — аргумент в стандартизированной форме, который передаётся в программу. Опция начинается с тире - или двойного тире –. Она переключает режим работы программы. Следующие друг за другом опции можно объединить в одну группу. Вот пример объединения опций `-l`, `-a` и `-h` утилиты ls:

```
ls -lah
```

П

Парадигма программирования[410] (programming paradigm[411]) — это набор идей, методов и принципов, которые определяют способ написания программ.

Параметр (parameter[412]) — сущность, которая хранит какое-то значение. Параметр в отличие от переменной может не иметь имени.

Параметр интерпретатора (shell parameter[413]) — именованная область памяти интерпретатора для хранения данных.

Параметр командной строки (command line parameter[414]) — вид аргумента команды. Он передаёт информацию в программу. Параметр также может быть часть опции. Например, чтобы указать выбранный режим работы.
Вот вызов утилиты find:

```
find ~/Documents -name README
```

Её первый параметр `~/Documents` сообщает путь начала поиска. Второй параметр `README` относится к опции `-name`.

Переменная[415] (variable[416]) — 1) это область памяти, обращаться к которой можно по имени.

[408] https://www.tldp.org/LDP/abs/html/options.html
[409] http://linuxcommand.org/lc3_wss0120.php
[410] https://ru.wikipedia.org/wiki/Парадигма_программирования
[411] https://en.wikipedia.org/wiki/Programming_paradigm
[412] http://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Shell-Parameters
[413] http://mywiki.wooledge.org/BashGuide/Parameters
[414] https://stackoverflow.com/a/36495940/6562278
[415] https://ru.wikipedia.org/wiki/Переменная_(программирование)
[416] https://www.gnu.org/software/bash/manual/html_node/Shell-Parameters.html#Shell-Parameters

2. в Bash это параметр, доступный по имени. Переменные задаются пользователем или интерпретатором. Пример объявления переменной:

```
filename="README.txt"
```

Переменные оболочки (shell variables[417]) — переменные, которые устанавливает интерпретатор. В них хранятся временные данные, настройки и состояния ОС или Unix-окружения. Пользователь может читать значения переменных оболочки. Для записи доступны только некоторые из них. Выводятся командой set. Пример — переменная `PATH`.

Переменные окружения (environment variables[418]) — неупорядоченный набор переменных, который копируется из родительского процесса в дочерний. Утилита env изменяет переменные окружения при запуске программы. При вызове её без параметров, она выводит переменные, объявленные в текущем командном интерпретаторе.

Подпрограмма[419] (subroutine[420]) — фрагмент программы, который выполняет одну задачу. Фрагмент выделяется в самостоятельный блок кода. Его можно вызвать из любого места программы.

Позиционные параметры (positional parameters[421]) — параметры со всеми аргументы командной строки, которые Bash-скрипт получил при вызове. Имена параметров соответствуют порядку аргументов. Пример использования позиционного параметра в скрипте:

```
cp "$1" ~
```

Порядок выполнения[422] (control flow[423]) — порядок выполнения инструкций программы и вызова функций в процессе её работы.

Приглашение командной строки (prompt[424]) — это последовательность символов. Командный интерпретатор выводит её, когда готов обработать следующую команду пользователя.

P

Разделение времени[425] (time-sharing[426]) — использование ресурсов компьютера несколькими пользователями одновременно. Достигается за счёт многозадачности и мультипрограммирования.

[417] http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_02.html
[418] http://mywiki.wooledge.org/Environment
[419] https://ru.wikipedia.org/wiki/Подпрограмма
[420] https://en.wikipedia.org/wiki/Subroutine
[421] https://www.gnu.org/software/bash/manual/html_node/Shell-Parameters.html#Shell-Parameters
[422] https://ru.wikipedia.org/wiki/Порядок_выполнения
[423] https://en.wikipedia.org/wiki/Control_flow
[424] https://www.gnu.org/software/bash/manual/html_node/Controlling-the-Prompt.html#Controlling-the-Prompt
[425] https://ru.wikipedia.org/wiki/Разделение_времени
[426] https://en.wikipedia.org/wiki/Time-sharing

Расширение файла[427] (filename extension) — часть имени файла. Добавляется в конец имени через точку. Расширение определяет тип файла.

Рекурсия[428] (recursion[429]) — вызов функции из неё же самой (простая рекурсия) или через другие функции (косвенная рекурсия).

С

Связанный список[430] (linked list[431]) — структура данных, состоящая из элементов или узлов. Порядок их размещения в списке не совпадает с порядком следования в памяти. Поэтому каждый узел содержит данные и адрес памяти следующего узла. Такая организация списка делает эффективными операции вставки и удаления.

Сетевой протокол[432] (communication protocol[433]) — соглашение о формате сообщений между узлами компьютерной сети.

Символьная ссылка[434] (symbolic link[435]) — файл специального типа. Вместо данных он содержит указатель на другой файл или каталог.

Синхронный (synchronous[436]) — обозначает события или действия, которые происходят в одном потоке выполнения программы.

Специальные параметры (special parameters[437]) — устанавливаются интерпретатором для хранения своего состояния, передачи параметров в вызываемые приложения (позиционные параметры) и чтения их кода возврата. Специальные параметры доступны только для чтения. Пример такого параметра — `$?`.

Стандартные потоки[438] (standard streams[439]) — программные каналы коммуникации приложения с окружением, в котором оно работает. Потоки — это абстракции физических каналов ввода с клавиатуры и вывода на экран монитора. Обращение к каналу происходит по дескриптору, который назначается ОС.

У

Управляющая последовательность[440] (escape sequence[441]) — набор символов, которые не

[427] https://ru.wikipedia.org/wiki/Расширение_имени_файла
[428] https://ru.wikipedia.org/wiki/Рекурсия#В_программировании
[429] https://en.wikipedia.org/wiki/Recursion_(computer_science)
[430] https://ru.wikipedia.org/wiki/Связный_список
[431] https://en.wikipedia.org/wiki/Linked_list
[432] https://ru.wikipedia.org/wiki/Протокол_передачи_данных
[433] https://en.wikipedia.org/wiki/Communication_protocol
[434] https://ru.wikipedia.org/wiki/Символическая_ссылка
[435] https://en.wikipedia.org/wiki/Symbolic_link
[436] https://en.wiktionary.org/wiki/synchronous
[437] http://mywiki.wooledge.org/BashGuide/Parameters
[438] https://ru.wikipedia.org/wiki/Стандартные_потоки
[439] https://en.wikipedia.org/wiki/Standard_streams
[440] https://ru.wikipedia.org/wiki/Управляющая_последовательность
[441] https://en.wikipedia.org/wiki/Escape_sequence

имеют собственного значения. Вместо этого они управляют устройством вывода. Например, символ перевода строки `\n` даёт команду устройству вывода начать новую строку.

Управляющие символы[442] — другое название для управляющей последовательности.

Условный оператор[443] (conditional statement[444] или conditional expression) — конструкция языка программирования. Она выбирает набор команд для выполнения в зависимости от значения логического выражения.

Утилита[445] (utility software[446]) — вспомогательная программа для работы с ОС или оборудованием.

Уязвимость[447] (vulnerability[448]) — ошибка или недостаток в системе. Используя уязвимость, можно выполнить несанкционированные действия.

Ф

Файловый дескриптор[449] (file descriptor[450]) — абстрактный указатель на файл или канал коммуникации (поток, конвейер или сетевой сокет). Дескрипторы являются частью POSIX-интерфейса. Они представляют собой целые неотрицательные числа.

Файловая система[451] (file system[452]) — способ хранения и чтения данных с носителей информации.

Функция[453] (function) — другое название для подпрограммы.

Х

Хеш-таблица[454] (hash table[455]) — структура данных. Каждый её элемент — это пара ключ-значение (key-value). Говорят, что хеш-таблица отображает ключи на значения. Роль ключей напоминает индексы элементов в массиве. Ключи рассчитываются хеш-функцией[456].

Хеш-функция[457] (hash function[458]) — функция для генерации уникальной последовательности байтов из переданных на вход данных.

[442] https://ru.wikipedia.org/wiki/Управляющие_символы
[443] https://ru.wikipedia.org/wiki/Ветвление_(программирование)
[444] https://en.wikipedia.org/wiki/Conditional_(computer_programming)
[445] https://ru.wikipedia.org/wiki/Утилита
[446] https://en.wikipedia.org/wiki/Utility_software
[447] https://ru.wikipedia.org/wiki/Уязвимость_(компьютерная_безопасность)
[448] https://en.wikipedia.org/wiki/Vulnerability_(computing)
[449] https://ru.wikipedia.org/wiki/Файловый_дескриптор
[450] https://en.wikipedia.org/wiki/File_descriptor
[451] https://ru.wikipedia.org/wiki/Файловая_система
[452] https://en.wikipedia.org/wiki/File_system
[453] https://ru.wikipedia.org/wiki/Функция_(программирование)
[454] https://ru.wikipedia.org/wiki/Хеш-таблица
[455] https://en.wikipedia.org/wiki/Hash_table
[456] https://ru.wikipedia.org/wiki/Хеш-функция
[457] https://ru.wikipedia.org/wiki/Хеш-функция
[458] https://en.wikipedia.org/wiki/Hash_function

Ш

Шаблон поиска[459] (glob[460]) — поисковый запрос. Вместе с обычными символами в него входят символы подстановки[461]: * и ?. Символы подстановки соответствуют любым символам. Например, шаблон `R*M?`. Он соответствует строкам, которые начинаются с R и предпоследняя буква которых M.

Шебанг[462] (shebang[463]) — последовательность из символов решётки и восклицательного знака #! в начале скрипта. Загрузчик программ рассматривает строку после шебанг как имя интерпретатора. Дальше загрузчик запускает интерпретатор и передаёт ему скрипт на выполнение. Пример шебанг для Bash-скриптов:

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

Я

Язык программирования общего назначения (general-purpose programming language[464]) — язык, на котором можно разрабатывать приложения для разных прикладных областей. В нём нет конструкций, полезных для одной области и бесполезных в других.

A

alias[465] (псевдоним) — встроенная команда Bash для сокращения длинных строк. Применяется в режиме командного интерпретатора.

Application Programming Interface[466] или API (интерфейс прикладного программирования[467]) — набор соглашений о взаимодействии компонентов информационной системы. Соглашения отвечают на следующие вопросы:

- Какую функцию выполнит вызываемый компонент?
- Какие данные передать на вход функции?
- Какие данные функция возвращает?

Arithmetic Expansion[468] (арифметическая подстановка) — в Bash вычисление арифметического выражения и подстановка его результата. Например:

[459] https://ru.wikipedia.org/wiki/Шаблон_поиска
[460] https://en.wikipedia.org/wiki/Glob_(programming)
[461] https://ru.wikipedia.org/wiki/Символ_подстановки
[462] https://ru.wikipedia.org/wiki/Шебанг_(Unix)
[463] https://en.wikipedia.org/wiki/Shebang_(Unix)
[464] https://en.wikipedia.org/wiki/General-purpose_programming_language
[465] https://www.gnu.org/software/bash/manual/html_node/Aliases.html#Aliases
[466] https://en.wikipedia.org/wiki/Application_programming_interface
[467] https://ru.wikipedia.org/wiki/API
[468] https://www.gnu.org/software/bash/manual/html_node/Arithmetic-Expansion.html#Arithmetic-Expansion

```
echo $((4+3))
```

ASCII-кодировка[469] — восьмибитная система кодировки символов. Включает в себя:

- десятичные цифры
- латинский алфавит
- национальный алфавит
- знаки препинания
- управляющие символы

B

Background[470] (фоновый режим) — в Bash режим исполнения процесса. В этом режиме его идентификатор не относится к группе идентификаторов[471] процесса терминала. Исполняемый процесс не обрабатывает прерывания клавиатуры.

Bash[472] (Bourne again shell) — интерпретатор командной строки, разработанный Брайаном Фоксом. Bash заменил интерпретатор Bourne shell в Linux-дистрибутивах и некоторых проприетарных Unix-системах. Bash совместим с POSIX-стандартом. Некоторые его расширения стандартом не предусмотрены.

Bash-скрипт (**Bash script**[473]) — текстовый файл, содержащий команды интерпретатора. Bash исполняет скрипты в не интерактивном режиме.

Best Practices[474] (хорошая практика) — рекомендованные приемы использования языка программирования или какой-то технологии. Пример для языка Bash — заключение строк в двойные кавычки, чтобы избежать word splitting.

Bottleneck[475] (букв. бутылочное горло, узкое место) — компонент или ресурс информационной системы, который ограничивает её производительность или пропускную способность.

Bourne shell[476] — интерпретатор командной строки, разработанный Стивеном Борном. Он заменил оригинальный интерпретатор Кена Томпсона[477] в Unix версии 7[478]. Все функции Bourne shell соответствуют POSIX-стандарту[479]. Но некоторые упомянутые в стандарте возможности отсутствуют.

[469] https://ru.wikipedia.org/wiki/ASCII
[470] https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html#Job-Control-Basics
[471] https://en.wikipedia.org/wiki/Process_group
[472] https://en.wikipedia.org/wiki/Bash_(Unix_shell)
[473] https://www.gnu.org/software/bash/manual/html_node/Shell-Scripts.html#Shell-Scripts
[474] http://mywiki.wooledge.org/BashGuide/Practices
[475] https://en.wikipedia.org/wiki/Bottleneck_(software)
[476] https://en.wikipedia.org/wiki/Bourne_shell
[477] https://en.wikipedia.org/wiki/Thompson_shell
[478] https://en.wikipedia.org/wiki/Version_7_Unix
[479] https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html

Brace Expansion[480] (подстановка фигурных скобок) — в Bash генерация слов из заданных частей. Эта функция отсутствует в стандарте POSIX. Например, следующие две команды эквивалентны:

```
cp test.{txt,md,log} Documents
cp test.txt test.md test.log Documents
```

C

Coding Style[481] (стандарт оформления кода[482]) — набор правил и соглашений для написания исходного кода программ. Задача стандарта — помочь нескольким программистам писать, читать и понимать общий исходный код.

Command Substitution[483] (подстановка команды) — подстановка вместо команды её вывода в stdout. Вывод получается после исполнения команды в subshell. Пример:

```
echo "$(date)"
```

E

Endianness[484] (порядок байтов[485]) — порядок байтов при хранении чисел в памяти компьютера. Он определяется свойствами центрального процессора. Сегодня используются порядки от старшего к младшему (big-endian) и от младшего к старшему (little-endian). Некоторые CPU поддерживают оба варианта (bi-endian). Переключение между ними происходит при запуске компьютера. Пример хранения четырёхбайтового числа 0x0A0B0C0D для разных порядков:

```
0A 0B 0C 0D     big-endian
0D 0C 0B 0A     little-endian
```

Error-prone[486] (подверженный ошибкам) — характеристика неудачных приёмов программирования и решений. Эти решения работают корректно в частных случаях, но приводят к ошибкам при определённых входных данных или условиях. Пример error-prone решения — обработка вывода утилиты ls в конвейере:

[480] http://mywiki.wooledge.org/BraceExpansion
[481] https://en.wikipedia.org/wiki/Programming_style
[482] https://ru.wikipedia.org/wiki/Стандарт_оформления_кода
[483] https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html#Command-Substitution
[484] https://en.wikipedia.org/wiki/Endianness
[485] https://ru.wikipedia.org/wiki/Порядок_байтов
[486] https://en.wiktionary.org/wiki/error-prone

```
ls | grep "test"
```

Exit Status[487] (код возврата[488]) — в Bash целочисленное значение от 0 до 255, которое команда возвращает интерпретатору при завершении. Код возврата 0 означает успешное выполнение команды. Все остальные коды указывают на ошибку.

F

Filename Expansion[489] (подстановка имён файлов) — в Bash подстановка имён файлов вместо шаблонов, содержащих символы ?, * и [. Пример:

```
rm -rf *
```

Foreground[490] (активный режим) — в Bash режим исполнения процесса. При этом его идентификатор относится к группе идентификаторов[491] процесса терминала. Исполняемый процесс обрабатывает прерывания клавиатуры.

G

Globbing[492] или glob — в Bash другое название для filename expansion.

I

Input Field Separator[493] или IFS (разделитель поля ввода) — список следующих друг за другом символов. Bash использует их как разделители при обработке вводимых строк (в том числе и для word splitting). По умолчанию это символы пробела, табуляции и перевода строки.

L

Linux-окружение (Linux Environment) — другое название для POSIX-окружения.

P

Parameter Expansion[494] (подстановка параметров или подстановка переменных) — в Bash подстановка вместо имени переменной или параметра его значения. Примеры:

[487] https://www.gnu.org/software/bash/manual/html_node/Exit-Status.html#Exit-Status
[488] https://ru.wikipedia.org/wiki/Код_возврата
[489] https://www.gnu.org/software/bash/manual/html_node/Filename-Expansion.html#Filename-Expansion
[490] https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html#Job-Control-Basics
[491] https://en.wikipedia.org/wiki/Process_group
[492] https://mywiki.wooledge.org/glob?action=show&redirect=globbing
[493] https://mywiki.wooledge.org/IFS
[494] https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html#Shell-Parameter-Expansion

```
echo "$PATH"
echo "${var:-empty}"
```

Portable Operating System Interface[495] или POSIX (переносимый интерфейс операционных систем) — набор стандартов. Они описывают интерфейсы взаимодействия прикладных программ с ОС, командный интерпретатор и интерфейсы утилит. POSIX поддерживает совместимость ОС семейства Unix. Благодаря этому, между ними легче переносить прикладные программы.

POSIX-окружение (POSIX environment) — программная среда полностью или частично совместимая со стандартом POSIX. Для полной совместимости нужна поддержка ядром ОС, командной оболочкой и файловой системой. Для частичной совместимости достаточно окружения наподобие Cygwin[496].

POSIX Shell[497] — стандарт для POSIX-систем с описанием минимального набора функций командного интерпретатора. Если интерпретатор имеет эти функции, он считается POSIX-совместимым. При этом стандарт никак не ограничивает дополнительные возможности и расширения. В основу стандарта легла реализация ksh88 Korn shell[498]. Этот интерпретатор появился позже, чем Bourne shell. Поэтому некоторые функции стандарта POSIX отсутствуют в Bourne shell.

Process Substitution[499] (подстановка процесса) — в Bash аналог подстановки команды. В отличие от неё исполнение происходит асинхронно. При этом ввод и вывод команды привязаны к файлам. Содержимое этих файлов Bash перенаправляет родительскому процессу. Пример:

```
diff <(sort file1.txt) <(sort file2.txt)
```

Q

Quote Removal[500] (удаление кавычек) — подстановка, которую Bash выполняет последней. Она удаляет неэкранированные символы \, ‘ и “, которые не были получены в результате предыдущих подстановок.

S

Short-circuit evaluation[501] (короткое замыкание) — вычисление только тех операндов логического оператора, которые достаточны для вывода значения всего выражения.

[495] https://ru.wikipedia.org/wiki/POSIX
[496] https://ru.wikipedia.org/wiki/Cygwin
[497] https://www.grymoire.com/Unix/Sh.html
[498] https://en.wikipedia.org/wiki/KornShell
[499] https://www.gnu.org/software/bash/manual/html_node/Process-Substitution.html#Process-Substitution
[500] https://www.gnu.org/software/bash/manual/html_node/Quote-Removal.html#Quote-Removal
[501] https://en.wikipedia.org/wiki/Short-circuit_evaluation

Subshell[502] (подоболочка) — способ группирования команд. Команды исполняются в интерпретаторе, запущенном в дочернем процессе. Переменные, определённые в дочернем процессе, не доступны в родительском. Пример выполнения команд в subshell:

```
(ps aux | grep "bash")
```

T

Tilde Expansion[503] (подстановка тильды) — в Bash подстановка вместо символа тильда ~ домашнего каталога пользователя. Путь до домашнего каталога читается из переменной `HOME`.

U

Unix-окружение (Unix environment[504]) — другое название для POSIX-окружения.

W

Word Splitting[505] (разделение слов) — в Bash разделение аргументов командной строки на слова и передача их отдельными параметрами. Символы из переменной `IFS` используются как разделители. Аргументы, заключённые в кавычки, не обрабатываются. Пример:

```
cp file1.txt file2.txt "my file.txt" ~
```

[502] http://mywiki.wooledge.org/SubShell
[503] https://www.gnu.org/software/bash/manual/html_node/Tilde-Expansion.html#Tilde-Expansion
[504] https://ccrma.stanford.edu/guides/planetccrma/Unix.html
[505] https://www.gnu.org/software/bash/manual/html_node/Word-Splitting.html#Word-Splitting

Ответы

Общая информация

Упражнение 1-1. Перевод чисел из BIN в HEX

```
* 10100110100110 = 0010 1001 1010 0110 = 2 9 A 6 = 29A6

* 10110001110101000100 11 = 0010 1100 0111 0101 0001 0011 = 2 C 7 5 1 3 = 2C7513

* 11111011100010010101001100000000110101101 = 1111 1011 1000 1001 0101 0011 0000 0001\
 1010 1101 = F B 8 9 5 3 0 1 A D = FB895301AD
```

Упражнение 1-2. Перевод чисел из HEX в BIN

```
* FF00AB02 = F F 0 0 A B 0 2 = 1111 1111 0000 0000 1010 1011 0000 0010 = 11111111000\
0000010101011000000010

* 7854AC1 = 7 8 5 4 A C 1 = 0111 1000 0101 0100 1010 1100 0001 = 1111000010101001010\
11000001

* 1E5340ACB38 = 1 E 5 3 4 0 A C B 3 8 = 0001 1110 0101 0011 0100 0000 1010 1100 1011\
 0011 1000 = 11110010100110100000010101100101100111000
```

Командный интерпретатор Bash

Упражнение 2-1. Шаблоны поиска

Правильный ответ: `README.md`.

Строка `00_README.txt` не подходит. Согласно шаблону `*ME.??`, после точки идут два символа. В строке `00_README.txt` их три.

В строке `README` нет точки. Поэтому она тоже не подходит.

Упражнение 2-2. Шаблоны поиска

Шаблону поиска `*/doc?openssl*` соответствуют три строки:

- /usr/share/doc/openssl/IPAddressChoice_new.html
- /usr/share/doc_openssl/IPAddressChoice_new.html
- /doc/openssl

Строка doc/openssl не подходит. В ней нет символа / перед doc.

Упражнение 2-3. Поиск файлов утилитой find

Вот команда для поиска текстовых файлов в системных каталогах:

```
find /usr -name "*.txt"
```

Текстовые файлы хранятся только в /usr. Поэтому нет смысла проверять остальные системные каталоги.

Подсчитаем число строк в найденных файлах. Для этого добавим действие с вызовом утилиты wc:

```
find /usr -name "*.txt" -exec wc -l {} +
```

Чтобы найти все текстовые файлы на диске, начните поиск с корневого каталога. Например, так:

```
find / -name "*.txt"
```

Если к этому вызову добавить действие с утилитой wc, произойдёт ошибка. То есть следующая команда не сработает в окружении MSYS2:

```
find / -name "*.txt" -exec wc -l {} +
```

Проблема связана с ошибкой на иллюстрации 2-17. Текст ошибки передаётся утилите wc. Утилита рассматривает каждое полученное на вход слово как путь до файла. Текст — это не путь. Поэтому wc завершит работу с ошибкой.

Упражнение 2-4. Поиск файлов утилитой grep

Ищите информацию о лицензиях приложений в системном каталоге с документацией /usr/share/doc.

В документации на приложение с лицензией GNU General Public License[506] встречается строка “General Public License”. Найдём такие документы следующей командой:

[506] https://ru.wikipedia.org/wiki/GNU_General_Public_License

```
grep -Rl "General Public License" /usr/share/doc
```

Также проверим файлы каталога `/usr/share/licenses`:

```
grep -Rl "General Public License" /usr/share/licenses
```

В окружении MSYS2 есть два дополнительных каталога установки приложений: `/mingw32` и `/mingw64`. Они не соответствуют POSIX-стандарту. Проверим установленные в них программы следующими командами:

```
grep -Rl "General Public License" /mingw32/share/doc
grep -Rl "General Public License" /mingw64/share
```

Чтобы найти приложения с лицензией MIT, подойдёт строка поиска “MIT license”. Для Apache лицензии — строка “Apache license”, а для BSD — “BSD license”.

Упражнение 2-6. Работа с файлами и каталогами

Для начала создайте каталоги для каждого года и месяца. Например, так:

```
mkdir -p ~/photo/2019/11
mkdir -p ~/photo/2019/12
mkdir -p ~/photo/2020/01
```

Предположим, что фотографии хранятся в каталоге `D:\Photo`. С помощью утилиты find найдём там файлы, созданные в ноябре 2019 года. Чтобы проверить дату создания файла, используйте параметр `-newermt`. Например:

```
find /d/Photo -type f -newermt 2019-11-01 ! -newermt 2019-12-01
```

Эта команда ищет файлы в каталоге `/d/Photo`. Он соответствует пути `D:\Photo` в Windows-окружении.

Первое выражение `-newermt 2019-11-01` означает искать только файлы, изменённые начиная с 1 ноября 2019 года. За ним следует выражение `! -newermt 2019-12-01`. Оно исключает из результата файлы, модифицированные начиная с 1 декабря 2019 года. Восклицательный знак перед выражением — это отрицание. Между выражениями нет условия. Но утилита find подставит логическое И по умолчанию. В результате получится выражение: “файлы, созданные после 1 ноября 2019 года, но не позднее 30 ноября 2019 года”. Другими словами — “файлы за ноябрь месяц”.

Команда поиска файлов готова. Добавим к ней действие копирования. Получим следующее:

```
find /d/Photo -type f -newermt 2019-11-01 ! -newermt 2019-12-01 -exec cp {} ~/photo/\
2019/11 \;
```

Эта команда скопирует файлы за ноябрь 2019 года в каталог ~/photo/2019/11.

Вот аналогичные команды для копирования файлов за декабрь и январь:

```
find /d/Photo -type f -newermt 2019-12-01 ! -newermt 2020-01-01 -exec cp {} ~/photo/\
2019/12 \;
find /d/Photo -type f -newermt 2020-01-01 ! -newermt 2020-02-01 -exec cp {} ~/photo/\
2020/01 \;
```

Предположим, что файлы в каталоге D:\Photo не нужны. Тогда заменим копирование на переименование. Получим такие команды:

```
find /d/Photo -type f -newermt 2019-11-01 ! -newermt 2019-12-01 -exec mv {} ~/photo/\
2019/11 \;
find /d/Photo -type f -newermt 2019-12-01 ! -newermt 2020-01-01 -exec mv {} ~/photo/\
2019/12 \;
find /d/Photo -type f -newermt 2020-01-01 ! -newermt 2020-02-01 -exec mv {} ~/photo/\
2020/01 \;
```

Обратите внимание на масштабируемость нашего решения. Количество файлов в каталоге D:\Photo неважно. Чтобы разбить их на три месяца, нужно три команды.

Упражнение 2-7. Конвейеры и перенаправление потоков ввода-вывода

Выясним, как работает утилита bsdtar. Вызовите её с опцией --help. На экран выведется справка по опциям и параметрам. Из справки следует, что утилита создаст архив каталога, если передать ей опции -c и -f. После опций идёт имя архива. Вот пример вызова утилиты:

```
bsdtar -c -f test.tar test
```

Эта команда создаст архив с именем test.tar и содержимым каталога test. Обратите внимание, что команда не сожмёт[507] файлы. То есть архив займёт столько же места на диске, сколько и собранные в него файлы.

Цели операций архивирования и сжатия разные. Архивирование нужно для хранения и копирования большого числа файлов. Сжатие уменьшает объём, занимаемый данными на диске. Часто эти операции совмещают в одну.

Чтобы создать архив и сжать его, добавьте в вызов bsdtar опцию -j. Например, так:

[507] https://ru.wikipedia.org/wiki/Сжатие_данных

```
bsdtar -c -j -f test.tar.bz2 test
```

Опции `-c`, `-j` и `-f` можно объединить в одну группу. Получится следующее:

```
bsdtar -cjf test.tar.bz2 test
```

Напишем команду для прохода по каталогу фотографий. Для каждого месяца она создаст отдельный архив.

Следующий вызов утилиты find найдёт каталоги с месяцами:

```
find ~/photo -type d -path */2019/* -o -path */2020/*
```

Вывод этой команды перенаправим на вход утилиты xargs. Она сформирует вызов bsdtar. Получится такая команда:

```
find ~/photo -type d -path */2019/* -o -path */2020/* | xargs -I% bsdtar -cf %.tar %
```

Чтобы bsdtar сжимала файлы, добавьте опцию `-j`. Получим:

```
find ~/photo -type d -path */2019/* -o -path */2020/* | xargs -I% bsdtar -cjf %.tar.\
bz2 %
```

Сжатие длится дольше чем архивация.

Мы передаём параметр `-I` утилите xargs. Он указывает место подстановки аргументов в сформированную команду. В вызове утилиты `bsdtar` таких мест два: имя создаваемого архива и путь до обрабатываемого каталога.

Не забывайте про имена файлов с символами перевода строки. Чтобы обработать их корректно, добавим опцию `-print0` в вызов утилиты find. Получим:

```
find ~/photo -type d -path */2019/* -o -path */2020/* -print0 | xargs -0 -I% bsdtar \
-cjf %.tar.bz2 %
```

Предположим, что файлы в архивах должны храниться без относительных путей (например `2019/11`). Для удаления путей используйте опцию bsdtar `--strip-components`. Например, так:

```
find ~/photo -type d -path */2019/* -o -path */2020/* -print0 | xargs -0 -I% bsdtar \
--strip-components=3 -cjf %.tar.bz2 %
```

Упражнение 2-8. Логические операторы

Реализуем алгоритм по шагам. Первое действие — копирование файла `README` в домашний каталог пользователя. Это делает следующая команда:

```
cp /usr/share/doc/bash/README ~
```

С помощью оператора && и echo выведем результат команды в лог-файл. Получим:

```
cp /usr/share/doc/bash/README ~ && echo "cp - OK" > result.log
```

Для архивации файла вызовем утилиту bsdtar или tar. Например, так:

```
bsdtar -cjf ~/README.tar.bz2 ~/README
```

Результат утилиты выведем в лог-файл с помощью оператора && и echo:

```
bsdtar -cjf ~/README.tar.bz2 ~/README && echo "bsdtar - OK" >> result.log
```

Теперь команда echo дописывает строку в конец существующего лог-файла.

Объединим вызовы утилит cp и bsdtar в одну команду. Утилита bsdtar вызывается только после успешного копирования файла `README`. Чтобы добиться такой зависимости, поставим между командами оператор &&. Получим:

```
cp /usr/share/doc/bash/README ~ && echo "cp - OK" > result.log && bsdtar -cjf ~/READ\
ME.tar.bz2 ~/README && echo "bsdtar - OK" >> result.log
```

Добавим последнее действие — удаление файла `README`:

```
cp /usr/share/doc/bash/README ~ && echo "cp - OK" > ~/result.log && bsdtar -cjf ~/RE\
ADME.tar.bz2 ~/README && echo "bsdtar - OK" >> ~/result.log && rm ~/README && echo "\
rm - OK" >> ~/result.log
```

Запустите эту команду. Если она выполнится без ошибок, в лог-файл запишется следующее:

```
cp - OK
bsdtar - OK
rm - OK
```

Команда с вызовами трёх утилит подряд выглядит громоздко. Её неудобно читать и редактировать. Разобьём команду на строки. Для этого есть несколько способов.

Способ первый — перенос строк после логических операторов. Применим его и получим следующее:

```
cp /usr/share/doc/bash/README ~ && echo "cp - OK" > ~/result.log &&
bsdtar -cjf ~/README.tar.bz2 ~/README && echo "bsdtar - OK" >> ~/result.log &&
rm ~/README && echo "rm - OK" >> ~/result.log
```

Попробуйте скопировать эту команду в окно терминала и исполнить. Она выполнится без ошибок.

Второй способ разбить команду на строки — использовать символ обратный слэш . Сразу после него поставьте перенос строки. Применяйте этот способ, когда в команде нет логических операторов.

Для примера поставим обратные слэши перед операторами && в нашей команде. Получим:

```
cp /usr/share/doc/bash/README ~ && echo "cp - OK" > ~/result.log \
&& bsdtar -cjf ~/README.tar.bz2 ~/README && echo "bsdtar - OK" >> ~/result.log \
&& rm ~/README && echo "rm - OK" >> ~/result.log
```

Разработка Bash-скриптов

Упражнение 3-2. Полная форма подстановки параметров

Утилита find рекурсивно ищет файлы, начиная с указанного пути. Используйте параметр `-maxdepth`, чтобы исключить из поиска подкаталоги.

Команда поиска TXT файлов в текущем каталоге выглядит так:

```
find . -maxdepth 1 -type f -name "*.txt"
```

Добавим действие для копирования найденных файлов в домашний каталог пользователя. Получим:

```
find . -maxdepth 1 -type f -name "*.txt" -exec cp -t ~ {} \;
```

Создайте скрипт с именем `txt-copy.sh`. Скопируйте в него команду поиска.

В скрипт будем передавать параметр. В зависимости от него, выбирается действие: копирование или переименование. В качестве параметра удобнее передать имя утилиты: cp или mv. Скрипт вызовет утилиту по имени для каждого файла, найденного find.

Рассмотрим интерфейс скрипта `txt-copy.sh`. Копирование выполняется следующей командой:

```
./txt-copy.sh cp
```

Команда для переименования файлов такая:

```
./txt-copy.sh mv
```

Первый параметр скрипта сохраняется в переменной `$1`. Подставим её в вызов утилиты find. Получится следующее:

```
find . -maxdepth 1 -type f -name "*.txt" -exec "$1" -t ~ {} \;
```

Если действие не указано, скрипт должен копировать файлы. То есть допустим такой вызов:

```
./txt-copy.sh
```

Чтобы это сработало, добавим в подстановку параметра `$1` значение по умолчанию. Получим скрипт `find-txt.sh` из листинга 5-1.

Листинг 5-1. Скрипт для поиска TXT-файлов

```
#!/bin/bash

find . -maxdepth 1 -type f -name "*.txt" -exec "${1:-cp}" -t ~ {} \;
```

Упражнение 3-4. Оператор if

Исходная команда выглядит так:

```
( grep -RlZ "123" target | xargs -0 cp -t . && echo "cp - OK" || ! echo "cp - FAILS"\
 ) && ( grep -RLZ "123" target | xargs -0 rm && echo "rm - OK" || echo "rm - FAILS" \
)
```

Обратите внимание на отрицание вызова echo “cp - FAILS”. Из-за него утилита grep вызовется второй раз только, если первый вызов выполнился успешно.

Заменим логический оператор && между вызовами grep на конструкцию if-else. Получится следующее:

```
if grep -RlZ "123" target | xargs -0 cp -t .
then
  echo "cp - OK"
  grep -RLZ "123" target | xargs -0 rm && echo "rm - OK" || echo "rm - FAILS"
else
  echo "cp - FAILS"
fi
```

Теперь заменим операторы || во втором вызове grep на if-else. Получим:

```
if grep -RlZ "123" target | xargs -0 cp -t .
then
  echo "cp - OK"
  if grep -RLZ "123" target | xargs -0 rm
  then
    echo "rm - OK"
  else
    echo "rm - FAILS"
  fi
else
  echo "cp - FAILS"
fi
```

Чтобы избежать вложенных конструкций if-else, применим технику раннего возврата. Также добавим в начале скрипта шебанг. Листинг 5-2 демонстрирует результат.

Листинг 5-2. Скрипт для поиска стоки в файлах

```
#!/bin/bash

if ! grep -RlZ "123" target | xargs -0 cp -t .
then
  echo "cp - FAILS"
  exit 1
fi

echo "cp - OK"

if grep -RLZ "123" target | xargs -0 rm
then
  echo "rm - OK"
else
  echo "rm - FAILS"
fi
```

Упражнение 3-5. Оператор [[

Сравним содержимое двух каталогов. Результатом сравнения будет список файлов, которыми они различаются.

Для начала надо пройти по всем файлам каждого каталога. Применим утилиту find. Поиск файлов в каталоге `dir1` выглядит так:

```
find dir1 -type f
```

Вот пример вывода этой команды:

```
dir1/test3.txt
dir1/test1.txt
dir1/test2.txt
```

Мы получили список файлов в каталоге `dir1`. Проверим, что каждый из них есть в каталоге `dir2`. Для этого добавим действие `-exec` в вызов find.

Есть одна проблема. Утилита find добавляет имя каталога `dir1` к каждому найденному файлу. Чтобы исключить `dir1` в списке файлов, перейдём в этот каталог перед запуском find. Получим такие команды:

```
1 cd dir1
2 find . -type f
```

Теперь вывод find выглядит так:

```
./test3.txt
./test1.txt
./test2.txt
```

С помощью действия `-exec` и команды test проверим, что каждый найденный файл есть в каталоге `dir2`. Получим:

```
1 cd dir1
2 find . -type f -exec test -e ../dir2/{} \;
```

Здесь мы используем команду test вместо оператора [[. Дело в том, что встроенный интерпретатор find не способен обработать этот оператор корректно. Это одно из исключений, когда [[надо заменить на test. В общем случае предпочитайте оператор [[.

Если файла не оказалось в каталоге `dir2`, выведем его имя на экран. Для этого инвертируем проверку test и добавим второе действие `-exec` с выводом echo. Между действиями поставим логический оператор И. В результате получим следующие команды:

```
cd dir1
find . -type f -exec test ! -e ../dir2/{} \; -a -exec echo {} \;
```

Добавим аналогичный вызов find для проверки файлов каталога `dir2` в каталоге `dir1`.

Листинг 5-3 демонстрирует полный скрипт `dir-diff.sh` для сравнения каталогов.

Листинг 5-3. Скрипт для сравнения каталогов

```
#!/bin/bash

cd dir1
find . -type f -exec test ! -e ../dir2/{} \; -a -exec echo {} \;

cd ../dir2
find . -type f -exec test ! -e ../dir1/{} \; -a -exec echo {} \;
```

Мы написали скрипт сравнения каталогов для учебных целей. Не используйте его для реальных задач. Предпочитайте специальную GNU-утилиту diff.

Упражнение 3-6. Оператор case

Скрипт переключения между конфигурационными файлами будет создавать **символьные ссылки**[508]. Символьная ссылка — это файл специального типа. Вместо данных он содержит указатель на другой файл или каталог.

В Unix-окружении, запущенном на OS Windows, создать символьную ссылку нельзя. Вместо этого утилита ln копирует файл или каталог, соответствующий ссылке. На ОС Linux и macOS утилита работает правильно.

Символьные ссылки удобны, когда нужен доступ к файлу или каталогу из разных мест файловой системы. Открыв ссылку на файл, вы редактируете файл, на который она указывает. Так же работает ссылка на каталог. Любые изменения отразятся на целевом каталоге.

Напишем алгоритм скрипта переключения между файлами конфигурации. Он выглядит так:

1. Удалить существующую символьную ссылку или файл по пути `~/.bashrc`.
2. Проверить опцию командной строки, переданную в скрипт.
3. В зависимости от опции создать символьную ссылку на файл `.bashrc-home` или `.bashrc-work`.

Реализуем этот алгоритм с помощью оператора case. Листинг 5-4 демонстрирует результат.

[508] https://ru.wikipedia.org/wiki/Символическая_ссылка

Листинг 5-4. Скрипт для переключения конфигурационных файлов

```
#!/bin/bash

file="$1"

rm ~/.bashrc

case "$file" in
  "h")
    ln -s ~/.bashrc-home ~/.bashrc
    ;;

  "w")
    ln -s ~/.bashrc-work ~/.bashrc
    ;;

  *)
    echo "Указана недопустимая опция"
    ;;
esac
```

Конфигурационный файл выбираем в зависимости от переданного в скрипт параметра `$1`.

Команды вызова утилиты ln отличаются только именем файла. Такое подобие подсказывает, что оператор case можно заменить на ассоциативный массив. Тогда получится скрипт, похожий на листинг 5-5.

Листинг 5-5. Скрипт для переключения конфигурационных файлов

```
#!/bin/bash

option="$1"

declare -A files=(
  ["h"]="~/.bashrc-home"
  ["w"]="~/.bashrc-work")

if [[ -z "$option" || ! -v files["$option"] ]]
then
  echo "Указана недопустимая опция"
  exit 1
fi

rm ~/.bashrc
```

```
16
17 ln -s "${files["$option"]}" ~/.bashrc
```

Рассмотрим последнюю строку скрипта. Двойные кавычки при подстановке элемента массива здесь не обязательны. Но они предотвратят ошибку, если в будущем появится имя конфигурационного файла с пробелами.

Упражнение 3-7. Арифметические действия в дополнительном коде

Результаты сложения однобайтовых целых:

```
* 79 + (-46) = 0100 1111 + 1101 0010 = 1 0010 0001 -> 0010 0000 = 33

* -97 + 96 = 1001 1111 + 0110 0000 = 1111 1111 -> 1111 1110 -> 1000 0001 = -1
```

Результат сложения двухбайтовых целых:

```
* 12868 + (-1219) = 0011 0010 0100 0100 + 1111 1011 0011 1101 = 1 0010 1101 1000 000\
1 -> 0010 1101 1000 0001 = 11649
```

Чтобы проверить правильность перевода чисел в дополнительный код, используйте онлайн-калькулятор[509]

Упражнение 3-8. Modulo и остаток от деления

```
* 1697 % 13
q = 1697 / 13 ~ 130.5385 ~ 130
r = 1697 - 13 * 130 = 7

* 1697 modulo 13
q = 1697 / 13 ~ 130.5385 ~ 130
r = 1697 - 13 * 130 = 7

* 772 % -45
q = 772 / -45 ~ -17.15556 ~ -17
r = 772 - (-45) * (-17) = 7

* 772 modulo -45
q = (772 / -45) - 1 ~ -18.15556 ~ -18
r = 772 - (-45) * (-18) = -38
```

[509] https://planetcalc.com/747/

```
* -568 % 12
q = -568 / 12 ~ -47.33333 ~ -47
r = -568 - 12 * (-47) = -4

* -568 modulo 12
q = (-568 / 12) - 1 ~ -48.33333 ~ -48
r = -568 - 12 * (-48) = 8

* -5437 % -17
q = -5437 / -17 ~ 319.8235 ~ 319
r = -5437 - (-17) * 319 = -14

* -5437 modulo -17
q = -5437 / -17 ~ 319.8235 ~ 319
r = -5437 - (-17) * 319 = -14
```

Упражнение 3-9. Побитовое отрицание

Сначала вычислим побитовое отрицание для беззнаковых двухбайтовых целых.

```
 56 = 0000 0000 0011 1000
~56 = 1111 1111 1100 0111 = 65479

 1018 = 0000 0011 1111 1010
~1018 = 1111 1100 0000 0101 = 64517

 58362 = 1110 0011 1111 1010
~58362 = 0001 1100 0000 0101 = 7173
```

Если операция отрицания выполняется над знаковыми двухбайтовыми целыми, результаты отличаются:

```
 56 = 0000 0000 0011 1000
~56 = 1111 1111 1100 0111 -> 1000 0000 0011 1001 = -57

 1018 = 0000 0011 1111 1010
~1018 = 1111 1100 0000 0101 -> 1000 0011 1111 1011 = -1019
```

Число 58362 нельзя представить как знаковое двухбайтовое целое. Причина в переполнении. Если записать биты числа в переменную такого типа, получим -7174. Перевод этого числа в дополнительный код выглядит так:

```
58362 = 1110 0011 1111 1010 -> 1001 1100 0000 0110 = -7174
```

Теперь выполним побитовое отрицание:

```
  -7174  = 1110 0011 1111 1010
~(-7174) = 0001 1100 0000 0101 = 7173
```

Проверим результаты для знаковых целых с помощью Bash-команд:

```
$ echo $((~56))
-57
$ echo $((~1018))
-1019
$ echo $((~(-7174)))
7173
```

Проверить отрицание двухбайтового беззнакого целого 58362 с помощью Bash нельзя. Интерпретатор сохранит число в знаковом четырёхбайтовом целом. Тогда результат отрицания будет такой:

```
$ echo $((~58362))
-58363
```

Упражнение 3-10. Побитовые И, ИЛИ, исключающее ИЛИ

Вычислим битовые операции для беззнаковых двухбайтовый целых:

```
1122 & 908 = 0000 0100 0110 0010 & 0000 0011 1000 1100 = 0000 0000 000 0000 = 0

1122 | 908 = 0000 0100 0110 0010 | 0000 0011 1000 1100 = 0000 0111 1110 1110 = 2030

1122 ^ 908 = 0000 0100 0110 0010 ^ 0000 0011 1000 1100 = 0000 0111 1110 1110 = 2030


49608 & 33036 = 1100 0001 1100 1000 & 1000 0001 0000 1100 = 1000 0001 0000 1000 = 33\
032

49608 | 33036 = 1100 0001 1100 1000 | 1000 0001 0000 1100 = 1100 0001 1100 1100 = 49\
612

49608 ^ 33036 = 1100 0001 1100 1000 ^ 1000 0001 0000 1100 = 0100 0000 1100 0100 = 16\
580
```

Если целые знаковые, результаты битовых операций для первой пары чисел 1122 и 908 такие же. Для второй пары, вычисление отличается. Рассмотрим его.

Сначала получим значение чисел 49608 и 33036 в дополнительном коде:

```
49608 = 1100 0001 1100 1000 -> 1011 1110 0011 1000 = -15928

33036 = 1000 0001 0000 1100 -> 1111 1110 1111 0100 = -32500
```

Теперь выполним битовые операции:

```
-15928 & -32500 = 1100 0001 1100 1000 & 1000 0001 0000 1100 = 1000 0001 0000 1000 ->\
 1111 1110 1111 1000 = -32504

-15928 | -32500 = 1100 0001 1100 1000 | 1000 0001 0000 1100 = 1100 0001 1100 1100 ->\
 1011 1110 0011 0100 = -15924

-15928 ^ -32500 = 1100 0001 1100 1000 ^ 1000 0001 0000 1100 = 0100 0000 1100 0100 = \
16580
```

Вот Bash-команды для проверки результатов:

```
$ echo $((1122 & 908))
0
$ echo $((1122 | 908))
2030
$ echo $((1122 ^ 908))
2030

$ echo $((49608 & 33036))
33032
$ echo $((49608 | 33036))
49612
$ echo $((49608 ^ 33036))
16580

$ echo $((-15928 & -32500))
-32504
$ echo $((-15928 | -32500))
-15924
$ echo $((-15928 ^ -32500))
16580
```

Упражнение 3-11. Битовые сдвиги

Вычисление битовых сдвигов:

```
* 25649 >> 3 = 0110 0100 0011 0001 >> 3 = 0110 0100 0011 0 = 0000 1100 1000 0110 = 3\
206

* 25649 << 2 = 0110 0100 0011 0001 << 2 = 10 0100 0011 0001 -> 1001 0000 1100 0100 -\
> 1110 1111 0011 1100 = -28476

* -9154 >> 4 = 1101 1100 0011 1110 >> 4 = 1101 1100 0011 -> 1111 1101 1100 0011 -> 1\
000 0010 0011 1101 = -573

* -9154 << 3 = 1101 1100 0011 1110 << 3 = 1 1100 0011 1110 -> 1110 0001 1111 0000 ->\
 1001 1110 0001 0000 = -7696
```

Bash-команды для проверки результатов:

```
$ echo $((25649 >> 3))
3206
$ echo $((25649 << 2))
102596
$ echo $((-9154 >> 4))
-573
$ echo $((-9154 << 3))
-73232
```

Результаты Bash-команд отличаются для второго и четвертого сдвига. Причина в том, что Bash хранит все числа в восьми батах.

Проверьте свои расчёты с помощью онлайн-калькулятора[510].

Упражнение 3-12. Операторы цикла

Чтобы отгадать число, игроку даётся семь попыток. Каждая попытка обрабатывается по одному и тому же алгоритму. Поместим его в цикл for.

Алгоритм обработки действия игрока следующий:

1. Прочитать ввод с помощью команды read.
2. Сравнить введённое число с загаданным.
3. Если игрок ошибся, вывести подсказку и перейти к шагу 1.

[510] https://onlinetoolz.net/bitshift

4. Если игрок угадал число, завершить работу скрипта.

Чтобы загадать случайное число, обратимся к зарезервированной переменной RANDOM. При чтении она возвращает случайное значение от 0 до 32767. Нам нужно число от 1 до 100. Получим его из RANDOM по такому алгоритму:

1. Получим случайное число от 0 до 99. Для этого вычислим остаток от деления RANDOM на 100 по формуле:

```
number=$((RANDOM % 100))
```

2. Получим число в диапазоне от 1 до 100. Для этого прибавим единицу к предыдущему результату.

Полная формула вычисления случайного числа от 1 до 100 выглядит так:

```
number=$((RANDOM % 100 + 1))
```

Листинг 5-6 демонстрирует скрипт, выполняющий алгоритм игры.

Листинг 5-6. Скрипт игры Больше-Меньше

```
#!/bin/bash

number=$((RANDOM % 100 + 1))

for i in {1..7}
do
  echo "Введите число:"

  read input

  if (( input < number))
  then
    echo "Число $input меньше искомого"
  elif (( number < input))
  then
    echo "Число $input больше искомого"
  else
    echo "Вы отгадали число"
    exit 0
  fi
done

echo "Вы не отгадали число"
```

Чтобы отгадать число за семь попыток, примените **двоичный поиск**[511]. Его идея в разделении массива чисел на половины. Рассмотрим пример игры “Больше-Меньше” и двоичного поиска.

Как только игра началась мы отгадываем число в диапазоне от 1 до 100. Середина этого диапазона — число 50. Введите это значение первым. Программа даст подсказку, в какой половине диапазона находится загаданное число. Предположим, программа ответила, что 50 меньше искомого числа. Это означает, что искать надо в диапазоне от 50 до 100. Введём середину этого диапазона, то есть число 75. Получаем ответ, что 75 тоже меньше искомого. Вывод — искомое число находится между 75 и 100. Середина этого диапазона X рассчитывается так:

```
X = 75 + (100 - 75) / 2 = 87.5
```

Округлите результат в большую или меньшую сторону. Это неважно. Округлим в меньшую и получим число 87 для следующего ввода. Если число до сих пор не отгадано, продолжайте делить диапазон предполагаемых чисел пополам. Тогда вам хватит семь шагов, чтобы найти загаданное число.

Упражнение 3-13. Функции

Мы рассмотрели вариант функции `code_to_error` в примерах раздела “Функции в скриптах”.

Объединим код функций `print_error` и `code_to_error` в один файл. Получим скрипт из листинга 5-7.

Листинг 5-7. Скрипт для вывода сообщений об ошибках

```
#!/bin/bash

code_to_error()
{
  case $1 in
    1)
      echo "Не найден файл"
      ;;
    2)
      echo "Нет прав для чтения файла"
      ;;
  esac
}

print_error()
{
```

[511] https://ru.wikipedia.org/wiki/Двоичный_поиск

```
  echo "$(code_to_error $1) $2" >> debug.log
}

print_error 1 "readme.txt"
```

Сейчас функция `code_to_error` выводит сообщения на русском языке. Переименуем её на `code_to_error_ru`. Тогда язык сообщений станет понятен из имени функции.

Добавим в скрипт функцию `code_to_error_en`. Она печатает текст на английском языке для переданного в неё кода ошибки. Код функции выглядит так:

```
code_to_error_en()
{
  case $1 in
    1)
      echo "File not found:"
      ;;
    2)
      echo "Permission to read the file denied:"
      ;;
  esac
}
```

Теперь надо выбрать, какую функцию `code_to_error` вызывать из `print_error`. Проверим региональные настройки в переменной окружения `LANG`. Если значение переменной соответствует шаблону “ru_RU*”, вызовем функцию `code_to_error_ru`. В противном случае вызовем `code_to_error_en`.

Полный код скрипта приведён в листинге 5-8.

Листинг 5-8. Скрипт для вывода сообщений об ошибках

```
#!/bin/bash

code_to_error_ru()
{
  case $1 in
    1)
      echo "Не найден файл"
      ;;
    2)
      echo "Нет прав для чтения файла"
      ;;
  esac
```

```
}

code_to_error_en()
{
  case $1 in
    1)
      echo "File not found:"
      ;;
    2)
      echo "Permission to read the file denied:"
      ;;
  esac
}

print_error()
{
  if [[ "$LANG" == ru_RU* ]]
  then
    echo "$(code_to_error_ru $1) $2" >> debug.log
  else
    echo "$(code_to_error_en $1) $2" >> debug.log
  fi
}

print_error 1 "readme.txt"
```

Оператор if в функции `print_error` можно заменить на case. Например, так:

```
print_error()
{
  case $LANG in
    ru_RU*)
      echo "$(code_to_error_ru $1) $2" >> debug.log
      ;;
    en_US*)
      echo "$(code_to_error_en $1) $2" >> debug.log
      ;;
    *)
      echo "$(code_to_error_en $1) $2" >> debug.log
      ;;
  esac
}
```

Вариант с case удобнее, если надо поддерживать более двух языков.

Сейчас в функции `print_error` код дублируется. В каждом блоке оператора case вызывается одинаковая команда echo. Единственное различие между блоками — имя функции для конвертирования кода ошибки в текст. Чтобы избежать дублирования, сохраним имя функции в переменной `func`. Затем подставим эту переменную в вызов echo. Получится следующее:

```
print_error()
{
  case $LANG in
    ru_RU)
      local func="code_to_error_ru"
      ;;
    en_US)
      local func="code_to_error_en"
      ;;
    *)
      local func="code_to_error_en"
      ;;
  esac

  echo "$($func $1) $2" >> debug.log
}
```

Альтернативное решение проблемы дублирования кода — использовать индексируемый массив. Заменим операторы case в функциях `code_to_error_ru` и `code_to_error_en` на массивы. Например, так:

```
code_to_error_ru()
{
  declare -a messages

  messages[1]="Не найден файл"
  messages[2]="Нет прав для чтения файла"

  echo "${messages[$1]}"
}

code_to_error_en()
{
  declare -a messages

  messages[1]="The following file was not found:"
```

```
  messages[2]="You do not have permissions to read the following file:"

  echo "${messages[$1]}"
}
```

Можно упростить код и обойтись без функций `code_to_error`. Объединим сообщения на всех языках в один ассоциативный массив. Поместим его в функцию `print_error`. Ключами массива будут комбинации значения переменной `LANGUAGE` и кода ошибки. Получим такую функцию `print_error` как в листинге 5-9.

Листинг 5-9. Скрипт для вывода сообщений об ошибках

```
#!/bin/bash

print_error()
{
  declare -A messages

  messages["ru_RU",1]="Не найден файл"
  messages["ru_RU",2]="Нет прав для чтения файла"

  messages["en_US",1]="File not found:"
  messages["en_US",2]="Permission to read the file denied:"

  echo "${messages[$LANGUAGE,$1]} $2" >> debug.log
}

print_error 1 "readme.txt"
```

Упражнение 3-14. Область видимости переменных

Скрипт из листинга 3-37 выведет на консоль следующий текст:

```
main1: var =
foo1: var = foo_value
bar1: var = foo_value
bar2: var = bar_value
foo2: var = bar_value
main2: var =
```

Начнём с вывода “main1” и “main2”. Переменная `var` объявлена в функции `foo` как локальная. Поэтому она доступна только в `foo` и в вызываемой из неё функции `bar`. Следовательно, до и после вызова `foo` переменная `var`считается необъявленной.

В выводе “foo1” напечаталось значение `foo_value`. Оно было присвоено переменной только что.

Далее идёт вывод “bar1”. Переменная `var` объявлена в функции `foo`, а `bar` вызывается из неё. Поэтому тело функции `bar` тоже является областью видимости `var`.

Затем мы присваиваем `var` новое значение `bar_value`. Обратите внимание, что мы не объявляем новую глобальную переменной с именем `var`. Мы перезаписываем уже существующую локальную переменную. Её значение `bar_value` мы получим в выводах “bar2” и “foo2”.

Ссылки на ресурсы

Общая информация

На русском языке

- Цикл статей по истории электронных компьютеров[512].
- История ОС Unix[513].
- История ОС Linux[514].
- История ОС от Apple[515].
- История мейнфреймов[516].
- Лекция “История вычислительных машин”[517].
- Лекция “Ретроспектива развития вычислительной техники”[518].
- Статья “Статическая и динамическая типизация”[519].

На английском языке

- Цикл статей по истории электронных компьютеров[520]
- История ОС от Apple[521].
- Статья “Статическая и динамическая типизация”[522]

Bash

На русском языке

- Перевод книги Machtelt Garrels “Руководство по Bash для начинающих”[523].
- Перевод книги Mendel Cooper “Advanced Bash-Scripting Guide”[524] (“Искусство программирования на языке сценариев командной оболочки”).

[512] https://habr.com/ru/post/408611
[513] https://habr.com/ru/post/147774
[514] https://habr.com/ru/post/95646
[515] https://habr.com/ru/post/198016
[516] http://www.thg.ru/cpu/mainframe_history/index.html
[517] https://postnauka.ru/video/86550
[518] https://postnauka.ru/video/38501
[519] https://habr.com/ru/post/308484
[520] https://technicshistory.com/2017/08/29/the-electronic-computers-part-1-prologue
[521] http://web.archive.org/web/20180702193510/http://kernelthread.com/publications/appleoshistory/
[522] https://www.destroyallsoftware.com/compendium/types?share_key=baf6b67369843fa2
[523] http://rus-linux.net/nlib.php?name=/MyLDP/BOOKS/Bash-Guide-1.12-ru/bash-guide-index.html
[524] https://www.opennet.ru/docs/RUS/bash_scripting_guide

- Шпаргалка по Bash командам[525].
- Подводные камни Bash[526].

На английском языке

- Книга Machtelt Garrels “Руководство по Bash для начинающих”[527]
- Книга Mendel Cooper “Advanced Bash-Scripting Guide”[528]
- Шпаргалка по Bash командам[529]
- Bash Reference Manual[530]
- Wiki ресурс[531] по использованию Bash.
- Руководство по Bash[532].
- Подводные камни Bash[533].
- Примеры неправильного использования GNU-утилит Useless Use of Cat Award[534].
- Регулярные выражения в утилите grep[535].
- Best practices по использованию Bash[536].
- Bash coding style на основе mywiki.wooledge.org[537].
- Bash coding style от Google[538].
- Сервис для анализа Bash-команд[539].
- Инструкция по локализации Bash-скриптов[540]

Unix-окружение

На русском языке

- Книга Алексея Федосеева “Введение в администрирование UNIX”[541].
- Книга Эрика Реймонда “Искусство программирования в Unix”[542].

[525] https://tproger.ru/translations/bash-cheatsheet
[526] https://habr.com/ru/company/mailru/blog/311762/
[527] https://tldp.org/LDP/Bash-Beginners-Guide/html/
[528] http://tldp.org/LDP/abs/html
[529] https://github.com/NisreenFarhoud/Bash-Cheatsheet
[530] https://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents
[531] https://wiki.bash-hackers.org
[532] http://mywiki.wooledge.org/BashGuide
[533] http://mywiki.wooledge.org/BashPitfalls
[534] http://www.smallo.ruhr.de/award.html
[535] https://www.cyberciti.biz/faq/grep-regular-expressions
[536] http://mywiki.wooledge.org/BashGuide/Practices
[537] https://github.com/bahamas10/bash-style-guide
[538] https://google.github.io/styleguide/shellguide.html
[539] https://explainshell.com/#
[540] https://mywiki.wooledge.org/BashFAQ/098
[541] http://heap.altlinux.org/modules/unix_base_admin.dralex/index.html
[542] https://ru.wikipedia.org/wiki/Философия_Unix#Реймонд:_Искусство_программирования_в_Unix