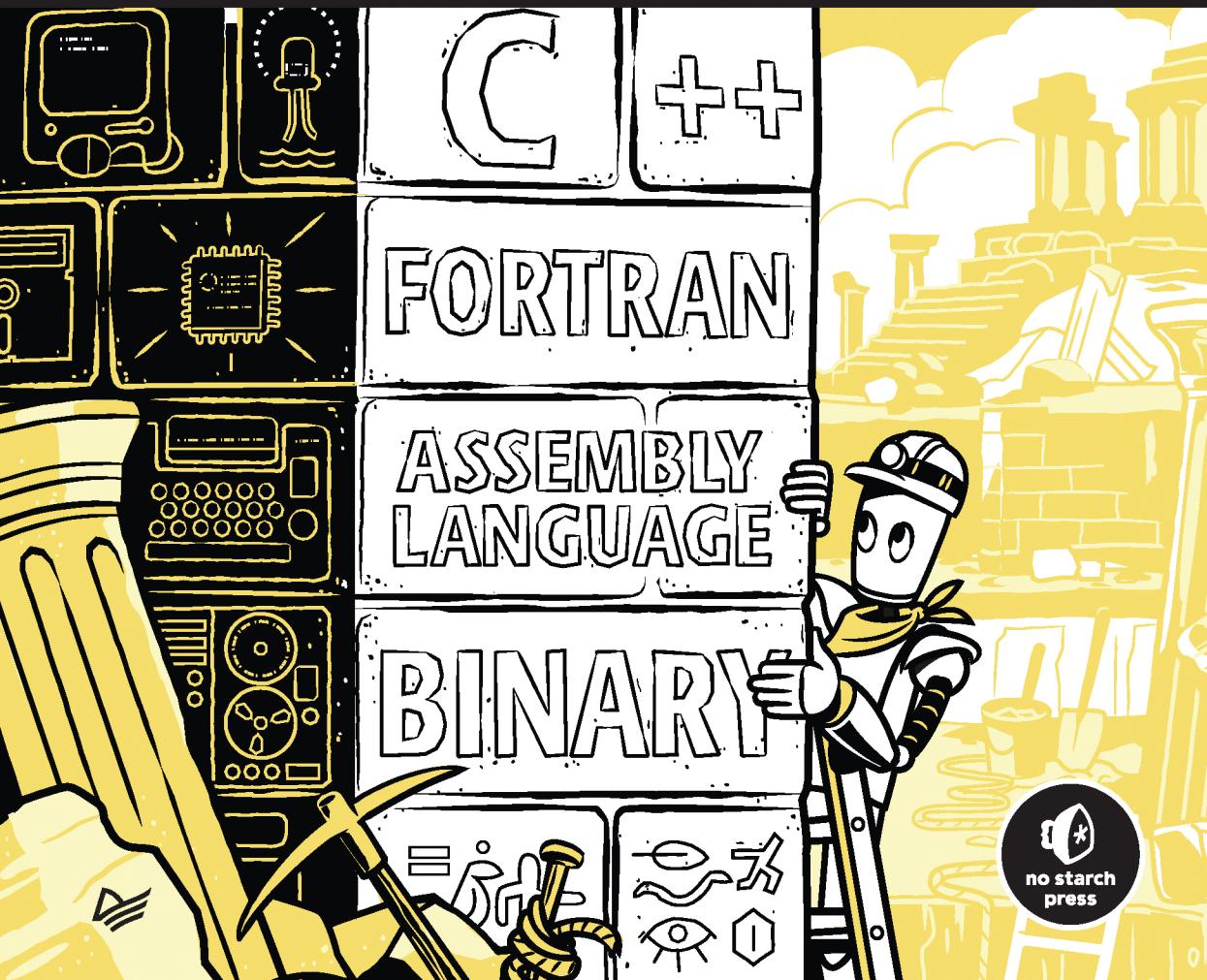


ТАЙНАЯ ЖИЗНЬ ПРОГРАММ

КАК СОЗДАТЬ КОД,
КОТОРЫЙ ПОНРАВИТСЯ ВАШЕМУ КОМПЬЮТЕРУ

ДЖОНАТАН СТЕЙНХАРТ



THE SECRET LIFE OF PROGRAMS

**Understand Computers —
Craft Better Code**

by Jonathan E. Steinhart



San Francisco

ТАЙНАЯ ЖИЗНЬ ПРОГРАММ

КАК СОЗДАТЬ КОД, КОТОРЫЙ
ПОНРАВИТСЯ ВАШЕМУ КОМПЬЮТЕРУ

ДЖОНАТАН СТЕЙНХАРТ



Санкт-Петербург · Москва · Минск

2023

Джонатан Стейнхарт

Тайная жизнь программ. Как создать код, который понравится вашему компьютеру

Перевел с английского С. Черников

Научный редактор А. Гаврилов

ББК 32.973.2-018

УДК 004.05

Стейнхарт Джонатан

C79 Тайная жизнь программ. Как создать код, который понравится вашему компьютеру. — СПб.: Питер, 2023. — 528 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1731-4

Знакомы ли вы с технологиями, лежащими в основе вашей собственной программы? Почему «правильный» код не хочет работать? Истина проста и банальна — нужно сразу создавать код, который будет работать хорошо и не будет прятать в себе трудноуловимые ошибки.

Для этого Джонатан Стейнхарт исследует фундаментальные концепции, лежащие в основе работы компьютеров. Он рассматривает аппаратное обеспечение, поведение программ на определенных устройствах, чтобы показать, как на самом деле должен работать ваш код.

Узнайте, что на самом деле происходит, когда вы запускаете код на компьютере, и вы научитесь программировать лучше и эффективнее.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1593279707 англ.

© 2019 by Jonathan E. Stienhart.

The Secret Life of Programs: Understand Computers — Craft Better Code
ISBN 978-1-59327-970-7, published by No Starch Press.

ISBN 978-5-4461-1731-4

Russian edition published under license by No Starch Press Inc.

© Перевод на русский язык ООО «Прогресс книга», 2023

© Издание на русском языке, оформление ООО «Прогресс книга», 2023

© Серия «Для профессионалов», 2023

Права на издание получены по соглашению с No Starch Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес:
194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 23.05.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 42,570. Тираж 700. Заказ 0000.

Краткое содержание

Благодарности	18
Предисловие	20
Введение	24
Глава 1. Внутренний язык компьютеров	38
Глава 2. Комбинаторная логика	74
Глава 3. Последовательная логика	111
Глава 4. Анатомия компьютера	136
Глава 5. Архитектура компьютера	161
Глава 6. Разбор связей	186
Глава 7. Организация данных	233
Глава 8. Обработка языка	273
Глава 9. Веб-браузер	296
Глава 10. Прикладное и системное программирование	320
Глава 11. Сокращения и приближения	347
Глава 12. Взаимоблокировки и состояния гонки	400
Глава 13. Безопасность	418
Глава 14. Машинный интеллект	459
Глава 15. Влияние реальных условий	491

Оглавление

Об авторе	17
О научном редакторе	17
Благодарности	18
Предисловие	20
От издательства	23
Введение	24
Почему важно программировать хорошо	25
Научиться писать код — только начало	26
Низкоуровневые знания важны	27
Кому стоит прочитать эту книгу?	28
Что такое компьютеры?	28
Что такое программирование компьютеров?	29
Кодинг, программирование, инженерия и computer science	31
Ландшафт	33
Структура книги	36
Глава 1. Внутренний язык компьютеров	38
Что такое язык?	38
Письменный язык	39
Бит	40
Логические операции	40
Булева алгебра	41
Закон де Моргана	42
Представление целых чисел с помощью битов	43
Представление положительных чисел	43
Сложение двоичных чисел	46
Представление отрицательных чисел	48

Представление действительных чисел	54
Представление с фиксированной точкой	54
Представление с плавающей точкой	55
Стандарт IEEE для чисел с плавающей точкой	57
Двоично-десятичная система счисления	58
Более простые способы работы с двоичными числами	59
Восьмеричное представление	59
Шестнадцатеричное представление	59
Представление контекста	60
Именованные группы битов	61
Представление текста	63
Американский стандартный код обмена информацией	63
Развитие других стандартов	65
8-битная форма представления Unicode	66
Использование символов для представления чисел	67
Кодировка Quoted-Printable	67
Кодировка Base64	68
Кодировка URL	69
Представление цветов	69
Добавление прозрачности	72
Кодирование цветов	73
Выводы	73
Глава 2. Комбинаторная логика	74
Задача для цифровых компьютеров	75
Разница между аналоговым и цифровым представлением	76
Почему для аппаратного обеспечения размер имеет значение	78
Цифровые решения для более стабильных устройств	79
Цифровые устройства в аналоговом мире	80
Почему вместо цифр используются биты	82
Знакомство с принципами работы электрического тока	83
Электрический ток на примере сантехники	83
Электрические переключатели	86
Создание аппаратного обеспечения, работающего с битами	90
Реле	90
Вакуумные лампы	93
Транзисторы	94
Интегральные схемы	95
Логические вентили	96
Повышение помехоустойчивости с помощью гистерезиса	97
Дифференциальная передача сигналов	99

Задержка распространения	100
Варианты выходов	101
Создание более сложных схем	104
Создание сумматора	104
Построение дешифраторов	107
Построение демультиплексоров	108
Построение селекторов	108
Выводы	110
Глава 3. Последовательная логика	111
Представление времени	111
Осцилляторы	112
Генераторы тактовых сигналов	113
Триггеры-защелки	113
Синхронный RS-триггер	115
Триггеры	116
Счетчики	119
Регистры	121
Организация памяти и обращение к памяти	122
Оперативная память	125
Постоянное запоминающее устройство	126
Блочные устройства	129
Флеш-память и твердотельные диски	132
Обнаружение и исправление ошибок	133
Аппаратное и программное обеспечение	134
Выводы	135
Глава 4. Анатомия компьютера	136
Память	136
Ввод и вывод	139
Центральный процессор	140
Арифметико-логическое устройство	140
Сдвиг	143
Исполнительное устройство	144
Набор инструкций	146
Инструкции	146
Режимы адресации	149
Инструкции кода состояния	150
Ветвление	150
Итоговый набор инструкций	151

Окончательный проект	153
Регистр команд	153
Передача данных и управляющие сигналы	154
Управление движением	154
Наборы команд RISC и CISC	158
Графические процессоры	160
Выводы	160
Глава 5. Архитектура компьютера	161
Основные архитектурные элементы	162
Ядра процессора	162
Микропроцессоры и микрокомпьютеры	163
Процедуры, подпрограммы и функции	164
Стеки	166
Прерывания	170
Относительная адресация	173
Блок управления памятью	175
Виртуальная память	177
Пространство системы и пользователя	178
Иерархия памяти и производительность	179
Сопроцессоры	181
Организация данных в памяти	182
Запуск программ	183
Мощность запоминающих устройств	184
Выводы	185
Глава 6. Разбор связей	186
Низкоуровневый ввод/вывод	187
Порты ввода/вывода	187
Нажми на кнопку	189
Да будет свет	192
Свет, камера, мотор...	193
Светлые идеи	194
2 ^н оттенка серого	195
Квадратура	196
Параллельная связь	197
Последовательная связь	198
Поймать волну	201
Универсальная последовательная шина	202

Сети	203
Современные локальные сети.....	205
Интернет.....	206
Аналоговые устройства в цифровом мире	207
Цифро-аналоговое преобразование	208
Аналого-цифровое преобразование	210
Цифровое аудио.....	214
Цифровые изображения	222
Видео	224
Устройства взаимодействия с человеком	226
Терминалы	226
Графические терминалы	228
Векторная графика	228
Растровая графика	230
Клавиатура и мышь	232
Выводы	232
Глава 7. Организация данных	233
Базовые типы данных	233
Массивы	235
Битовые матрицы	237
Строки	238
Составные типы данных	240
Односвязные списки	242
Динамическое выделение памяти	247
Более эффективное выделение памяти	248
Сборка мусора	249
Двусвязные списки	250
Иерархические структуры данных	251
Хранение данных на дисковых устройствах	255
Базы данных	258
Индексы	259
Перемещение данных	260
Векторный ввод/вывод	264
Подводные камни объектно-ориентированного программирования.....	265
Сортировка	267
Создание хешей	268
Эффективность и производительность	271
Выводы	272

Глава 8. Обработка языка	273
Язык ассемблера	273
Языки высокого уровня	275
Структурное программирование	276
Лексический анализ	277
Конечные автоматы	279
Регулярные выражения	281
От слов к предложениям	283
Клуб «Язык дня»	285
Деревья синтаксического анализа	286
Интерпретаторы	289
Компиляторы	291
Оптимизация	293
Осторожнее с аппаратной частью!	295
Выводы	295
Глава 9. Веб-браузер	296
Языки разметки	297
Унифицированные указатели ресурсов	299
HTML-документы	300
Объектная модель документа	302
Словарь древовидных структур данных	303
Интерпретация модели DOM	303
Каскадные таблицы стилей	304
XML и друзья	309
JavaScript	312
jQuery	314
SVG	316
HTML5	317
JSON	317
Выводы	318
Глава 10. Прикладное и системное программирование	320
«Угадай животное», версия 1: HTML и JavaScript	323
Каркас прикладного уровня	323
Тело веб-страницы	325
JavaScript	326
CSS	328

«Угадай животное», версия 2: С	329
Терминалы и командная строка	329
Создание программы	330
Терминалы и драйверы устройств	330
Переключение контекста	331
Стандартный ввод/вывод	333
Кольцевые буферы	334
Больше абстракций — лучше код	336
Важные мелочи	337
Переполнение буфера	338
Программа на языке С	339
Тренировка	345
Выводы	346
Глава 11. Сокращения и приближения	347
Поиск по таблице	347
Преобразование	348
Отображение текстур	349
Классификация символов	352
Целочисленные методы	354
Прямые линии	357
Кривые линии	362
Многочлены	365
Рекурсивное деление	366
Сpirали	366
Конструктивная геометрия	369
Сдвиг и наложение масок	376
Еще меньше математики	378
Приближения степенного ряда	378
Алгоритм Волдера	379
Парочка случайностей	384
Заполняющие пространство кривые	385
L-системы	387
Стохастические приемы	389
Квантование	390
Выводы	399
Глава 12. Взаимоблокировки и состояния гонки	400
Что такое состояние гонки?	401
Общие ресурсы	401

Процессы и потоки	402
Блокировки	404
Транзакции и детализация	405
Ожидание захвата ресурсов	406
Взаимоблокировки	407
Реализация кратковременного захвата ресурсов	408
Реализация долговременного захвата ресурсов	409
JavaScript в браузере	409
Асинхронные процессы и промисы	413
Выводы	417
Глава 13. Безопасность	418
Обзор безопасности и конфиденциальности	419
Модель угроз	419
Доверие	420
Физическая безопасность	423
Безопасность связей	424
Наше время	425
Метаданные и наблюдение	427
Социальный контекст	428
Аутентификация и авторизация	430
Криптография	431
Стеганография	431
Шифры подстановки	433
Шифры перестановки	435
Более сложные шифры	436
Одноразовые блокноты	437
Проблема обмена ключами	438
Криптография с открытым ключом	438
Прямая секретность	439
Криптографические хеш-функции	440
Цифровые подписи	441
Инфраструктура открытых ключей	441
Блокчейн	442
Управление паролями	443
Гигиена ПО	444
Защищайте только необходимое	444
Проверьте логику трижды	444
Поишите ошибки	445
Сведите к минимуму поверхности атаки	445

Не выходите за пределы	446
Генерировать хорошие случайные числа — сложно	447
Знайте свой код	449
Чрезвычайная изобретательность — ваш враг	451
Разберитесь с видимостью	451
Не переусердствуйте	452
Не копите	452
Не полагайтесь на динамическое выделение памяти	452
Не полагайтесь и на сборку мусора	454
Данные как код	456
Выводы	458
Глава 14. Машинный интеллект	459
Обзор	460
Машинное обучение	463
Байес	463
Гаусс	465
Собель	468
Кэнни	472
Выделение признаков	475
Нейронные сети	477
Использование данных машинного обучения	482
Искусственный интеллект (ИИ)	484
Большие данные	487
Выводы	490
Глава 15. Влияние реальных условий	491
Повышение ценности	492
Как мы до этого дошли	494
Краткая история	494
ПО с открытым исходным кодом	497
Creative Commons	499
Расцвет переносимости	500
Управление пакетами	500
Контейнеры	501
Java	502
Node.js	503
Облачные вычисления	504
Виртуальные машины	504
Портативные устройства	505

Среда разработки	505
Есть ли у вас опыт?	506
Учимся оценивать	506
Планируем проекты	507
Принимаем решения	508
Работаем с разными людьми	508
Создаем культуру поведения на работе	510
Делаем осознанный выбор	511
Методологии разработки	511
Проектирование	513
Ведение записей	513
Быстрое прототипирование	513
Разработка интерфейса	514
Использовать сторонний код или писать собственный?	518
Разработка	519
Серьезный разговор	519
Переносимый код	522
Управление версиями	523
Тестирование	524
Создание отчетов и отслеживание багов	524
Рефакторинг	524
Обслуживание	525
Позаботьтесь о стиле	525
Чините, а не создавайте заново	527
Выводы	527

*Джули и Ханалей за то, что научили меня
объяснять принципы работы
сложных технологий простым людям.*

*Удивительному месту — Bell Telephone Laboratories —
и всем, кто там работал, особенно Карлу, за то,
что он подыскал для меня там местечко,
когда я был еще подростком.*

Об авторе

Джонатан Э. Стейнхарт (Jonathan E. Steinhart) занимается разработкой с 1960-х годов. Он проектировал оборудование, обучаясь в средней школе, а программное обеспечение — в старших классах, что помогло ему найти подработку на лето в Bell Telephone Laboratories. Он получил степень бакалавра в области электротехники и computer science в Университете Кларксона (Clarkson University) в 1977 году. После выпуска Джонатан работал в Tektronix, а затем стал пробовать свои силы в компаниях-стартапах. Он стал консультантом в 1987 году, специализируясь на проектировании систем с повышенными требованиями к безопасности. В 1990-х он немного сбавил обороты — но только для того, чтобы основать Four Winds Vineyard.

О научном редакторе

Обри Андерсон (Aubrey Anderson) получил степень бакалавра в области электротехники и computer science в Университете Тафтса (Tufts University). Во время учебы он работал ассистентом кафедры и помогал улучшать учебные программы вводных курсов по компьютерным наукам. Он начал программировать в 14 лет и с тех пор вел проекты в области робототехники, системного дизайна и веб-программирования. В настоящее время Обри работает инженером-программистом в Google.

Благодарности

К созданию этой книги приложили руку множество людей. Все началось с моих родителей, Роберта (Robert) и Розалин Стейнхарт (Rosalyn Steinhart), — благодаря им я появился на свет, и они же затем поощряли мой интерес к науке, по крайней мере, пока он не начал их пугать. Много замечательных учителей не позволили этому интересу угаснуть, в том числе Беатрис Сигал (Beatrice Seagal), Уильям Малвахилл (William Mulvahill) и Миллер Бульяри (Miller Bugliari). Большое спасибо Полу Рубенфилду (Paul Rubenfield) за то, что он рассказал мне о гражданской обороне и об обществе скаутов Explorer Scout в Bell Labs.

Невозможно отплатить сполна моим помощникам в Explorer Scout, Карлу Кристиансену (Carl Christiansen) и Хайнцу Ликламе (Heinz Lycklama). Они изменили мою жизнь. Благодаря им я познакомился со многими удивительными людьми в Bell Telephone Laboratories, включая Джо Кондона (Joe Condon), Сэнди Фрейзера (Sandy Fraser), Дэйва Хагельбаргера (Dave Hagelbarger), Дика Хауса (Dick Hause), Джима Кайзера (Jim Kaiser), Хэнка Макдональда (Hank McDonald), Макса Мэтьюза (Max Mathews), Денниса Ричи (Dennis Ritchie), Кена Томпсона (Ken Thompson) и Дэйва Веллера (Dave Weller). Я многому научился у каждого из них.

Спасибо Обри Андерсону (Aubrey Anderson), Клему Коулу (Clem Cole), Ли Джаловеку (Lee Jalovec), А. С. Мендионесу (A.C. Mendiones), Эду Посту (Ed Post) и Бетси Зеллер (Betsy Zeller) за то, что они хоть раз прочитали мою книгу. И особенно Обри за научную редактуру.

Также спасибо Мэтту Блейзу (Matt Blaze), Адаму Чеккетти (Adam Cecchetti), Сэнди Кларк (Sandy Clark), Тому Даффу (Tom Duff), Натали Фрид (Natalie Freed), Фрэнку Хайдту (Frank Heidt), Д. В. Хенкель-Уоллесу (DV Henkel-Wallace) (он

же Гамби), Лу Кацу (Lou Katz), Саре-Джей Терп (Sara-Jaye Terp), Талин (Talin) и Полу Викси (Paul Vixie) за отзывы об отдельных главах.

И спасибо всем, кто отвечал на звонки, когда я хотел получить ответы на общие вопросы, включая Уорда Каннингема (Ward Cunningham), Джона Гилмора (John Gilmore), Эвелин Мэст (Evelyn Mast), Майка Перри (Mike Perry), Алекса Полви (Alex Polvi), Алана Вирфс-Брока (Alan Wirfs-Brock) и Майка Зула (Mike Zuhl). И конечно же, Ракель Хеллберг (Rakel Hellberg), девушке на горнолыжном подъёмнике, за то, что она подтолкнула меня к завершению этого проекта.

Эта книга не увидела бы свет без поддержки и поощрения людей из различных компьютерных сообществ, включая AMW, Hackers и TUHS.

Спасибо Ханалей Стейнхарт (Hanalei Steinhart) за композицию на рис. 6.36 и Джули Доннелли (Julie Donnelly) за шарф на рис. 11.41.

Спасибо коту Тони за то, что позволил использовать его фото, и за шерсть на моей клавиатуре.

Предисловие



Я родился гиком. Отец рассказывал, что я переключал воображаемый тумблер, чтобы включить качели, перед тем как на них сесть, и выключал их, когда переставал качаться. Механизмы сами рассказывали мне, как они устроены изнутри. Я напоминал С-ЗРО, понимающего «бинарный код испарителей». Мне повезло, что я вырос в то время, когда можно было изучать работу большинства вещей без микроскопа.

Оглядываясь назад, я понимаю, что в Нью-Джерси у меня было невероятное детство. Я разбирал все, что можно, часто испытывая на прочность мамины нервы. Родители покупали мне множество наборов «50 в одном», но им стало не по себе, когда я начал объединять их и собирать то, чего не было в инструкции. Кульминацией стала охранная сигнализация для подушки, которая застала зубную фею «на месте преступления», — неудачное решение в экономическом смысле, но тем не менее подарившее мне массу эмоций. Я таскал сломанные телевизоры и другую бытовую технику из мусорных баков, чтобы разбирать их, изучать, как они работают, и создавать новые из частей старых. Одной из моих любимых игрушек был отцовский конструктор 1929 года. Космическая программа еще больше подогрела мой интерес к технологиям; я помню, как однажды ночью мы с отцом стояли во дворе перед домом и смотрели, как по небу движется спутник «Эхо-1».

Большинство детей занимались разносной газетой; я ремонтировал телевизоры и стереосистемы. Мой отец работал в IBM, и я иногда ходил с ним на работу и восхищался большими компьютерами. Он взял меня с собой на электрошоу

в Атлантик-Сити, когда мне было восемь лет, и я помню, как играл с IBM 1620. Я также помню, как меня восхищало оборудование на стенде Tektronix — оно, возможно, повлияло на мой дальнейший выбор работы. Год спустя я посетил Всемирную выставку в Нью-Йорке и был очарован стендом Bell System; позже мне довелось работать с одним из его проектировщиков.

Я получил прекрасное образование в государственной школе пост-«спутникового» образца — таких больше нет в Америке. В пятом классе мы передавали из рук в руки банку со ртутью. В шестом классе я взорвал химическую лабораторию и извлек из этого урок. (Я до сих пор могу назвать формулу получения трехйодистого азота.) Я помню, как в восьмом классе учитель физики отвез нас в Нью-Йорк, чтобы показать фильм «Космическая одиссея 2001 года», потому что он считал это важным. Он сделал это, не оставив записок родителям и не получив разрешений; учитель, который сотворит подобное сегодня, скорее всего, потеряет работу или того хуже. На химии в старших классах мы делали порох, на физике пускали друг в друга ракеты на футбольном поле, на биологии протыкали пальцы, чтобы определить группу крови. Это так непохоже на день сегодняшний, когда столько людей уже утонуло в ведре воды, что теперь на ведрах пишут предупреждения, когда мокрые полы вселяют страх, а чиновники настолько не разбираются в науке, что не могут отличить лабораторный опыт от теракта.

Помимо учебы родители записали меня в школу бойскаутов, которая мне нравилась, и в Малую (бейсбольную. — *Примеч. ред.*) лигу, которую я ненавидел. Скаутинг научил меня многому — от верховой езды до безопасного обращения с огнем и выживания в дикой природе. Малая лига научила меня нелюбви к командным видам спорта.

В те дни были популярны кружки радиолюбителей; они были одними из тех мест, где можно было ставить опыты. Я пошел волонтером в местную группу экстренной радиосвязи гражданской обороны, специально чтобы повозиться с оборудованием. У них была примитивная радиотелеграфная система, которую я переделал и в итоге создал такие же для других районов. Мне нравилось трехмерное механическое устройство, которое собой представлял телеграфный аппарат.

Когда я учился в старшей школе, друг рассказал мне о скаутском отряде Explorer, который собирался каждый понедельник вечером в Bell Telephone Laboratories в соседнем Марри-Хилле. Я присоединился к нему и начал заниматься компьютерами, когда те еще были размером с дом. Меня зацепило. Вскоре я стал рано уходить из школы, добираться автостопом до лабораторий и уговаривать сотрудников впустить меня. Это превратилось в серию потрясающих летних подработок с невероятными людьми, изменившими мою жизнь. Я многому научился, просто заглядывая в лаборатории и спрашивая сотрудников, что они делают. В конце концов, я написал для них программы, хотя планировал изучать

электротехнику, потому что проекты, связанные с оборудованием, просто нельзя было завершить за лето.

Я чувствовал, что лучший способ отдать дань своим соратникам-скаутам — это пойти по их стопам, пытаясь по мере возможности помочь новым поколениям подающих надежды молодых технарей на их пути. Это оказалось непросто, поскольку расцвет американских исследований уступил место увеличению акционерной стоимости; сами продукты не ценятся так высоко, как прибыль, которую они приносят, — из-за этого трудно обосновать важность исследований. Компании редко позволяют детям разгуливать в производственных помещениях, потому что это большая ответственность. Изначально я думал, что буду работать через скаутинг, но понял, что не смогу, потому что скауты приняли некоторые правила, которые я не мог поддержать, так как никогда не получал значка за дискриминацию по половому признаку. Вместо этого я записался добровольцем в местную школу.

Я начал писать эту книгу, чтобы дополнить курс, который вызвался вести. Я сделал это до того, как интернет стал таким же легкодоступным, как сегодня. Я живу в довольно бедном фермерском регионе, поэтому первоначальный проект этой книги включал в себя почти все, что можно, поскольку я исходил из того, что ученики не смогут позволить себе дополнительные материалы. Но объять все оказалось невозможно.

Сейчас в интернете доступно множество материалов о языках программирования и концепциях, и у большинства людей есть доступ в интернет дома, в школе или в библиотеке. Я переписал материал в надежде, что теперь читателям будет намного легче найти дополнительную информацию в сети. Итак, если вам что-то неясно или нужны подробности, просто погуглите.

Недавно несколько моих знакомых студентов выразили разочарование по поводу того, как их учат программированию. Хотя они могут найти информацию в интернете, они все время спрашивают, есть ли место, в котором собрано все необходимое. Эта книга как раз и написана, чтобы стать таким единым источником.

Мне повезло, что я вырос одновременно с компьютерами. Мы развивались вместе. Мне трудно представить, каково это — перейти сразу в зрелую сферу современных вычислений, не имея опыта. Самым сложным в написании этой книги было решить, насколько далеко можно вернуться в прошлое ради примеров и какие элементы современных технологий выбрать для обсуждения. Я остановился на некоторых ретрообразцах, поскольку большую часть необходимого можно извлечь из более старых и простых технологий, которые легче понять. Новые, более сложные технологии строятся из тех же блоков, что и старые; знание этих блоков значительно упрощает понимание новых технологий.

Время изменилось. Гаджеты стало гораздо сложнее разбирать, ремонтировать и модифицировать. Компании нарушают законы, такие как Закон о защите

авторских прав в цифровую эпоху (Digital Millennium Copyright Act, DMCA), чтобы запретить людям ремонтировать принадлежащие им устройства, что, к счастью, иногда приводит к законам о «праве на ремонт». Мы, американцы, получаем неоднозначные сообщения от правительства; с одной стороны, нас поощряют за карьеру в технической сфере, а с другой — мы видим, как игнорируют научный подход, а технические функции передают на аутсорсинг. Неясно, стали бы США мощной технологической державой, если бы так было и полвека назад.

Но есть и положительные моменты. Пространства для творчества множатся. Некоторым детям разрешают создавать вещи, и малыши обнаруживают, что это весело. Электронные детали дешевле, чем когда-либо (это не касается деталей с проводами). Вычислительная мощность современного смартфона больше, чем у всех компьютеров в мире во времена моего детства, вместе взятых. Компьютеры дешевле, чем кто-либо мог себе представить; микрокомпьютеры, такие как Raspberry Pi и Arduino, стоят дешевле пиццы и имеют огромное количество доступных начинок.

Имея такую доступную мощность, заманчиво просто побаловаться высокоуровневой функциональностью. Это похоже на игру с LEGO. Мои родители подарили мне один из первых наборов LEGO; в нем были только прямоугольники. Но с помощью воображения я мог построить все, что хотел. Сегодня вы можете купить набор LEGO «Звездные войны», чтобы собрать модель мастера Йоды. Придумывать новых персонажей гораздо сложнее. Необычные вещи мешают воображению.

В классическом фильме 1939 года «Волшебник страны Оз» есть отличная сцена, в которой волшебник появляется и кричит: «Не обращайте внимания на того человека за ширмой». Эта книга для тех, кто не собирается его слушать и хочет узнать, что скрывается за ширмой. Моя цель — пролить свет на главные кирпичики, из которых строится высокуюровневая функциональность. Эта книга для тех, чье воображение не удовлетворяется только функциональностью высокого уровня; она для тех, кто стремится создавать *новые* высокуюровневые функции. Если вы хотите стать волшебником, а не только владеть магическими предметами, то эта книга для вас.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Введение



Несколько лет назад я ехал на подъемнике вместе с нашей шведской студенткой по обмену. Я спросил ее, думала ли она о том, чем будет заниматься после выпуска. Она сказала, что подумывает об инженерии и в прошлом году ходила на курсы программирования. Я поинтересовался, чему они учат. Она ответила: «Java». Я инстинктивно отреагировал: «Очень жаль».

Почему я так сказал? Мне потребовалось время, чтобы понять это. Дело не в том, что Java — плохой язык программирования; он на самом деле довольно неплох. Просто он (как и другие языки) обычно используется для обучения программированию *без передачи каких-либо знаний о компьютерах*. Если вам это кажется странным, моя книга — для вас.

Язык программирования Java был создан в 1990-х годах Джеймсом Гослингом (James Gosling), Майком Шериданом (Mike Sheridan) и Патриком Нейтоном (Patrick Naughton) из Sun Microsystems. Частично он был смоделирован по образцу языка C, который широко использовался в то время. Язык C не поддерживает автоматическое управление памятью, и потому связанные с этим ошибки были настоящей головной болью. Java намеренно исключил данный класс ошибок программирования; он скрыл от программиста управление памятью. В том числе поэтому он очень хорош для начинающих. Но для подготовки компетентных специалистов и создания качественных программ требуется нечто гораздо большее, чем просто хороший язык. Кроме того, оказалось, что Java привнес целый класс новых проблем программирования, которые труднее поддаются отладке, включая низкую производительность из-за скрытой системы управления памятью.

Как вы увидите в этой книге, понимание памяти — ключевой навык для программистов. Учась программировать, легко выработать привычки, от которых потом трудно избавиться. Исследования показали, что дети, которые выросли, играя на так называемых «безопасных» площадках, чаще травмируются в старшем возрасте, чем остальные (предположительно потому, что такие дети не знают, что падение — это больно). Аналогичная ситуация с программированием. Безопасная среда программирования снимает страх перед началом работы, но, помимо этого, нужно подготовиться к реальным условиям внешней среды. Эта книга поможет осуществить такой переход.

Почему важно программировать хорошо

Чтобы понять, почему сложно преподавать программирование без обучения тому, как работают компьютеры, сначала подумайте, как прочно компьютеры вошли в нашу жизнь. Цена на них упала настолько резко, что компьютер стал самым дешевым способом создания множества вещей. Например, для вывода на приборную панель автомобиля устаревшего аналогового циферблата дешевле использовать компьютер, чем настоящие механические часы. Это результат того, как производятся компьютерные микросхемы; они печатаются огромными партиями. Выпустить чип, содержащий миллиарды компонентов, больше не составляет труда. Заметьте, что я говорю о цене самих компьютеров, а не о цене объектов, в которые они включены. В целом компьютерный чип сегодня стоит меньше, чем упаковка, в которой он поставляется. Доступны компьютерные чипы, которые стоят копейки. Скорее всего, наступит время, когда будет сложно найти устройство, в котором нет электроники.

Огромное количество компьютеров, выполняющих массу задач, означает множество компьютерных программ. Поскольку компьютеры так широко распространены, программирование невероятно разнообразно. Подобно врачам, многие программисты становятся узкими специалистами. Можно сосредоточиться на таких областях, как техническое зрение, анимация, веб-страницы, телефонные приложения, промышленное управление, медицинские устройства и многое другое.

Но что самое странное — в отличие от медицины, в программировании можно стать узким специалистом, не являясь универсалом. Скорее всего, вам не нужен кардиохирург, который никогда не изучал анатомию, но среди программистов подобное — норма. Так ли это важно? На самом деле, множество фактов свидетельствуют о том, что это не очень хорошо, учитывая почти ежедневные сообщения о нарушениях безопасности и отзыве продуктов. Случалось, что люди, осужденные за вождение в нетрезвом виде на основе данных алкотестера, выигрывали суды о пересмотре кода алкотестера. Оказалось, что код содержал массу багов, и обвинения в таких случаях снимались. Недавно антивирусная программа вызвала отказ медицинского оборудования во время операции на

сердце. Проблемы с конструкцией самолета Boeing 737 MAX привели к человеческим жертвам. Большое количество подобных инцидентов не внушает особого доверия.

Научиться писать код — только начало

Одна из причин такого положения дел в том, что не так уж и сложно написать программу, которая *кажется* работоспособной или выполняется без проблем большую часть времени. Возьмем для сравнения изменения в музыке (не диско!) 1980-х годов. Раньше людям приходилось потрудиться, прежде чем писать музыку. Изучить теорию музыки, композицию и освоить музыкальный инструмент, натренировать слух и провести много часов за практикой. Затем появился стандарт цифрового интерфейса музыкальных инструментов (Musical Instrument Digital Interface, MIDI), предложенный Икутаро Какехаши (Ikutaro Kakehashi) из Roland и позволивший любому человеку создавать «музыку» на компьютере — без усилий. Я считаю, что лишь малая доля таких «произведений» является музыкой на самом деле; по большей части это шум. *Музыку* создают настоящие музыканты вне зависимости от того, применяют они MIDI для записи основы или нет. В наши дни программирование во многом похоже на использование MIDI. Больше не нужно потеть от усердия, или тратить годы на практику, или даже изучать теорию, чтобы писать программы. Но это не значит, что они будут хороши или надежны.

Ситуация может стать еще хуже, по крайней мере в Соединенных Штатах. Корыстные инвесторы, такие как владельцы компаний-разработчиков программного обеспечения, лоббируют закон, обязывающий изучать программирование уже в школе. В теории это звучит великолепно, но на практике это не лучшая идея, потому что не у всех есть способность к программированию. Мы не требуем, чтобы все учились играть в футбол, потому что знаем, что он не для всех. Вероятная цель этой инициативы не в том, чтобы готовить отличных программистов, а в том, чтобы увеличить прибыль компании-разработчика за счет наводнения рынка множеством неквалифицированных специалистов, что приведет к снижению заработной платы. Люди, стоящие за этой идеей, не очень заботятся о качестве кода — они также настаивают на принятии закона, ограничивающего ответственность за некачественные продукты. Конечно, вы можете программировать для развлечения так же, как и играть в футбол. Просто не ждите, что вас выберут на Суперкубок.

В 2014 году президент Обама сказал, что научился программировать. Он действительно переместил пару элементов в превосходном инструменте визуального программирования Blockly и даже напечатал строку кода на JavaScript (язык программирования, не связанный с Java и изобретенный в компании Netscape, предшественнице Mozilla Foundation, поддерживающей многочисленные программные пакеты, включая веб-браузер Firefox). Как вы думаете,

он действительно научился программировать? Подсказка: если вы ответите да, вам, вероятно, следует вдобавок к чтению этой книги поработать над оттачиванием навыков критического мышления. Наверняка, он кое-что узнал о программировании, но *нет, он не научился программировать*. Если бы он мог научиться программировать за час, это бы значило, что программирование очень простая штука и его не нужно преподавать в школах.

Низкоуровневые знания важны

Интересное и несколько отличающееся от общепринятого мнение о том, как обучать программированию, было выражено в статье из блога Стивена Вольфрама (Stephen Wolfram), создателя Mathematica и языка Wolfram, под названием «Как научить вычислительному мышлению». Вольфрам определяет такое мышление как «формулирование инструкций с достаточной ясностью и достаточным систематическим подходом, чтобы передавать их компьютеру». Я полностью согласен с этим определением. Фактически оно во многом мотивировало меня на написание этой книги.

Но я категорически не поддерживаю позицию Вольфрама, согласно которой будущие программисты должны развивать навыки вычислительного мышления с помощью мощных высокуюровневых инструментов (таких, как те, которые он разработал), а не изучать лежащие в основе дисциплины фундаментальные технологии. Например, из растущего интереса к статистике, а не к расчетам становится ясно, что «обработка данных» — это развивающаяся область. Но что происходит, когда люди просто загружают груды данных в причудливые программы, принцип работы которых не понимают до конца?

Есть вероятность, что они получают интересные, но бессмысленные или неверные результаты. Например, недавнее исследование («Gene Name Errors Are Widespread in the Scientific Literature» («Распространенные ошибки в названиях генов в научной литературе». — Примеч. ред.) Марка Циманна (Mark Ziemann), Йотама Эрена (Yotam Eren) и Ассама Эль-Оста (Assam El-Osta)) показало, что пятая часть опубликованных статей по генетике содержит неточности из-за неправильного использования электронных таблиц. Подумайте только, какие ошибки и их последствия могут вызвать более мощные инструменты в руках большего числа людей! Особенно важно принимать правильные решения, если речь идет о человеческих жизнях.

Понимание базовых технологий помогает определить, что может пойти не так. Знание только высокуюровневых инструментов ведет к постановке неправильных вопросов. Прежде чем взять в руки гвоздезабивной пистолет, стоит научиться обращаться с молотком. Еще одна причина для изучения базовых систем и инструментов заключается в том, что это дает возможность создавать новые инструменты. Это важно, ведь потребность в создателях будет всегда, даже

если их меньше, чем пользователей. Если вы изучите компьютеры и поведение программ уже не будет для вас загадкой, вы сможете создавать лучший код.

Кому стоит прочитать эту книгу?

Эта книга предназначена для тех, кто хочет стать отличным программистом. Что для этого требуется? Прежде всего хорошие навыки критического мышления и анализа. Чтобы решать сложные задачи, программист должен уметь оценивать, действительно ли его программа решает поставленную задачу. Это труднее, чем кажется. Нередко опытный профессионал смотрит на чужую программу и язвительно констатирует: «Эта штука сложна, но она не решает даже простую проблему, которая и не проблема вовсе».

Вам наверняка знаком классический фэнтезийный образ волшебника, который обретает власть над вещами, узнав их настоящие имена. И горе тем из них, кто забывает детали. Хороший программист похож на волшебника, способного держать в уме суть вещей, не опуская мелочей.

Подобно умелым ремесленникам, компетентные программисты — люди творческие. Нередко можно встретить совершенно непонятный код — точно так же многих англоговорящих сбивает с толку роман Джеймса Джойса «Поминки по Финнегану». Хорошие программисты пишут код, который не только работает, но понятен другим и прост в обслуживании.

Наконец, хороший программист должен прекрасно разбираться в том, как работают компьютеры. Невозможно решать сложные задачи, имея лишь поверхностное представление об основах. Эта книга предназначена для тех, кто изучает программирование, но при этом ощущает отсутствие глубины знаний. Она также подойдет тем, кто уже занимается программированием, но хочет большего.

Что такое компьютеры?

Обычно в ответ можно услышать что-то вроде: «Компьютеры — это устройства, с помощью которых люди проверяют электронную почту, совершают онлайн-покупки, работают с документами, сортируют фотографии и играют». Это определение отчасти является результатом терминологической небрежности, которая распространилась, когда компьютеры стали потребительским товаром. Другой популярный ответ: компьютеры — это мозги наших высокотехнологичных игрушек, таких как сотовые телефоны и музыкальные плееры. Второй вариант ближе к истине.

Отправлять электронную почту и играть в игры можно благодаря программам, работающим на компьютерах. Сам компьютер похож на новорожденного ребенка. Он на самом деле многое не умеет. Мы почти никогда не задумываемся

о механизмах физиологии человека, потому что прежде всего взаимодействуем с личностью, которая работает на их основе, точно так же как программы выполняются на компьютерах. Например, когда вы читаете страницу сайта, вы не используете для этого сам компьютер; вы читаете ее с помощью написанных кем-то программ, работающих на вашем компьютере, на компьютере, где размещена веб-страница, и на всех промежуточных компьютерах, которые обеспечивают функционирование интернета.

Что такое программирование компьютеров?

Преподаватели обучаются человека выполнять определенные задачи. Точно так же начать программировать означает стать учителем для компьютеров. Программисты учат компьютеры делать то, что от них хотят.

Умение обучать компьютеры полезно, особенно когда вы хотите, чтобы они делали нечто новое для них, и вы не можете просто купить нужную программу, ведь ее еще никто не создал. Например, вы, вероятно, воспринимаете Всемирную паутину как должное, но она была изобретена не так давно, когда сэру Тиму Бернерсу-Ли (Tim Berners-Lee) понадобился лучший способ организовать обмен информацией между учеными из Европейского центра ядерных исследований (Conseil Européen pour la Recherche Nucléaire, CERN). За это он был посвящен в рыцари. Круто, не правда ли?

Обучать компьютеры сложно, но это легче, чем учить людей. Мы знаем намного больше о том, как работают компьютеры. И вряд ли техника когда-нибудь взбунтуется.

Компьютерное программирование включает два этапа.

1. Понять Вселенную.
2. Объяснить ее трехлетнему ребенку.

Что это значит? Невозможно писать компьютерные программы, не разбираясь в задачах, которые они выполняют. Например, нельзя написать программу для проверки правописания, не зная правил орографии, а хорошую видеоигру — не зная физики. Итак, первый шаг к тому, чтобы стать хорошим программистом, — узнать как можно больше об окружающем мире. Решения часто приходят откуда не ждали, поэтому не игнорируйте что-то лишь потому, что оно не кажется актуальным.

Второй шаг процесса требует объяснения того, что вы знаете, машине, которая буквально воспринимает окружающий мир — как маленький ребенок. Эта детская прямолинейность очень наглядна в возрасте около трех лет. Допустим, вы пытаетесь выйти из дома и спрашиваете ребенка: «Где твоя обувь?» Ответ: «Вот она». Ребенок *ответил* на ваш вопрос. Проблема в том, что он не понимает, что

на самом деле вы просите его надеть туфли, чтобы куда-то пойти. Гибкость и способность делать выводы — навыки, которые дети усваивают по мере взросления. Но компьютеры похожи на Питера Пэна: они никогда не вырастают.

Компьютеры похожи на маленьких детей тем, что они не умеют обобщать. Они по-прежнему полезны, потому что, как только вы поймете, как им что-то объяснить, они будут делать это очень быстро и неутомимо, хотя и не будут обладать здравым смыслом. Компьютер может без устали делать то, о чем вы просите, не оценивая, правильно ли это, во многом как заколдованные метлы в отрывке «Ученик чародея» из мультфильма «Фантазия» 1940 года. Поручить компьютеру сделать что-то — все равно что просить джинна из волшебной лампы (не в версии ФБР¹) исполнить желание. Вы должны быть очень осторожны, формулируя запрос!

Вы, возможно, сомневаетесь в моих словах, потому что компьютеры кажутся более способными, чем они есть на самом деле. Например, они умеют рисовать, исправлять орфографические ошибки, понимать, что вы говорите, проигрывать музыку и т. д. Но имейте в виду, что это делает не компьютер, а сложный набор кем-то написанных программ, благодаря которым он выполняет все эти задачи. Компьютеры функционируют отдельно от программ, которые на них работают.

Это то же самое, что смотреть на машину, которая движется по дороге. Кажется, что она довольно хорошо останавливается и начинает движение в нужное время, обезжает препятствия, добирается туда, куда нужно, ест, когда проголодается, и т. д. Но машина не действует сама по себе. Все это происходит благодаря связке автомобиля и водителя. Компьютеры подобны машинам, а программы — водителям. Не имея нужных знаний, невозможно сказать, что делает автомобиль, а что — водитель.

В общем, программирование подразумевает изучение того, что вам нужно знать для решения задачи, а затем объяснение этих вещей маленькому ребенку. Поскольку существует множество способов решить задачу, программирование — это не только искусство, но и наука. Оно предполагает поиск элегантных решений, а не использование грубой силы. Да, вы *можете* выбраться из дома, пробив дыру в стене, но, вероятно, намного проще выйти через дверь. Многие могут создать ресурс наподобие HealthCare.gov в миллионы строк кода, но, чтобы сделать это в тысячах строк, требуются определенные навыки.

Однако, прежде чем обучать трехлетку, нужно узнать побольше о нем и о том, что он понимает. И это не обычный ребенок — это инопланетная форма жизни. Компьютер не играет по тем же правилам, что и мы. Возможно, вы слышали об искусственном интеллекте (ИИ), который пытается заставить компьютеры действовать похоже на людей. Прогресс в этой области идет намного медленнее,

¹ Отсылка к троянской программе Magic Lantern («Волшебная лампа») для чтения зашифрованной информации на компьютерах подозреваемых. — Примеч. ред.

чем предполагалось изначально. В основном потому, что в действительности мы не понимаем проблемы; мы недостаточно знаем о том, как работает наш мозг. Довольно сложно научить инопланетянина думать, как мы, если нам самим неизвестно, каково это.

Человеческий мозг позволяет нам совершать действия бессознательно. Мозг появился как аппаратное обеспечение, которое затем было запрограммировано. Например, вы научились шевелить пальцами, а затем — хватать предметы. После достаточной практики вы просто берете вещи в руки, не задумываясь о механизмах, которые делают это возможным. Философы, такие как Жан Пиаже (французский психолог, 1896–1980) и Ноам Хомский (американский лингвист, родившийся в 1928 году), разработали разные теории о том, как происходит процесс обучения. Является ли мозг простым инструментом или в нем есть специальные аппаратные средства для таких функций, как языки? Этот вопрос все еще изучается.

Наша невероятная способность выполнять действия бессознательно мешает учиться программированию, потому что оно требует разбиения задач на более мелкие шаги, которые может выполнять компьютер. Например, вы наверняка умеете играть в крестики-нолики. Соберите группу людей и попросите каждого самостоятельно перечислить шаги, которые должен предпринять игрок, чтобы сделать хороший ход для любой конфигурации доски. (Ответ можно найти в интернете, но постарайтесь этого не делать.) После того как все составят списки, проведите соревнование. Узнайте, чьи правила лучше! А хороши ли были ваши правила? Что вы пропустили? Правда ли вы знаете, что делаете, когда играете в игру? Скорее всего, вы не объяснили ряд условий, потому что понимаете их интуитивно.

Итак, если это еще не очевидно: первый шаг — понимание Вселенной — гораздо важнее, чем второй — объяснение ее трехлетнему ребенку. Подумайте: что хорошего в том, чтобы уметь говорить, если вы не знаете, что сказать? Несмотря на это, нынешнее образование делает упор на второй шаг. Все потому, что гораздо легче обучать механическим аспектам задачи, чем творческим, так же как и оценивать усвоение материала. И в целом учителя не имеют достаточной подготовки в этой области и работают по кем-то созданным и предоставленным им программам. Однако в этой книге основное внимание уделяется первому шагу. Хотя книга не может охватить всё вокруг, она исследует задачи и их решения в компьютерной вселенной вместо разбора точного синтаксиса программирования, необходимого для реализации этих решений.

Кодинг, программирование, инженерия и computer science

Для описания работы с программами используются разные термины. У них нет точных определений, но есть примерные.

Кодинг – относительно новый термин, ставший популярным как часть «обучения программированию». Кодинг можно в определенном смысле рассматривать как работу переводчика. Сравним это с использованием кодов Международной классификации болезней (МКБ). Постановка диагноза врачом – сравнительно простая задача. Сложнее всего перевести этот диагноз в один из более чем 100 000 кодов в стандартах МКБ (МКБ-10 на момент написания книги). Сертифицированный специалист, который выучил эти коды, знает, что, когда врач ставит диагноз «лягнула корова», этому диагнозу нужно присвоить код W55.2XA. Эта работа на самом деле сложнее, чем большая часть кодинга в программировании, потому что МКБ-кодов огромное множество. Но процесс аналогичен тому, что сделал бы кодер, если бы его попросили «выделить текст жирным» на веб-странице: он знает, какой код использовать, чтобы выполнить задание.

Стандарт МКБ-10 настолько сложен, что немногие сертифицированные специалисты знают его полностью. Медицинские кодировщики получают сертификаты по отдельным специальностям, например «Заболевания нервной системы» или «Психические и поведенческие расстройства». Это аналогично тому, что программисты становятся специалистами по отдельно взятым языкам, например HTML или JavaScript.

Но *программирование*, то есть работа программиста, означает знать больше, чем одну-две области специализации. Врач в этом сценарии подобен программисту. Врач ставит диагноз, оценивая пациента. Это может быть довольно сложно. Например, если у пациента есть ожоги и он промок насекомым, это «неестественный внешний вид» (код R46.1) или «ожог горячими водными лыжами, первый контакт» (код V91.07XA)? После того как врач поставит диагноз, можно разработать план лечения. План должен быть эффективным; врач, вероятно, не захочет снова принять пациента, страдающего от тяжелой степени «чрезмерной родительской опеки» (Z62.1).

Программист, как и врач, оценивает проблему и находит решение. Например, есть потребность в веб-сайте, который позволил бы людям ранжировать коды МКБ-10 с точки зрения глупости. Программист определит лучшие алгоритмы хранения и обработки данных, структуру взаимодействия между веб-клиентом и сервером, пользовательский интерфейс и т. д. Это не простая «вставка кода».

Инженерия – это следующий уровень сложности. В общем, инженерия – это искусство извлекать знания и использовать их для достижения чего-либо. Можно рассматривать создание стандартов МКБ как инженерную разработку; обширная область медицинских диагнозов была сведена к набору кодов, которые было легче отслеживать и анализировать, чем записи врача. Представляет ли такая сложная система *хорошую* разработку – вопрос открытый. В качестве примера компьютерной инженерии – много лет назад я работал над проектом создания недорогого медицинского монитора, такого как те, которые вы видите в больницах. Мне было поручено создать систему, в которой врач или медсестра могли

разобраться менее чем за 5 минут без чтения документации. Как вы понимаете, для этого требовалось гораздо больше, чем просто знание программирования. И я добился цели — знакомство с моим решением занимает около 30 секунд.

Программирование часто путают с computer science. В то время как многие специалисты в области компьютерных наук программируют, большинство программистов не являются специалистами в computer science. *Computer science* — это наука о компьютерных технологиях и вычислениях. Открытия в этой сфере используют инженеры и программисты.

Кодинг, программирование, инженерия и computer science — независимые, но связанные дисциплины, которые различаются по типу и объему требуемых знаний. Специалист по компьютерным наукам, разработчик или кодер не становится хорошим программистом автоматически. Хотя книга дает представление о том, как думают разработчики и специалисты по computer science, она не сделает вас ими; для этого обычно требуется высшее образование и определенный наработанный опыт. Разработка и программирование похожи на музыку или живопись — они отчасти являются навыками, а отчасти искусством. Рассмотрение обоих аспектов в этой книге должно помочь вам улучшить навыки программирования.

Ландшафт

Компьютерное проектирование и программирование — большая область для изучения, которую я не смогу охватить здесь в полной мере. Ее можно схематично представить так, как показано на рис. 1.

Имейте в виду, что рис. 1 — упрощенное представление и что линии, разделяющие разные слои, на самом деле не такие четкие.

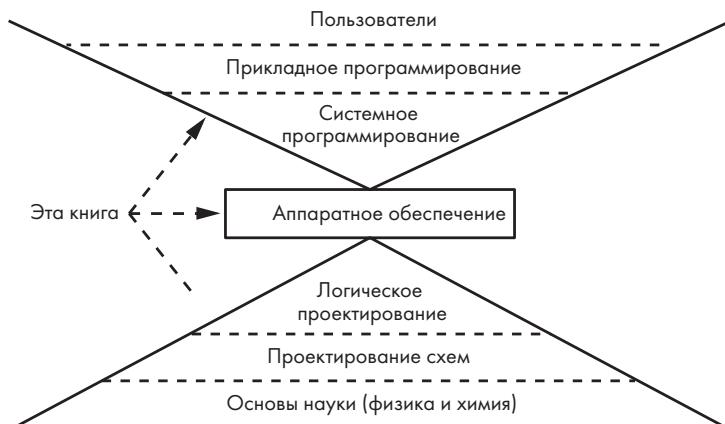


Рис. 1. Компьютерный ландшафт

Большинство людей являются *пользователями* компьютерных систем. Вы наверняка тоже сейчас принадлежите к ним. Существуют особые пользователи, называемые *системными администраторами*, которые поддерживают работу компьютерных систем. Они устанавливают программное обеспечение, управляют учетными записями пользователей, создают резервные копии и т. д. Обычно они обладают особыми полномочиями, недоступными обычным пользователям.

Людей, которые пишут такие программы, как веб-страницы, приложения для телефонов и музыкальные проигрыватели, называют *прикладными программистами*. Они пишут программное обеспечение для взаимодействия пользователей с компьютерами, используя блоки, созданные другими специалистами. Прикладное программирование преподается на большинстве курсов по «обучению программированию», как будто все программисты должны научиться импортировать эти блоки и склеивать их вместе. Хотя в большинстве случаев это не столь обязательно, гораздо лучше понять суть самих «блоков» и «клея» между ними.

Прикладные программы не взаимодействуют с компьютерным оборудованием напрямую; вот где в игру вступает *системное программирование*. Системные программисты создают строительные блоки, используемые программистами прикладных задач. Системным программистам необходимо разбираться в оборудовании, потому что их код взаимодействует с ним. Одна из целей этой книги — научить вас тому, что нужно знать, чтобы стать хорошим системным программистом.

Аппаратное обеспечение компьютера включает в себя не только ту часть, которая выполняет фактические вычисления, но и то, как эта часть соединяется с внешним миром. Аппаратное обеспечение выражается в виде *логики*. Это та же логика, которая используется при написании программ, и она является ключом к пониманию работы компьютера. Логика построена из различных типов *электронных схем*. Проектирование схем выходит за рамки этой книги, но вы можете узнать о нем больше, изучая схемотехнику. Если вы хотите править миром, подумайте о двойной специальности в области схемотехники и computer science.

Конечно, в основе всего лежит фундаментальная наука, дающая знания обо всем, начиная от нашего понимания электричества до химии, необходимой для создания микросхем.

Как показано на рис. 1, каждый уровень основывается на предыдущем. Это означает, что неправильный выбор проектирования или ошибки на более низких уровнях влияют на все высшие уровни. Например, ошибка проектирования в процессорах Intel Pentium около 1994 года привела к тому, что некоторые операции деления дали неверные результаты. Это коснулось всего программного обеспечения, которое использовало деление с плавающей запятой в этих процессорах.

Как видите, системное программирование находится в нижней части иерархии программного обеспечения. Оно похоже на инфраструктуру — дороги, электричество и вода. Всегда важно быть хорошим программистом, но еще важнее быть им, если вы — системный программист, потому что другие полагаются на вашу инфраструктуру. Также видно, что системное программирование находится между прикладным программированием и аппаратным обеспечением, а это означает, что для работы необходимы знания обеих этих областей. Слово *йога* переводится с санскрита как «союз», и так же, как практикующие йогу стремятся объединить свой разум и тело, системные программисты являются техногами, соединяющими аппаратное и программное обеспечение.

Вам не нужно изучать системное программирование, чтобы работать на одном из других уровней. Но если вы этого не сделаете, вам придется найти кого-то, кто поможет вам справиться с проблемами вне вашей предметной области, так как вы не сможете решить их самостоятельно. Понимание основ технологии также приводит к лучшим решениям на более высоких уровнях. Это не только мое мнение; см. пост в блоге Вилле-Матиаса Хейккиля (Ville-Matias Heikkilä) The Resource Leak Bug of Our Civilization за 2014 год — он высказывает похожее мнение.

Эта книга тоже стремится охватить часть истории. Большинство программистов не изучают историю своего дела, потому что им нужно усвоить очень много материала. В результате многие делают ошибки, которые случались и раньше. Знание истории по крайней мере позволяет делать новые ошибки, а не повторять старые. Помните, что новейшие технологии, которые вы используете сегодня, уже завтра устареют.

Если говорить об истории, то эта книга до отказа набита интересными технологиями и именами их создателей. Найдите время, чтобы узнать больше как о технологиях, так и о людях. Большинство упомянутых специалистов решили по крайней мере одну интересную задачу, и стоит узнать, как они воспринимали мир и как подходили к проблемам и решали их. В романе Нила Стивенсона (Neal Stephenson) «Анафема» (2008) есть отличный диалог:

«Нам угрожает инопланетный корабль, начиненный атомными бомбами. У нас есть транспортир».

«Ладно, я сбегаю домой за линейкой и куском бечевки».

Стоит отметить опору на основы. Это не «Давай посмотрим, что делать, в “Википедии”», или «Я спрошу на Stack Overflow», или «Я найду какой-нибудь пакет на GitHub». Научитьсяправляться с проблемами, которые еще никто не решил, — важнейший навык.

Во многих примерах в этой книге разбираются старые технологии, такие как 16-битные компьютеры. Ведь с их помощью можно узнать почти все, что нужно, и они более простые.

Структура книги

Книга концептуально разделена на три части. В первой части исследуется компьютерное оборудование: что это такое и как оно устроено. Вторая часть изучает поведение программ, запущенных на оборудовании. Последняя часть посвящена искусству программирования — совместной работе над созданием хороших программ.

- **Глава 1. Внутренний язык компьютеров.** В этой главе начинается исследование менталитета трехлетнего ребенка. Компьютеры — битовые игроки; их жизнь — управление битами. Вы узнаете, что они собой представляют и что с ними можно сделать. Мы будем присваивать битам и совокупностям битов гипотетические значения.
- **Глава 2. Комбинаторная логика.** В этой главе исследуется обоснование использования битов вместо цифр и применение цифровых компьютеров. Глава включает в себя обсуждение некоторых старых технологий, которые проложили путь к тому, что мы имеем сегодня. Охватываются основы комбинаторной логики. Вы узнаете, как создавать более сложные функции из битов и логики.
- **Глава 3. Последовательная логика.** Здесь вы узнаете, как использовать логику для построения памяти. Мы будем учиться генерировать время, потому что память — это не что иное, как состояние, которое сохраняется во времени. В этой главе рассматриваются основы последовательной логики и обсуждаются различные технологии памяти.
- **Глава 4. Анатомия компьютера.** В этой главе показано, как компьютеры собираются из логических элементов и элементов памяти, которые обсуждались в предыдущих главах. Рассматриваются различные методологии реализации.
- **Глава 5. Архитектура компьютера.** В этой главе мы рассмотрим некоторые дополнения к базовому компьютеру, изученному в главе 4. Вы узнаете, как они обеспечивают необходимую функциональность и эффективность.
- **Глава 6. Разбор связей.** Компьютеры должны взаимодействовать с внешним миром. В этой главе изучаются ввод и вывод. Эта часть книги также пересматривает разницу между цифровыми и аналоговыми компьютерами и то, как мы обеспечиваем работу цифровых компьютеров в аналоговом мире.
- **Глава 7. Организация данных.** Теперь, когда вы увидели, как работают компьютеры, мы узнаем, как их эффективно использовать. Компьютерные программы манипулируют данными в памяти, и важно представить способ ее использования для решаемой задачи.
- **Глава 8. Обработка языка.** Языки были изобретены, чтобы людям было проще программировать компьютеры. В главе обсуждается процесс преобразования языков в то, что действительно работает на компьютерах.

- **Глава 9. Веб-браузер.** Для веб-браузеров было запрограммировано множество всего. В этой главе рассматривается, как работает веб-браузер, и выделяются его основные компоненты.
- **Глава 10. Прикладное и системное программирование.** В этой главе мы напишем две версии программы, которая работает на двух разных уровнях, показанных на рис. 1. Здесь раскрываются многие различия между программированием на уровне приложений и на уровне систем.
- **Глава 11. Сокращения и приближения.** Очень важно сделать программы эффективными. В этой главе исследуются некоторые способы, с помощью которых можно этого добиться, освободив программы от ненужной работы.
- **Глава 12. Взаимоблокировки и состояния гонки.** Во многих системах используется более одного компьютера. В этой главе рассматриваются некоторые проблемы взаимодействия компьютеров.
- **Глава 13. Безопасность.** Компьютерная безопасность — это непростая тема. Здесь рассматриваются основы работы со сложной математикой.
- **Глава 14. Машинный интеллект.** В этой главе также поднимается более сложная тема. Новые приложения являются результатом сочетания больших данных, искусственного интеллекта и машинного обучения — от рождения автомобиля до искусства сводить с ума рекламой.
- **Глава 15. Влияние реальных условий.** Программирование — очень методичный и логичный процесс. Но в определении того, что и как программируют, участвуют люди, а им часто не хватает логики. В этой главе обсуждаются некоторые вопросы программирования в реальном мире.

Читая эту книгу, имейте в виду, что многие объяснения упрощены и, следовательно, в мельчайших деталях могут быть неточны. Чтобы сделать объяснения идеальными, потребуется слишком много отвлекающих подробностей. Не удивляйтесь, если вы поймете это по мере того, как будете узнавать больше. Считайте эту книгу глянцевым путеводителем по стране компьютеров. Он не может охватить все подробно, и, когда вы начнете углубляться в детали, вы обнаружите множество тонких различий.

1

Внутренний язык компьютеров



Основная задача языка — передавать информацию. Ваша работа как программиста — давать инструкции компьютерам. Они не понимают нашего языка, поэтому нам придется выучить их собственный.

Человеческий язык — результат тысячелетней эволюции.

Мы мало знаем о том, как он развивался, поскольку на ранних этапах развития языков еще не умели записывать историю. (Повидимому, никто не сочинил баллад о развитии языков.) С компьютерными языками все иначе, поскольку они изобретены относительно недавно и спустя долгое время после развития человеческого языка, что позволяет нам писать о них.

Человеческий и компьютерный языки имеют много общего, например письменные символы и правила их корректного расположения и использования. Одна вещь, которая их различает, — это неписьменные формы; язык компьютеров исключительно письменный.

В этой главе вы начнете изучать язык компьютеров. Этот процесс происходит поэтапно, как и в случае с человеческим языком. Нужно начинать с букв, а затем переходить к словам и предложениям. К счастью, компьютерные языки гораздо более просты, чем их человеческие аналоги.

Что такое язык?

Язык — это удобное сокращение. Он позволяет описывать сложные концепции, не демонстрируя их. Он также позволяет передавать концепции на расстоянии, даже через посредников.

Каждый язык — будь то письменный, разговорный или включающий серию жестов — имеет значение, *закодированное* в виде набора символов. Однако кодирования значения символами недостаточно. Язык работает только в том случае, если все общающиеся стороны располагают одинаковым *контекстом*, позволяющим присвоить одинаковое значение одним и тем же символам. Например, видя слово *Toto*, многие вспоминают собаку из «Волшебника страны Оз», в то время как другие представляют японского производителя сидений для унитазов с подогревом. Недавно я столкнулся с большим недопониманием, обсуждая одежду с одним из своих французских студентов по обмену. Оказывается, слово *camisole* в Америке обычно обозначает майку, а во Франции — смирительную рубашку! В обоих примерах одни и те же символы можно различить только по контексту, и этот контекст не всегда легко определить. С компьютерными языками та же проблема.

Письменный язык

Письменный язык — это последовательность символов. Мы образуем слова, располагая символы в определенном порядке. Например, в английском языке мы образуем слово *ut*, разместив три символа (то есть буквы) в порядке слева направо следующим образом: *u* и *t*.

Существует множество символов и их комбинаций. В английском языке 26 основных символов (*A* — *Z*) — без учета прописных и строчных букв, знаков препинания, лигатур и т. д., — которые носители английского языка изучают в раннем детстве. В других языках используются другие типы и количество символов. В некоторых иероглифических языках, таких как китайский и японский, символов очень много, и каждый символ представляет собой отдельное слово.

В языках также разный порядок расположения элементов, например справа налево на иврите и вертикально на китайском. Важен и порядок символов: *dog* («пёс») — это не то же самое, что *god* («бог»).

Хотя стиль в некотором смысле можно рассматривать как язык сам по себе, мы не различаем символы по шрифту: *a*, *a* и **a** — это один и тот же символ.

Технологию письменного языка, включая компьютерный язык, формируют три компонента:

- контейнеры, содержащие символы;
- допустимые символы в контейнерах;
- порядок контейнеров.

Некоторые языки включают более сложные правила, которые ограничивают разрешенные символы в контейнерах на основе символов в других контейнерах. Например, некоторые символы не могут находиться в соседних контейнерах.

Бит

Начнем с контейнера. Он может называться *символом* на человеческом языке и *битом* на языке компьютера. Термин «бит» представляет собой неудачное сочетание понятий «двоичный» (binary) и «цифровой» (digit). Неудачное, потому что двоичный код означает что-то состоящее из двух частей, тогда как слово «цифра» означает любой из 10 символов (0–9) повседневной системы счисления. В следующей главе вы узнаете, почему мы используем биты; пока ограничимся тем, что они дешевы и просты в сборке.

Бит является двоичным — битовый контейнер может содержать только один из двух символов наподобие точки и тире из азбуки Морзе. В азбуке Морзе используются всего два символа для представления сложной информации путем объединения этих символов в различные комбинации. Например, буква *A* — это точка-тире.

B — это тире-точка-точка-точка, *C* — тире-точка-тире-точка и т. д. Порядок символов важен, как и в человеческом языке: тире-точка означает *N*, а не *A*.

Понятие символов абстрактно. На самом деле, не так важно, что они означают; они могут значить что угодно — включение и выключение, день и ночь, утка и гусь. Но помните, что язык не работает без контекста. Было бы странно, если бы отправитель думал, что он говорит *U* (точка-точка-тире), а получатель в это время слышал «утка-утка-гусь».

В оставшейся части этой главы вы узнаете о некоторых распространенных способах присваивать битам значения для вычислений. Имейте в виду, что мы часто будем использовать гипотетические примеры, например, такие как: «Представим, что этот бит означает «синий»». На самом деле программирование так и работает, поэтому, даже изучая некоторые стандартные способы использования битов, не бойтесь изобретать свои, если это уместно.

Логические операции

Биты используются для представления ответов на вопросы типа «да/нет», например: «Холодно?» или «Тебе нравится моя шляпа?». Мы используем термины *истина* для обозначения ответа «да» и *ложь* для обозначения ответа «нет». На вопросы вроде «Где тут вечеринка с собаками?» нельзя ответить «да» или «нет», поэтому их нельзя представить в виде единственного бита.

В человеческом языке мы часто объединяем несколько предложений с «да/нет» в одно. Мы вполне можем сказать: «Наденьте пальто, если холодно или идет дождь» или «Катайтесь на лыжах, если идет снег и не нужно идти в школу». Можно выразить это по-другому: «Наденьте пальто — это истина, если холодно — истина или идет дождь — истина» или «Катайтесь на лыжах — это истина,

если идет снег — истина, а нужно идти в школу — ложь». Это и есть **логические операции**, каждая из которых создает новый бит на основе содержимого других битов.

Булева алгебра

Подобно тому как алгебра — это набор правил для работы с числами, *булева алгебра*, изобретенная в XIX веке английским математиком Джорджем Булем, представляет собой набор правил, которые используются для работы с битами. Как и в случае с обычной алгеброй, в булевой алгебре применяются правила ассоциативности, коммутативности и дистрибутивности.

Существуют три основные логические операции НЕ, И и ИЛИ, а также одна составная операция — исключающее ИЛИ:

- **НЕ (NOT).** Эта операция означает «противоположность». Например, если бит ложный, *НЕ* этот бит будет истинным. Если бит истинен, *НЕ* этот бит будет ложным.
- **И (AND).** В этой операции задействовано 2 бита или более. В 2-битной операции результат будет истинным, только если *И* первый, *И* второй бит истинны. Если задействовано более 2 бит, результат будет истинным только в том случае, если *все* биты истинны.
- **ИЛИ (OR).** Эта операция также включает 2 бита или более. В 2-битной операции результат будет истинным, если первый *ИЛИ* второй бит истинен; в противном случае результат будет ложным. Если бит больше, чем 2, результат будет истинным, если какой-либо из битов истинен.
- **Исключающее ИЛИ (XOR).** Результат операции «исключающее ИЛИ» будет истинным, если первый и второй биты имеют разные значения.

На рис. 1.1 логические операции представлены графически в виде так называемых *таблиц истинности*. Входы находятся снаружи квадратов, а выходы — внутри. В этих таблицах И означает Истина, а Л — Ложь.

НЕ	И	ИЛИ	Исключающее ИЛИ							
л <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>и</td></tr></table>	и	л <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>л</td><td>л</td></tr></table>	л	л	л <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>л</td><td>и</td></tr></table>	л	и	л <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>л</td><td>и</td></tr></table>	л	и
и										
л	л									
л	и									
л	и									
и <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>л</td></tr></table>	л	и <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>л</td><td>и</td></tr></table>	л	и	и <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>и</td><td>и</td></tr></table>	и	и	и <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>и</td><td>л</td></tr></table>	и	л
л										
л	и									
и	и									
и	л									

Рис. 1.1. Таблицы истинности для логических операций

На рис. 1.2 показано, как читать таблицы истинности для операций НЕ и И. Определить выход можно, проследив путь от входа или входов.

НЕ $a = \overline{a}$	НЕ $a \wedge b = \overline{a} \wedge b$	$a \wedge b = a \wedge b$	НЕ $a \wedge b = \overline{a} \wedge b$	$a \wedge b = a \wedge b$

Рис. 1.2. Использование таблиц истинности

Как видите, операция НЕ просто меняет значение входа на противоположное. В то же время, операция И возвращает истину только тогда, когда оба входа истинны.

ПРИМЕЧАНИЕ

Операция «исключающее ИЛИ» основана на других операциях. Например, исключающее ИЛИ двух битов a и b означает то же, что $(a \text{ ИЛИ } b) \text{ И } \text{НЕ } (a \text{ И } b)$. Это показывает, что базовые логические операции можно комбинировать по-разному для получения одного и того же результата.

Закон де Моргана

В XIX веке британский математик Огастес де Морган (Augustus De Morgan) добавил закон, применимый только к булевой алгебре, названный *законом де Моргана*. Этот закон гласит, что операция $a \text{ И } b$ эквивалентна операции НЕ (НЕ a ИЛИ НЕ b), как показано на рис. 1.3.

a	b	$a \wedge b$	НЕ a	НЕ b	НЕ a ИЛИ НЕ b	НЕ (НЕ a ИЛИ НЕ b)
л	л	л	и	и	и	л
л	и	л	и	л	и	л
и	л	л	л	и	и	л
и	и	и	л	л	л	и

Рис. 1.3. Таблица истинности закона де Моргана

Обратите внимание, что результаты $a \text{ И } b$ во втором столбце идентичны результатам, перечисленным в последнем столбце НЕ (НЕ a ИЛИ НЕ b). Это означает, что при достаточном количестве операций НЕ можно заменить операции И операциями ИЛИ (и наоборот). Это полезно, потому что компьютеры работают с настоящими входными данными, которые они не могут контролировать. Хотя было бы неплохо, если бы входные данные всегда были представлены в виде *холодно* или *дождь*, но они часто приходят в виде НЕ *холодно* и НЕ *дождь*. Подобно двойному отрицанию в естественных языках («Это не может не радовать»), закон де Моргана — это инструмент, который позволяет оперировать положениями *отрицательной логики* в дополнение к *положительной логике*, которую мы уже

рассмотрели. На рис. 1.4 показано решение задачи «надеть ли пальто» как для положительной, так и для отрицательной логики.

холодно	дождь	надеть пальто	не холодно	нет дождя	не надевать пальто
Л	Л	Л	Л	Л	Л
Л	И	И	Л	И	Л
И	Л	И	И	Л	Л
И	И	И	И	И	И

Рис. 1.4. Положительная и отрицательная логика

С левой стороны (положительная логика) можно принять решение, используя единственную операцию ИЛИ. С правой стороны (отрицательная логика) закон де Моргана позволяет принимать решение, используя единственную операцию И. Если бы закона де Моргана не существовало, пришлось бы реализовать случай отрицательной логики как *НЕ* не-холодно *ИЛИ* *НЕ* не-дождь. Хотя и это сработает, каждая операция влечет за собой затраты в стоимости и производительности, поэтому сокращение операций минимизирует затраты. Оборудование, которое выполняет операцию *НЕ*, стоит реальных денег, и, как вы узнаете из следующей главы, каскадные операции замедляют работу.

Де Морган говорит, что наш пример эквивалентен варианту «холодно и дождь», что намного проще.

Представление целых чисел с помощью битов

Продвинемся вверх по иерархии и научимся использовать биты для представления чисел. Числа сложнее логики, но намного проще слов.

Представление положительных чисел

Мы обычно используем *десятичную* систему счисления, потому что она соответствует нашей анатомии (у нас 10 пальцев. — *Примеч. ред.*). В контейнеры помещаются десять различных символов, называемых *цифрами*: 0123456789. Контейнеры заполняются справа налево. Каждый контейнер имеет имя, отличное от его содержимого; крайний правый контейнер мы называем единицами, следующий — десятками, затем сотнями, тысячами и т. д. Они являются производными от степеней десятки; 10^0 — один, 10^1 — десять, 10^2 — сто, 10^3 — тысяча. Эта система называется *десятичной*, поскольку 10 является основанием степени. Значение числа получается из суммы произведения каждого значения контейнера и значения его содержимого. Например, число 5028 — это сумма 5 тысяч, 0 сотен, 2 десятков и 8 единиц, или $5 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 8 \times 10^0$, как показано на рис. 1.5.

10^3	10^2	10^1	10^0
5	0	2	8

Рис. 1.5. Число 5028 в десятичной системе счисления

Аналогичный подход можно использовать для записи чисел с помощью битов. Поскольку вместо цифр используются биты, мы имеем только два символа: 0 и 1. Но это не проблема. В десятичном формате новый контейнер добавляется всякий раз, когда не хватает места; 9 можно поместить в один контейнер, но для 10 уже понадобятся 2 контейнера. То же самое в двоичном формате — нам просто нужен новый контейнер для чего-то большего, чем 1. В самом правом контейнере все равно будут единицы, но что мы поместим в следующий? Двойки. Значение контейнера в десятичном формате, в котором содержится 10 символов, в 10 раз больше, чем значение в контейнере справа от него. Таким образом, в двоичном формате с двумя символами значение в выбранном контейнере в два раза больше, чем значение в контейнере справа. Вот и все! Значения контейнера являются степенями двойки, что означает, что это система с *основанием 2*, а не с основанием 10.

В табл. 1.1 перечислены некоторые степени числа 2. Рассмотрим ее как справочный материал, чтобы изучить двоичное представление числа 5028.

Таблица 1.1. Степени двойки

Расширение	Степень	Десятичное число
$2 \div 2$	2^0	1
2	2^1	2
2×2	2^2	4
$2 \times 2 \times 2$	2^3	8
$2 \times 2 \times 2 \times 2$	2^4	16
$2 \times 2 \times 2 \times 2 \times 2$	2^5	32
$2 \times 2 \times 2 \times 2 \times 2 \times 2$	2^6	64
$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$	2^7	128
$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$	2^8	256
$2 \times 2 \times 2$	2^9	512
$2 \times 2 \times 2$	2^{10}	1024

Расширение	Степень	Десятичное число
$2 \times 2 \times 2$	2^{11}	2048
$2 \times 2 \times 2$	2^{12}	4096
$2 \times 2 \times 2$	2^{13}	8192
$2 \times 2 \times 2$	2^{14}	16 384
$2 \times 2 \times 2$	2^{15}	32 768

Каждое число в крайнем правом столбце табл. 1.1 представляет значение двоичного контейнера. На рис. 1.6 показано, как число 5028 может быть записано в двоичной системе с использованием, по сути, тех же вычислений, что и для десятичной записи выше.

2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	1	1	0	1	0	0	1	0	0

Рис. 1.6. Число 5028 в двоичной системе

Результат преобразования в двоичный формат:

$$1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + \\ + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 5028.$$

Как видите, число 5028 в двоичном формате состоит из одного числа 4096 (2^{12}), нуля чисел 2048 (2^{11}), нуля чисел 1024 (2^{10}), одного числа 512 (2^9), одного числа 256 (2^8) и т. д., пока не получится 1001110100100. Выполняя те же вычисления, что и для десятичных чисел, запишем $1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$. Подставляя десятичные числа из табл. 1.1, получим $4096 + 512 + 256 + 128 + 32 + 4$, что равно 5028.

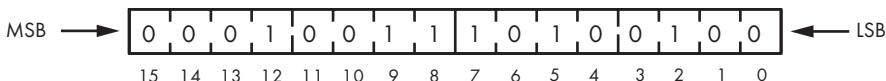
5028 – четырехзначное десятичное число. В двоичном формате его можно представить 13-битным числом.

Количество цифр определяет диапазон значений, которые можно представить в десятичном виде. Например, 100 различных значений в диапазоне 0–99 могут быть представлены двумя цифрами. Точно так же количество битов определяет диапазон значений, которые можно представить в двоичном формате. Например, 2 бита могут представлять четыре значения в диапазоне 0–3. Таблица 1.2 отображает как количество, так и диапазон значений, которые можно представить с разным количеством битов.

Таблица 1.2. Диапазоны значений двоичных чисел

Количество бит	Количество значений	Диапазон значений
4	16	0...15
8	256	0...255
12	4096	0...4095
16	65 536	0...65 535
20	1 048 576	0...1 058 575
24	16 777 216	0...16 777 215
32	4 294 967 296	0...4 294 967 295
64	18 446 744 073 709 551 616	0...18 446 744 073 709 551 615

Крайний правый бит в двоичном числе называется *наименьшим значащим битом*, а крайний левый — *наибольшим значащим битом*, потому что изменение значения самого правого бита оказывает наименьшее влияние на значение числа, а изменение значения самого левого — наибольшее. Компьютерщики любят трехбуквенные аббревиатуры, или TLA (three-letter acronym), поэтому эти биты обычно называют LSB (least significant bit) и MSB (most significant bit) соответственно. На рис. 1.7 показан пример числа 5028, записанного в 16 битах.

**Рис. 1.7. MSB и LSB**

Вы заметите, что, хотя двоичное представление 5028 занимает 13 бит, рис. 1.7 показывает его в 16-битном формате. Как и в десятичной системе счисления, всегда можно использовать больше контейнеров, чем требуется, добавляя *ведущие нули* слева.

В десятичном виде 05 028 имеет то же значение, что и 5028. Двоичные числа часто представлены таким образом, потому что компьютеры созданы на основе битовых блоков.

Сложение двоичных чисел

Теперь, когда вы знаете, как представлять числа в двоичном формате, перейдем к простой арифметике с двоичными числами. В десятичном сложении мы складываем все цифры справа (наименьшая значащая цифра) налево (наибольшая значащая цифра) и, если результат больше 9, переносим 1. Точно так же мы

складываем все биты в двоичных числах, переходя от наименьшего значащего к наибольшему значащему биту, и, если результат больше 1, переносим 1.

На самом деле сложение в двоичном формате немного проще, поскольку существуют только 4 возможные комбинации из 2 бит по сравнению со 100 комбинациями из 2 цифр. Например, на рис. 1.8 показано, как сложить 1 и 5, используя двоичные числа, с учетом цифр над каждым столбцом.

$$\begin{array}{r}
 & \overset{0}{1} & \overset{0}{0} & \overset{0}{1} \\
 & 1 & 0 & 1 \\
 5 & & 1 & 0 & 1 \\
 \hline
 6 & & 1 & 1 & 0
 \end{array}$$

Рис. 1.8. Сложение двоичных чисел

Число 1 — 001 в двоичном формате, а число 5 — 101, потому что $(1 \times 4) + (0 \times 2) + (1 \times 1) = 5$. Чтобы сложить двоичные числа 001 и 101, мы начинаем с наименьшего значащего бита в крайнем правом столбце. Сложение двоичных чисел 1 и 1 в этом столбце дает 2, но символа для 2 в двоичном формате не существует. Однако мы знаем, что 2 на самом деле равно 10 в двоичном формате ($[1 \times 2] + [0 \times 1] = 2$), поэтому пишем 0 в качестве суммы и переносим 1 к следующей цифре. Поскольку средние биты — нули, мы получаем в качестве суммы только 1, перенесенную из предыдущего столбца. Затем складываем цифры в крайнем левом столбце: 0 плюс 1 — просто 1 в двоичном формате. В результате получится двоичное число 110, или 6 в десятичной системе счисления — то же самое, что мы получили бы, сложив 1 и 5.

Вы могли заметить, что правила сложения двоичных чисел могут быть выражены в терминах логических операций, которые мы обсуждали ранее, как показано на рис. 1.9. В главе 2 мы увидим, что именно так компьютеры и выполняют двоичное сложение.

A	B	A И B	A + B	исключающее		A	B
				A ИЛИ B	В		
0	0	0	00	0	0	0	0
0	1	0	01	1	0	0	1
1	0	0	01	1	1	1	0
1	1	1	10	0	1	1	1

Рис. 1.9. Сложение двоичных чисел с использованием логических операций

Результат сложения двух бит тот же, что и для операции «исключающее ИЛИ» с этими двумя битами, а переносимое значение содержит то же, что ИЛИ этих двух бит. Вы можете проверить эту теорию по рис. 1.9, где сложение 1 и 1 в двоичном формате дает 10. В этом примере переносимое значение равно 1 — то же самое, что

и результат операции «1 И 1». Точно так же выражение (1 исключающее ИЛИ 1) дает 0, то есть значение, которое мы присваиваем самой позиции бита.

Сложение двух битов — операция, которая редко выполняется изолированно. Вернитесь к рис. 1.8 — кажется, что мы складываем по 2 бита в каждом столбце, но на самом деле мы складываем 3 бита из-за переноса. К счастью, чтобы сложить 3 бита, не придется изучать ничего нового (потому что $A + B + C$ эквивалентно $(A + B) + C$ согласно сочетательному закону), поэтому можно сложить 3 бита вместе, используя два сложения двух битов.

Что произойдет, если результат сложения не поместится в имеющемся количестве битов? Это приведет к *переполнению*, которое происходит всякий раз, когда мы переносим цифру из наиболее значащего бита. Например, если мы складываем 4-битные числа, например прибавляем $1001 (9_{10})$ к $1000 (8_{10})$, в результате должно получиться число $10001 (17_{10})$, но в итоге получится $0001 (1_{10})$, потому что для наиболее значащего бита не осталось места. Позже мы более подробно рассмотрим, что в компьютерах есть *регистр кодов условий*, который представляет собой место, где хранятся некие фрагменты информации. Один из таких — *бит переполнения*, который содержит значение переноса из наиболее значащего бита. Мы можем проверить это значение, чтобы определить, произошло ли переполнение.

Вы, наверное, знаете, что можно вычесть одно число из другого, прибавив к первому числу отрицательный вариант второго. В следующем разделе мы узнаем, как представлять отрицательные числа. Зимствование за пределами наиболее значащего бита называется *потерей значимости*. Для этого у компьютеров тоже есть код условия.

Представление отрицательных чисел

Все числа, двоичный формат которых мы разбирали в предыдущем разделе, были положительными. Но многие задачи в реальной жизни связаны как с положительными, так и с отрицательными числами. Посмотрим, как можно использовать биты для представления отрицательных чисел. Предположим, что у нас есть 4 бита. Как вы узнали в предыдущем разделе, 4 бита могут представлять 16 чисел в диапазоне от 0 до 15.

Тот факт, что мы можем хранить 16 чисел в 4 битах, не означает, что эти числа должны находиться в диапазоне от 0 до 15. Помните, что язык работает через значение и контекст. Это означает, что можно изобретать новые контексты для интерпретации битов.

Прямой код со знаком

Знак обычно используется, чтобы отличать отрицательные числа от положительных. Знак имеет два значения, плюс и минус, поэтому его можно представить

с помощью бита. Мы произвольно будем использовать крайний левый бит (MSB) для знака, оставив 3 бита, которые могут представлять число от 0 до 7. Если бит знака равен 0, мы считаем это число положительным. Если он равен 1, то число считается отрицательным. Это позволяет представить всего 15 различных положительных и отрицательных чисел, а не 16, потому что существует и положительный, и отрицательный 0. Таблица 1.3 показывает, как с учетом этого можно представить числа от -7 до $+7$.

Это называется представлением в виде *прямого кода со знаком*, потому что есть бит, который представляет знак, и биты, которые представляют величину — то, насколько значение далеко от нуля.

Представление в виде прямого кода со знаком мало используется по двум причинам.

Во-первых, создание битов требует денег, поэтому не стоит тратить их зря на два разных представления для нуля; лучше использовать эту комбинацию битов для представления другого числа. Во-вторых, арифметика с использованием исключающего ИЛИ и И не работает в таком представлении.

Таблица 1.3. Прямой код со знаком для двоичных чисел

Знак	2^2	2^1	2^0	Десятичное число
0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0
1	0	0	0	-0
1	0	0	1	-1
1	0	1	0	-2
1	0	1	1	-3
1	1	0	0	-4
1	1	0	1	-5
1	1	1	0	-6
1	1	1	1	-7

Допустим, нужно прибавить $+1$ к -1 . Мы ожидали получить 0, но при использовании представления в виде прямого кода со знаком получается другой результат, как показано на рис. 1.10.

$$\begin{array}{r}
 & 0 & 0 & 0 & 1 & +1 \\
 + & 1 & 0 & 0 & 1 & -1 \\
 \hline
 & 1 & 0 & 1 & 0 & -2
 \end{array}$$

Рис. 1.10. Сложение с использованием прямого кода со знаком

Как видно, 0001 представляет положительную 1 в двоичной системе, потому что его бит знака равен 0. 1001 представляет -1 в двоичной системе, потому что бит знака равен 1. Сложение их с использованием арифметики исключающего ИЛИ и И дает 1010. Этот результат эквивалентен -2 в десятичной системе счисления, что не равно сумме $+1$ и -1 .

Мы могли бы работать с арифметикой прямого кода, используя более сложную логику, но наша задача — максимально все упростить. Рассмотрим несколько других способов представления чисел и выберем лучший подход.

Обратный код

Еще один способ получить отрицательные числа — взять положительные и инвертировать все биты, что называется *обратным кодом*. Мы разделяем биты аналогично примеру прямого кода со знаком. В этом контексте мы получаем дополнение с помощью операции НЕ. Таблица 1.4 показывает числа в диапазоне от -7 до 7 с использованием обратного кода.

Таблица 1.4. Двоичные числа с обратным кодом

Знак	2^2	2^1	2^0	Десятичное число
0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0
1	1	1	1	-0

Знак	2^2	2^1	2^0	Десятичное число
1	1	1	0	-1
1	1	0	1	-2
1	1	0	0	-3
1	0	1	1	-4
1	0	1	0	-5
1	0	0	1	-6
1	0	0	0	-7

Как видите, инвертирование каждого бита 0111 (+7) дает 1000 (-7).

При использовании обратного кода все еще сохраняется проблема двух представлений нуля, что по-прежнему не позволяет легко выполнять сложение.

Чтобы обойти это, мы применим *круговой перенос*, чтобы добавить 1 к LSB при переносе из наиболее значащей позиции и получить правильный результат (рис. 1.11).

	1	0	1	0	0	0	+2
+		1	1	1	0		-1
	0	0	0	0		0	
+		0	0	0	1		1
	0	0	0	1		1	+1

Рис. 1.11. Сложение в обратном коде

Чтобы сложить $+2$ и -1 с использованием обратного кода, выполним обычное сложение двоичных чисел 0010 и 1110 . Поскольку при добавлении цифр в самый значащий бит (бит знака) получается результат 10 , записываем 0 и круговым переносом переносим 1 для каждой цифры. Но у нас всего лишь 4 бита, поэтому, перейдя к MSB, мы возвращаем перенос к первому биту, чтобы получить 0001 , или $+1$, что является правильной суммой $+2$ и -1 . Как видите, все эти вычисления значительно всё усложняют.

Такой способ работает, но это все же не лучшее решение, потому что нам нужно дополнительное оборудование, чтобы добавить бит для кругового переноса.

В современных компьютерах не используются ни прямой, ни обратный коды. Арифметика с применением этих методов не работает без дополнительного оборудования, а оно стоит денег. Посмотрим, сможем ли мы придумать решение этой проблемы.

Дополнительный код

Что произойдет, если не добавлять никакое оборудование, а просто использовать операции «исключающее ИЛИ» и И? Выясним, какой набор битов при добавлении к $+1$ даст 0 , и назовем его -1 . Если придерживаться 4-битных чисел, число $+1$ будет равно 0001 в двоичной системе. Добавление к нему 1111 даст 0000 , как показано на рис. 1.12, поэтому мы будем использовать этот набор битов для представления -1 .

$$\begin{array}{r}
 +1 \quad \begin{smallmatrix} 1 \\ 0 \end{smallmatrix} \quad \begin{smallmatrix} 1 \\ 0 \end{smallmatrix} \quad \begin{smallmatrix} 1 \\ 0 \end{smallmatrix} \quad \begin{smallmatrix} 0 \\ 1 \end{smallmatrix} \\
 -1 \quad \begin{smallmatrix} 1 \\ 1 \end{smallmatrix} \\
 \hline
 0 \quad \begin{smallmatrix} 0 \\ 0 \end{smallmatrix} \quad \begin{smallmatrix} 0 \\ 0 \end{smallmatrix} \quad \begin{smallmatrix} 0 \\ 0 \end{smallmatrix} \quad \begin{smallmatrix} 0 \\ 0 \end{smallmatrix}
 \end{array}$$

Рис. 1.12. Нахождение -1

Этот способ представления называется *дополнительным кодом*, и это наиболее часто используемое двоичное представление для целых чисел со знаком. Можно получить отрицательное число, дополнив число (то есть выполнив операцию НЕ для каждого бита), а затем добавив 1, отбрасывая любой перенос из MSB. Дополнение к $+1$ (0001) равно 1110 , а добавление 1 дает 1111 вместо -1 . Аналогично $+2$ — это 0010 , его дополнение -1101 , а добавление 1 дает 1110 , представляя -2 . В табл. 1.5 показаны значения в диапазоне от -8 до 7 с использованием дополнительного кода.

Таблица 1.5. Двоичные числа с дополнительным кодом

Знак	2^2	2^1	2^0	Десятичное число
0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0
1	1	1	1	-1
1	1	1	0	-2
1	1	0	1	-3
1	1	0	0	-4

Знак	2^2	2^1	2^0	Десятичное число
1	0	1	1	-5
1	0	1	0	-6
1	0	0	1	-7
1	0	0	0	-8

Проверим то же самое для 0, чтобы увидеть, решает ли дополнительный код проблему двух разных представлений нуля. Если взять 0000 и инвертировать каждый бит, то получим 1111 в качестве его дополнения. Добавление 1 к 1111 дает [1]0000, но, поскольку это 5-битное число, превышающее количество доступных битов, можно не учитывать 1 в бите переноса. Остается 0000, с чего мы и начали, так что ноль имеет только одно представление в дополнительном коде.

Программистам необходимо знать, сколько битов требуется для хранения нужных им чисел. Со временем это войдет в привычку. А пока можно обратиться к табл. 1.6, где показан диапазон значений, который можно представить с помощью дополнительного кода для чисел разного размера.

Таблица 1.6. Диапазоны значений двоичных чисел с дополнительным кодом

Количество битов	Количество значений	Диапазон значений
4	16	-8...7
8	256	-128...127
12	4096	2048...2047
16	65 536	-32 768...32 767
20	1 048 576	-524 288...524 287
24	16 777 216	-8 388 608...8 388 607
32	4 294 967 296	-2 147 483 648...2 137 483 647
64	18 446 744 073 709 551 616	-9 223 372 036 854 775 808... ...9 223 372 036 854 775 807

Как видно из табл. 1.6, по мере увеличения количества битов диапазон представляемых значений увеличивается экспоненциально. Важно помнить, что всегда нужен контекст, чтобы определить, является ли рассматриваемое 4-битное число числом 15 вместо -1 с использованием дополнительного кода, -7 с использованием прямого кода или -0 с использованием обратного кода. Вы должны знать, какое представление используете.

Представление действительных чисел

До сих пор мы представляли целые числа в двоичном формате. А как насчет действительных чисел? Действительные числа включают десятичную точку в основании 10. Нам нужен способ представить эквивалентную двоичную точку в основании 2. Опять же, этого можно добиться путем интерпретации битов в разных контекстах.

Представление с фиксированной точкой

Один из способов представления дробей в двоичном формате — выбор произвольного места для двоичной точки, двоичного эквивалента десятичной точки. Например, если у нас есть 4 бита, мы можем представить, что два из них находятся справа от двоичной точки, представляя четыре дробных значения, а два — слева, представляя четыре целых значения. Это называется представлением *с фиксированной точкой*¹, потому что положение двоичной точки фиксировано (табл. 1.7).

Таблица 1.7. Двоичные числа с фиксированной точкой

Целая часть			Дробная часть		Значение
0	0	.	0	0	0
0	0	.	0	1	$\frac{1}{4}$
0	0	.	1	0	$\frac{1}{2}$
0	0	.	1	1	$\frac{3}{4}$
0	1	.	0	0	1
0	1	.	0	1	$1\frac{1}{4}$
0	1	.	1	0	$1\frac{1}{2}$
0	1	.	1	1	$1\frac{3}{4}$
1	0	.	0	0	2
1	0	.	0	1	$2\frac{1}{4}$
1	0	.	1	0	$2\frac{1}{2}$
1	0	.	1	1	$2\frac{3}{4}$
1	1	.	0	0	3
1	1	.	0	1	$3\frac{1}{4}$
1	1	.	1	0	$3\frac{1}{2}$
1	1	.	1	1	$3\frac{3}{4}$

¹ В российской нотации — числа с фиксированной запятой. — Примеч. ред.

Целые числа слева от точки должны быть знакомы по двоичной системе счисления. Подобно рассмотренным ранее целым числам, мы имеем четыре значения из двух битов справа от точки; они представляют собой четверти вместо привычных десятичных долей.

Хотя этот подход работает довольно хорошо, он нечасто используется в компьютерах общего назначения, поскольку для представления полезного диапазона чисел требуется слишком много битов. Некоторые специализированные компьютеры, называемые *процессорами для цифровой обработки сигналов* (digital signal processor, DSP), по-прежнему используют числа с фиксированной точкой. И, как вы увидите в главе 11, числа с фиксированной точкой особенно полезны в некоторых приложениях.

Компьютеры общего назначения созданы для решения общих задач, работающих с широким диапазоном чисел. Вы можете получить представление об этом диапазоне, полистав учебник физики. Существуют крошечные числа, такие как постоянная Планка (6.63×10^{-34} джоуля в секунду), и огромные числа, например постоянная Авогадро (6.02×10^{23} моль $^{-1}$), то есть диапазон чисел составляет от 10^{57} до 2^{191} . Это почти 200 бит! Биты не настолько дешевы, чтобы использовать их сотнями для представления каждого числа, поэтому нам нужен другой подход.

Представление с плавающей точкой

Эта проблема решается использованием двоичной версии экспоненциальной записи — она применяется для представления широкого диапазона чисел, включая постоянные Планка и Авогадро. Экспоненциальная запись представляет большой диапазон чисел (а как иначе?), создавая новый контекст для интерпретации. Она задействует число с одной цифрой слева от десятичной точки, называемое *мантиссой*, умноженное на 10 в некоторой степени, называемое *экспонентой*. Компьютеры используют эту же систему, за исключением того, что мантисса и экспонента представлены в двоичных числах, а вместо 10 используется 2.

Это называется представлением *с плавающей точкой*¹, что сбивает с толку, потому что двоичная (или десятичная) точка всегда находится в одном и том же месте: между единицей и половинками (десятичные доли в десятичной системе счисления). Представление с плавающей точкой — это просто еще один способ научной записи, которая позволяет записывать 1.2×10^{-3} вместо 0.0012.

Обратите внимание, что нам не нужны дополнительные биты, чтобы указать, что основание равно 2, потому что определение с плавающей точкой подразумевает это по умолчанию. Благодаря отделению значащих цифр от экспонент системы с плавающей точкой позволяет представлять очень маленькие или очень большие числа без необходимости хранить все эти нули.

¹ В российской нотации — числа с плавающей запятой. — Примеч. ред.

В табл. 1.8 показан пример 4-битного представления с плавающей точкой с двумя битами мантиссы и двумя битами экспоненты.

Таблица 1.8. Двоичные числа с плавающей точкой

Мантисса			Экспонента		Значение
0	0	.	0	0	0 (0×2^0)
0	0	.	0	1	0 (0×2^1)
0	0	.	1	0	0 (0×2^1)
0	0	.	1	1	0 (0×2^3)
0	1	.	0	0	0.5 ($S \times 2^0$)
0	1	.	0	1	1.0 ($S \times 2^1$)
0	1	.	1	0	2.0 ($S \times 2^2$)
0	1	.	1	1	4.0 ($S \times 2^3$)
1	0	.	0	0	1.0 (1×2^0)
1	0	.	0	1	2.0 (1×2^1)
1	0	.	1	0	4.0 (1×2^2)
1	0	.	1	1	8.0 (1×2^3)
1	1	.	0	0	1.5 ($1S \times 2^0$)
1	1	.	0	1	3.0 ($1S \times 2^1$)
1	1	.	1	0	6.0 ($1S \times 2^2$)
1	1	.	1	1	12.0 ($1S \times 2^3$)

Хотя в этом примере используется всего несколько битов, мы уже можем заметить некоторые недостатки, присущие системе с плавающей точкой. Во-первых, появляется множество бесполезных комбинаций битов. Например — четыре способа представления 0 и два способа представления 1.0, 2.0 и 4.0. Во-вторых, не существует наборов битов для каждого возможного числа; из-за экспоненты расстояние между числами возрастает по мере их увеличения. Один из побочных эффектов заключается в том, что, хотя мы можем сложить 0.5 и 0.5, чтобы получить 1.0, мы не можем сложить 0.5 и 6.0, потому что не существует набора битов, представляющего 6.5. (Есть целая отрасль математики, называемая *численным анализом*, которая включает отслеживание погрешности вычислений.)

Стандарт IEEE для чисел с плавающей точкой

Как ни странно, система с плавающей точкой является стандартным способом представления действительных чисел в вычислениях. В ней используется больше битов, чем в табл. 1.8, а кроме них — два знака: один для мантиссы и один скрытый (он является частью экспоненты). Также существует множество способов убедиться, что операции наподобие округления работают максимально хорошо, и свести к минимуму количество бесполезных комбинаций битов. Все это описывается в стандарте IEEE 754. IEEE означает Институт инженеров электротехники и электроники (Institute of Electrical and Electronic Engineers) — это организация, деятельность которой включает разработку стандартов.

Мы хотим максимизировать точность с учетом доступных битов. Один из способов сделать это называется *нормализацией* — корректировка мантиссы так, чтобы избежать ведущих (находящихся слева) нулей. Каждая корректировка мантиссы влево требует соответствующей корректировки экспоненты. Еще один прием от Digital Equipment Corporation (DEC) удваивает точность, отбрасывая крайний левый бит мантиссы, поскольку мы знаем, что он всегда будет равен 1, что дает место для еще одного бита.

Вам не нужно подробно разбирать IEEE 754 (по крайней мере, пока). Но стоит учитывать два типа чисел с плавающей точкой, с которыми вы будете часто сталкиваться: числа с одинарной и двойной точностью. Числа с одинарной точностью используют 32 бита и могут представлять числа примерно в диапазоне $\pm 10^{\pm 38}$ с точностью около 7 цифр. Числа с двойной точностью используют 64 бита и могут представлять более широкий диапазон чисел, приблизительно $\pm 10^{\pm 308}$, с точностью около 15 цифр (рис. 1.13).



Рис. 1.13. Форматы чисел с плавающей точкой стандарта IEEE

В обоих форматах учитывается бит знака для мантиссы — буква З на рис. 1.13. Видно, что в числах с двойной точностью имеется на три разряда больше,

чем в числах с одинарной, что дает в восемь раз больший диапазон. Кроме того, в числах с двойной точностью мантисса на 29 бит больше, чем в числах с одинарной, что обеспечивает большую точность. Однако все это возможно за счет использования вдвое большего количества битов, чем для чисел с одинарной точностью.

Вы могли заметить, что для экспонент нет явного бита знака. Разработчики IEEE 754 решили, что экспоненты всех нулей и всех единиц будут иметь особое значение, поэтому фактические экспоненты необходимо втиснуть в оставшиеся наборы битов. Им это удалось благодаря использованию *смещенного* значения экспоненты. Для чисел с одинарной точностью смещение составляет 127. Это означает, что набор битов для 127 (01111111) представляет собой экспоненту 0. Набор битов для 1 (00000001) представляет экспоненту -126, а 254 (11111110) представляет +127. То же самое действует и для двойной точности, за исключением того, что смещение составляет 1023.

Еще одно удобство IEEE 754 заключается в следующем. В него добавлены специальные наборы битов для представления таких операций, как деление на ноль, которое дает положительную или отрицательную бесконечность. Стандарт также определяет специальное значение, называемое *NaN* — от английского «not a number» («не число») — поэтому, если вы получили значение *NaN*, то это, вероятно, означает, что совершенная операция недопустима. Эти специальные комбинации битов используют зарезервированные значения экспоненты, о которых мы уже говорили.

Двоично-десятичная система счисления

Мы рассмотрели некоторые наиболее распространенные способы представления чисел в двоичном формате, однако существует и множество альтернативных систем счисления. Одна из них — *двоично-десятичная система* (binary-coded decimal, BCD), в которой используется 4 бита для представления всех десятичных цифр. Например, число 12 в двоичном формате равно 1100. Но в двоично-десятичном формате оно представлено как 0001 0010, где 0001 — 1 в разряде десятков, а 0010 — 2 в разряде единиц. Это гораздо более привычное и удобное представление для людей, привыкших работать с десятичными числами.

Раньше компьютеры знали, как работать с BCD-числами, но эта система утратила свою популярность. Тем не менее ее все еще можно встретить, поэтому стоит иметь о ней представление. В частности, многие устройства, взаимодействующие с компьютерами, например дисплеи и акселерометры, используют BCD.

Основная причина, по которой BCD потеряла популярность, заключается в том, что она не так эффективно использует биты, как двоичная. Для представления BCD-числа требуется больше битов, чем для его двоичного варианта. Хотя сейчас биты стоят намного дешевле, чем раньше, они все еще не настолько дешевы,

чтобы можно было просто выбросить 6 бит из всех 16-битных комбинаций, поскольку это было бы равносильно потере колоссального количества доступных битов (37.5 %).

Более простые способы работы с двоичными числами

Доподлинно известно, что работа с двоичными числами может привести к слепоте — настолько она визуально утомительна! Люди придумали несколько способов облегчить чтение двоичных чисел. Рассмотрим некоторые из них.

Восьмеричное представление

Один из подходов, которые не испортят вам зрение, — *восьмеричное представление*. Восьмеричный означает «имеющий основание 8», и идея состоит в том, чтобы сгруппировать биты в тройки. К этому моменту вы уже должны знать, что 3 бита можно использовать для представления 2^3 , или восьми значений от 0 до 7. Допустим, есть чудовищное двоичное число, такое как 100101110001010100. На него просто больно смотреть. На рис. 1.14 показано, как преобразовать его в восьмеричное представление.

100	101	110	001	010	100
4	5	6	1	2	4

Рис. 1.14. Восьмеричное представление двоичных чисел

Таким образом, мы делим биты на группы по три бита, присваиваем каждой группе восьмеричное значение и в результате получаем число 456124, прочитать которое гораздо проще. Например, чтобы получить восьмеричное значение 100, мы просто переводим его в двоичное число: $(1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4$.

Шестнадцатеричное представление

Восьмеричное представление все еще используется, но не так часто, как раньше. Пальма первенства перешла к *шестнадцатеричному представлению* (основание 16), потому что в наши дни начинка компьютеров создается с точностью до 8 бит, которые делятся без остатка на 4, но не на 3.

Было легко воспринимать некоторые знакомые цифры как двоичные, потому что мы использовали всего две цифры — 0 и 1. Для восьмеричной системы счисления понадобилось 8 цифр из 10. Для шестнадцатеричной системы нужно взять 16 цифр, но у нас столько нет. Нам нужен отдельный символ для

представления 10, еще один для 11, и так до 15. Предположим (я уже говорил, что мы будем так делать), что символы abcdef (или ABCDEF) представляют значения от 10 до 16. Допустим, у нас есть еще одно страшное двоичное число — 1101001111111000001.

На рис. 1.15 показано, как преобразовать его в шестнадцатеричный формат.

1101	0011	1111	1100	0001
d	3	f	c	1

Рис. 1.15. Шестнадцатеричное представление двоичных чисел

В этом примере мы делим биты на четыре группы. Затем присваиваем каждой группе один из 16 символов (0123456789abcdef). Например, 1101 (первая группа из 4 бит) будет представлена как d, потому что она вычисляется как $1(2^3) + 1(2^2) + 0(2^1) + 1(2^0) = 13$ в десятичной системе счисления, а число 13 соответствует символу d. Сопоставляем следующую группу из 4 бит (0011) с другим символом и т. д. Например, 1101001111111000001 преобразуется в d3fc1 в шестнадцатеричном формате. В табл. 1.9 представлен удобный список шестнадцатеричных значений, к которому вы можете обращаться, пока не привыкнете переводить их из одной системы в другую самостоятельно.

Таблица 1.9. Преобразование двоичного числа в шестнадцатеричное

Двоичное	Шестнадцатеричное	Двоичное	Шестнадцатеричное
0000	0	1000	8
0001	1	1001	9
0010	2	1010	a
0011	3	1011	b
0100	4	1100	c
0101	5	1101	d
0110	6	1110	e
0111	7	1111	f

Представление контекста

Откуда узнать, как интерпретировать число? Например, число 10 обозначает 2, если это двоичное число, 8, если восьмеричное, 10, если десятичное, и 16, если

шестнадцатеричное. В книгах по математике используются индексы, поэтому можно применять их, чтобы различать числа: 10_2 , 10_8 , 10_{10} или 10_{16} . Но индексы неудобно набирать с клавиатуры. Было бы неплохо использовать одни и те же обозначения, но, к сожалению, многие думают, что найдут лучший способ, и продолжают изобретать новые. Во многих языках программирования встречаются следующие обозначения:

- Число, начинающееся с 0, является восьмеричным, например 017.
- Число, которое начинается с одной из цифр от 1 до 9, является десятичным числом, например 123.
- Число с префиксом 0x является шестнадцатеричным, например 0x12f.

Обратите внимание, что мы не можем отличить восьмеричный и десятичный 0, но это не важно, потому что они имеют одинаковое значение. И лишь в немногих языках есть обозначение для двоичного кода, потому что числа в двоичной системе на самом деле используются редко и обычно их можно определить по контексту. Некоторые языки, такие как C++, используют префикс 0b для представления двоичных чисел.

Именованные группы битов

Компьютеры — это не просто неорганизованные связки битов. Люди, которые проектируют компьютеры, должны принимать решения о количестве битов и их организации из соображений стоимости. Как и в случае с представлениями чисел, было опробовано много идей, но только некоторые из них прижились.

Биты слишком малы, чтобы быть полезными сами по себе, поэтому их чаще всего собирают в более крупные блоки. Например, компьютеры серии Honeywell 6000 использовали 36-битные блоки в качестве базовой структуры и позволяли разбивать их на 18-, 9- или 6-битные блоки или объединять в 72-битные блоки. DEC PDP-8, первый коммерческий мини-компьютер (представленный в 1965 году), использовал 12-битные блоки. Со временем мир остановился на 8-битных блоках как на фундаментальной единице, которая была названа *байтом*.

Фрагменты разного размера имеют собственные имена, чтобы на них было легче ссылаться. В табл. 1.10 приведены названия и размеры некоторых распространенных блоков, используемых сегодня.

Вы спросите, почему мы используем полуслова, длинные и двойные слова, но не простые слова. Слово используется для описания естественного размера объектов в конкретном компьютерном дизайне. Естественный размер относится к самому большому фрагменту, с которым можно быстро работать. Например, на DEC PDP-11 можно было работать с байтами, полусловами и длинными словами, но

его внутренняя структура была 16-битной, поэтому и естественный размер для этого компьютера был 16-битным. Такие языки программирования, как C и C++, позволяют объявлять переменные как `int` (сокращение от *integer*) — переменные естественного размера. Также можно объявлять переменные, используя набор поддерживаемых определенных размеров.

Таблица 1.10. Названия коллекций битов

Имя	Число битов
Полубайт	4
Байт	8
Полуслово	16
Длинное слово	32
Двойное слово	64

Существует несколько стандартных терминов, которые позволяют легко ссылаться на большие числа. Сначала был один стандарт, а позже его заменили на новый. У инженеров есть привычка находить слова, которые означают что-то близкое к тому, что они хотели бы выразить, а затем использовать их, как если бы эти слова действительно имели нужное значение. Например, в метрической системе *кило* означает тысячу, *мега* означает миллион, *гига* — миллиард, а *тера* — триллион. Эти термины были заимствованы, но немного изменены, потому что в вычислениях используется основание 2 вместо 10.

Однако когда мы говорим о *килобите* или *килобайте* (Кбит или Кбайт) в вычислениях, то на самом деле не имеем в виду тысячу. Мы имеем в виду число, ближайшее к тысяче в основании 2, что равно 1024, или 2^{10} . То же самое касается *мегабайта* (Мбайт), который равен 2^{20} , *гига* (Гбайт) — 2^{30} , и *тера* (Тбайт) — 2^{40} .

Но в некоторых случаях имеется в виду версия с основанием 10. Чтобы определить нужную интерпретацию, необходим контекст. Традиционно версия с основанием 10 использовалась для обозначения размера дисковых накопителей. Американский адвокат притворился, что ничего не знает об этом, и подал в суд (на компанию *Safier v. WDC*), утверждая, что объем дискового накопителя меньше заявленного. (На мой взгляд, это было так же глупо, как и судебные иски, утверждающие, что размер пиломатериалов 2×4 в действительности не 2 на 4 дюйма, несмотря на то что именно так всегда обозначались размеры неоструктурированных, необработанных досок.) Это привело к созданию новых стандартных префиксов IEC: киби (КиБ) для 2^{10} , меби (МиБ) для 2^{20} , гиби (ГиБ) для 2^{30}

и теби (ТиБ) для 2⁴⁰. Они постепенно завоевывают популярность, хотя «киби» для меня звучит как название корма для собак.

Термин *символ* часто используется как синоним слова *байт*, потому что, как мы увидим в следующем разделе, коды символов обычно разрабатывались так, чтобы соответствовать байтам. Теперь, с развитием поддержки неанглийских языков, часто возникает необходимость говорить о многобайтовых символах.

Представление текста

На этом этапе вы узнали, что биты — это все, с чем нам нужно работать в компьютерах, и что их можно использовать для представления чего-либо, например чисел.

Пришло время перейти на следующий уровень и использовать числа для обозначения чего-то еще, например букв и других символов на клавиатуре.

Американский стандартный код обмена информацией

Как и в случае с числами, появилось несколько конкурирующих идей для представления текста. Победителем еще в далеком 1963 году стал *Американский стандартный код обмена информацией* (American Standard Code for Information Interchange, ASCII). В его версии всем символам на клавиатуре были присвоены 7-битные числовые значения. Например, 65 означает заглавную А, 66 — заглавную В и т. д. Проиграл же стандарт IBM, *Расширенный двоично-десятичный код обмена информацией* (Extended Binary-Coded Decimal Interchange Code, EBCDIC), основанный на кодировке, используемой для перфокарт. И да, часть «BCD» в EBCDIC обозначает ту же двоично-десятичную систему, которую мы видели ранее. В табл. 1.11 показаны коды ASCII.

Найдем в этой таблице букву А. Видим, что она имеет значение 65 в десятичной системе счисления, что составляет 0x41 в шестнадцатеричной системе, а также 0101 в восьмеричной. Как оказалось, коды символов ASCII — это то место, где восьмеричные числа по-прежнему широко используются по историческим причинам.

В таблице ASCII много забавных кодов. Их называют управляющими символами, потому что они чем-то управляют, а не выводятся на печать. В табл. 1.12 показаны их значения.

Многие из них предназначались для управления передачей сообщений. Например, ACK (подтверждение) означает «я получил сообщение», а NAK (отрицательное подтверждение) — «я не получил сообщение».

Таблица 1.11. Таблица кодов ASCII

Де- ся- тич- ное	Вось- ме- рич- ное	Сим- вол	Де- ся- тич- ное	Вось- ме- рич- ное	Сим- вол	Де- ся- тич- ное	Вось- ме- рич- ное	Сим- вол	Деся- тич- ное	Вось- ме- рич- ное	Сим- вол
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	\$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(72	48	H	104	68	h
9	09	HT	41	29)	73	49	I	105	69	i
10	0A	NL	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	5	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Таблица 1.12. Управляющие символы ASCII

NUL	Пустой символ	SOH	Начало заголовка
STX	Начало текста	ETX	Конец текста
EOT	Конец передачи	ENQ	Запрос
ACK	Подтверждение	BEL	Звуковой сигнал
BS	Возврат на шаг	HT	Горизонтальная табуляция
NL	Новая строка	VT	Вертикальная табуляция
FF	Прогон страницы	CR	Возврат каретки
SO	Переключение на другую ленту	SI	Переключение на исходную ленту
DLE	Экранирование управляющих символов	DC1	1-й код управления устройством
DC2	2-й код управления устройством	DC3	3-й код управления устройством
DC4	4-й код управления устройством	NAK	Отрицательное подтверждение
SYN	Пустой символ для синхронного режима передачи	ETB	Конец блока передаваемых данных
CAN	Отмена	EM	Конец носителя
SUB	Символ замены	ESC	Управляющая последовательность
FS	Разделитель файлов	GS	Разделитель групп
RS	Разделитель записей	US	Разделитель полей
SP	Пробел	DEL	Удаление

Развитие других стандартов

Некоторое время было достаточно стандарта ASCII, потому что он содержал символы, необходимые для английского языка. Большинство первых компьютеров были американскими, а остальные – британскими. Потребность в поддержке других языков росла по мере того, как компьютеры становились более доступными. *Международная организация по стандартизации* (International Standards Organization, ISO) приняла стандарты ISO-646 и ISO-8859, которые в основном представляют собой ASCII с некоторыми расширениями для символов ударения и других диакритических знаков, используемых в европейских языках. Комитет по *японским промышленным стандартам* (Japanese Industrial Standards, JIS) разработал стандарт JIS X 0201 для японских иероглифов. Также существуют китайские, арабские стандарты и многие другие.

Одна из причин создания всех этих стандартов заключается в том, что они были разработаны в то время, когда биты стоили намного дороже, чем сегодня, поэтому символы были упакованы в 7 или 8 бит. Когда цена битов начала падать, был разработан новый стандарт *Unicode*, который назначал символам 16-битные коды.

В то время считалось, что 16 бит будет достаточно, чтобы вместить все символы на всех языках Земли с запасом места. С тех пор Unicode был расширен до 21 бита (из которых действительны 1 112 064 значения), что должно сработать, но, скорее, всего ненадолго, учитывая нашу любовь к созданию эмодзи с котиками.

8-битная форма представления Unicode

Компьютеры используют 8 бит для хранения символа ASCII, потому что они не предназначены для обработки 7-битных величин. Опять же, хотя биты стали намного доступнее, чем раньше, все же они не настолько дешевы, чтобы использовать 16 из них для хранения одной буквы, если можно обойтись восемью. Unicode решает эту проблему, используя разные кодировки для кодов символов. Кодировка — это комбинация битов, которая представляет другую комбинацию битов. Совершенно верно — мы используем абстракции (биты) для создания чисел, представляющих символы, а затем используем другие числа для представления этих чисел! Понимаете, что я имел в виду под *гипотетическим значением*? В частности, существует кодировка, называемая *8-битной формой представления Unicode* (Unicode Transformation Format-8 bit, UTF-8), придуманная американским программистом Кеном Томпсоном (Ken Thompson) и его канадским коллегой Робом Пайком (Rob Pike). Эта кодировка чаще всего используется благодаря своей эффективности и обратной совместимости. UTF-8 назначает 8 бит каждому символу ASCII, поэтому они не занимают дополнительного места для данных ASCII. Символы, отличные от ASCII, кодируются таким образом, чтобы не нарушать работу программ, ожидающих ввода ASCII-символов.

UTF-8 кодирует символы как последовательность 8-битных блоков, часто называемых *октетами*. Одно из удобств UTF-8 заключается в том, что количество наиболее значащих единиц в первом блоке определяет длину последовательности, потому первый блок легко распознать. Это полезно, поскольку программам легко находить границы символов. Все символы ASCII умещаются в 7 битах, поэтому на каждый символ приходится ровно один фрагмент, что довольно удобно для англоговорящих, поскольку для хранения символов английского языка нужно меньше места, чем для других языков, которым необходимы не-ASCII-символы.

На рис. 1.16 показано, как UTF-8 кодирует символы по сравнению с Unicode.

На рис. 1.16 видно, что числовой код буквы *A* одинаков в ASCII и Unicode. Чтобы закодировать *A* в UTF-8, мы обговариваем условие, что любые коды, которые умещаются в 7 бит, включаются в один фрагмент UTF-8, и задаем MSB равным 0. Вот почему в кодировке UTF-8 для буквы *A* в начале стоит 0. Далее видно, что Unicode-вариант для символа π не умещается в 7 битах, ему нужно 11. Для кодирования π в UTF-8 мы используем два 8-битных фрагмента, первый из которых начинается со 110, а второй — с 10, в результате чего в каждом фрагменте остается 5 и 6 бит соответственно для хранения оставшегося кода. Наконец, Unicode для символа ♣ помещается в 16 битах и поэтому занимает три фрагмента UTF-8.

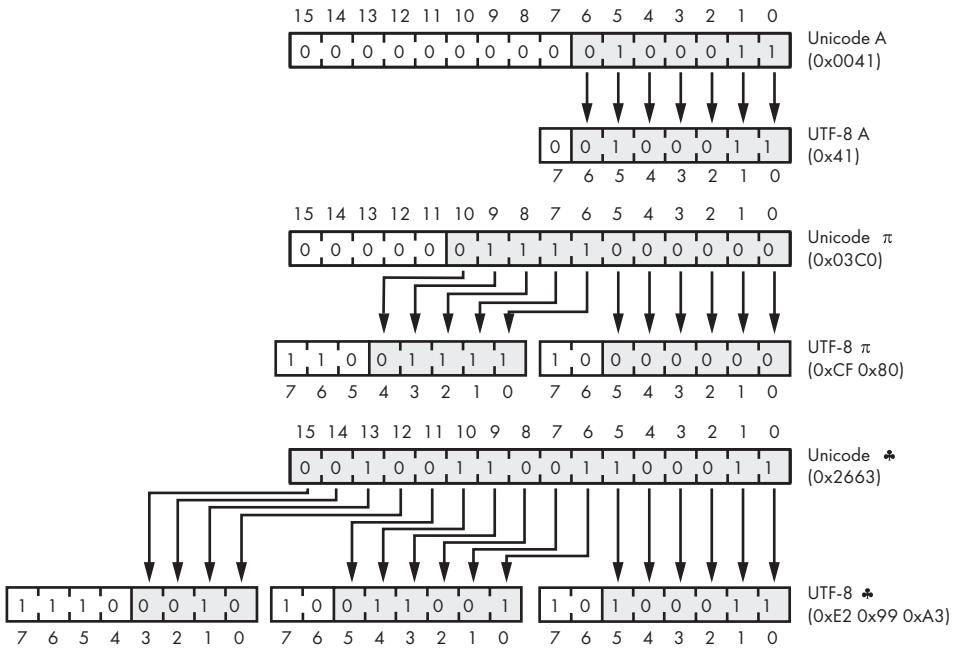


Рис. 1.16. Примеры кодировки UTF-8

Использование символов для представления чисел

UTF-8 использует числа для представления чисел, представляющих числа, состоящие из битов, представляющих символы. Но и это еще не все! Теперь мы собираемся применять символы для обозначения некоторых из этих чисел. На заре межкомпьютерных коммуникаций люди хотели передавать между компьютерами нечто большее, чем просто текст; нужно было отправлять двоичные данные. Но сделать это было непросто, потому что, как мы видели ранее, многие значения ASCII были зарезервированы для управляющих символов и не обрабатывались последовательно между системами. Также некоторые системы поддерживали передачу только 7-битных символов.

Кодировка Quoted-Printable

Кодировка *Quoted-Printable*, также известная как *QP*-кодировка, — это механизм, позволяющий передавать 8-битные данные по пути, который поддерживает только 7-битные данные. Она была создана для вложений электронной почты. Эта кодировка позволяет представить любое 8-битное байтовое значение тремя символами: символом **=**, за которым следует пара шестнадцатеричных чисел, по одному для

каждого полубайта. Конечно, при этом знак = получает особое значение и поэтому должен быть представлен с использованием =3D, его значения из табл. 1.11.

В кодировке Quoted-Printable есть несколько дополнительных правил. Символы табуляции и пробела должны быть представлены как =09 и =20 соответственно, если они расположены в конце строки. Длина закодированных строк не может превышать 76 символов. Знак = в конце строки — это мягкий разрыв строки, который удаляется при декодировании данных получателем.

Кодировка Base64

Хотя кодировка Quoted-Printable работает, она не очень эффективна, поскольку для представления единственного байта требуются три символа. Более эффективна кодировка Base64, и она действительно имела большое значение, когда обмен данными между компьютерами занимал гораздо больше времени, чем сегодня. Кодировка Base64 упаковывает 3 байта данных в 4 символа. 24 бита данных в трех байтах разделены на четыре 6-битных блока, каждому из которых назначается печатный символ, как показано в табл. 1.13.

Таблица 1.13. Кодировка символов Base64

Число	Символ	Число	Символ	Число	Символ	Число	Символ
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	а	42	q	58	6
11	Л	27	б	43	р	59	7
12	М	28	с	44	с	60	8
13	Н	29	д	45	т	61	9
14	О	30	е	46	у	62	+
15	Р	31	ф	47	в	63	/

Байты 0, 1, 2 будут закодированы как ААЕС (рис. 1.17).

0	1	2
0 0 0 0 0 0	0 0 0 0	0 0 0 1
0(A)	0(A)	4(E)

0 0 0 0 0 0	0 0 0 0	0 0 0 1	0 0	0 0 0 1 0
0(A)	0(A)	4(E)		2(C)

Рис. 1.17. Кодировка Base64

Эта кодировка преобразует каждый набор из трех байтов в четыре символа. Но при этом не гарантируется, что длина данных будет кратна трем байтам. Проблема решается с помощью символов *заполнения*; символ = будет добавлен в конец строки, если строка занимает только 2 байта, либо же будут добавлены символы ==, если строка займет всего 1 байт.

Кодировка Base64 до сих пор часто используется для вложений электронной почты.

Кодировка URL

В одном из предыдущих разделов вы видели, что кодировка Quoted-Printable наделила особой силой символ = и что эта же кодировка включала механизм для представления обычного символа =. Практически идентичная схема используется в URL-адресах веб-страниц.

Если вы когда-нибудь рассматривали URL-адрес веб-страницы, вы могли заметить такие последовательности символов, как %26 и %2F. Они существуют потому, что определенные символы имеют особое значение в контексте URL-адреса. Но иногда нам нужно использовать эти символы как литералы — другими словами, без специальных значений.

Как мы уже видели, символы представлены в виде последовательности 8-битных блоков. Каждый фрагмент может быть представлен двумя шестнадцатеричными символами, как показано на рис. 1.16. *Кодировка URL*, также известная как *процентная кодировка*, заменяет символ на знак %, за которым следует шестнадцатеричное представление этого символа.

Например, косая черта (/) имеет особое значение в URL-адресах. Она имеет код 47 в ASCII, что равно 2F в шестнадцатеричном формате. Если нужно использовать / в URL-адресе, не учитывая его особое значение, мы заменим его на %2F. (И поскольку мы только что придали особое значение символу %, его необходимо заменить на %25, если мы буквально имеем в виду %.)

Представление цветов

Еще одно распространенное использование чисел — представление цветов. Вы уже знаете, что числа можно использовать для отображения координат на

графике. Компьютерная графика предполагает создание изображений путем нанесения капель цвета на эквивалент электронной миллиметровой бумаги. Капля, нанесенная на каждую пару координат, называется *элементом изображения*, или, чаще, *пикселием*.

Компьютерные мониторы генерируют цвета, смешивая красный, зеленый и синий основные цвета, таким образом применяя *цветовую модель RGB* (от red, green, blue — «красный, зеленый, синий»). Ее можно представить в виде *цветового куба*, в котором каждая ось представляет основной цвет, как показано на рис. 1.18. Значение 0 означает, что конкретный основной цвет не используется, а 1 означает, что он максимально яркий.

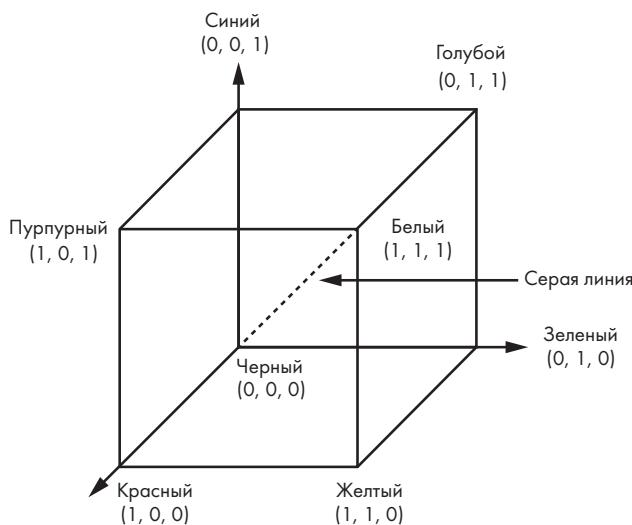


Рис. 1.18. Цветовой куб RGB

Видим, что черный цвет обозначает, что ни один основной цвет не используется, а белый — что все основные цвета имеют максимальную яркость. Оттенок красного цвета получается, если включается только основной красный цвет. Смешивание красного и зеленого дает желтый оттенок. Серый цвет получается в результате выбора одного и того же уровня для всех трех источников цвета. Этот способ смешивания называется *системой сложения цветов*, поскольку сложение основных цветов дает разный результат. На рис. 1.19 показаны координаты нескольких цветов в кубе.

Если вы пробовали рисовать, то наверняка лучше знакомы с *субтрактивной* системой цветов, в которой основными цветами являются голубой, пурпурный и желтый. Субтрактивная система цветов определяет цвета путем удаления длин волн белого света, а не добавления цветного света, как в системе сложения. Хотя ни одна из систем не может воспроизвести все цвета, которые видят глаза,

в субтрактивной системе доступно больше оттенков, чем в системе сложения. Существует целый набор *предпечатных* технологий, благодаря которым дизайн будет выглядеть одинаково как на экране монитора, так и в печатном издании. Если вас действительно интересует тема цвета, прочтите книгу Морин Стоун (Maureen Stone) «A Field Guide to Digital Color».

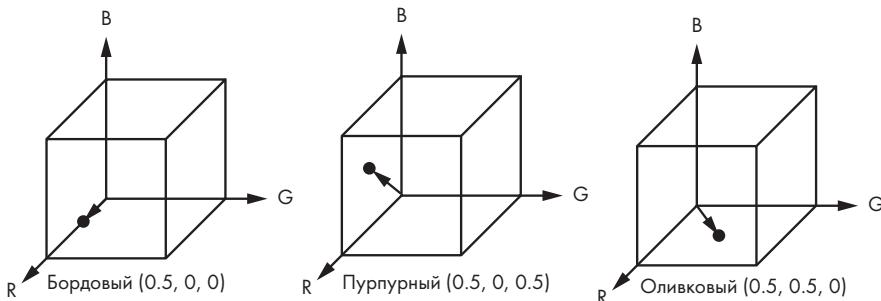


Рис. 1.19. Примеры цветового куба RGB

Человеческий глаз — очень сложный механизм, созданный для выживания, а не для вычислений. Он может различать около 10 миллионов цветов, но при этом он нелинейен; удвоение уровня освещенности не обязательно приводит к удвоению воспринимаемой яркости. Хуже того, реакция глаза медленно меняется со временем в зависимости от общего уровня освещенности. Это называется *адаптацией к темноте*. К тому же глаз реагирует на цвета по-разному; он очень чувствителен к изменениям зеленого цвета, в отличие от синего, — этот феномен использовался в стандарте Национального комитета по телевизионным системам (National Television System Committee, NTSC). В современных компьютерах решено округлить 10 миллионов до ближайшей степени двойки и использовать 24 бита для представления цвета. Эти 24 бита разделены на три 8-битных поля, по одному для каждого из основных цветов.

Можно заметить, что в табл. 1.10 нет названия для 24 бит. Все потому, что современные компьютеры не предназначены для работы на 24-битных устройствах (хотя существовало несколько 24-битных машин, таких как Honeywell DDP-224). В результате цвета упаковываются в ближайший стандартный размер, который составляет 32 бита (*длинное слово*), как показано на рис. 1.20.



Рис. 1.20. Хранение цветов RGB

Видим, что эта схема оставляет 8 неиспользуемых бит для каждого цвета. Это много, учитывая, что в компьютерных мониторах сегодня более 8 миллионов

пикселей. Нельзя просто позволить этим битам пропасть зря, так что же с ними делать? Можно использовать их для чего-то, чего не хватает в обсуждении цвета выше: *прозрачности*, то есть способности «видеть сквозь» цвет. До сих пор мы обсуждали только непрозрачные цвета, но их нельзя использовать, например, для розовых очков.

Добавление прозрачности

В ранних анимационных фильмах каждый кадр рисовали вручную. Во-первых, это была очень трудоемкая работа, а во-вторых, появлялся эффект визуального «дрожания», потому что было невозможно точно воспроизвести фон на каждом кадре.

Американские аниматоры Джон Брэй (John Bray) (1879–1978) и Эрл Херд (Earl Hurd) (1880–1940) решили эту проблему: в 1915 году они изобрели *целлулоидную анимацию*. Движущихся персонажей рисовали на прозрачных листах целлулоида, которые затем можно было перемещать по статическому фоновому изображению.

Хотя компьютерная анимация уходит корнями в 1940-е годы, она стала по-настоящему популярной в 1970-х и 1980-х годах. В то время компьютеры были недостаточно быстрыми, чтобы делать все, что хотели режиссеры (подозреваю, для режиссеров они никогда не будут достаточно быстрыми). И нам понадобился механизм для объединения объектов, сгенерированных разными алгоритмами. Как и целлулоидная анимация, прозрачность позволяет создавать *композиции* или комбинировать изображения из разных источников. Вам, вероятно, знакомо это понятие, если вы когда-нибудь работали в графическом редакторе, таком как GIMP или Photoshop.

В 1984 году Том Дафф (Tom Duff) и Томас Портер (Thomas Porter) из Lucasfilm изобрели способ реализации прозрачности и композиции, который стал стандартом. Они добавили значение прозрачности под названием *альфа* (α) к каждому пикселию. α — математическое значение от 0 до 1, где 0 означает, что цвет полностью прозрачен, а 1 — что цвет полностью непрозрачен. Набор уравнений *композиционной алгебры* определяет, как цвета с разными альфа-значениями объединяются для создания новых цветов.

Дафф и Портер продумали грамотную реализацию. Поскольку они не применяют систему с плавающей точкой, значение α , равное 1, они представляют с числом 255, используя преимущества этих дополнительных 8 бит (см. рис. 1.20). Вместо того чтобы сохранять красный, зеленый и синий цвета, Дафф и Портер сохраняют значения цвета, умноженные на α . Например, если бы цвет был средне-красным, он имел бы значение 200 для красного и 0 для зеленого и синего. Значение красного было бы равно 200, если бы цвет был непрозрачным, потому что α было бы равно 1 (со значением α , равным 255). Но α наполовину прозрачного среднего красного цвета будет равно 0.5, поэтому сохраненное значение для красного

цвета будет равно $200 \times 0.5 = 100$, а сохраненное значение $\alpha = 127$ ($255 \times 0.5 = 127$). На рис. 1.21 показано, как работает хранение пикселей с α .

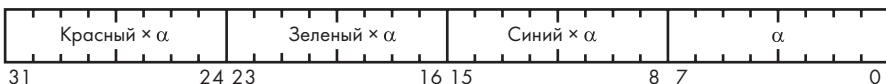


Рис. 1.21. Хранение цветов RGB α

Следовательно, композиция изображений включает в себя умножение значений цвета на α . Хранение цветов в предварительно умноженной форме означает, что нам не нужно выполнять это умножение каждый раз, когда используется пиксель.

Кодирование цветов

Поскольку веб-страницы — это в основном *текстовые* документы, то есть представляют собой последовательность удобочитаемых символов (часто в UTF-8), нам нужен способ представления цветов с помощью текста.

Это делается аналогично кодировке URL: цвета определяются с помощью *шестнадцатеричных троек*. Шестнадцатеричная тройка — это символ #, за которым следуют шесть шестнадцатеричных значений в формате #*rrggbb*, где *rr* — значение красного цвета, *gg* — зеленого, а *bb* — синего. Например, #ffff00 означает желтый цвет, #000000 — черный, а #ffffff — белый. Каждое из трех 8-битных значений цвета преобразуется в двухсимвольное шестнадцатеричное представление.

Хотя α также доступно на веб-страницах, не существует краткого формата для его представления. Оно полностью использует дополнительный набор схем.

Выводы

Из этой главы вы узнали, что, несмотря на концептуальную простоту битов, их можно использовать для представления сложных вещей, таких как очень большие числа, символы и даже цвета. Вы разобрались, как представлять десятичные числа в двоичном формате, выполнять простую арифметику с использованием двоичных чисел и представлять отрицательные числа и дроби. Вы также узнали о различных стандартах кодирования букв и символов с помощью битов.

Гики шутят: «В мире есть 10 типов людей — те, кто понимает двоичные числа, и те, кто нет». Теперь вы в первой группе.

В главе 2 вы познакомитесь с основами работы аппаратной части компьютеров — мы разберем их физические компоненты и то, почему они вообще используют биты.

2

Комбинаторная логика



В эпизоде «Город на краю вечности» из сериала «Звездный путь» 1967 года Спок говорит: «Мэм, я пытаюсь построить мнемоническую схему памяти с помощью каменных ножей и медвежьих шкур». Подобно Споку, люди придумали всевозможные гениальные способы создания вычислительных устройств с использованием доступных ресурсов. Некоторые фундаментальные технологии были созданы специально для вычислений; а многие — изобретены для других целей, а затем *адаптированы* для вычислений. В этой главе мы рассмотрим некоторые аспекты этой эволюции, в том числе удобное и довольно молодое изобретение — электричество.

Из главы 1 вы узнали, что современные компьютеры используют двоичные контейнеры, называемые *битами*, в качестве внутреннего языка. Вам может быть интересно, почему компьютеры применяют именно их, а не удобные для людей десятичные числа. Чтобы понять, почему в современных технологиях принято использовать биты, эту главу мы начнем с рассмотрения некоторых ранних вычислительных устройств, которые этого *не* делали. Биты сами по себе неудобны для вычислений, поэтому мы поговорим о том, как упростить их использование. Мы разберем некоторые более старые и простые технологии — механические реле и вакуумные лампы, — а затем сравним их с современной реализацией битов в аппаратном обеспечении с применением электричества и интегральных схем.

В главе 1 биты обсуждались в довольно абстрактной форме, а здесь мы перейдем к деталям. Физические устройства, в том числе те, которые работают с битами, называются *аппаратным обеспечением*. Мы поговорим об аппаратном

обеспечении, которое реализует *комбинаторную логику*, или булеву алгебру, которую мы обсуждали в главе 1. Как и в предыдущей главе, сначала рассмотрим простые строительные блоки, а затем объединим их, чтобы получить расширенную функциональность.

Задача для цифровых компьютеров

Для начала рассмотрим некоторые механические вычислительные устройства на основе шестерен, которые появились еще до современной эпохи. Для двух соединенных шестерен соотношение количества их зубьев определяет относительную скорость, что делает их полезными для умножения, деления и других вычислений. Одним из механических устройств с зубчатой передачей является антикитерский механизм, старейший известный пример компьютера, найденный на греческом острове и датируемый примерно 100 годом до н. э. С помощью этого механизма выполнялись астрономические расчеты — пользователь вводил дату, поворачивая циферблат, а затем вращал рукоятку, чтобы получить положение Солнца и Луны в этот день. Другой пример — компьютеры управления артиллерией времен Второй мировой войны, которые выполняли тригонометрические и дифференциальные вычисления с использованием множества шестерен странной формы со сложной конструкцией, что делало их настоящими произведениями искусства.

Примером механического компьютера без шестеренок является *логарифмическая линейка*, изобретенная английским священником и математиком Уильямом Отредом (William Oughtred) (1574–1660). Она основана на логарифмах, придуманных шотландским физиком, астрономом и математиком Джоном Напьером (John Napier) (1550–1617). Главная функция логарифмической линейки — выполнять умножение, используя тот факт, что $\log(x \times y) = \log(x) + \log(y)$.

Логарифмическая линейка имеет фиксированную и подвижную шкалы. Произведение двух чисел вычисляется путем совмещения фиксированной шкалы x с подвижной шкалой y , как показано на рис. 2.1.

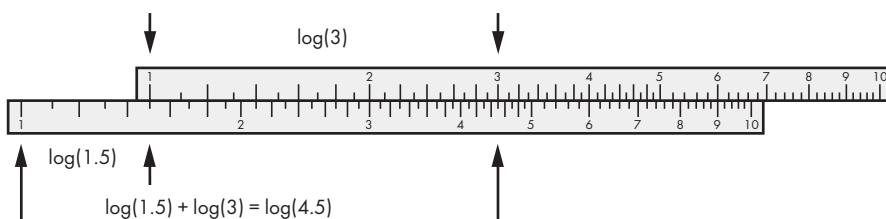


Рис. 2.1. Сложение с помощью логарифмической линейки

Логарифмическая линейка считается первым массовым вычислительным устройством. Она представляет собой отличный пример того, как люди решали

проблему с помощью доступных им в то время технологий. Сегодня пилоты самолетов все еще используют в качестве резервного устройства круговую версию логарифмической линейки — бортовой компьютер, выполняющий вычисления, связанные с навигацией.

Подсчет — исторически важное применение вычислительных устройств. Поскольку количество пальцев у человека ограничено и они нужны ему для других целей, кости и палки с надрезами, называемые счетными палочками, использовались в качестве вычислительных средств еще за 18 000 лет до н. э. Существует даже теория, что египетский глаз Гора применялся для представления двоичных дробей.

Английский ученый Чарльз Бэббидж (Charles Babbage) (1791–1871) убедил британское правительство профинансировать строительство сложного десятичного механического вычислителя, названного *разностной машиной*, которая первоначально была разработана гессенским военным инженером Иоганном Хельфрихом фон Мюллером (Johann Helfrich von Müller) (1746–1830). Снискавшая популярность благодаря роману Уильяма Гибсона (William Gibson) и Брюса Стерлинга (Bruce Sterling)¹, названному в ее честь, разностная машина опередила свое время, потому что технологии обработки металла в тот период не подходили для изготовления деталей с требуемой точностью.

Однако уже тогда выпускались простые десятичные механические калькуляторы, поскольку они не требовали такой же сложной металлообработки. Например, счетные машины, которые могли складывать десятичные числа, были созданы в середине XVII века для бухгалтерского учета и счетоводства. Было выпущено много различных моделей, а в более поздних версиях счетных машин ручные рычаги были заменены на электродвигатели, что упростило их работу. Фактически легендарный старомодный кассовый аппарат представлял собой счетную машину с денежным ящиком.

Все эти исторические примеры делятся на две категории, о чем мы поговорим дальше.

Разница между аналоговым и цифровым представлением

Существует важное различие между такими устройствами, как логарифмическая линейка (счетные палочки) и счетная машина. На рис. 2.2 показано сравнение одной из шкал логарифмической линейки с рис. 2.1 с пронумерованными пальцами.

И шкала логарифмической линейки, и количество пальцев измеряются от 1 до 10. Мы можем представить значения наподобие 1.1 на шкале, что довольно удобно, но не можем сделать это с помощью пальцев, не применив фантазию

¹ В России книга была издана под не совсем корректным названием «Машина различий». — Примеч. ред.

(или ловкость рук). Это потому, что шкала, как говорят математики, *непрерывна* — она может представлять действительные числа. А вот пальцы математики назвали бы *дискретными*, потому что с их помощью можно представить только целые числа. Между такими числами нет других значений. Мы можем только переходить от одного целого числа к другому, как с пальца на палец.

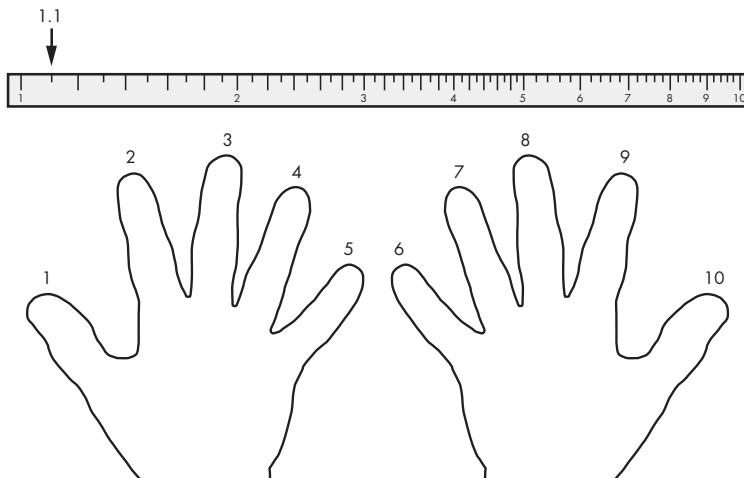


Рис. 2.2. Непрерывные и дискретные измерения

Говоря об электронике, мы используем слово «аналоговый» в значении «непрерывный» и «цифровой» в значении «дискретный» (легко запомнить, что пальцы цифровые (digital), потому что на латинский слово «палец» переводится как *digitus* (цифра)). Вы, наверное, уже знакомы с терминами «аналоговый» и «цифровой». Несомненно, вы учились программировать с помощью цифровых компьютеров, однако вы наверняка не знали, что существуют и аналоговые компьютеры, такие как логарифмические линейки.

Аналоговые вычисления кажутся более удобными, потому что в них можно оперировать действительными числами. Однако в этом случае появляются проблемы с точностью. Например, мы можем выбрать число 1.1 на шкале логарифмической линейки на рис. 2.2, потому что эта часть шкалы шире, чем остальные, и для 1.1 есть отдельная отметка. Но найти 9.1 намного сложнее, потому что эта часть шкалы гораздо более узкая, а число находится где-то между отметками 9.0 и 9.2. Разницу между 9.1 и 9.105 было бы трудно различить даже с помощью микроскопа.

Конечно, можно увеличить масштаб. Например, можно получить более точные данные, если использовать шкалу длиной с футбольное поле. Однако сложно создать портативный компьютер со 120-ярдовой шкалой, не говоря уже о том, сколько энергии потребуется для управления таким большим объектом. Нам

нужны небольшие, быстрые компьютеры с низким энергопотреблением. В следующем разделе я объясню еще одну причину, по которой размер важен.

Почему для аппаратного обеспечения размер имеет значение

Представьте, что вам нужно отвозить детей в школу, которая находится в 10 милях от вас, и забирать их оттуда со средней скоростью 40 миль в час. Расстояние и скорость таковы, что за час вы успеете проехать туда и обратно только два раза. Нельзя завершить поездку быстрее, если только не увеличить скорость или не выбрать другую школу поблизости.

Современные компьютеры вместо детей перевозят электроны. Электрическое поле в проводнике распространяется со скоростью света — около 300 миллионов метров в секунду (за исключением США, где скорость составляет около миллиарда футов в секунду). Поскольку мы еще не нашли способ обойти это физическое ограничение, единственный способ минимизировать время в пути в компьютерах — расположить его части близко друг к другу.

Современные компьютеры имеют тактовую частоту около 4 ГГц, что означает, что они могут выполнять четыре миллиарда операций в секунду. За четыре миллиардные доли секунды электрическое поле преодолеет всего 75 миллиметров.

На рис. 2.3 показан типичный центральный процессор (ЦП) размером около 18 миллиметров с каждой стороны. Чтобы совершить два полных обхода этого процессора, как раз достаточно четырех миллиардных долей секунды. Отсюда следует, что уменьшение размеров позволяет повысить производительность.

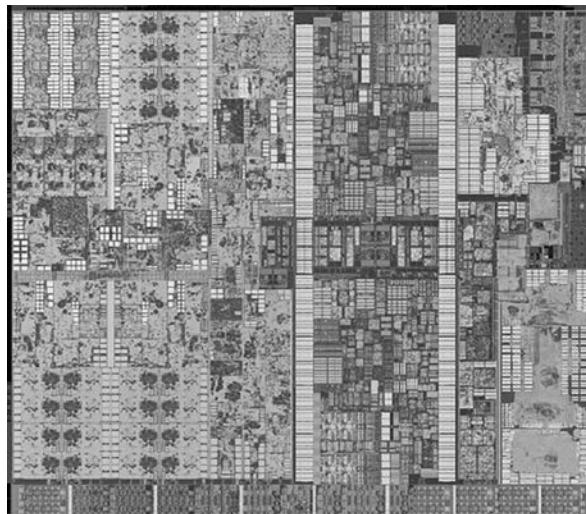


Рис. 2.3. Микрофотография ЦП (любезно предоставлена корпорацией Intel)

Кроме того, подобно перевозке детей в школу и обратно, передача данных требует энергии, и одного кофе для этого недостаточно. Уменьшение размера вещей сокращает количество обходов, что снижает объем необходимой энергии. В итоге мы получаем более низкое энергопотребление и меньшее тепловыделение — чтобы телефон не прожег дыру в кармане. Это одна из причин, почему создатели вычислительных устройств всегда стремились уменьшить размеры аппаратного обеспечения. Но если делать вещи очень маленькими, возникают другие проблемы.

Цифровые решения для более стабильных устройств

Несмотря на то что уменьшение размеров обеспечивает быстроту и эффективность, нарушить работу очень маленьких объектов довольно легко. Немецкий физик Вернер Гейзенберг (Werner Heisenberg) (1901–1976) был абсолютно уверен в этом.

Представьте стеклянный мерный стакан с делениями от 1 до 10 унций.

Если вы нальете немного воды в стакан и поднимете его, будет трудно сказать, сколько унций в стакане, из-за того что рука немного дрожит. А теперь представьте мерный стакан в миллиард раз меньше. Никто не сможет удерживать его достаточно неподвижно, чтобы получить точные показания. Фактически, даже если поставить этот крошечный стакан на стол, это все равно не поможет, потому что при таком размере движение атомов не позволит ему оставаться неподвижным. В крошечных масштабах Вселенная очень нестабильна.

И мерный стакан, и линейка являются аналоговыми (непрерывными) устройствами — с них очень легко считать неверные показания. Таких отклонений, как случайное космическое излучение, достаточно, чтобы создавать волны в микроскопических мерных стаканчиках, но они с меньшей вероятностью влияют на дискретные устройства: пальцы, счетные палочки или механические калькуляторы. Это потому, что дискретные устройства используют *критерии принятия решения*. При счете на пальцах не бывает «промежуточных» значений. Можно изменить логарифмическую линейку, включив в нее критерии принятия решения — добавив *фиксаторы* (что-то вроде механических закрепок) в целочисленных позициях. Но как только мы это сделаем, мы превратим ее в дискретное устройство, потерявшее способность представлять действительные числа. Фактически критерии принятия решения мешают представлению определенных диапазонов значений. Математически это похоже на округление чисел до ближайшего целого.

До сих пор мы говорили только о помехах извне, поэтому вы можете подумать, что можно минимизировать их, используя что-то вроде защитного экрана. В конце концов, свинец защитил Супермена от криптонита. Но существует и другой, более коварный источник помех. Электрический ток воздействует на

объекты на расстоянии, как гравитация, — и это хорошо, иначе у нас не было бы радио. Но это также означает, что сигнал, идущий по проводникам внутри микросхемы, может влиять на сигналы на других проводниках, особенно когда они расположены так близко друг к другу. Расстояние между проводниками внутри современного микропроцессора составляет несколько нанометров (10^{-9} метров). Для сравнения: диаметр человеческого волоса составляет около 100 000 нанометров. Эти помехи немного похожи на ветер, который вы чувствуете, когда проезжаете мимо встречного автомобиля. Поскольку не существует простого способа защиты от эффекта *перекрестных помех*, важно использовать цифровую схему, которая имеет более высокую помехозащищенность от критериев принятия решения. Можно, конечно, уменьшить влияние помех, увеличив размеры объектов, чтобы провода были дальше друг от друга, но это противоречит другим целям. Дополнительная энергия, необходимая для преодоления критерия принятия решения, дает степень защищенности от шума, которую не получить при использовании непрерывных устройств.

На самом деле стабильность, полученная благодаря критериям принятия решений, является основной причиной, по которой мы создаем цифровые (дискретные) компьютеры. Но, как вы, возможно, заметили, мир — это аналоговое (непрерывное) место, если не брать в расчет вещи, которые настолько малы, что к ним применима квантовая физика. В следующем разделе вы узнаете, как мы управляем аналоговым миром, чтобы добиться цифрового поведения, необходимого для создания стабильных вычислительных устройств.

Цифровые устройства в аналоговом мире

Многие инженерные разработки включают в себя умное применение естественных *передаточных функций*, обнаруженных учеными. Они подобны функциям, которые вы изучаете на уроках математики, за исключением того, что они представляют явления реального мира. Например, на рис. 2.4 показан график передаточной функции для датчика цифровой камеры (или пленки в аналоговой камере старого образца, если на то пошло).

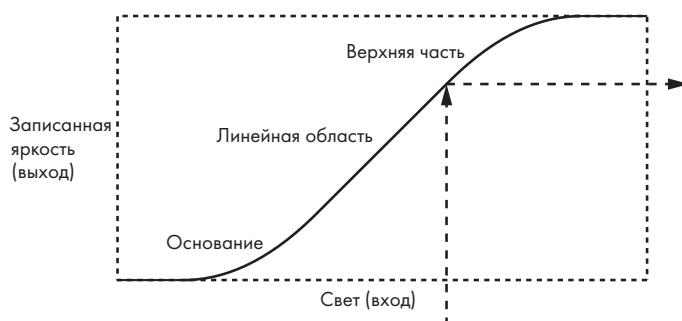


Рис. 2.4. Передаточная функция датчика камеры или пленки

Ось X показывает количество входящего света (вход), а ось Y представляет количество записанной яркости, или света, зарегистрированного датчиком (выход). Кривая показывает отношения между ними.

Подберем значения X и Y для функции, перемещая воображаемый входной шарик по кривой и получая разные выходы. Видим, что передаточная функция дает разные значения записанной яркости для различных значений света. Обратите внимание, что кривая — это не прямая линия. Если слишком много света попадает на *верхнюю часть* кривой, изображение будет передержано, поскольку записанные значения яркости будут ближе друг к другу, чем в реальности. Точно так же, если свет попадает на *основание* кривой, снимок будет недодержан. Обычно, если только не требуется какой-то особый эффект, цель состоит в регулировании экспозиции таким образом, чтобы она попала в *линейную область* — тогда снимок будет максимально приближен к реальности.

Инженеры придумали всевозможные уловки для использования передаточных функций — например, настройка выдержки и диафрагмы камеры так, чтобы свет попадал в линейную область. Еще один пример — схемы усилителя, например те, которые управляют динамиками или наушниками в плеере.

На рис. 2.5 показано влияние изменения громкости на передаточную функцию усилителя.

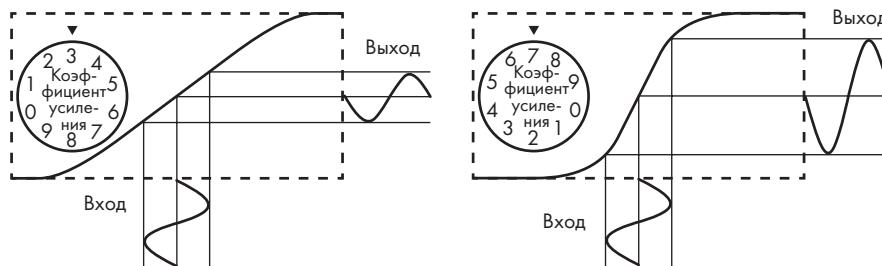


Рис. 2.5. Влияние коэффициента усиления на передаточную функцию усилителя

Регулятор громкости регулирует *коэффициент усиления*, или крутизну кривой. Как видите, чем больше коэффициент, тем круче кривая и тем громче выход. А что было бы, если бы у нас был один из тех специальных усилителей из фильма «Это — Spinal Tap!» 1984 года, где можно увеличить усиление до 11? Тогда сигнал не ограничивался бы линейной областью. Данный результат приводит к *искажению*, поскольку выходной сигнал уже не является точным воспроизведением входного, что ухудшает звучание. На рис. 2.6 видно, что выход не похож на вход, потому что вход нарушает границы линейной области передаточной функции.

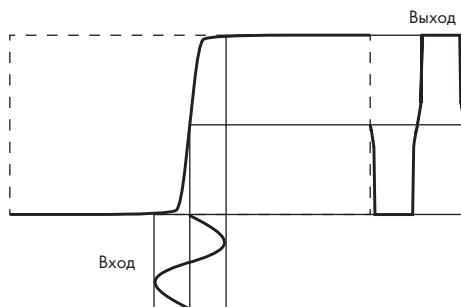


Рис. 2.6. Ограничение усилителя

Небольшое изменение входных данных вызывает скачок выходных данных на крутой части кривой. Это похоже на момент перехода с одного пальца на другой — на критерий принятия решения, называемый *порогом*. Это искажение является полезным явлением, потому что выходные значения находятся по обе стороны от порогового значения; трудно поймать значения, близкие к порогу. Непрерывное пространство при этом разделяется на дискретные области, что нам и нужно для стабильности и помехоустойчивости — способности работать при наличии помех. Аналоговое представление стремится к использованию большой линейной области, а цифровое — малой.

Возможно, вы естественным образом открыли этот феномен, качаясь на качелях в детстве (если вам посчастливилось вырасти до того, как детские игровые площадки стали считаться опасными). Гораздо стабильнее находиться у основания (в самом низу) качели или ее верхней части (в самом верху), чем пытаться балансировать где-то между ними.

Почему вместо цифр используются биты

Мы говорили о том, почему цифровые технологии лучше подходят для компьютеров, чем аналоговые. Но почему компьютеры используют биты вместо цифр? В конце концов, люди используют цифры, и нам удобно считать до десяти, потому что у нас 10 пальцев.

Очевидная причина заключается в том, что у компьютеров нет пальцев. (Компьютер с пальцами — жуткое зрелище.) С одной стороны, счет на пальцах интуитивно понятен, но это не очень эффективное использование пальцев, потому что для одной цифры используется один палец. С другой стороны, если применять каждый палец для представления значения, как в случае с битами, можно сосчитать более чем до тысячи. Эта идея не нова; фактически китайцы использовали 6-битные числа для работы с гексаграммами в «Книге Перемен» еще в IX в. до н. э. Применение битов вместо пальцев повышает эффективность счета более чем в 100 раз. Даже использование групп из четырех пальцев для представления десятичных чисел

с применением двоично-десятичной системы (BCD), которую мы рассматривали в главе 1, превосходит наш обычный метод подсчета по эффективности.

Еще одна причина, по которой в аппаратном обеспечении лучше использовать биты, чем цифры, заключается в том, что не существует простого способа настроить передаточную функцию с помощью цифр так, чтобы получить 10 различных пороговых значений. Можно создать аппаратное обеспечение, реализующее левую часть рис. 2.7, но оно будет намного сложнее и дороже, чем 10 копий устройств, создающих правую часть рисунка.

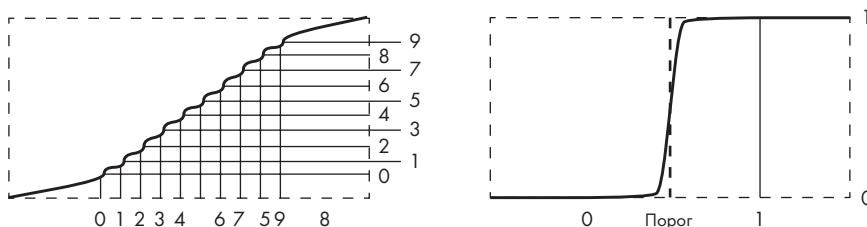


Рис. 2.7. Двоичные и десятичные пороги

Конечно, если бы мы *могли* построить 10 порогов в одном пространстве, мы бы это сделали. Но, как мы уже поняли, лучше задействовать 10 бит вместо одной цифры. Именно так и работает современное аппаратное обеспечение. Мы используем преимущества передаточной функции в основании и верхней части, которые на языке электротехники называются *отсечением* и *насыщением* соответственно. При этом у нас есть много места для маневра; получить неправильные выходы можно только при значительных помехах. Кривая передаточной функции настолько крата, что выходной сигнал переключается с одного значения на другое.

Знакомство с принципами работы электрического тока

Современные компьютеры работают благодаря электричеству. Оно позволяет собирать устройства быстрее и проще, чем при использовании других современных технологий. Из этого раздела вы узнаете об электричестве достаточно, чтобы понять, как оно используется в компьютерном аппаратном обеспечении.

Электрический ток на примере сантехники

Электричество невидимо, поэтому его трудно визуализировать, так что представим вместо него воду. Электричество поступает из источника энергии (например, батареи) точно так же, как вода поступает из резервуара. Батареи разряжаются и нуждаются в подзарядке — так же и резервуары для воды высыхают и нуждаются в повторном наполнении. Солнце — единственный главный источник

энергии, который у нас есть; в примере с водой тепло от Солнца вызывает испарение, а водяной пар затем превращается в дождь, наполняющий резервуар. Начнем с простого водяного клапана, изображенного на рис. 2.8.



Рис. 2.8. Водяной клапан

Как видите, у клапана есть ручка для открывания и закрывания. На рис. 2.9 показан настоящий запорный клапан, получивший свое название от задвижки, которая открывается и закрывается при помощи ручки. Вода может пройти по клапану, когда он открыт. Представим, что 0 означает «закрыто», а 1 — «открыто».



Рис. 2.9. Закрытый и открытый запорный клапан

Используем два клапана и несколько труб, чтобы проиллюстрировать операцию И, как показано на рис. 2.10.

Как видите, вода течет только тогда, когда оба клапана открыты или равны 1, что, как мы выяснили в главе 1, является определением операции И. Когда выход одного клапана соединяется со входом другого, как на рис. 2.10, это называется *последовательным соединением*, которое реализует операцию И. *Параллельное соединение*, показанное на рис. 2.11, получается в результате соединения входов и выходов клапанов, что реализует операцию ИЛИ.

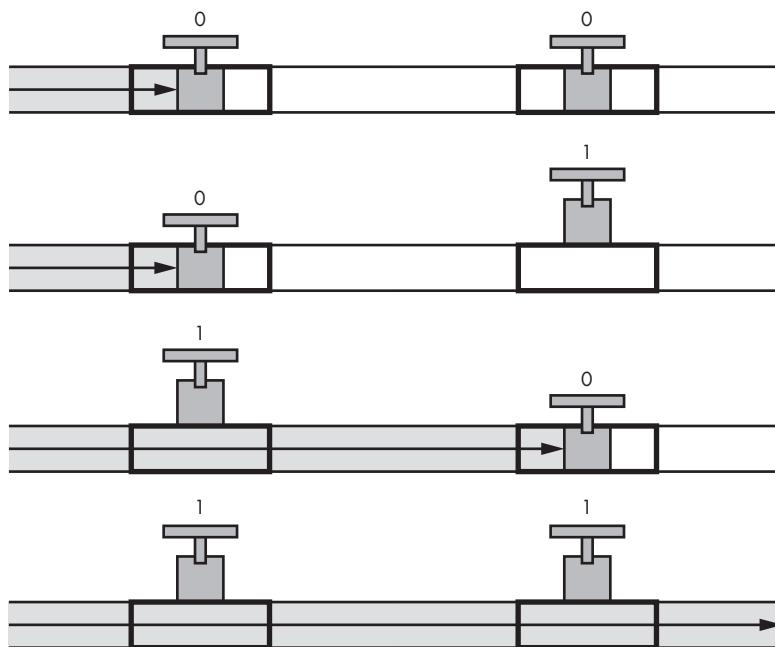


Рис. 2.10. Операция И на примере сантехники

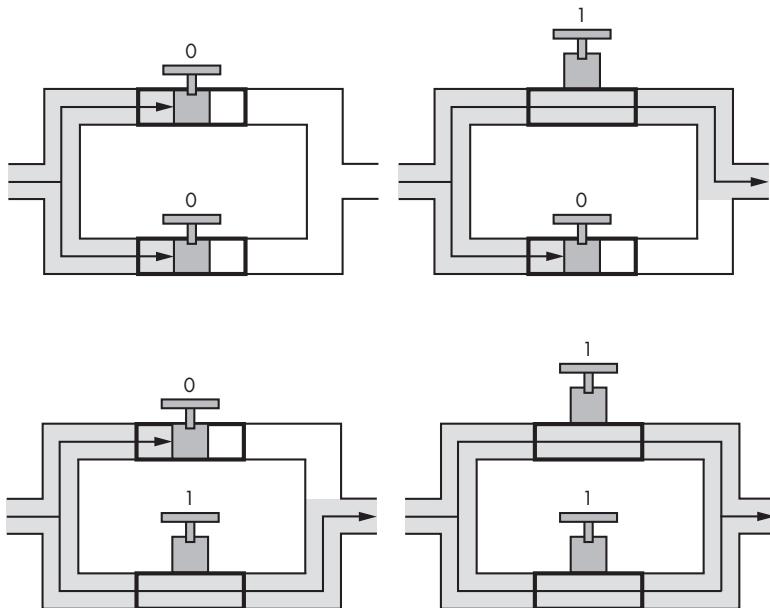


Рис. 2.11. Операция ИЛИ на примере сантехники

Электрическому току требуется время на прохождение через компьютерную микросхему — так и воде нужно время, чтобы распространиться по трубам. Вы, вероятно, знаете из собственного опыта, что температура воды в душе меняется не сразу после поворота ручки. Этот эффект называется *задержкой распространения*, и мы скоро поговорим о нем подробнее. Задержка непостоянна; в случае с водой изменение температуры заставляет трубы расширяться или сжиматься, что изменяет скорость потока и, следовательно, время задержки.

Ток проходит по проводу, как вода по трубе. Он представляет собой поток электронов. Кусок провода состоит из двух частей: металл внутри, как и пространство внутри трубы, является *проводником*, а покрытие снаружи, как и сама водопроводная труба, является *изолятором*. Поток можно включать и выключать с помощью клапанов. В мире электричества клапаны называют *переключателями*. Они настолько похожи между собой, что устаревшее устройство под названием «вакуумная трубка» также было известно как термоэлектронный клапан.

Вода не просто пассивно течет по водопроводным трубам; она приводится в движение *давлением*, которое может различаться по силе. Электрический эквивалент давления воды — это *напряжение (V)*, измеряемое в *вольтах (В)*, названных в честь итальянского физика Александра Вольта (1745–1827). Величина потока называется *силой тока (I)* и измеряется в *амперах*, названных в честь французского математика Андре Мари Ампера (1775–1836).

Вода может течь по широким или по узким трубам, но чем уже труба, тем больше сопротивление ограничивает количество воды, которая может пройти по трубе. Даже при большом напряжении (давлении воды) вы не сможете получить большой ток (поток) из-за сопротивления при слишком узком проводнике (трубе). *Сопротивление (R)* измеряется в *Омах (Ω)*, названных в честь немецкого математика и физика Георга Симона Ома (1789–1854).

Эти три переменные — напряжение, сила тока и сопротивление — связаны *законом Ома*, который гласит, что $I = U/R$, что читается как «сила тока равна напряжению, деленному на сопротивление». Таким образом, как и в случае с водопроводными трубами, большее сопротивление означает меньшую силу тока. Сопротивление также превращает электричество в тепло — по этому принципу работает все, от тостеров до электрических одеял. На рис. 2.12 показано, как сопротивление затрудняет проталкивание тока напряжением.

Легкий способ понять закон Ома — выпить молочный коктейль через трубочку.

Электрические переключатели

Чтобы сделать переключатель (клапан) для тока, достаточно вставить или удалить изолятор между проводниками. Представьте переключаемые вручную выключатели освещения. Они состоят из двух металлических частей, которые либо соприкасаются, либо разделяются при помощи рукоятки переключателя.

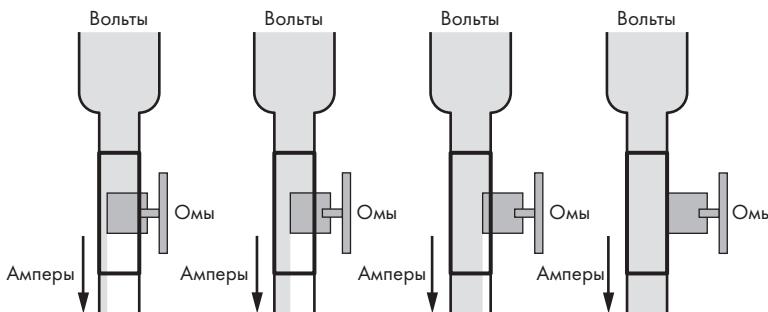


Рис. 2.12. Закон Ома

Оказывается, воздух — неплохой изолятор; ток не может течь между двумя металлическими предметами, если они не соприкасаются. (Обратите внимание: я сказал, что воздух — «довольно хороший» изолятор; при достаточно высоком напряжении воздух ионизируется и превращается в проводник. Молния — хороший пример.)

Водопроводную систему в здании можно показать на чертеже. Электрические системы, называемые *цепями*, изображаются с помощью *принципиальных схем*, в которых используются условные обозначения для каждого из компонентов. На рис. 2.13 показано условное обозначение для простого переключателя.

Такой переключатель похож на подъемный мост: ток (автомобили) не может перейти с одной стороны моста на другую, когда стрелка на схеме (мосте) направлена вверх. Это легко проиллюстрировать с помощью старомодных *рубильников*, показанных на рис. 2.14, — их часто показывают в глупых научно-фантастических фильмах. Рубильники по-прежнему используются для таких вещей, как электрические разъединители, но в наши дни их обычно прячут в защитных контейнерах, чтобы поджарить себя было не так-то просто.

На рис. 2.13 и 2.14 показаны *однополюсные переключатели на одно направление* (single-pole, single-throw, SPST). *Полюса* — это количество соединенных вместе переключателей, которые перемещаются вместе. Водяные клапаны в предыдущем разделе были однополюсными; можно сделать *двухполюсный* клапан, приварив стержень между ручками пары клапанов, чтобы они оба двигались вместе при перемещении стержня. Переключатели и клапаны могут иметь любое количество полюсов. Термин «*в одном направлении*» означает, что существует только одна точка соприкосновения: что-то может быть либо включено, либо выключено, но не то и другое одновременно. Для этого нам понадобится *однополюсное устройство на два направления* (single-pole, double-throw, SPDT). На рис. 2.15 показано условное обозначение для этого монстра.



Рис. 2.13. Условное обозначение однополюсного переключателя на одно направление



Рис. 2.14. Однополюсный рубильник на одно направление



Рис. 2.15. Условное обозначение SPDT-переключателя

Он похож на железнодорожную стрелку, которая направляет поезд на тот или иной путь, или трубу, разделенную на две отдельные трубы, как показано на рис. 2.16.

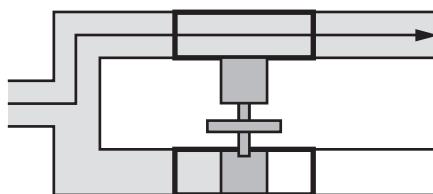


Рис. 2.16. Водяной SPDT-клапан

Когда ручка опущена, вода течет через верхний клапан. Вода будет течь через нижний клапан, если поднять ручку.

Терминологию переключателя можно расширить, чтобы описать любое количество полюсов и направлений. Например, *двуухполюсный переключатель на два направления* (double-pole, double-throw, DPDT) должен быть нарисован, как показано на рис. 2.17, с пунктирной линией, указывающей, что полюса объединены, то есть перемещаются вместе.

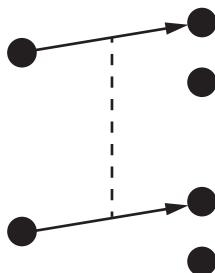


Рис. 2.17. Условное обозначение DPDT-переключателя

На рис. 2.18 показан реальный вид DPDT-рубильника.

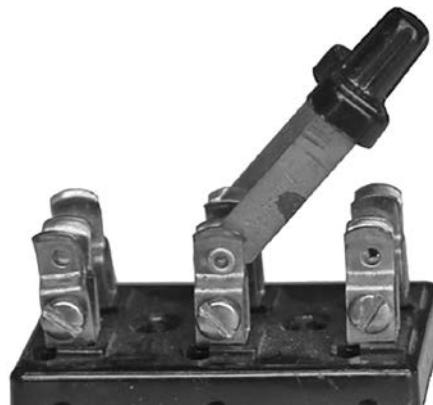


Рис. 2.18. DPDT-рубильник

Ранее я опустил некоторые подробности, касающиеся наших гидротехнических сооружений: система не будет работать, если воде будет некуда течь. Вода не может течь, если слив забит. Поэтому должен быть способ вернуть воду из слива обратно в резервуар для воды, иначе вся система окажется бесполезной.

То же верно и для электрических систем. Ток от источника энергии проходит через элементы схемы и возвращается к источнику. Вот почему это называется *электрической цепью*. Представьте это так: бегун на трассе должен вернуться на стартовую линию, чтобы пробежать еще один круг.

Посмотрите на простую электрическую схему на рис. 2.19. В ней появляются два новых условных обозначения: один для источника напряжения (слева) и один для лампочки (справа). Построив такую схему, можно включать и выключать свет с помощью переключателя.

Ток не будет течь, если переключатель разомкнут. Когда переключатель замкнут, ток течет от источника напряжения через переключатель, лампочку и обратно

к источнику напряжения. Последовательные и параллельные переключатели работают так же, как и их аналоги с водяными клапанами.

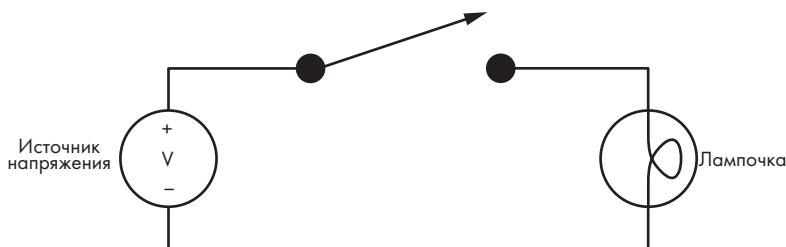


Рис. 2.19. Простая электрическая схема

Теперь вы узнали немного об электричестве и некоторых основных схемах. Хотя их можно использовать для реализации некоторых простых логических функций, сами по себе они недостаточно мощны, чтобы делать что-то еще. В следующем разделе вы узнаете о дополнительном устройстве, благодаря которому были созданы первые компьютеры с электрическим питанием.

Создание аппаратного обеспечения, работающего с битами

Теперь, когда мы узнали, почему в компьютерной технике используются именно биты, пришла пора разобраться с тем, каким образом они создаются. Погрузиться сразу в современные электронные технологии реализации может быть непросто, поэтому я обращусь к другим, более понятным историческим технологиям. Некоторые из них не используются в современных компьютерах, но встречаются в системах, работающих с компьютерами, поэтому о них стоит знать.

Реле

Электричество использовалось для питания компьютеров задолго до изобретения электроники. Электричество и магнетизм тесно взаимосвязаны, что обнаружил датский физик Ганс Кристиан Эрстед (1777–1851) в 1820 году. Если свернуть моток провода в катушку и пропустить через нее ток, то катушка станет **электромагнитом**. Электромагниты можно включать и выключать, а также использовать для перемещения вещей. Они подойдут и для управления водяными клапанами — так работает большинство автоматических систем пожаротушения. Кроме того, электромагнетизм позволяет нам создавать различные электродвигатели. Если вокруг катушки с проволокой вращать магнит, будет вырабатываться электричество — таков принцип работы генераторов;

фактически именно так мы получаем большую часть электроэнергии. На всякий случай, если вам захочется это проверить, получить электричество от электромагнита можно, если очень быстро вращать магнит вокруг катушки. Подобный опыт может шокировать, но этот эффект, называемый *обратной ЭДС* (электродвижущей силой), очень полезен; так катушка зажигания автомобиля производит искру для свечей зажигания. Таким же образом работают электрические заборы.

Реле – это устройство, которое использует электромагнит для переключения контактов. На рис. 2.20 показано условное обозначение реле с одной группой контактов на переключение, которое, как видим, очень похоже на обозначение переключателя, прикрепленного к катушке.

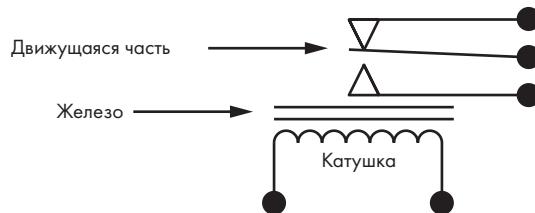


Рис. 2.20. Условное обозначение SPDT-реле

На рис. 2.21 показан реальный вид реле с одной группой контактов на замыкание. Переключатель открыт, когда на катушку не подается питание, поэтому это реле называется *нормально разомкнутым*. Если бы переключатель был замкнут без питания, это было бы *нормально замкнутое* реле.

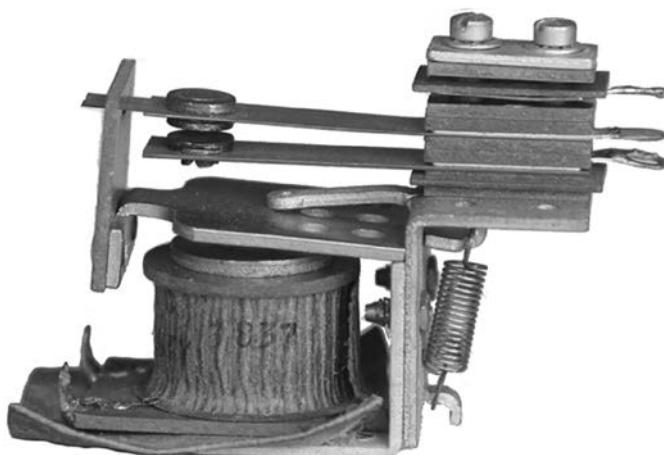


Рис. 2.21. Нормально разомкнутое SPST-реле

Контакты внизу идут к катушке реле; остальное очень похоже на вариацию переключателя. Контакт посередине перемещается в зависимости от того, находится ли катушка под напряжением. Мы можем реализовать логические функции с помощью реле, как показано на рис. 2.22.

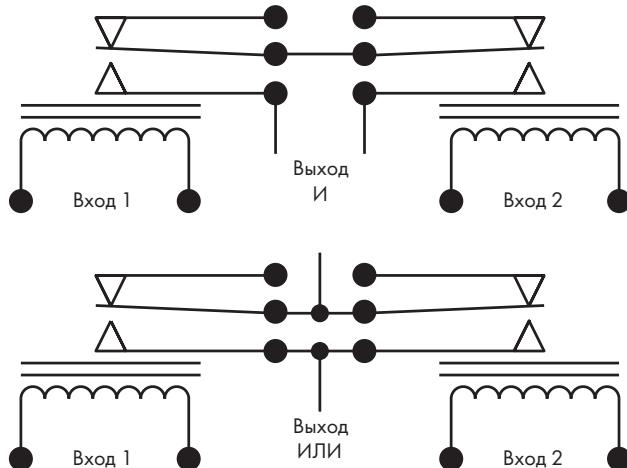


Рис. 2.22. Схемы соединений реле для логических операций И и ИЛИ

Видно, что в верхней части рис. 2.22 два выходных провода соединяются только в том случае, если *оба* реле активированы, что определяет логическую операцию И. Точно так же внизу провода соединяются, если срабатывает *какое-либо* реле, что представляет логическую операцию ИЛИ. Обратите внимание на маленькие черные точки на рисунке. Они указывают на соединения проводов на схемах; провода, пересекающиеся без точки, не соединены.

Реле позволяют делать то, что невозможно сделать с переключателями. Например, можно построить *инверторы*, реализующие логическое НЕ, без которого возможности булевой алгебры очень ограничены. Можно использовать выход схемы И наверху, чтобы управлять одним из входов на схеме ИЛИ внизу. Именно эта способность заставляет переключатели управлять другими переключателями позволяет создавать сложную логику компьютеров.

С помощью реле создано множество удивительных вещей. Например, существует шаговое реле с одной группой контактов на 10 шагов с двумя катушками. Одна катушка перемещает контакт в следующее положение при каждом включении, а другая сбрасывает реле, возвращая контакт в исходную позицию. Огромные здания, заполненные шаговыми реле, раньше использовались для подсчета цифр телефонных номеров при наборе. На телефонных станциях всегда было очень шумно. Шаговые реле также придают очарование старым автоматам для игры в пинбол.

Еще один интересный факт о реле — порог передаточной функции вертикален; независимо от того, как медленно увеличивается напряжение на катушке, переключатель всегда переходит из одного положения в другое. Это озадачило меня в детстве; только изучая уравнения Лагранжа — Гамильтона на первом курсе колледжа, я узнал, что значение передаточной функции не определено на пороге, что вызывает переход.

Минусы реле заключаются в том, что они работают медленно, потребляют много электроэнергии и перестают работать, если грязь (или жучки) попадает на контакты переключателя. Фактически термин *bug* (ошибка или «жучок») был популяризирован американским ученым-компьютерщиком Грейс Хоппер (Grace Hopper) в 1947 году, когда оказалось, что ошибка в компьютере Harvard Mark II возникла из-за мотылька, застрявшего в контактах реле. Еще одна интересная проблема возникает во время использования контактов реле для управления другими реле. Вспомните, что внезапное отключение питания катушки на мгновение генерирует очень высокое напряжение и что воздух становится проводящим при высоких напряжениях. Это явление часто ведет к появлению искры между контактами реле, что приводит к их износу. Из-за этих недостатков люди начали искать что-то, что выполняло бы ту же работу, что и реле, но не имело бы движущихся частей.

Вакуумные лампы

Британский физик и инженер-электрик сэр Джон Амброуз Флеминг (1849–1945) изобрел вакуумную лампу. Она была основана на принципе *термоэлектронной эмиссии*, который гласит, что, если нагреть предмет до достаточной температуры, это приведет к излучению электронов. У вакуумных ламп есть *нагреватель*, который нагревает *катод*, действующий подобно питчеру в бейсболе. В вакууме электроны (бейсбольные мячи) текут от катода к *аноду* (кетчеру). Примеры ламп показаны на рис. 2.23.



Рис. 2.23. Вакуумные лампы

Электроны обладают некоторыми общими с магнитами свойствами, в том числе тем, что противоположные заряды притягиваются, а одинаковые — отталкиваются. Вакуумная лампа может содержать дополнительный элемент-дозатор, называемый *сеткой*, который отталкивает электроны, идущие от катода, чтобы предотвратить их попадание на анод. Вакуумная лампа, содержащая три элемента (катод, сетку и анод), называется *триодом*. На рис. 2.24 показано схематическое обозначение триода.

Здесь нагреватель нагревает катод, что приводит к излучению электронов. Если сетка не отбрасывает их назад, они приземляются на анод. Можно мысленно соотнести сетку с ручкой переключателя.

Преимущество вакуумных ламп в том, что в них нет движущихся частей и поэтому они работают намного быстрее реле. Но у них есть и недостатки — лампы сильно нагреваются и такие же хрупкие, как и обычные лампы накаливания. Нагреватели перегорают, как нити в лампах накаливания. Но вакуумные лампы все равно были лучше реле и позволяли создавать более быстрые и надежные компьютеры.

Транзисторы

В наши дни правят транзисторы. *Транзистор* (от transfer resistor — перенос сопротивления) похож на вакуумную лампу, но использует особый тип материала, называемый *полупроводником*, который может быть как проводником, так и изолятором. Именно это свойство позволило создавать клапаны для электричества, не требующие наличия нагревателя и не имеющие движущихся частей. Но, конечно, транзисторы несовершенны. Их можно сделать очень маленькими, и это хорошо, но тонкие проводники имеют большее сопротивление, из-за чего выделяется тепло. Избавиться от тепла в транзисторе — настоящая проблема, потому что полупроводники легко плавятся.

Все об устройстве транзисторов знать не обязательно. Важно знать, что транзистор сделан на *подложке* или пластине из полупроводникового материала, обычно кремния. В отличие от других технологий, таких как шестерни, клапаны, реле и вакуумные лампы, транзисторы не производятся индивидуально. Они создаются с помощью процесса под названием «*фотолитография*» — проецирования изображения транзистора на кремниевую пластину и затем его проявления. Этот процесс подходит для массового производства, потому что большое количество транзисторов можно спроектировать на одну кремниевую подложку, проявить, а затем разделить на отдельные компоненты.

Существует много различных типов транзисторов, из них выделяют два основных — *биполярный* (bipolar junction transistor, BJT) и *полевой* (field effect

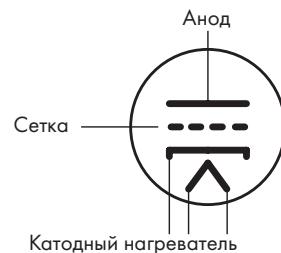


Рис. 2.24. Условное обозначение триода

transistor, FET). Процесс производства включает в себя *добавление примеси*, при котором материал подложки пропитывается неприятными химическими веществами, такими как мышьяк, для изменения его характеристик. Примеси создают области материала *p*- и *n*-типа. Конструкция транзистора предполагает изготовление бутербродов из *p*- и *n*-типов. На рис. 2.25 показаны условные обозначения для некоторых типов транзисторов.

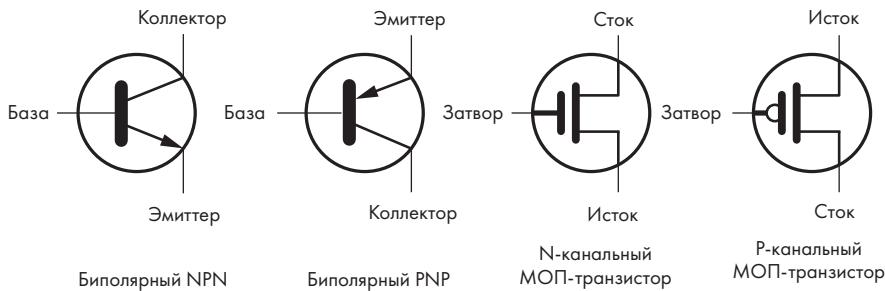


Рис. 2.25. Условные обозначения различных типов транзисторов

Термины *NPN*, *PNP*, *N-канальный* и *P-канальный* относятся к многослойным конструкциям. Можно представить транзистор как клапан или переключатель, а *затвор* (или *базу*) — как ручку. Электрический ток течет сверху вниз, когда ручка поднимается, подобно тому как катушка в реле перемещает контакты. Но в отличие от переключателей и клапанов, которые мы рассматривали до сих пор, в биполярных транзисторах ток может течь только в одном направлении.

Видно, что между затвором и остальной частью транзистора в обозначениях полевых транзисторов есть зазор. Он символизирует то, что полевые транзисторы работают при помощи электрического поля (отсюда и название — полевой); это все равно, что использовать статическое прилипание (склонность легких предметов прилипать к другим предметам из-за статического электричества) для перемещения переключателя.

Полевой транзистор со структурой «металл-оксид-полупроводник» (полевой МОП-транзистор), или *MOSFET* (metal-oxide semiconductor field effect transistor), представляет собой разновидность полевого транзистора, очень часто используемую в современных компьютерных микросхемах благодаря низкому энергопотреблению. Варианты с *N*- и *P*-каналами часто используются в дополнительных парах, отсюда и происходит термин *CMOS* (complementary metal oxide semiconductor) — комплементарная структура металл-оксид-полупроводник (КМОП).

Интегральные схемы

Транзисторы позволили делать более компактные, быстрые и надежные логические схемы, потребляющие меньше энергии. Но создание даже простой схемы,

вроде схемы для реализации логической операции И, по-прежнему требовало множества компонентов.

Ситуация изменилась в 1958 году, когда Джек Килби (Jack Kilby) (1923–2005), американский электротехник, и Роберт Нойс (Robert Noyce) (1927–1990), американский математик, физик и соучредитель Fairchild Semiconductor и Intel, изобрели *интегральную схему*. Благодаря интегральным схемам создание сложных систем становится не дороже сборки одного транзистора. Интегральные схемы стали называть *микросхемами* из-за их внешнего вида.

Как вы, наверное, заметили, многие из схем одного типа можно построить с использованием реле, вакуумных ламп, транзисторов или интегральных схем. И с каждой новой технологией эти схемы становились меньше, дешевле и энергоэффективнее. В следующем разделе рассказывается об интегральных схемах, разработанных для комбинаторной логики.

Логические вентили

В середине 1960-х годов работодатель Джека Килби, компания Texas Instruments, представила семейства интегральных схем 5400 и 7400. Эти микросхемы содержали готовые схемы, выполняющие логические операции. Эти конкретные схемы, называемые *логическими вентилями*, или просто *вентилями*, являются аппаратными реализациями логических операций, которые мы называем комбинаторной логикой. Компания Texas Instruments продала миллиарды таких схем. Они доступны и сегодня.

Логические вентили очень пригодились разработчикам аппаратного обеспечения: им больше не приходилось делать все с нуля, и они могли создавать сложные логические схемы так же просто, как собирать многосоставные сантехнические системы. Аналогично тому, как сантехники находят корзины с тройниками, коленами и соединительными муфтами в хозяйственном магазине, разработчики логики находят «корзины» с вентилями И, ИЛИ, исключающего ИЛИ и инверторов (штук, выполняющих операцию НЕ). На рис. 2.26 показаны обозначения этих вентилей.

Как и следовало ожидать, выход Y логического элемента И истинен, если оба параметра на входах A и B также истинны. (Информация о работе других вентилей представлена в таблицах истинности на рис. 1.1.)

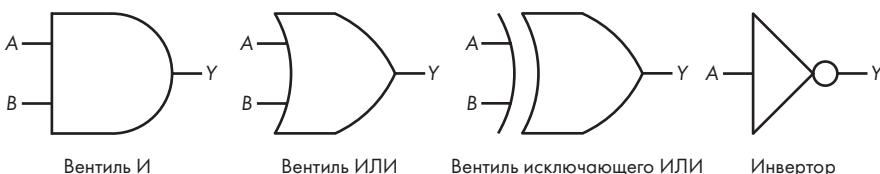


Рис. 2.26. Обозначения вентилей

Ключевой частью условного обозначения инвертора на рис. 2.26 является \circ (круг), а не треугольник, к которому он прикреплен. Треугольник без круга называется *буфером*, и он просто передает свой вход на выход. Обозначение инвертора в основном применяется только там, где инвертор не используется в сочетании с чем-либо еще.

Создание логических элементов И и ИЛИ с использованием технологии *транзисторно-транзисторной логики* (transistor-transistor logic, TTL) компонентов серий 5400 и 7400 неэффективно, потому что выходной сигнал простой схемы вентиля по умолчанию уже инвертирован, и для его правильного вывода требуется лишний инвертор. Это сделало бы их более дорогими, медленными и энергоемкими. Итак, в качестве основных вентиляй использовались схемы *И-НЕ* и *ИЛИ-НЕ* с инвертирующим кругом, как на рис. 2.27.

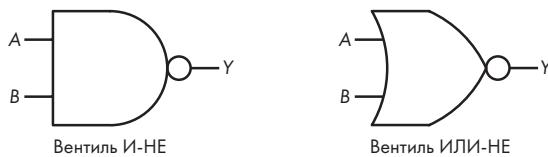


Рис. 2.27. Вентили И-НЕ и ИЛИ-НЕ

К счастью, эта дополнительная инверсия не влияет на нашу способность разрабатывать логические схемы благодаря закону де Моргана. Он применяется на рис. 2.28, чтобы показать, что логический элемент И-НЕ эквивалентен логическому элементу ИЛИ с инвертированными входами.



Рис. 2.28. Перерисовка логического элемента И-НЕ с использованием закона де Моргана

Все рассмотренные ранее вентили имели два входа, не считая инвертора, но на самом деле у вентиляй может быть более двух входов. Например, вентиль И с тремя входами будет иметь на выходе истину, если все три входа истинны. Теперь, когда вы знаете, как работают вентили, рассмотрим некоторые сложности, которые возникают при их использовании.

Повышение помехоустойчивости с помощью гистерезиса

Как вы уже видели, лучшую помехозащищенность можно получить с использованием цифровых (дискретных) устройств благодаря критериям принятия решения. Но бывают ситуации, когда этого недостаточно. Легко предположить,

что логические сигналы мгновенно переходят из 0 в 1 и наоборот. В большинстве случаев это верно, особенно когда мы соединяем вентили друг с другом. Но в реальном мире сигналы меняются гораздо медленнее.

Посмотрим, что произойдет при медленно меняющемся сигнале. На рис. 2.29 показаны два сигнала, которые постепенно переходят из 0 в 1.

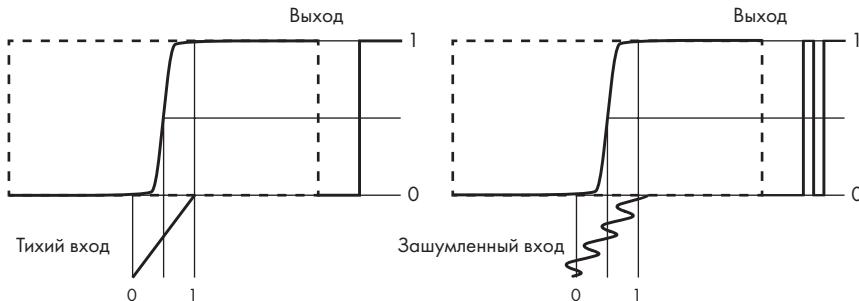


Рис. 2.29. Сбой из-за шума

Вход слева тихий и не имеет шума, а к сигналу справа добавлен некоторый шум. Видно, что зашумленный сигнал вызывает *сбой* на выходе, потому что шум заставляет сигнал пересекать пороговое значение более одного раза.

Этого можно избежать, используя *гистерезис*, в котором критерий принятия решения зависит от истории. Как показано на рис. 2.30, передаточная функция не симметрична. Существуют разные передаточные функции для сигналов нарастания (из 0 в 1) и спада (из 1 в 0), как показано стрелками. Когда выход равен 0, применяется кривая справа, и наоборот.

В результате мы получаем два разных порога: один для нарастающих сигналов и один для ниспадающих. Это означает, что, когда сигнал пересекает одно из пороговых значений, ему нужно пройти намного дальше, прежде чем он пересечет другое, благодаря чему получается более высокая помехоустойчивость.

Существуют специальные вентили с гистерезисом. Их называют *триггерами Шмитта* в честь американского ученого Otto X. Шмитта (Otto H. Schmitt) (1913–1998), который изобрел данную схему. Поскольку они сложнее и дороже обычных вентилей, их используют только там, где они действительно нужны. Гистерезис добавляется к условному обозначению, как показано для инвертора на рис. 2.31.

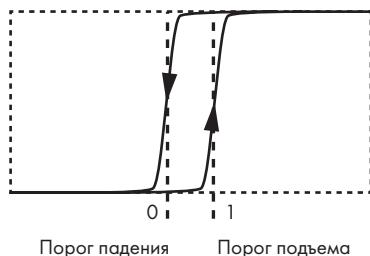


Рис. 2.30. Передаточная функция гистерезиса

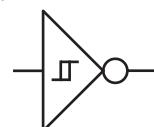


Рис. 2.31. Условное обозначение триггера Шмитта

Дифференциальная передача сигналов

Иногда шума бывает так много, что даже гистерезиса недостаточно. Представьте прогулку по тротуару. Назовем правый край тротуара *положительным порогом*, а левый — *отрицательным*. Вы думаете о своем, когда кто-то, толкая коляску для двойни, сбивает вас с правого края тротуара, а затем группа бегунов оттесняет вас слева. В этом случае нам понадобится защита.

До сих пор мы измеряли сигнал относительно абсолютного порога или пары порогов в случае триггера Шмитта. Но бывают ситуации, когда шум настолько велик, что оба порога триггера Шмитта пересекаются, что делает его неэффективным.

Добавим в систему друзей. Представьте, что вы идете по тротуару с другом. Если ваш друг находится слева от вас, назовем его 0; если справа — 1. Теперь коляска и бегуны оттолкнут в сторону не только вас, но и вашего друга. Но вы не сменили позицию, так что шум в итоге не повлиял на вас. Конечно, если вы просто гуляете рядом, можно столкнуться с тротуара только одного из вас. Вот почему лучше держаться за руки или обнимать друг друга за талию. Да, больше объятий — больше помехозащищенности! Это называется *дифференциальной передачей сигналов*, потому что мы измеряем разницу между парой *комплементарных* сигналов. На рис. 2.32 показана схема дифференциальной передачи сигналов.

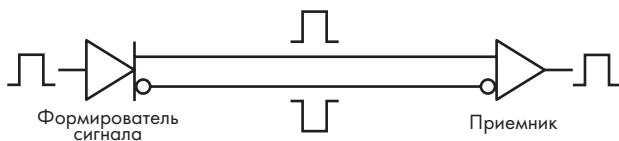


Рис. 2.32. Дифференциальная передача сигналов

Видим, что существует *формирователь*, который преобразует входной сигнал в комплементарную пару, и *приемник*, который преобразует комплементарную пару обратно в *несимметричный* выход. Обычно приемник включает триггер Шмитта для дополнительной помехоустойчивости.

Конечно, существуют ограничения. Слишком сильный шум может вытолкнуть электронные компоненты за пределы их указанного рабочего диапазона — представьте, что рядом с тротуаром есть здание и вас с другом вдавливают в стену. *Коэффициент подавления синфазного сигнала* (common-mode rejection ratio, CMRR) является частью спецификации компонента и указывает количество шума, с которым компонент может справиться. Сигнал называется «*синфазным*», потому что это название относится именно к шуму, который является общим для обоих сигналов в паре.

Дифференциальная передача сигналов используется во многих местах, например на телефонных линиях. Такое применение стало необходимым в 1880-х годах, когда появились электрические трамваи, поскольку они генерировали много электрических шумов, мешающих телефонным сигналам. Шотландский изобретатель Александр Грэйам Белл (1847–1922) изобрел *кабельную витую пару*, в которой пары проводов были скручены вместе для электрического эквивалента объятий (рис. 2.33).

Он также запатентовал телефон. Сегодня витая пара распространена повсеместно; вы найдете ее в кабелях USB, SATA (для подключения жестких дисков) и Ethernet.

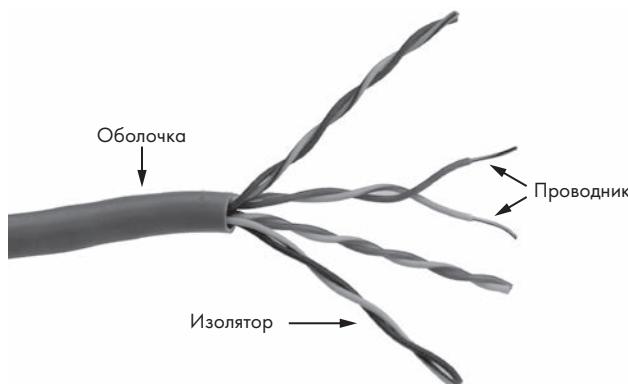


Рис. 2.33. Кабель Ethernet на основе витой пары

Интересное применение дифференциальной передачи сигналов можно найти в концертной аудиосистеме Wall of Sound, используемой американской группой The Grateful Dead (1965–1995). Таким образом была решена проблема обратной связи в микрофоне с помощью попарно соединенных микрофонов — выходной сигнал одного микрофона вычитался из выхода другого. Любой звук, попадавший в оба микрофона, был синфазным и подавлялся. Вокалисты пели в один из микрофонов в паре, чтобы звучал только их голос. Искажение этой системы, которое можно заметить, прослушав живые записи группы, состоит в том, что шум публики получается каким-то металлическим. Это потому, что низкочастотные звуки имеют большую длину волн, чем высокочастотные; низкочастотный шум с большей вероятностью будет синфазным, чем высокочастотный.

Задержка распространения

Я упомянул задержку распространения в разделе «Электрический ток на примере сантехники» на с. 83. *Задержка распространения* — это время, необходимое

для того, чтобы изменение входа отразилось на выходе. Это статистическая мера, которая нужна в связи с различиями в производственных процессах и температуре, а также количествах и типах компонентов, подключенных к выходу вентиля. Вентили имеют как минимальную, так и максимальную задержку; фактическая задержка находится где-то между этими значениями. Задержка распространения — один из факторов, ограничивающих максимальную скорость, которая может быть достигнута в логических схемах. Разработчики должны использовать *наихудшие* значения, чтобы их схемы работали. Это означает, что схемы должны проектироваться с учетом как самых коротких, так и самых длительных задержек.

На рис. 2.34 серые области показывают места, где нельзя полагаться на выходные данные из-за задержки распространения.

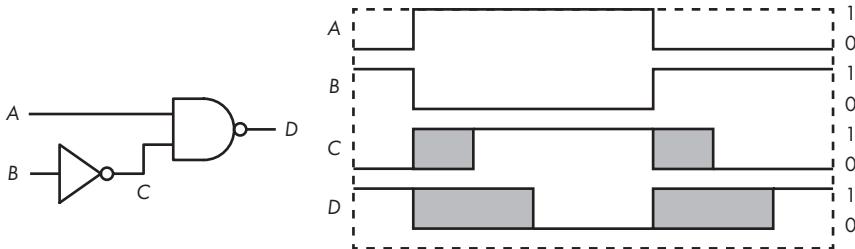


Рис. 2.34. Пример задержки распространения

Выходные данные могут измениться уже у левого края серых областей, но нет гарантий, что они изменятся до правого края. Длина серых областей будет увеличиваться по мере объединения всех большего количества вентилей.

Существует множество диапазонов времени задержки распространения, которые зависят от технологического процесса. Отдельные компоненты, такие как микросхемы серии 7400, могут иметь задержки в диапазоне 10 наносекунд (то есть 10 миллиардных долей секунды). Задержки логического вентиля внутри современных крупных компонентов, таких как микропроцессоры, могут составлять пикосекунды (триллионные доли секунды). Если вы читаете спецификации компонентов, задержки распространения обычно указываются как t_{PLH} и t_{PHL} для времени распространения от низкого к высокому (low to high) и от высокого к низкому уровню (high to low) соответственно.

Обсудив входы и то, что происходит на путях к выходам, перейдем непосредственно к выходам.

Варианты выходов

Мы уже говорили о входах вентиля, но практически не упоминали выходы. Существует несколько разных типов выходов, использующихся в разных ситуациях.

Каскадный выход

Обычный выход вентиля называется *каскадным*, потому что транзисторы, установленные один поверх другого, напоминают каскад. Мы можем смоделировать этот тип выхода с помощью переключателей, как показано на рис. 2.35.

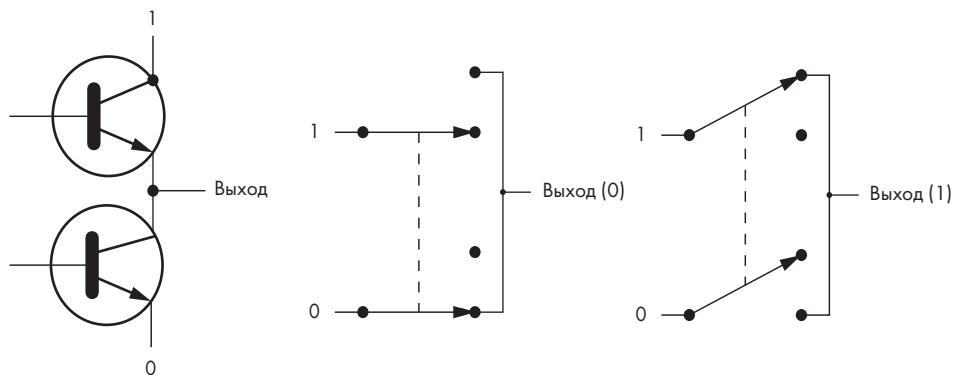


Рис. 2.35. Каскадный выход

Схема слева показывает, как каскадные выходы получили свое название. Верхний переключатель на рисунке называется *активным подтягивающим*, потому что он подключает выход к высокому логическому уровню, чтобы в результате получить 1. Каскадные выходы нельзя соединять вместе. Как видно на рис. 2.35, если соединить выход 0 с выходом 1, будут объединены положительный и отрицательный источники питания, что так же плохо, как скрещивание потоков в фильме «Охотники за привидениями» 1984 года, — это приведет к расплавлению компонентов.

Выход с открытым коллектором

Другой тип выхода называется выходом *с открытым коллектором* или *открытым стоком* в зависимости от типа используемого транзистора. Схема и модель переключателя для этого выхода показаны на рис. 2.36.

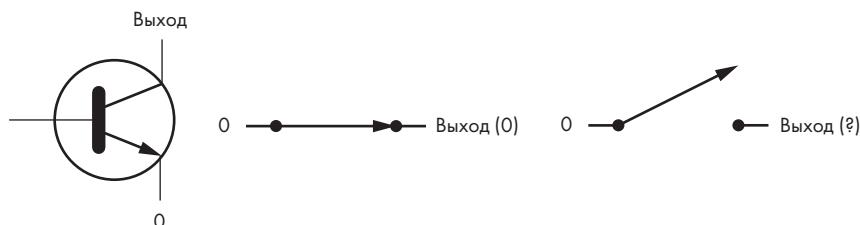


Рис. 2.36. Выход с открытым коллектором/стоком

Такой подход кажется странным на первый взгляд. Все хорошо, если мы хотим получить результат 0, но в другом случае выход просто *плавает*, поэтому мы не знаем, каково его значение.

Поскольку версии с открытым коллектором и открытым стоком не имеют активных подтягиваний, можно без вреда соединить их выходы. Можно использовать *пассивный подтягивающий резистор*, который представляет собой просто *подтягивающий резистор*, соединяющий выход с питающим напряжением, которое является источником единиц. Это называется V_{CC} для биполярных транзисторов и V_{DD} для МОП-транзисторов. Пассивное подтягивание создает эффект *монтажного И*, как показано на рис. 2.37.

Здесь происходит следующее: когда ни на одном из выходов с открытым коллектором не установлен низкий логический уровень, резистор устанавливает выход в 1. Резистор ограничивает силу тока, чтобы предотвратить перегрев компонентов схемы. Выход устанавливается в 0, когда на любом из выходов с открытым коллектором присутствует логический 0. Таким образом можно соединить много элементов, не используя логический вентиль И с большим количеством входов.

Другое использование выходов с открытым коллектором и открытым стоком — управление устройствами, такими как светодиоды (light-emitting diode, LED). Устройства с открытым коллектором и открытым стоком часто предназначены для поддержки этого использования и могут выдерживать более высокий ток, чем устройства с каскадным выходом. Некоторые версии позволяют подтянуть выход до уровня напряжения, превышающего уровень логической 1, что позволяет взаимодействовать с другими типами схем. Это важно, потому что, несмотря на согласованность порогов внутри семейства логических элементов, например в серии 7400, пороги у других семейств различаются.

Выход с тремя состояниями

Хотя схемы с открытым коллектором позволяют соединять выходы вместе, они не так быстры, как схемы с активным подтягиванием. Итак, отойдем от решения с двумя состояниями и представим выходы с *тремя состояниями*. Третье состояние — это состояние, когда выход выключен. Добавлен дополнительный вход *включения*, который включает и выключает выход, как показано на рис. 2.38.

Выключенное состояние известно как состояние *hi-Z*, или состояние с высоким сопротивлением (импедансом). Z — это символ *импеданса*, комплексного числа, обозначающего электрическое сопротивление цепи. Можно представить выход

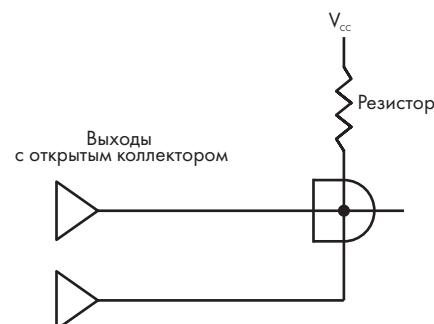


Рис. 2.37. Монтажное И

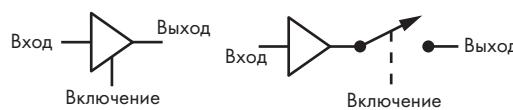


Рис. 2.38. Выход с тремя состояниями

с тремя состояниями в виде схемы, показанной на рис. 2.35. Управление вентилями по отдельности дает четыре комбинации: 0, 1, hi-Z и «выход из строя». Очевидно, что разработчики схем должны убедиться, что комбинация, при которой схема выйдет из строя, не может быть выбрана.

Выходы с тремя состояниями позволяют соединять вместе большое количество устройств. Важно помнить, что одновременно может быть включено только одно устройство.

Создание более сложных схем

Введение вентилей значительно упростило процесс проектирования аппаратного обеспечения. Мы избавились от необходимости разрабатывать всё из отдельных компонентов. Например, если для создания логического элемента И-НЕ с двумя входами требовалось около 10 электронных компонентов, то компонент из серии 7400 включал четыре из них в одном корпусе, что называлось *мелкомасштабной интеграцией* (small-scale integration, SSI), так что один корпус микросхемы мог заменить 40 электронных компонентов.

Разработчики аппаратного обеспечения могли построить что угодно из интегрированных логических вентилей, так же как и в случае с дискретными компонентами. Это делало вещи дешевле и компактнее. И поскольку определенные комбинации вентилей используются часто, были введены компоненты *интеграции среднего масштаба* (medium-scale integration, MSI), включающие эти комбинации, что дополнительно снизило количество необходимых частей. Позже появились *крупномасштабная интеграция* (large-scale integration, LSI), *сверхбольшая степень интеграции* (very large-scale integration, VLSI) и т. д.

Мы рассмотрим некоторые комбинации вентилей в следующих разделах, но и это еще не все. Мы используем эти функциональные строительные блоки высокого уровня для создания компонентов еще более высокого уровня, подобно тому как сложные компьютерные программы конструируются из более мелких программ.

Создание сумматора

Построим двоичный сумматор дополнительного кода. Возможно, вам никогда не придется разрабатывать что-то подобное, но этот пример демонстрирует, как умное управление логикой повышает производительность — что верно как для аппаратного, так и для программного обеспечения.

В главе 1 мы видели, что сумма двух битов представляет собой их исключающее ИЛИ, а перенос в старший разряд — И этих битов. На рис. 2.39 показана реализация вентиля.

Видим, что логический элемент исключающего ИЛИ получает сумму, а вентиль И — перенос. На рис. 2.39 представлен всего лишь *полусумматор*, потому что чего-то не хватает. Можно добавить два бита, но нужен третий вход, чтобы можно было осуществлять перенос. То есть чтобы получить сумму для каждого бита, необходимы два сумматора. Мы делаем перенос, когда по крайней мере два входа равны 1. Таблица 2.1 содержит таблицу истинности для *полного сумматора*.

Таблица 2.1. Таблица истинности для полного сумматора

A	B	C	Сумма	Перенос
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Построить полный сумматор немного сложнее (рис. 2.40).

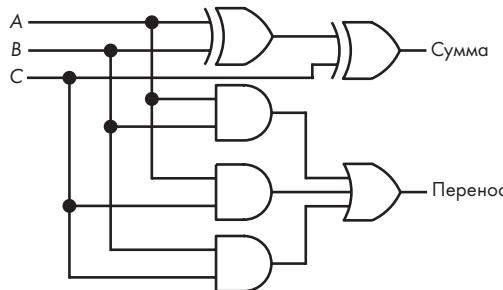


Рис. 2.40. Полный сумматор

Как видите, для него нужно гораздо больше логических вентилей. Но теперь, создав полный сумматор, мы можем использовать его, чтобы создать сумматор для более чем одного бита. На рис. 2.41 показана конструкция под названием «*каскадный сумматор*».

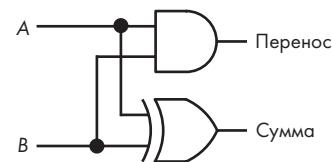


Рис. 2.39. Полусумматор

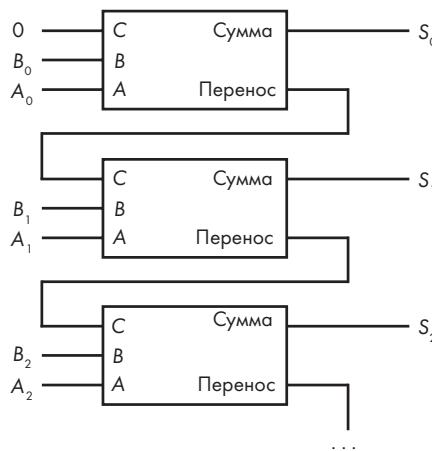


Рис. 2.41. Каскадный сумматор

Каскадный сумматор, или *сумматор со сквозным переносом*, получил свое название по тому, как перенос переходит от одного бита к другому. Процесс похож на волну. Пока он нормален, но можно заметить, что на один бит приходится две задержки затвора и их число быстро растет, если создавать 32- или 64-битный сумматор. Эти задержки можно устранить с помощью *сумматора с упреждающим переносом*. Несколько базовых арифметических операций помогут понять, как это сделать.

На рис. 2.40 видно, что перенос выхода полного сумматора для бита i передается как входной перенос в бит $i + 1$:

$$C_{i+1} = (A_i \text{ И } B_i) \text{ ИЛИ } (A_i \text{ И } C_i) \text{ ИЛИ } (B_i \text{ И } C_i).$$

Камнем преткновения здесь является то, что нам нужно знать C_i , чтобы получить C_{i+1} , что приводит к каскадной зависимости. Это видно из следующего уравнения для C_{i+2} :

$$C_{i+2} = (A_{i+1} \text{ И } B_{i+1}) \text{ ИЛИ } (A_{i+1} \text{ И } C_{i+1}) \text{ ИЛИ } (B_{i+1} \text{ И } C_{i+1}).$$

Можно устранить эту зависимость, подставив первое уравнение во второе, как показано ниже:

$$\begin{aligned} C_{i+2} &= (A_{i+1} \text{ И } B_{i+1}) \\ &\text{ИЛИ } (A_{i+1} \text{ И } ((A_i \text{ И } B_i) \text{ ИЛИ } (A_i \text{ И } C_i) \text{ ИЛИ } (B_i \text{ И } C_i))). \\ &\text{ИЛИ } (B_{i+1} \text{ И } ((A_i \text{ И } B_i) \text{ ИЛИ } (A_i \text{ И } C_i) \text{ ИЛИ } (B_i \text{ И } C_i))). \end{aligned}$$

Обратите внимание, что, хотя здесь намного больше операторов И и ИЛИ, задержка распространения по-прежнему зависит лишь от двух вентилей. C_n зависит только от входов A и B , поэтому время переноса и, следовательно, время

суммирования не зависят от количества битов. C_n всегда можно сгенерировать из C_{n-1} , который использует все большее количество вентилей по мере увеличения n . Хотя логические вентили недороги, они потребляют электроэнергию, поэтому нужно учитывать компромисс между скоростью и потребляемой мощностью.

Построение дешифраторов

В разделе «Представление целых чисел с помощью битов» на с. 43 мы создали, или *зашифровали*, числа из битов. *Дешифратор* делает обратное, снова превращая зашифрованное число в набор отдельных битов. Одно из применений дешифраторов — управление дисплеями. Возможно, вы видели *газоразрядные индикаторы* (показанные на рис. 2.42) в старых научно-фантастических фильмах; это и в самом деле классный ретродисплей для чисел. По сути, это набор неоновых вывесок, по одной на каждую цифру. Каждый светящийся провод имеет собственное соединение, поэтому необходимо преобразовать 4-битное число в 10 отдельных выходов.

Вспомним, что восьмеричное представление принимает восемь разных значений и кодирует их в 3 бита. На рис. 2.43 показан дешифратор 3 : 8, который преобразует восьмеричное значение обратно в набор отдельных битов.

Если вход равен 000, выход Y_0 истинен; когда на входе 001, Y_1 истинен; и т. д. Дешифраторы в основном называются по количеству входов и выходов. Пример на рис. 2.43 имеет три входа и восемь выходов, так что это дешифратор 3 : 8. Обычно подобные дешифраторы изображаются, как показано на рис. 2.44.

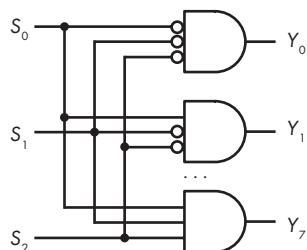


Рис. 2.43. Дешифратор 3 : 8



Рис. 2.42. Газоразрядный индикатор

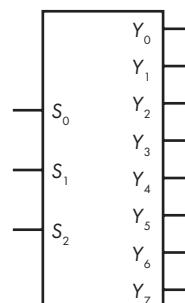


Рис. 2.44. Условное обозначение дешифратора 3 : 8

Построение демультиплексоров

Дешифратор можно использовать для создания *демультиплексора*, обычно обозначаемого как *dmux* (от demultiplexer), который позволяет направлять входной сигнал на один из нескольких выходов (как если бы вы распределяли студентов Хогвартса по факультетам). Демультиплексор включает в себя дешифратор с дополнительными логическими вентилями, как показано на рис. 2.45.

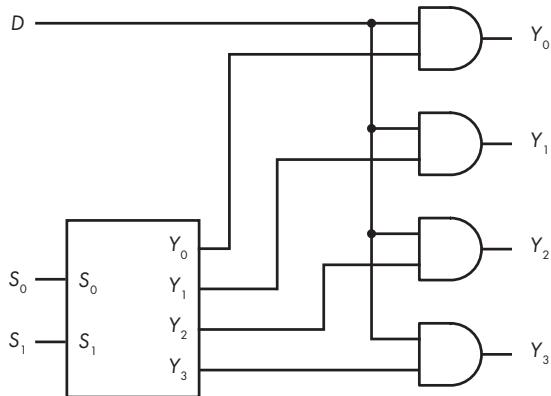


Рис. 2.45. Демультиплексор 1 : 4

Демультиплексор направляет входной сигнал D на один из четырех выходов Y_{0-3} на основе входов дешифратора S_{0-1} . Обозначение на рис. 2.46 используется в схемах для обозначения демультиплексора.

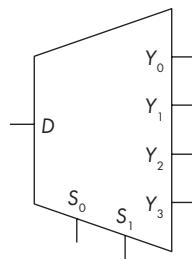


Рис. 2.46. Условное обозначение демультиплексора

Построение селекторов

Выбор одного входа из ряда входов — еще одна часто выполняемая функция. Например, у нас может быть несколько источников operandов для сумматора, и нужно выбрать только один. Используя вентили, можно создать еще один

функциональный блок, называемый *селектором* или *мультплексором* (*тих*, от *multiplexer*).

Селектор объединяет дешифратор с дополнительными вентилями, как показано на рис. 2.47.

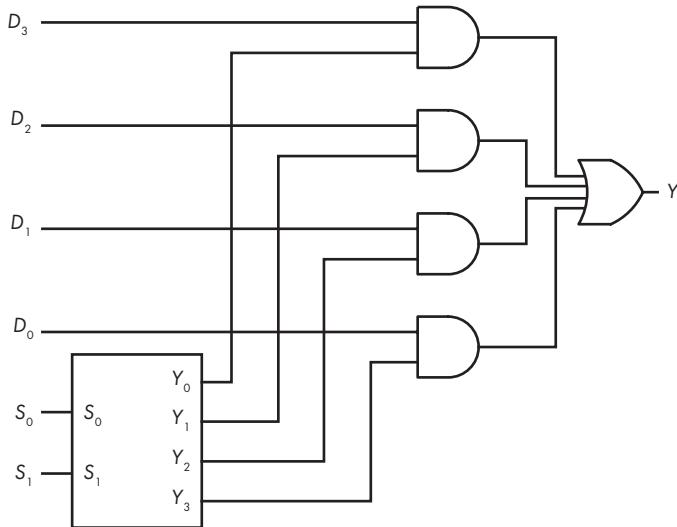


Рис. 2.47. Селектор 4 : 1

Селекторы также часто используются и имеют свое условное обозначение. На рис. 2.48 показано обозначение селектора 4 : 1, которое в значительной степени является обратным обозначением дешифратора.

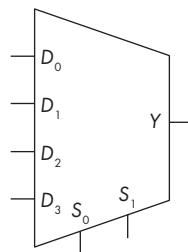


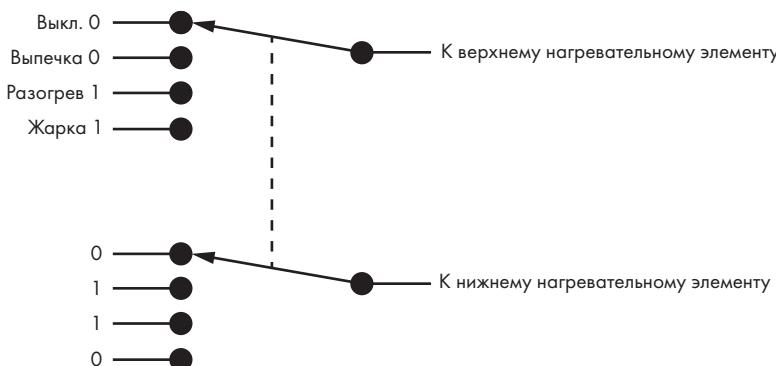
Рис. 2.48. Условное обозначение селектора 4 : 1

Вы, наверное, уже знакомы с селекторами, но не знаете об этом. Возможно, у вас есть тостер со шкалой с обозначениями «Выкл.», «Разогрев», «Выпечка» и «Жарка». Это и есть *селекторный переключатель* с четырьмя положениями. В тостере есть два нагревательных элемента: один сверху, а другой снизу. Логика работы тостера показана в табл. 2.2.

Таблица 2.2. Логика работы тостера

Настройка	Верхний элемент	Нижний элемент
Выкл.	Выкл.	Выкл.
Выпечка	Выкл.	Вкл.
Разогрев	Вкл.	Вкл.
Жарка	Вкл.	Выкл.

Можно реализовать эту логику, используя пару селекторов 4 : 1, объединенных, как показано на рис. 2.49.

**Рис. 2.49.** Селекторный переключатель тостера

Выводы

В этой главе вы узнали, почему мы используем биты вместо цифр для создания аппаратного обеспечения. Мы рассмотрели некоторые достижения в области техники, полезные для реализации битов и комбинаторной цифровой логики. Мы изучили современные обозначения логических элементов и то, как можно комбинировать простые логические элементы для создания более сложных устройств. Мы выяснили, что выходы комбинаторных устройств функционально зависят от входов, но, поскольку выходы меняются при изменении входов, запомнить что-либо невозможно. Для запоминания требуется способность «заморозить» выход, чтобы он не менялся при изменении входа. В главе 3 обсуждается последовательная логика, которая позволяет постепенно запоминать значения.

3

Последовательная логика



Комбинаторная логика, о которой вы узнали в предыдущей главе, «плывет по течению». Другими словами, выходные данные изменяются в зависимости от входных. Но невозможно создавать компьютеры только на основе комбинаторной логики, потому что она не предполагает возможности удалить что-то из потока и запомнить это. Например, вы не сможете сложить все числа от 1 до 100, если не будете отслеживать, в каком месте последовательности находитесь.

В этой главе вы узнаете о *последовательной логике*. Термин происходит от слова «последовательность», что означает «одно следует за другим во времени». Будучи человеком, вы интуитивно понимаете время и счет на пальцах, но это не значит, что время является естественным для цифровых схем. Мы должны как-то создать его.

Комбинаторная логика имеет дело только с текущим состоянием входных данных. А вот последовательная логика учитывает как текущее состояние, так и прошлое. В этой главе вы узнаете о методиках создания времени и методиках запоминания того, что было в прошлом. Мы проследим некоторые технологии, которые использовались для этих целей, от их зарождения до наших дней.

Представление времени

Люди измеряют время с помощью некой *периодической* функции — вращения Земли. Мы называем один полный оборот сутками и делим его на более мелкие единицы — часы, минуты и секунды. Секунду можно определить как 1/86 400 оборота Земли, поскольку в сутках 86 400 секунд.

В дополнение к использованию внешнего события, такого как вращение Земли, можно генерировать собственные периодические функции, применяя определенные понятия физики — например, время, необходимое для одного движения маятника. Именно благодаря этой технике старые дедовские часы издавали звук «тик-так». Конечно, чтобы маятник был полезен, его необходимо откалибровать по измеренной длине секунды.

Работая с компьютерами, мы используем электронику, поэтому нам нужен периодический электрический сигнал. Можно создать что-то подобное, поместив переключатель так, чтобы маятник ударял по нему. Но, если только вы не фанат стимпанка, вам, вероятно, не нужен компьютер с маятниковым питанием. В следующем разделе мы узнаем о более современных подходах.

Осцилляторы

Посмотрим, что можно сделать с инвертором: можно подключить его выход ко входу, как показано на рис. 3.1.

Это приведет к появлению *обратной связи*, такой же, как если поставить микрофон слишком близко к громкоговорителю. Выходной сигнал инвертора качается взад-вперед или *колеблется* между 0 и 1.

Скорость, с которой он колеблется, является функцией задержки распространения (см. раздел «Задержка распространения» на с. 100), которая имеет тенденцию меняться в зависимости от температуры. Полезно иметь генератор со стабильной частотой, чтобы генерировать точную привязку ко времени.

Экономически эффективный способ сделать это — использовать кристалл. Да, очень современно. Кристаллы, как и магниты, связаны с электричеством. Если прикрепить электроды (проводы) к кристаллу и сжать его, он будет генерировать электрический ток. И если пустить ток по этим проводам, кристалл изогнется. Это называется *пьезоэлектрическим* эффектом, он был открыт братьями Полем-Жаком (1855–1941) и Пьером (1859–1906) Кюри в конце XIX века. Пьезоэлектрический эффект имеет множество применений. Кристалл может улавливать звуковые колебания, превращаясь в микрофон. Звуковые колебания, возникающие при подаче электричества на кристаллы, вызывают раздражающие звуковые сигналы, которые можно услышать от многих приборов. На принципиальных электрических схемах кристалл обозначается символом, показанным на рис. 3.2.

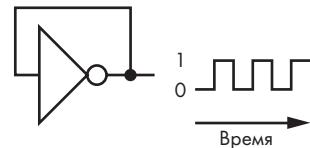


Рис. 3.1. Осциллятор



Рис. 3.2. Условное обозначение кристалла

Кварцевый осциллятор попеременно подает напряжение на кристалл и принимает его обратно с помощью электронных однополюсных переключателей с группой контактов на переключение. Время, необходимое кристаллу для этого действия, предсказуемо и очень точно. Кристаллический материал, лучше всего подходящий для такой цели, — кварц. Вот почему так часто можно видеть рекламу точных кварцевых часов. В следующий раз, обратив внимание на цену новых модных часов, вспомните, что действительно хороший кристалл продается всего за 25 центов.

Генераторы тактовых сигналов

Как вы убедились, осцилляторы позволяют измерять время. Очевидно, что компьютерам требуется точно отсчитывать его, например, чтобы воспроизводить видео с постоянной скоростью. Но на это есть еще одна причина более низкого уровня. В главе 2 мы обсудили, как задержка распространения влияет на время, необходимое схемам для выполнения действий. Время дает возможность дождаться, например, наихудшей задержки в сумматоре, прежде чем изучать полученный результат — чтобы удостовериться, что он стабильный и правильный.

Осцилляторы обеспечивают в компьютерах тактовые сигналы. Компьютерные тактовые сигналы подобны барабанщику в оркестре; они задают темп схемотехнике. Максимальная тактовая частота или самый быстрый темп определяется задержками распространения.

Производство компонентов связано с большим количеством статистических данных, потому что от детали к детали появляются большие различия. В процессе сортировки компоненты помещаются в разные ячейки или стопки в зависимости от их измеренных характеристик. Самые быстрые части по самой высокой цене пойдут в одну ячейку; более медленные и менее дорогие — в другую и т. д. Нечелесообразно иметь бесконечное количество ячеек, поэтому элементы в ячейке могут немного отличаться друг от друга, хотя эта разница меньше, чем различия между деталями во всей партии. Это одна из причин, по которой задержки распространения указываются в виде диапазона; производители предоставляют минимальные и максимальные значения в дополнение к типовому значению. Распространенной ошибкой проектирования логической схемы является использование типовых значений вместо минимумов и максимумов. Люди, разгоняющие свои компьютеры, делают ставку на то, что их деталь статистически находится в середине ячейки — то есть ее тактовый сигнал может быть увеличен на некоторую величину, и это не приведет к поломке детали.

Триггеры-защелки

Разобравшись с источниками времени, попробуем запомнить хотя бы небольшое количество информации. Это можно сделать с помощью обратной связи,

например, привязав выход логического элемента ИЛИ ко входу, как показано на рис. 3.3. При этом не создается осциллятор, подобный тому, который мы видели на рис. 3.1, поскольку здесь нет инверсии. Предположим, что *выход* начинается с 0 в схеме на рис. 3.3. Теперь, если на *входе* мы имеем 1, *выход* тоже будет равен 1, и, поскольку он подключен к другому входу, он останется одинаковым, даже если поменять *вход* на 0. Другими словами, выход запоминает данные.



Рис. 3.3. Триггер-защелка логического элемента ИЛИ

Конечно, эта схема требует некоторой доработки, потому что невозможно повторно вычислить значение 0 для *выхода*. Нам нужен способ сбросить значение, отключив обратную связь, как показано на рис. 3.4.

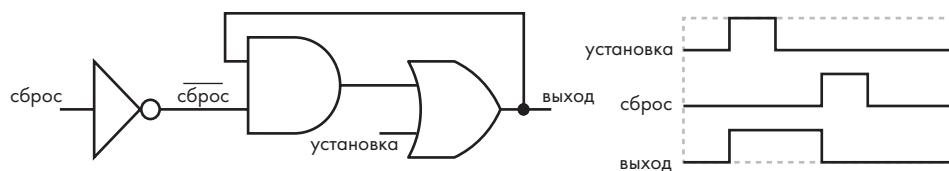


Рис. 3.4. Триггер-защелка логического элемента И-ИЛИ

Обратите внимание, что мы обозначили выход инвертора как *сброс*. Линия поверх символа на аппаратном языке означает «противоположность». То есть что-то истинно, если равно 0, и ложно, если равно 1. Иногда это называют *активным низким уровнем*, а не *активным высоким уровнем*, что означает выполнение логики при 0, а не 1. Линия произносится как «чертка», поэтому в устной речи сигнал будет называться «сброс с чертой».

Когда *сброс* находится на низком уровне, *сброс*, наоборот, находится на высоком, поэтому выход логического элемента ИЛИ возвращается на вход. Когда *сброс* переходит на высокий уровень, *сброс* оказывается на низком, нарушая эту обратную связь, так что *выход* становится равным 0.

На рис. 3.5 показан *RS-триггер* – чуть более умный способ создания бита памяти. *RS* означает *установка-сброс* (set-reset). Он имеет активные входы низкого уровня и дополнительные выходы, это означает, что один из них активен на низком уровне, а другой – на высоком. Можно создать версию с активными высокими входами, используя вентили ИЛИ-НЕ, но они часто более энергоемкие, чем вентили И-НЕ, и собирать их сложнее и дороже.



Рис. 3.5. RS-триггер

Ситуация, когда активны и установка, и сброс, обычно не возникает, потому что оба выхода будут истинными. Кроме того, если оба входа становятся неактивными (то есть переходят с 0 на 1) одновременно, состояние выходов невозможно предсказать, поскольку оно зависит от задержек распространения.

Схема на рис. 3.5 имеет приятное свойство, которого нет в схеме на рис. 3.4, — ее конструкция симметрична. Это означает, что задержки распространения одинаковы как для сигналов установки, так и для сигналов сброса.

Синхронный RS-триггер

Получив новый способ запоминания информации, посмотрим, что нужно, чтобы запомнить что-то в определенный момент времени. В схеме на рис. 3.6 ко входам добавлена дополнительная пара логических вентилей.

Как видно, когда вход синхронизации (синхр.) неактивен (высокий), не имеет значения, что происходит при установке и сбросе; выходы не изменяются, потому что оба входа для вентилей S и R будут равны 1.

Поскольку необходимо запомнить один бит информации, можно улучшить схему следующим образом — добавить инвертор между входами установки и сброса, так что останется один вход данных, который мы будем обозначать как D . Это изменение показано на рис. 3.7.

Теперь, если D равен 1, когда синхр. низкий, выход Q будет установлен как 1. Аналогично, если D равен 0, когда синхр. низкий, Q будет установлен как 0. Изменения на D при высоком синхр. не имеют никакого эффекта. Это означает, что можно помнить состояние D . То же самое показано на временной диаграмме, изображенной на рис. 3.8.

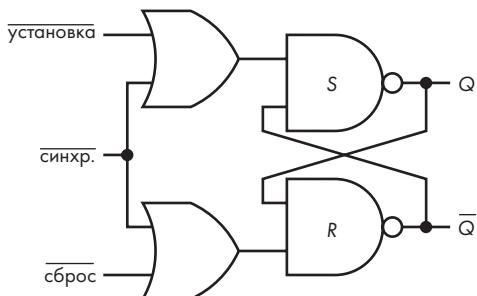


Рис. 3.6. Синхронный RS-триггер

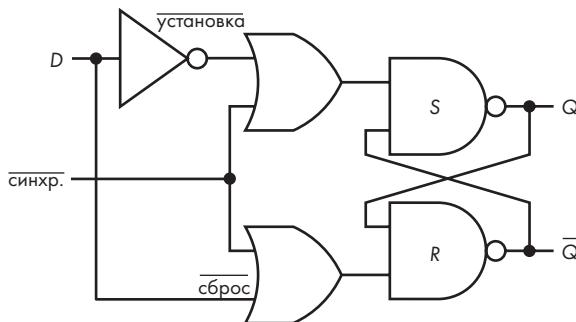


Рис. 3.7. Инвертирующий D-триггер

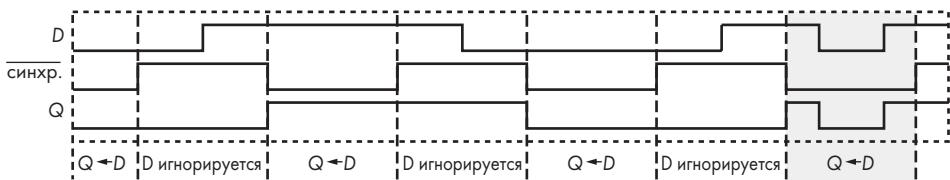


Рис. 3.8. Временная диаграмма инвертирующего D-триггера

Проблема с этой схемой заключается в том, что изменения D проходят всякий раз при низком синхр., как видно в выделенной серым цветом части. Поэтому приходится рассчитывать, что D будет «вести себя хорошо» и не изменится при «открытии» «входа синхронизации». Возможность мгновенно открывать вентиль предпочтительнее. Рассмотрим, как это сделать, в следующем разделе.

Триггеры

Как мы обсуждали в разделе выше, нужно свести к минимуму вероятность получения неверных результатов из-за изменения данных. Обычно это делается с помощью захвата данных во время перехода между логическими уровнями, а не на самом логическом уровне, который имеет определенное значение. Эти переходы называются *фронтами*. Фронт можно рассматривать как критерий выбора для времени. Вернувшись к рис. 3.8, видим, что переход между логическими уровнями происходит почти мгновенно. Такие типы электронных устройств с запуском по фронту называются *триггерами*.

RS-триггеры — это строительные блоки, из которых собираются другие типы триггеров. Можно сконструировать триггер с запуском по положительному фронту — он называется *прямой D-триггер*, или просто *D-триггер*. Для этого нужно скомбинировать три RS-триггера, как показано на рис. 3.9. Запуск по положительному фронту означает, что триггер сработает при переходе от логического 0 к логической 1; триггер с запуском по отрицательному фронту будет работать при переходе от логической 1 к логическому 0.

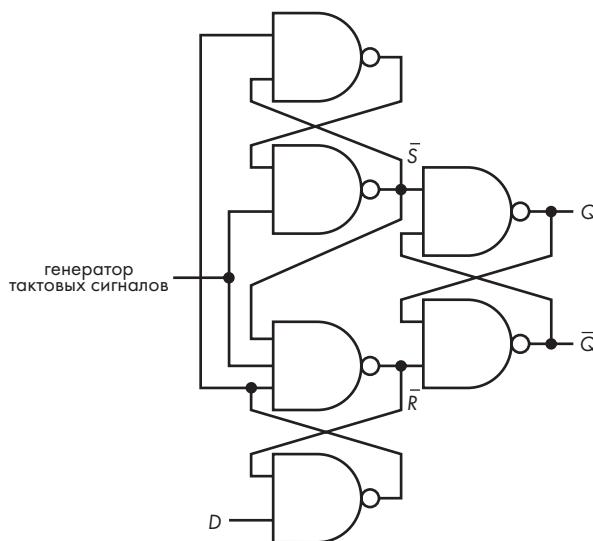


Рис. 3.9. Проект D-триггера

Эту схему не просто понять с первого взгляда. Два вентиля справа образуют RS-триггер. Из рис. 3.5 мы знаем, что выходы не изменятся, если \bar{S} или \bar{R} не станут низкими.

На рис. 3.10 показано, как схема ведет себя при различных значениях D и *тактовой частоты*. Тонкие линии показывают логические нули, толстые — логические единицы. Начиная с левого края, видим, что, когда на генераторе 0, значение D не так важно, потому что и \bar{S} и \bar{R} имеют высокий уровень, поэтому состояние защелки в правой части рис. 3.9 не изменилось. Двигаясь вправо, по следующим двум схемам видим, что при низком \bar{R} изменение значения D не влияет на

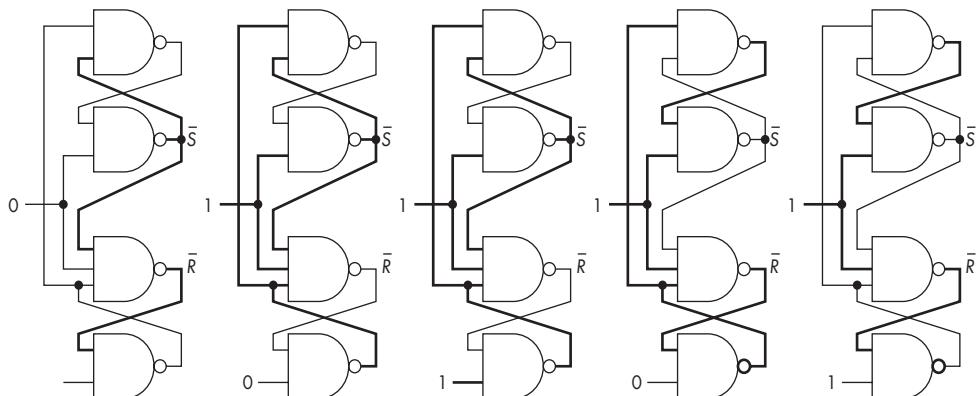


Рис. 3.10. Работа D-триггера

результат. Аналогичным образом две крайние правые схемы показывают, что при низком \bar{R} изменение значения D не влияет на результат. То есть изменения на D не важны, если генератор тактовых сигналов имеет высокий или низкий уровень.

Теперь посмотрим, что произойдет, когда генератор сменит уровень с низкого на высокий, как показано на рис. 3.11.

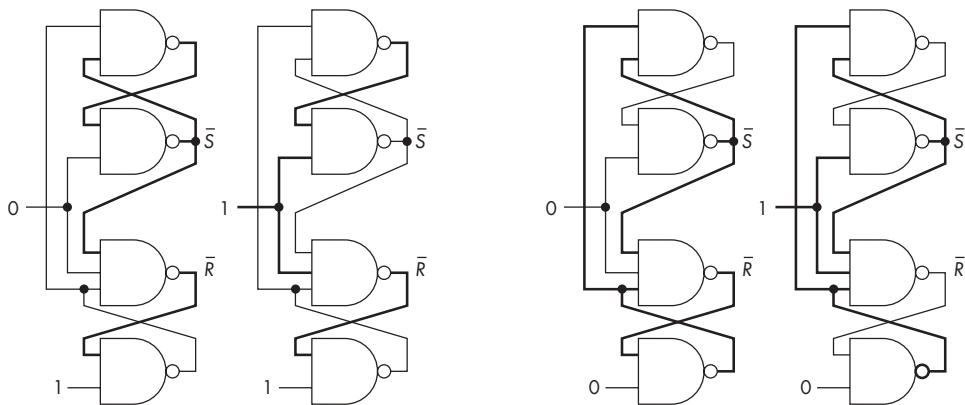


Рис. 3.11. Работа D-триггера с положительным фронтом

Слева видим, что при низком уровне генератора и высоком D \bar{S} и \bar{R} также имеют высокий уровень, поэтому ничего не меняется. Но когда генератор меняет значение на 1, \bar{S} переходит на низкий уровень, что меняет состояние триггера. Справа видим аналогичное поведение — при низком D и высоком уровне генератора \bar{R} становится низким и меняет состояние триггера. Из рис. 3.10 понятно, что никакие другие изменения не влияют на результат.

В 1918 году британские физики Уильям Экклс (William Eccles) и Фрэнк Джордан (Frank Jordan) изобрели первую электронную версию триггера, в которой использовались вакуумные лампы. На рис. 3.12 показано условное обозначение чуть менее стационарного *D-триггера* под названием 7474.

D-триггер имеет комплементарные выходы Q и \bar{Q} , комплементарные входы \bar{S} (установка) и \bar{R} (сброс). Это немного сбивает с толку, поскольку на схеме показаны S и R ; комбинация со знаком \circ делает их \bar{S} и \bar{R} . Итак, это похоже на уже изученный *RS-триггер*, за исключением непонятных обозначений в левой части. Непонятные обозначения — это два дополнительных входа: D — для данных (data) и CK — для тактовых сигналов (clock), которые представлены треугольником. Это триггер с запуском по положительному фронту, поэтому значение входа D сохраняется всякий раз, когда сигнал на CK изменяется от 0 до 1.

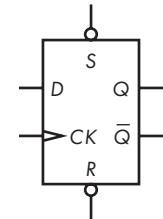


Рис. 3.12.
D-триггер

Устройства, запускаемые по фронту, имеют и другие нюансы в подсчете времени, кроме задержки распространения. Учитывается *время установки*, которое представляет собой промежуток времени до фронта тактового сигнала, в течение которого сигнал должен быть стабильным, и *время удержания*, которое представляет собой промежуток времени после фронта тактового сигнала, в течение которого сигнал должен быть стабильным. Они показаны на рис. 3.13.



Рис. 3.13. Время установки и удержания

Как видите, нам не нужно заботиться о том, что происходит на входе D , за исключением времени установки и удержания по фронту тактового сигнала. И, как и в случае со всей другой логикой, выходной сигнал становится стабильным после времени задержки распространения и остается таковым независимо от входа D . Время установки (setup) и удержания (hold) обычно обозначается как t_{setup} и t_{hold} .

Поведение триггеров на фронтах хорошо сочетается с генераторами тактовых сигналов. Мы убедимся в этом на примере в следующем разделе.

Счетчики

Триггеры часто применяются для подсчета. Например, можно отсчитывать время от осциллятора и управлять дисплеем с помощью дешифратора, получив в результате цифровые часы. На рис. 3.14 показана схема, которая выдает 3-битное число (C_2, C_1, C_0) , то есть количество раз, когда сигнал изменяется с 0 на 1. Сигнал сброса может использоваться для установки счетчика на 0.

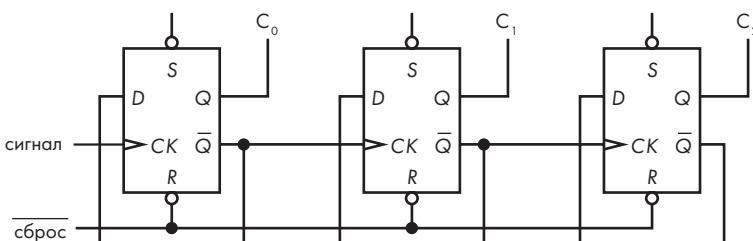


Рис. 3.14. 3-битный счетчик со сквозным переносом

Данный счетчик называется *счетчиком со сквозным переносом*, и его результат передается слева направо. C_0 изменяет C_1 , C_1 изменяет C_2 и т. д., если есть другие биты. Поскольку вход D каждого триггера соединен с его выходом \bar{Q} , он будет менять состояние при каждом положительном переходе сигнала CK .

Этот счетчик еще называют *асинхронным*, потому что все его элементы срабатывают только тогда, когда до них дойдет сигнал. Проблема асинхронных систем заключается в том, что трудно понять, когда получен конечный результат. Выходы (C_2, C_1, C_0) не активны во время сквозной передачи сигнала. Вы можете увидеть на рис. 3.15, что для получения результата для каждого последующего бита требуется все больше времени. Серые области на рисунке представляют неопределенные значения из-за задержки распространения.

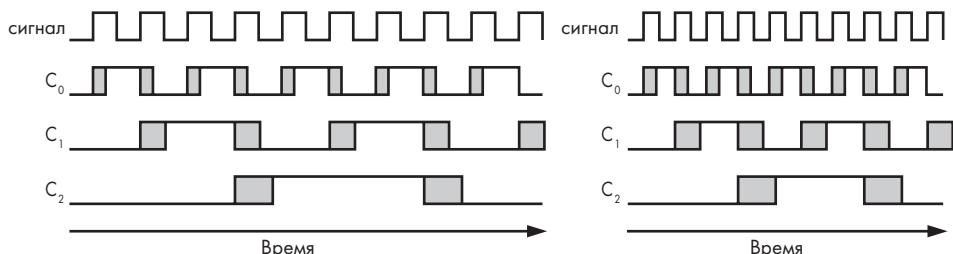


Рис. 3.15. Временная диаграмма счетчика со сквозным переносом

Временная диаграмма слева показывает, что мы получаем действительное 3-битное число, после того как устанавливаются задержки распространения. Но справа видно, что мы пытаемся считать результат быстрее, чем позволяют задержки распространения, поэтому бывают случаи, когда действительное число не получается.

Это похоже на проблему, которую мы рассматривали при изучении каскадного сумматора на рис. 2.41. Подобно тому как мы смогли решить ее с помощью упреждающего переноса, мы можем исправить проблему со сквозным переносом, используя *синхронный* счетчик.

В отличие от счетчика со сквозным переносом все выходы синхронного счетчика изменяются одновременно (синхронно). Это означает, что все триггеры синхронизируются параллельно. Трехбитный синхронный счетчик показан на рис. 3.16.

Видим, что все триггеры в состоянии счетчика меняются одновременно, потому что все они одновременно синхронизируются. Хотя задержка распространения по-прежнему определяет правильность результатов, каскадный эффект устраняется.

Счетчики — это еще один функциональный строительный блок, то есть у них имеется свое схематическое обозначение. В данном случае это еще один прямоугольник, как видно на рис. 3.17.

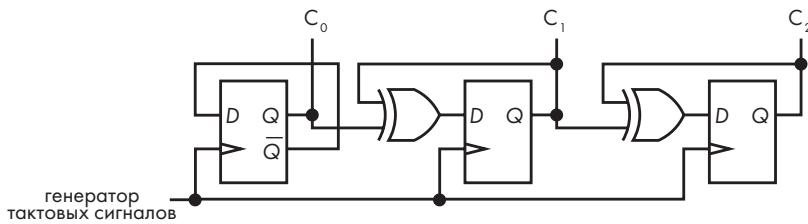


Рис. 3.16. 3-битный синхронный счетчик



Рис. 3.17. Условное обозначение счетчика на схеме

На рисунке показан ряд входов, которые мы раньше не видели. Счетчики могут иметь все или только некоторые из этих входов. Большинство счетчиков имеют вход *CLR*, который очищает счетчик, устанавливая его значение равным 0. Также распространен вход *EN*, который включает счетчик (тот не ведет счет, если не включен). Некоторые счетчики могут вести счет в любом направлении; вход *U/D* выбирает направление вверх или вниз. Наконец, иногда счетчики имеют входы данных *D_{0-n}* и сигнала нагрузки *LD*, позволяющие задать определенное значение для счетчика.

Теперь можно использовать счетчики для учета времени. Но это не единственное, что умеют триггеры. Мы разберемся, как запоминать большие объемы информации, в следующем разделе.

Регистры

D-триггеры хорошо подходят для запоминания информации. Часто с той же целью используются *регистры*, представляющие собой набор *D*-триггеров в связке с общим генератором тактовых сигналов. На рис. 3.18 показан пример регистра, хранящего результат сложения с использованием описанной ранее схемы сумматора.

После синхронизации результата работы сумматора с регистром можно изменять операнды, не меняя результата. Обратите внимание, что регистры часто имеют входы разрешения, аналогичные таким же для счетчиков.

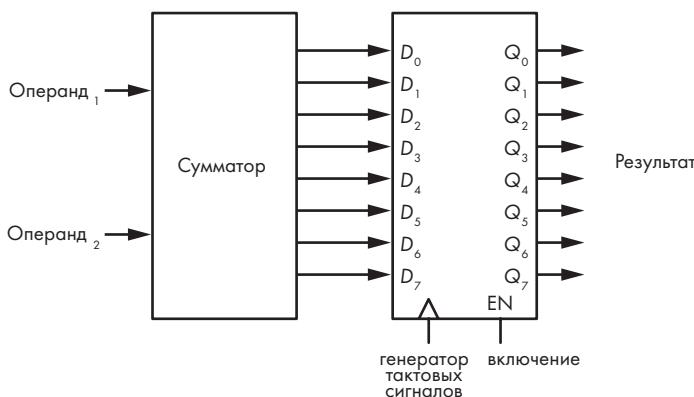


Рис. 3.18. Регистр, хранящий результат работы сумматора

Организация памяти и обращение к памяти

Мы уже убедились, что триггеры полезны, когда нужно запомнить немного информации, и что регистры удобны для запоминания набора битов. Но что использовать, когда нужно запомнить намного больше? Например, если необходимо хранить несколько разных результатов сложения?

Что ж, можно начать с большой кучи регистров. Однако здесь возникает новая проблема: как указать регистр, который нужно использовать? Эта ситуация проиллюстрирована на рис. 3.19.

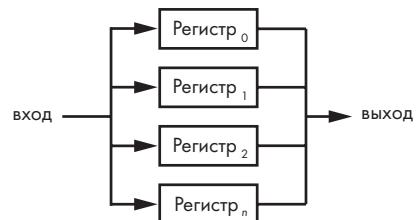


Рис. 3.19. Несколько регистров

Один из способов решить эту проблему — присвоить каждому регистру номер, как показано на рисунке. Можно указать этот номер, или *адрес*, чтобы выбрать регистр, используя один из стандартных строительных блоков — дешифратор из раздела «Построение дешифраторов» на с. 107. Выходы дешифратора подключены ко входам включения регистров.

Далее нужно предусмотреть возможность выбирать выход из указанного регистра. К счастью, мы узнали, как создавать селекторы, в разделе «Построение селекторов» на с. 108, и они как раз нам пригодятся.

Системы часто имеют несколько компонентов памяти, которые необходимо соединить. Пришло время для еще одного из стандартных строительных блоков: *выхода с тремя состояниями*.

В совокупности элемент памяти выглядит, как показано на рис. 3.20.

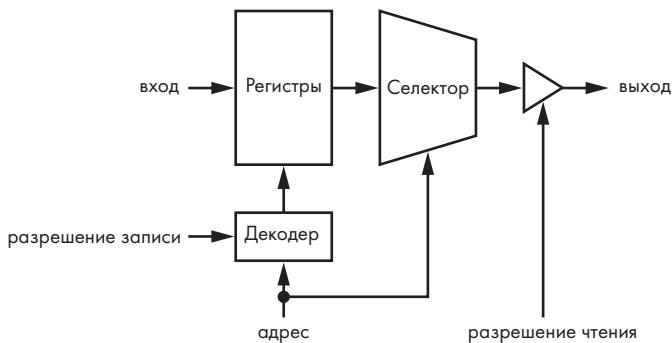


Рис. 3.20. Элемент памяти

В элементах памяти множество электрических соединений. Если мы оперируем 32-битными числами, нам потребуется по 32 соединения для каждого входа и выхода плюс соединения для адреса, сигналов управления и питания. Программистам не нужно беспокоиться о том, как уместить схемы в ячейки или как проложить провода, — это делают разработчики аппаратного обеспечения. Можно сократить количество подключений, так как вряд ли понадобится читать и записывать в память одновременно. Тогда можно обойтись одним набором подключений к данным плюс разрешения на *чтение/запись*. На рис. 3.21 показана упрощенная схема микросхемы памяти. Она управляется элементом управления *включение*, так что можно соединить вместе несколько микросхем памяти.

Видим, что на рисунке вместо отдельных сигналов используются большие толстые стрелки для адреса и данных. Мы называем группы связанных сигналов *шинами*, поэтому микросхема памяти имеет *адресную шину* и *шину данных*. Да, это аналогия с общественным транспортом для битов (bus (шина) – автобус).

Еще одна проблема при упаковке микросхем памяти возникает, когда размер памяти увеличивается и требуется подключение большого количества битов адреса. Если свериться с табл. 1.2 в главе 1, нам потребуется 32 адресных соединения для компонента памяти объемом 4 ГиБ.

Проектировщики памяти и планировщики дорог решают аналогичные проблемы управления движением. Многие города организованы в сети, и точно так же устроены внутри микросхемы памяти. На микрофотографии ЦП, показанной на рис. 2.3, видим несколько прямоугольных областей, которые представляют собой блоки памяти. Адрес разделен на две части: адрес *строки* и адрес *столбца*.

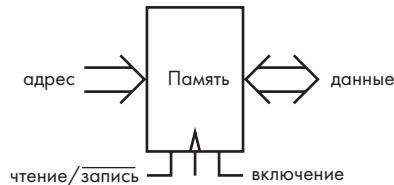


Рис. 3.21. Упрощенная схема микросхемы памяти

Адрес ячейки памяти внутренним образом определяется по пересечению строки и столбца, как показано на рис. 3.22.

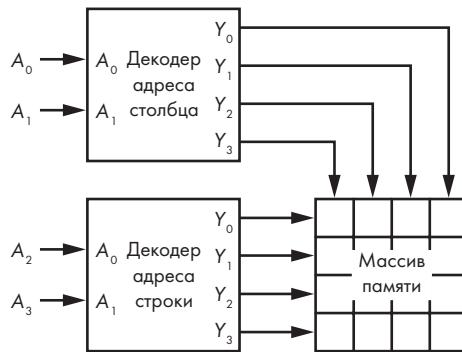


Рис. 3.22. Адресация строк и столбцов

Очевидно, нам не нужно беспокоиться о количестве адресных строк в 16-разрядной памяти, показанной на этом рисунке. Но что, если разрядов намного больше? Можно вдвое сократить количество адресных строк, *мультиплексируя* адреса строк и столбцов. Все, что нам потребуется, это регистры на микросхеме памяти для их сохранения, как показано на рис. 3.23.

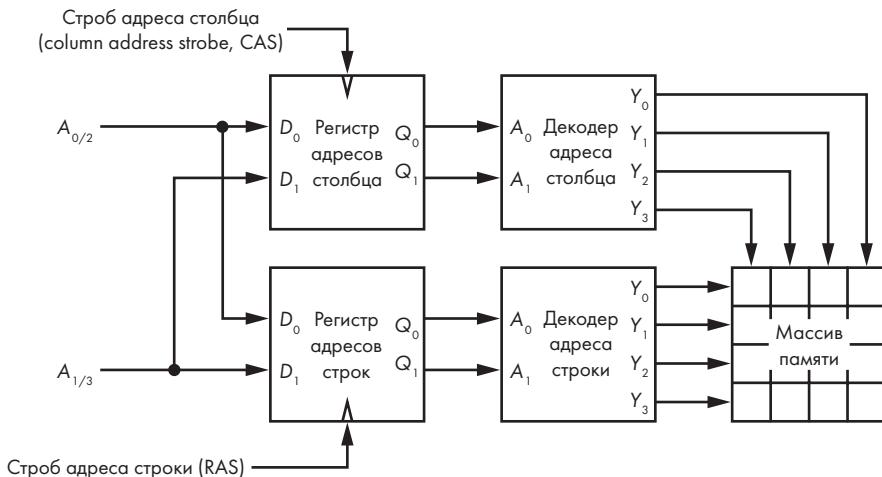


Рис. 3.23. Память с адресными регистрами

Поскольку адрес состоит из двух частей, отсюда следует, что производительность улучшилась бы, если бы нам нужно было изменить только одну часть, например, установив адрес строки и затем изменив адрес столбца. Такой подход используется в современных больших микросхемах памяти.

Микросхемы памяти описываются по их размеру в формате глубина × ширина. Например, микросхема 256×8 будет иметь 256 ячеек памяти шириной 8 бит; микросхема $64 \text{ МиБ} \times 1$ будет иметь 64 мебибита.

Оперативная память

Память, о которой мы уже говорили, называется *памятью на основе оперативно-запоминающего устройства, ОЗУ* (random-access memory, RAM). С помощью ОЗУ полная ширина любой области памяти может быть прочитана или записана в любом порядке.

Статическая память (static RAM, SRAM) – это дорого, но быстро. На каждый бит требуется шесть транзисторов. Поскольку транзисторы занимают место, SRAM не очень хорошо подходит для хранения миллиардов или триллионов битов.

Динамическая память (dynamic memory, DRAM) – хитрый прием. Электроны хранятся в микроскопических емкостях, называемых *конденсаторами*, с использованием только одного транзистора в качестве крышек емкостей. Проблема в том, что в этих емкостях происходят утечки, поэтому требуется время от времени *обновлять* память – регулярно пополнять емкости. Необходимо следить, чтобы обновление не произошло в критический момент, вызвав конфликт с доступом к памяти; это было проблемой в работе с одним из первых компьютеров на базе DRAM, DEC LSI-11. Одним из интересных побочных эффектов DRAM является то, что емкости пропускают больше, если на них падает свет. Это позволяет использовать их в качестве цифровых фотоаппаратов.

DRAM применяется для больших микросхем памяти благодаря своей высокой плотности (количество битов на область). Большие микросхемы памяти означают множество адресов, в связи с чем микросхемы DRAM используют схему мультиплексированной адресации, описанную в предыдущем разделе. Из-за других соображений внутреннего дизайна адрес строки можно сохранить быстрее с помощью строба адреса строки, а затем изменить адрес столбца с помощью строба адреса столбца. Строки – часто используемый термин, но их иногда еще называют *страницами*. Поэтому процесс можно сравнить с чтением книги – сканировать страницу намного проще, чем перелистывать. Это очень важный принцип программирования: хранение вещей, которые используются вместе, в одном ряду значительно повышает производительность.

И SRAM, и DRAM являются *энергозависимой* памятью, а значит, данные могут быть потеряны при отключении питания. *Память на магнитных сердечниках* – старинный *энергонезависимый* тип ОЗУ, в котором биты хранятся в *тороидальных* (пончиковидных) железных элементах, которые показаны на рис. 3.24. Тороиды были намагниченны: одно направление означало 0, а другое – 1. Физическая природа тороидов интересна, потому что они очень устойчивы к электромагнитным помехам извне. При таком типе памяти ферритовые сердечники располагались в сетке проводников, называемой *плоскостью*, через

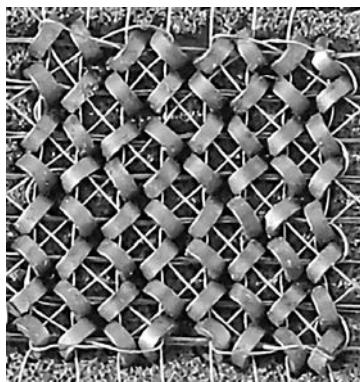


Рис. 3.24. Память на магнитных сердечниках

которую проходили вертикальные и горизонтальные проводники. Был также третий провод — проводник «ощущений». Он назывался так потому, что единственный способ прочитать состояние бита — попытаться его изменить, а затем «почувствовать», что произошло. Конечно, выяснив, что бит изменился, вы должны были вернуть его значение, иначе данные терялись — и бит оказывался бесполезным. Вся система основывалась на множестве дополнительных схем. Память на магнитных сердечниках на самом деле была трехмерной, поскольку плоскости собирались в блоки.

Несмотря на то что память на магнитных сердечниках не нова, ее энергонезависимые свойства по-прежнему ценятся и продолжаются исследования по созданию коммерчески практичной *магниторезистивной* памяти, которая сочетает в себе лучшее из технологий памяти на магнитных сердечниках и оперативной памяти.

Постоянное запоминающее устройство

Постоянное запоминающее устройство, или ПЗУ (read-only memory, ROM), — не очень точное название¹. Память, которую можно только прочитать, но никогда не записать, бесполезна. Несмотря на то что название прижилось, точнее будет сказать, что ПЗУ — это память с однократной записью. ПЗУ можно записать один раз, а затем многократно читать. ПЗУ важно для устройств, которые должны иметь встроенную программу, например для микроволновой печи; представьте, что было бы, если бы вам приходилось перед приготовлением попкорна каждый раз программировать микроволновую печь заново.

Одной из первых форм ПЗУ была карта Холлерита (Hollerith card), которая позже стала известна как *перфокарта*, показанная на рис. 3.25. Биты обозначались

¹ Read-only memory — в переводе с английского «память только для чтения». — *Примеч. ред.*

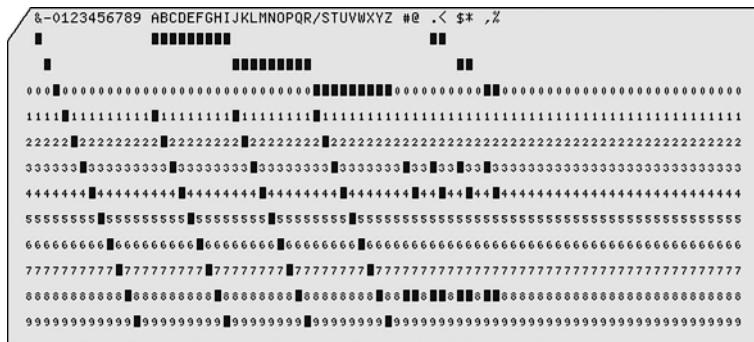


Рис. 3.25. Перфокарта

проколами на бумаге. Да, именно так! Они были довольно дешевыми, потому что американский изобретатель Герман Холлерит (1860–1929) умел экономить на качестве. Холлерит изобрел карту в конце XIX века, хотя было бы точнее сказать, что он позаимствовал идею жаккардового ткацкого станка, изобретенного Жозефом Мари Жаккардом в 1801 году. В жаккардовом ткацком станке для управления узором ткачества использовались перфокарты. Конечно, Жаккард позаимствовал эту идею у Базиля Бушона, который в 1725 году изобрел ткацкий станок с перфолентой. Иногда бывает трудно отличить изобретение от присвоения, потому что будущее строится на прошлом. Помните об этом, когда слышите, как люди выступают за более длительные и строгие законы о патентах и авторском праве; прогресс замедлится, если мы не сможем опираться на прежние изобретения.

Ранние устройства для чтения перфокарт использовали переключатели, чтобы считывать биты. Карты просовывались под ряд пружинящих проводов, которые проходили через отверстия и соприкасались с куском металла с обратной стороны. Более поздние версии, направлявшие свет через отверстия на ряд фотоприемников на обратной стороне, работали значительно быстрее.

Перфорированная бумажная лента — технология, родственная ПЗУ; рулоны бумажной ленты с пробитыми в ней отверстиями использовались для обозначения битов (рис. 3.26). Лента имела преимущество перед карточками в том, что в случае падения колоды карт данные искажались. В то же время лента могла порваться, и ее было трудно ремонтировать; во время ремонта данные на ленте могли быть повреждены.

Карты и лента работали очень медленно, потому что их нужно было физически перемещать, чтобы прочитать.

Вариант ПЗУ под названием «*веревочная ферритовая матрица*» использовался в бортовом компьютере Apollo (рис. 3.27). Поскольку такую схему можно было записать только с помощью шитья, память была невосприимчива к помехам, что важно в суровых условиях космоса.

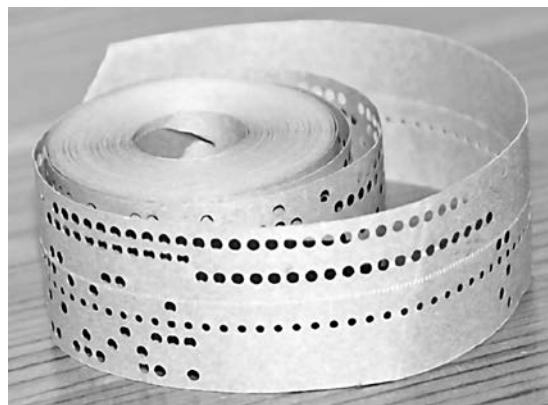


Рис. 3.26. Перфорированная бумажная лента

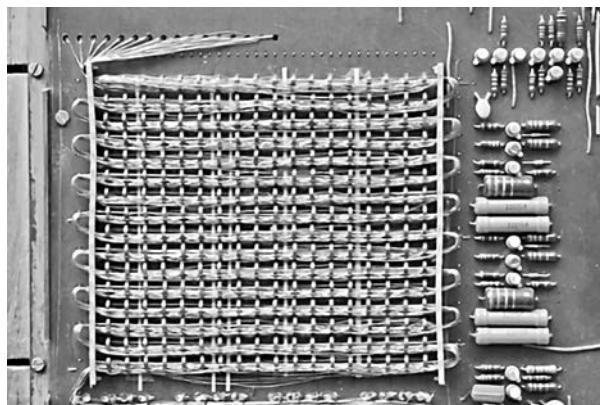


Рис. 3.27. Веревочная ферритовая матрица на управляющем компьютере Apollo

Перфокарты и бумажная лента представляли собой *последовательную* память — данные считывались по порядку. Считывающее устройство не могло работать в обратном порядке, поэтому такая память действительно годилась только для длительного хранения данных. Прежде чем использовать содержимое, его нужно было прочитать в оперативную память. Первое коммерческое применение однокристального микропроцессора Intel 4004 в 1971 году вызвало потребность в более совершенной технологии хранения программ. Первые микропроцессоры использовались для таких устройств, как калькуляторы, которые запускали фиксированную программу. Им на смену пришла *масочная ПЗУ*. Маска — это трафарет, используемый в процессе изготовления интегральной схемы. Написав программу, вы отправили бы битовую комбинацию производителю полупроводников, сопроводив ее немаленьким чеком. Далее эту комбинацию превратили бы в маску, а вы бы получили микросхему, содержащую программу. Такая

микросхема была доступна только для чтения, потому что не было возможности изменить ее, не выписав еще один большой чек и не сделав другую маску. Масочную ПЗУ можно было читать в произвольном порядке.

Маски были настолько дорогими, что их использование можно было оправдать только для массового применения. Появились *программируемые ПЗУ* (ППЗУ, programmable read-only memory, PROM) — микросхемы ПЗУ, которые можно было программировать самостоятельно, но только один раз. Первоначальный механизм ППЗУ включал плавку никрома (никель-хромового сплава) на кристалле. Никром — это то же самое, из чего сделаны светящиеся провода в тостере.

При разработке программ использовались бы огромные кучи микросхем ППЗУ. Инженеры не потерпели таких неудобств, поэтому ППЗУ сменила *стираемая ППЗУ* (erasable programmable read-only memory, EEPROM). Эти микросхемы были похожи на ППЗУ, за исключением того, что наверху у них было кварцевое окошко и их можно было стереть, поместив под специальный ультрафиолетовый свет.

Жизнь стала проще с появлением *электрически стираемой ППЗУ* (что за набор слов!), или *EEPROM* (electrically erasable programmable read-only memory). Это микросхема ППЗУ, которую можно стереть электрически — без света и кварцевого окна. Однако стирание EEPROM происходит сравнительно медленно, так что делать этого не стоит. EEPROM технически представляет собой ОЗУ, так как ее содержимое можно читать и записывать в любом порядке. Но поскольку данные в EEPROM записываются медленно и дороже, чем в ОЗУ, они используются только вместо ПЗУ.

Блочные устройства

Чтобы обратиться к памяти, нужно время. Представьте, что вам пришлось бы ходить в магазин каждый раз, когда понадобится стакан муки. Гораздо практичнее сходить в магазин один раз и принести домой целый пакет. В более крупных устройствах памяти используется именно этот принцип. Представьте себе оптовые закупки битов.

Дисковые накопители, также известные как *носители информации*, отлично подходят для хранения огромных объемов данных. На момент написания этой книги диск на 8 Тбайт стоил менее 200 долларов. Дисковые накопители часто называют просто *носителями информации*. Дисководы хранят биты на вращающихся магнитных пластинах, как закуски на вращающихся подносах. Закуски (биты) периодически перемещаются к вашему месту за столом, и вы можете дотянуться до них рукой. В дисководе вместо руки используется *головка диска*.

Дисковые накопители относительно медленны по сравнению с другими типами памяти. Если вы хотите снова прочитать только что найденную головкой часть,

вам придется подождать почти целый оборот, пока головка не найдет то же место. Современные диски вращаются со скоростью 7200 оборотов в минуту (об/мин), это означает, что вращение занимает чуть больше 8 миллисекунд. Большой минус дисковых накопителей заключается в том, что они механические и постепенно изнашиваются. Износ подшипников — одна из основных причин выхода диска из строя. Разница между коммерческими и потребительскими устройствами заключается прежде всего в количестве смазки в подшипнике: производители могут брать сотни долларов за то, что стоит меньше копейки. Дисковые накопители хранят данные, намагничивая области на диске, что делает их энергонезависимыми, как память на магнитных сердечниках.

Дисковые накопители — это компромисс между скоростью и плотностью записи. Они медленны из-за того, что битам нужно время, прежде чем они дойдут до головки. Однако, поскольку данные перемещаются прямо под головку, нам не требуется место для хранения информации об адресах и связях между данными, в отличие, например, от DRAM. На рис. 3.28 показаны внутренние части жесткого диска. Они заключены в герметичные контейнеры, потому что пыль и грязь могут привести к их поломке.



Рис. 3.28. Жесткий диск

Диски имеют блочную, а не байтовую адресацию. Блок (исторически называемый *сектором*) — это наименьшая единица, к которой можно получить доступ. Исторически диски имели секторы по 512 байт, хотя новые устройства разделяются на секторы по 4096 байт. Это означает, что для изменения одного байта на диске придется прочитать весь блок, изменить один байт и перезаписать весь блок. Диски содержат одну или несколько *пластин*, расположенных, как показано на рис. 3.29.

Поскольку все секторы содержат одинаковое количество битов, *плотность битов* (бит/мм²) в центре каждой пластины больше, чем на внешнем крае. Это

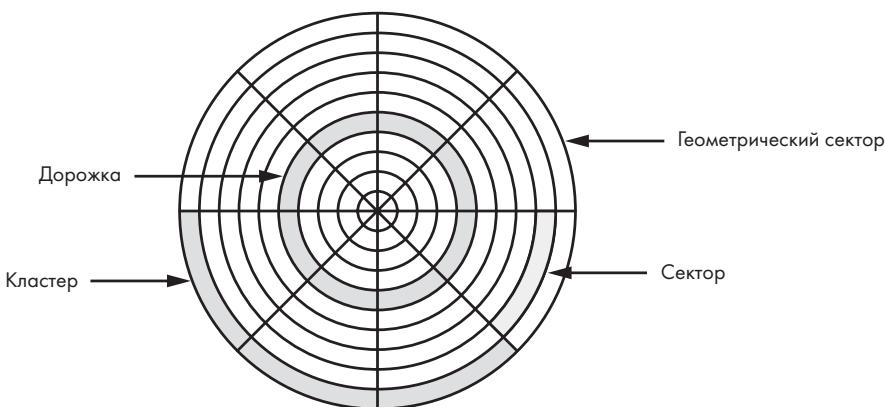


Рис. 3.29. Структура диска

неэкономно, потому что на внешних дорожках явно есть место, чтобы втиснуть еще больше битов. Новые диски решают эту проблему, разделяя диск на набор *радиальных зон* — фактически они имеют больше секторов во внешних зонах, чем во внутренних.

Производительность дисковых накопителей описывается некоторыми цифрами. Современные диски имеют головку на приводном рычаге, который движется радиально по диску; положение головки разделяет диски на дорожки. *Время поиска* — это время, необходимое для перемещения головки с одной дорожки на другую. Конечно, наличие одной головки на каждую дорожку ускорило бы поиск; такой способ подошел бы для очень старых дисков, но на современных дисках дорожки расположены слишком близко друг к другу, чтобы это можно было реализовать. Помимо времени поиска, указывается время, необходимое для поворота нужной части диска таким образом, чтобы она оказалась под головкой, — это называется *задержкой вращения*, которая, как мы видели выше, измеряется в миллисекундах.

Дисковые накопители часто называют *жесткими дисками*. Изначально все дисковые накопители были жесткими. Различие возникло, когда на сцене появились дешевые съемные устройства хранения данных под названием «дискеты». Они были гибкими, поэтому другой тип и назвали «жестким», чтобы их можно было легко отличить друг от друга.

Появился еще накопитель на *магнитных барабанах* — устаревшая разновидность дисковых накопителей, которая работала согласно описанию: вращающийся магнитный барабан с полосами головок.

Магнитная лента — еще одна технология энергонезависимого хранения, в которой используются катушки намагниченной ленты. Она намного медленнее, чем дисковод, и на то, чтобы перемотать ленту в требуемое положение, может

понадобиться много времени. В ранних компьютерах Apple для хранения магнитной ленты использовались аудиокассеты потребительского уровня.

Оптические диски похожи на магнитные, за исключением того, что они используют свет вместо магнетизма. Они знакомы вам как CD и DVD. Большим преимуществом оптических дисков является то, что их можно массово выпускать посредством печати. Предварительно отпечатанные диски — это ПЗУ. Версии, эквивалентные ППЗУ, которые можно записать один раз (CD-R, DVD-R), также доступны, как и версии, которые можно стирать и перезаписывать (CD-RW). На рис. 3.30 показан крупный план части оптического диска.

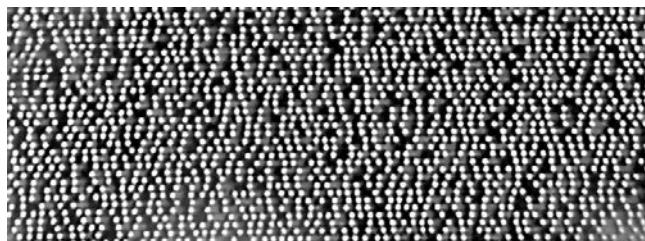


Рис. 3.30. Данные на оптическом диске

Флеш-память и твердотельные диски

Флеш-память — самое последнее воплощение EEPROM. Она прекрасно подходит для некоторых случаев, например для музыкальных плееров и цифровых фотоаппаратов. Принцип работы флеш-памяти основан на хранении электронов в емкостях, подобно DRAM. В этом случае емкости имеют больший размер и построены так, чтобы не протекать. Но петли на крышках емкостей со временем изнашиваются, если их открывать и закрывать слишком много раз. Флеш-память стирается быстрее, чем EEPROM, но ее изготовление дешевле. Она работает как ОЗУ для чтения, а также для записи пустого устройства, заполненного нулями. Но хотя нули можно превратить в единицы, их нельзя обратить, предварительно не стирая. Флеш-память внутренне разделена на блоки, и стирать можно только блоки, а не отдельные места в памяти. Устройства флеш-памяти имеют произвольный доступ для чтения и блочный доступ для записи.

Дисковые накопители постепенно заменяются *твердотельными накопителями*, которые в значительной степени представляют собой просто флеш-память, упакованную так, чтобы выглядеть как дисковый накопитель. Сейчас их цена за бит намного выше, чем у вращающихся дисков, но это должно вскоре измениться. Поскольку флеш-память изнашивается, твердотельные накопители включают в себя процессор, который отслеживает использование различных блоков и пытается выровнять их так, чтобы все блоки изнашивались с одинаковой скоростью.

Обнаружение и исправление ошибок

Вы никогда не предугадаете, когда случайный космический луч поразит часть памяти и испортит данные. Было бы неплохо узнать, когда это произойдет, и еще лучше — иметь возможность устраниить повреждения. Конечно, такие улучшения стоят денег и обычно не встречаются в устройствах потребительского уровня.

Мы хотим иметь возможность обнаруживать ошибки без необходимости хранить вторую полную копию данных. Но даже такой подход не сработает, потому что мы не можем узнать, какая из копий верная. Можно сохранить две дополнительные копии и предположить, что соответствующая пара (если она есть) верна. Так делают компьютеры, предназначенные для работы в очень суровых условиях. Они также используют более дорогую схему, которая не сгорает при попадании в нее протона. Например, космический шаттл имел резервные компьютеры и систему автоматического выбора в случае обнаружения ошибки.

Мы можем проверить наличие 1-битной ошибки, используя метод под названием «четность». Идея в том, чтобы сложить количество битов, для которых установлено значение 1, и использовать дополнительный бит для хранения информации о четности или нечетности этой суммы. Можно сделать это, вычислив исключающее ИЛИ битов. Есть две формы проверки: при *положительной четности* используется сумма битов, а при *нечетной* — дополнение к сумме битов. Этот выбор может показаться странным, но его номенклатура основана на количестве единиц или нулей, включая бит четности.

Левая половина рис. 3.31 показывает расчет положительной четности; мы получили четыре единицы, поэтому четность равна 0. Правая половина показывает проверку четности; 0 означает, что данные хороши или, по крайней мере, настолько хороши, насколько можно судить по четности. Большой минус четности заключается в том, что две ошибки наверняка будут выглядеть как правильные данные; этот метод выявляет только нечетное количество ошибок.

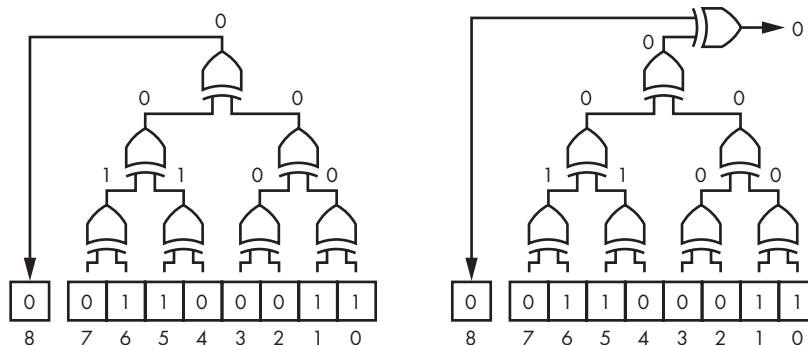


Рис. 3.31. Генерация и проверка четности

Существуют более сложные методы, такие как коды Хэмминга, изобретенные американским математиком Ричардом Хэммингом (Richard Hamming) (1915–1998). Они занимают больше битов, но позволяют обнаруживать больше ошибок и исправлять некоторые из них. Доступны микросхемы памяти для *проверки и исправления ошибок* (error checking and correcting, ECC), которые используют данный подход. Обычно они используются в крупных центрах обработки данных, а не в потребительских устройствах.

Методы наподобие четности хороши для постоянно изменяющихся данных. Существуют менее дорогие методы, позволяющие проверять данные в статических блоках — например, компьютерных программах. Самым простым из них является *контрольная сумма*, где содержимое каждой ячейки данных суммируется в некоторое n -битное значение, а биты переполнения отбрасываются. Контрольную сумму можно сравнить с программой, обычно непосредственно перед ее запуском. Чем больше значение контрольной суммы (то есть больше n), тем меньше вероятность получения ложного срабатывания.

Циклический избыточный код (cyclic redundancy check, CRC) — математически более верная замена контрольных сумм. Еще один вариант — хеш-коды. Их цель — вычислить проверочный номер, который достаточно уникален для данных, чтобы при большинстве изменений проверка могла выявить несоответствие.

Аппаратное и программное обеспечение

Методы, используемые для создания ППЗУ, EEPROM и флеш-памяти, не ограничиваются памятью. Скоро мы увидим, как из логических схем создается компьютерное аппаратное обеспечение. А поскольку вы изучаете программирование, то знаете, что программы включают логику в код, и вы можете знать, что компьютеры предоставляют логику программам через свои наборы команд. В чем разница между аппаратным и программным обеспечением? Граница нечеткая. По большому счету, различий между ними мало, за исключением того, что создавать программное обеспечение гораздо проще, поскольку это не требует дополнительных затрат, кроме времени разработки.

Вы, наверное, слышали термин «прошивка», который изначально относился к программному обеспечению в ПЗУ. Но большая часть прошивок теперь живет во флеш- или даже в оперативной памяти, поэтому разница между обычным ПО и прошивкой минимальна. На самом деле все еще сложнее. Раньше микросхемы разрабатывались компьютерными гиками, которые строили схемы, наклеивая цветную малярную ленту на большие листы прозрачной майларовой пленки. В 1979 году американские ученые и инженеры Карвер Мид (Carver Mead) и Линн Конвей (Lynn Conway) изменили мир своей публикацией «Introduction to VLSI Systems», которая дала толчок развитию индустрии автоматизации

электронного проектирования (electronic design automation, EDA). Проектирование микросхем превратилось в проектирование программного обеспечения. Сегодня микросхемы разрабатываются с использованием специализированных языков программирования, таких как Verilog, VHDL и SystemC.

В большинстве случаев программисту просто предоставляют аппаратное обеспечение. Но вы можете получить возможность поучаствовать в разработке системы, включающей как аппаратное, так и программное обеспечение. Проектирование интерфейса между аппаратным и программным обеспечением имеет решающее значение. Существует бесчисленное множество примеров микросхем с непривычными для использования, непрограммируемыми и ненужными функциями.

Интегральные схемы дороги в изготовлении. Вначале все микросхемы были *полностью заказными*. Микросхемы собираются слоями: сами компоненты находятся внизу, а металлические слои — вверху, а затем они соединяются. *Вентильные матрицы* были попыткой снизить стоимость для некоторых случаев использования; был доступен набор предварительно спроектированных компонентов, и только металлические слои проектировались по заказу. Как и в случае с памятью, они были заменены версиями, эквивалентными ППЗУ, которые можно было программировать самостоятельно. Существовал и эквивалент стираемой ППЗУ, который можно было стереть и перепрограммировать.

Современные, *программируемые пользователем вентильные матрицы* (ППВМ, field-programmable gate array, FPGA) являются эквивалентом флеш-памяти; их можно перепрограммировать с помощью соответствующего ПО. Во многих случаях использование ППВМ дешевле, чем использование других компонентов. ППВМ очень функциональны; например, можно получить большую ППВМ, содержащую пару процессорных ядер ARM. Компания Intel недавно приобрела Altera и теперь может включать ППВМ в микросхемы процессора. Существует ненулевой шанс, что вы будете работать над проектом, содержащим одно из этих устройств, поэтому будьте готовы превратить свое программное обеспечение в аппаратное.

Выводы

В этой главе вы узнали, как у компьютеров появилось чувство времени. Вы познакомились с последовательной логикой, которая, наряду с комбинаторной логикой из главы 2, предоставляет все фундаментальные строительные блоки для аппаратного обеспечения. А еще вы кое-что узнали о том, как устроена память. В главе 4 мы объединим все эти знания и создадим компьютер.

4

Анатомия компьютера



Вы узнали о свойствах битов и способах их использования для представления разных объектов в главе 1. В главах 2 и 3 вы узнали, почему мы используем биты и как они реализованы в аппаратном обеспечении. Вы также рассмотрели ряд основных строительных блоков и то, как их можно объединить в более сложные конфигурации.

В этой главе мы изучим, как объединить эти блоки в схему, которая может управлять битами. Эта схема называется *компьютером*.

Есть много способов собрать компьютер. Вариант, который мы создадим в этой главе, был выбран для удобства объяснения, а не потому, что он лучше всего спроектирован. Конечно, простые компьютеры функционируют, но заставить их работать *хорошо* не так просто из-за множества сложностей. Эта глава посвящена простому компьютеру, а следующие две — некоторым дополнительным сложностям.

В современном компьютере есть три больших элемента. Это *память*, *ввод и вывод* (input and output, I/O) и *центральный процессор* (ЦП). В этом разделе рассказывается, как эти части соотносятся друг с другом. Память представлена в главе 3, а в главе 5 компьютеры и память рассматриваются более подробно. Ввод/вывод является предметом главы 6. ЦП находится в так называемом «центре города» в этой главе.

Память

Компьютерам нужно место для хранения битов, которыми они управляют. Это место — память, как вы узнали из главы 3. Теперь пора выяснить, как компьютеры ее используют.

Память похожа на длинную улицу, полную домов. Каждый дом абсолютно одинакового размера, и в нем есть место для определенного количества битов. Строительные нормы в значительной степени ограничились одним байтом на дом. И, как и на настоящей улице, у каждого дома есть *адрес*, который представляет собой просто номер. Если на вашем компьютере 64 МиБ памяти, это $64 \times 1024 \times 1024 = 67\,108\,864$ байта (или 536 870 912 бит). Байты имеют адреса от 0 до 67 108 863. Такая нумерация имеет смысл, в отличие от нумерации на многих улицах в реальной жизни.

Довольно часто ссылаются на *ячейку* памяти, которая представляет собой просто память по определенному адресу, например «улица Памяти, 3» (рис. 4.1).

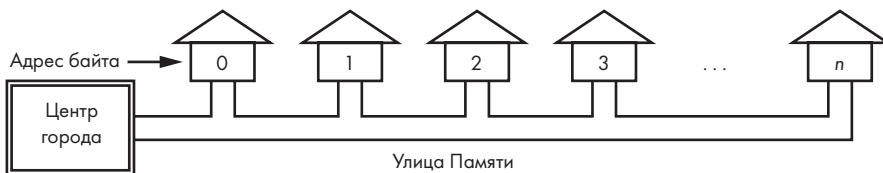


Рис. 4.1. Улица памяти

Тот факт, что основной единицей памяти является байт, не означает, что мы всегда рассматриваем память сквозь призму байтов. Например, 32-разрядные компьютеры обычно организуют память в виде блоков по 4 байта, в то время как 64-разрядные компьютеры – в виде блоков по 8 байт. Почему это имеет значение? Объединение превращает однополосные улицы в четырех- или восьмиполосные шоссе. Больше полос может обрабатывать больше трафика, потому что больше битов может попасть в шину данных. Когда мы обращаемся к памяти, нам нужно знать, к чему именно мы обращаемся. Адресация длинных слов отличается от адресации байтов, поскольку длинное слово на 32-битном компьютере составляет 4 байта, а на 64-битном компьютере – 8 байт. На рис. 4.2, например, адрес длинного слова 1 содержит байтовые адреса 4, 5, 6 и 7.

Я попытаюсь объяснить иначе: улица в 32-битном компьютере содержит четыре комплекса, а не отдельные дома, и каждый четырехэтажный дом содержит два дуплекса. Это означает, что можно назвать адрес отдельной единицы, дуплекса или всего здания.

Вы, возможно, заметили, что все здания стоят на шоссе, так что каждый байт движется по собственной полосе, а длинное слово занимает всю дорогу. Биты ездят в центр города и обратно нашине (автобусе) с четырьмя сиденьями, по одному на каждый байт. Двери расположены так, чтобы на каждую полосу приходилось по одному месту. На большинстве современных компьютеров шина останавливается только у одного здания для каждой поездки из центра города. Это означает, что нельзя, например, сформировать длинное слово из байтов 5, 6, 7 и 8, потому что в таком случае шине придется совершить две поездки: в здание 0 и здание 1.

В старых компьютерах была сложная загрузочная док-станция, которая позволяла это делать, но планировщики заметили, что она не так уж и полезна, и поэтому вырезали ее из бюджета на новые модели. Попытка попасть в два здания одновременно, как показано на рис. 4.3, называется *невыровненным доступом*.

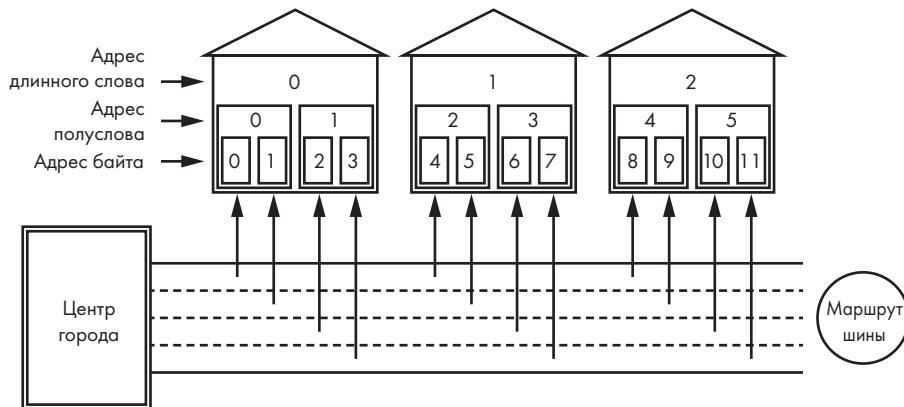


Рис. 4.2. Магистраль памяти

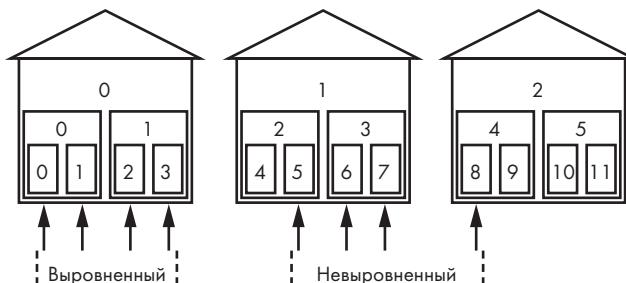


Рис. 4.3. Выровненный и невыровненный доступ

Как мы узнали из предыдущей главы, существует множество различных видов памяти с разным соотношением цены/качества. Например, SRAM быстрая и дорогая, как автомагистрали, рядом с которыми живут политики. Диск дешевый и медленный — как грунтовая дорога.

Кто и на каком сиденье сидит в шине? Может ли байт 0 или байт 3 занять крайнее левое место, когда в город направляется длинное слово? Это зависит от используемого процессора, потому что проектировщики воплотили в жизнь оба способа. Оба работают, так что это в значительной степени теологический спор. Фактически термин «конечный» (endian; по аналогии с названиями партий остроконечников и тупоконечников из романа Джонатана Свифта «Путешествие Гулливера», которые спорили о том, с какого конца лучше всего разбивать сваренное всмятку яйцо) используется для описания разницы.

Байт 0 занимает крайнее правое место в машинах с прямым порядком байтов (little-endian), таких как процессоры Intel. Он же занимает крайнее левое место в машинах с обратным порядком байтов (big-endian) — например, в процессорах Motorola. На рис. 4.4 сравниваются две схемы.



Рис. 4.4. Размещение при прямом и обратном порядке байтов

При передаче информации с одного устройства на другое следует помнить о порядке байтов — вы же не хотите случайно перемешать данные. Такое уже случалось, когда операционная система UNIX была перенесена с PDP-11 на компьютер IBM Series/1. Программа, которая должна была выводить «Unix», вместо этого выводила «pUxi», поскольку байты в 16-битных словах поменялись местами. Это было довольно забавно, поэтому в обиход вошел термин «синдром nuxi» для обозначения проблем с порядком следования байтов.

Ввод и вывод

Компьютер, который не может взаимодействовать с внешним миром, не очень полезен. Нам нужен способ вводить в компьютер данные и выводить их из него. Это называется *вводом/выводом*, или *I/O* (input/output). Устройства, которые подключаются ко вводу/выводу, называются *устройствами ввода/вывода*. Поскольку они находятся на периферии компьютера, их также часто называют *периферийными устройствами*, или просто *периферией*.

Раньше у компьютеров был отдельный канал ввода/вывода, как показано на рис. 4.5, который был похож на «улицу Памяти». Это имело смысл, когда компьютеры были огромных размеров, потому что они не помещались в маленькие корпуса с ограниченным количеством электрических соединений. Кроме того, «улица Памяти» была не очень длинной, поэтому не имело смысла ограничивать количество адресов только для поддержки ввода-вывода.



Рис. 4.5. Отдельные шины памяти и ввода/вывода

Теперь «Улица Памяти» намного длиннее, как в 32-, так и в 64-разрядных компьютерах. Она настолько длинная, что существуют адреса без домов — доступно много пустых участков. Другими словами, есть адреса, с которыми не

связана память. В результате имеет смысл выделить часть «улицы Памяти» для устройств ввода/вывода. Похоже на промышленный район на окраине города. Кроме того, поскольку в связку, имеющую ограниченное количество соединений, втиснуто как можно больше схем, логично, чтобы ввод/вывод находился на той же шине, что и память.

Многие компьютеры имеют стандартные *слоты* ввода-вывода, поэтому периферийные устройства можно подключать к ним единообразно. Это делается примерно так же, как распределялась собственность на Диком Западе; территория, не имеющая статуса городской, разделялась на набор земельных участков, как показано на рис. 4.6. Каждый держатель слота может использовать все адреса в пределах границ слота. Часто в каждом слоте есть определенный адрес, который содержит какой-то идентификатор, так что центр города может провести перепись, чтобы определить, кто проживает в каждом слоте.

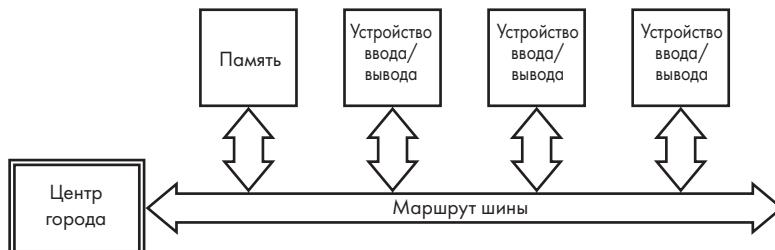


Рис. 4.6. Общая память и шина ввода/вывода

Мы часто используем метафору из судоходства и говорим, что все пришвартованы к *портам ввода/вывода*.

Центральный процессор

Центральный процессор (ЦП) — это часть компьютера, которая выполняет фактические вычисления. По нашей аналогии, он живет в центре города. Все остальное — актеры второго плана. ЦП состоит из множества отдельных частей, о которых мы узнаем в этом разделе.

Арифметико-логическое устройство

Арифметико-логическое устройство (АЛУ) — одна из основных частей ЦП. Оно умеет выполнять арифметические операции, операции булевой алгебры и др. На рис. 4.7 показана простая схема АЛУ.

Операнды — это просто биты, которые могут представлять числа. *Код операции* — это число, которое сообщает АЛУ, какой *оператор* применять

к операндам. *Результат*, конечно же, получается, когда мы применяем оператор к операндам.

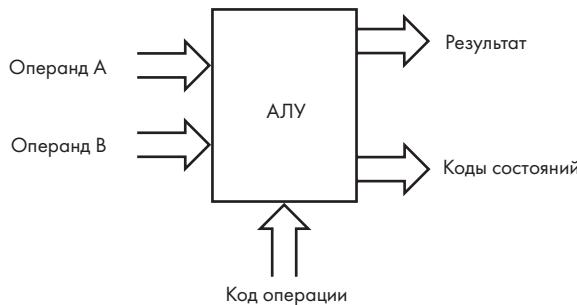


Рис. 4.7. Пример АЛУ

Коды состояний содержат дополнительную информацию о результате. Обычно они хранятся в *регистре кодов состояния*. Регистр, который мы упоминали еще в главе 3, — это просто особая часть памяти, которая располагается отдельно от других — на улице с дорогими домами нетиповой застройки. Типичный регистр кода состояния показан на рис. 4.8. Цифры над прямоугольниками — это номера битов, что удобно для их обозначения. Обратите внимание, что некоторые биты не используются; в этом нет ничего необычного.



Рис. 4.8. Регистр кода состояния

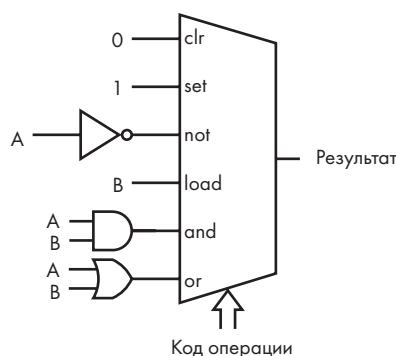
N устанавливается в 1, если результат последней операции — отрицательное число. Бит *Z* устанавливается в 1, если этот результат равен 0. Бит *O* устанавливается в 1, если результат последней операции привел к переполнению или потере значимости.

Таблица 4.1 показывает, что может делать АЛУ.

АЛУ может показаться странным, но на самом деле оно представляет собой лишь несколько логических элементов, управляющих селектором, который мы уже рассматривали. На рис. 4.9 показана общая конструкция АЛУ (для простоты мы убрали некоторые более сложные функции).

Таблица 4.1. Примеры кодов операций АЛУ

Код операции	Мнемокод	Описание
0000	clr	Игнорировать операнды; сделать каждый бит результата равным 0 (очистить)
0001	set	Игнорировать операнды; сделать каждый бит результата равным 1
0010	not	Игнорировать B; превратить нули из A в 1 и наоборот
0011	neg	Игнорировать B; получить в результате дополнение до двух для A, $-A$
0100	shl	Сдвиг A влево на младшие 4 бита B (см. следующий раздел)
0101	shr	Сдвиг A вправо на младшие 4 бита B (см. следующий раздел)
0110		Не используется
0111		Не используется
1000	load	Передать операнд B в результат
1001	and	Результат равен A И B для каждого бита операндов
1010	or	Результат равен A ИЛИ B для каждого бита операндов
1011	xor	Результат равен A исключающее ИЛИ B для каждого бита операндов
1100	add	Результат равен A + B
1101	sub	Результат равен A – B
1110	cmp	Установить коды состояний на основе B – A (сравнить)
1111		Не используется

**Рис. 4.9.** Часть внутреннего устройства АЛУ

Сдвиг

Возможно, вы заметили операции сдвига в табл. 4.1. Сдвиг влево перемещает каждый бит на одну позицию влево, отбрасывая крайний левый бит и перемещая 0 в освободившуюся крайнюю правую позицию. Если сдвинуть 01101001 (105_{10}) на 1 влево, мы получим 11010010 (210_{10}). Это очень удобно, потому что сдвиг влево позиции номер один умножает ее на 2.

Сдвиг вправо перемещает каждый бит вправо на одну позицию, отбрасывая крайний правый бит и перемещая 0 в освободившееся крайнее левое положение. Если сдвинуть 01101001 (105_{10}) на 1 вправо, мы получим 00110100 (52_{10}). Это эквивалентно делению числа на 2 и отбрасыванию остатка.

Значение MSB (наибольшего значащего бита), потерянного при сдвиге влево, или LSB (наименьшего значащего бита) — при сдвиге вправо, часто требуется сохранить, для этого и нужен регистр кода состояния. Представим, что наш ЦП сохраняет его в бите O .

Вы могли заметить, что, похоже, все элементы АЛУ могут быть реализованы с помощью комбинаторной логики, за исключением этих инструкций сдвига. Можно создать *регистры сдвига* из триггеров, где содержимое сдвигается на одну битовую позицию за такт.

Последовательный регистр сдвига (показанный на рис. 4.10) работает медленно, потому что в худшем случае ему требуется один такт на бит.

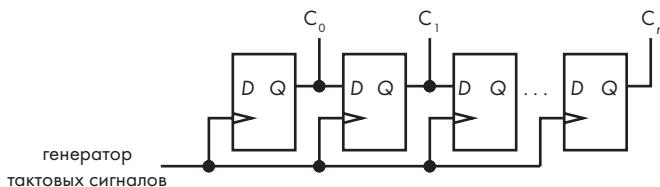


Рис. 4.10. Последовательный регистр сдвига

Можно решить эту проблему, создав *устройство быстрого (циклического) сдвига* полностью на комбинаторной логике, используя один из ранее рассмотренных логических строительных блоков — селектор (см. рис. 2.47). Чтобы построить 8-битное устройство циклического сдвига, нам понадобится восемь селекторов 8 : 1.

На каждый бит имеется один селектор, как показано на рис. 4.11.

Величина сдвига вправо указана на $S_{0:2}$. Видим, что без сдвига (000 для S) входной бит 0 (I_0) передается в выходной бит 0 (O_0), I_1 в O_1 и т. д. Когда S равно 001, выходы сдвигаются вправо на единицу, потому что таким образом входы

подключаются к селектору. Когда S равно 010, выходы сдвигаются вправо на два и т. д. Другими словами, мы имеем все восемь возможностей и просто выбираем необходимую.

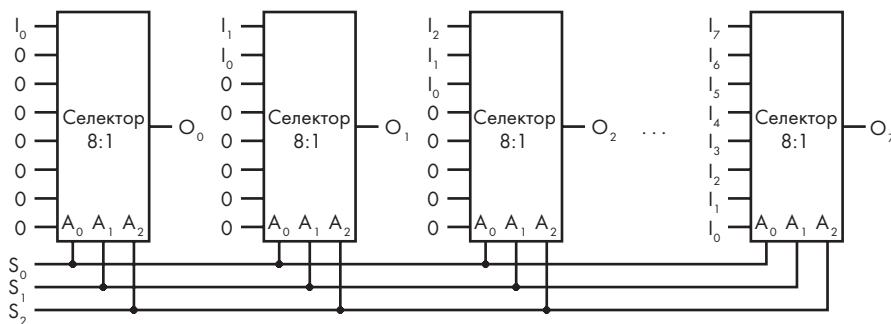


Рис. 4.11. Комбинаторное многорегистровое устройство циклического сдвига

Вам может быть интересно, почему я продолжаю показывать эти логические схемы, как будто они построены из старых деталей серии 7400. Такие функции, как вентили, селекторы, демультиплексоры, сумматоры, триггеры и т. д., доступны как предопределенные компоненты в системах проектирования интегральных схем. Они используются так же, как и старые компоненты, за исключением того, что вместо размещения множества деталей серии 7400, о которых я упоминал в главе 2, на плате теперь мы собираем аналогичные компоненты в одну микросхему с помощью проектировочного программного обеспечения.

Возможно, вы заметили, что в нашем простом АЛУ нет операций умножения и деления. Это потому, что они намного сложнее, но на самом деле не показывают нам ничего нового. Вы знаете, что умножение можно производить повторным сложением — это его последовательная версия. Можно также построить комбинаторный множитель путем каскадного наслаждения устройств циклического сдвига и сумматоров, имея в виду, что сдвиг влево умножает число на 2.

Устройства сдвига – ключевой элемент в реализации арифметики с плавающей точкой; экспоненты используются для сдвига мантисс, чтобы выровнять двоичные точки, – тогда числа можно будет складывать, вычитать и т. д.

Исполнительное устройство

Исполнительное устройство компьютера, также известное как *управляющий блок*, — вот настоящий начальник. В конце концов, от АЛУ мало пользы — ему нужно указывать, что делать. Исполнительное устройство захватывает коды операций и операнды из нужных мест в памяти, сообщает АЛУ, какие операции

выполнять, и помещает результаты обратно в память. Надеюсь, он делает все это в порядке, который служит полезной цели. (Кстати, мы используем значение «выполнить задачу» для термина «execute» («выполнить»). На самом деле мы не убиваем биты¹.)

Как исполнительное устройство может это сделать? Мы даем ему список инструкций, таких как «добавьте число из ячейки 10 к числу из ячейки 12 и поместите результат в ячейку 14». Где исполнительный модуль находит эти инструкции? В памяти! Техническое название того, что мы используем, — *компьютер с хранимой программой*. Он был создан благодаря работе гениального английского ученого Алана Тьюринга (1912–1954).

Да, мы получили еще один способ рассматривать и интерпретировать биты. *Инструкции* — это битовые комбинации, которые говорят компьютеру, что делать. Битовые комбинации являются частью конструкции конкретного процессора. Они не имеют общего стандарта, например представления с помощью чисел, поэтому процессор Intel Core i7, вероятно, будет иметь другую битовую комбинацию для инструкции `inc A`, чем процессор ARM Cortex-A.

Как исполнительное устройство узнает, где искать инструкцию в памяти? Оно использует *счетчик команд* (часто сокращается как PC, от program counter), который похож на почтальона или на большую стрелку с надписью «Вы здесь». Как показано на рис. 4.12, счетчик команд — это еще один регистр, одна из тех частей памяти, которые располагаются на особой «улице». Он состоит из счетчика (см. «Счетчики» на с. 119), а не простого регистра (см. «Регистры» на с. 121). Счетчик можно рассматривать как регистр с дополнительной функцией подсчета.

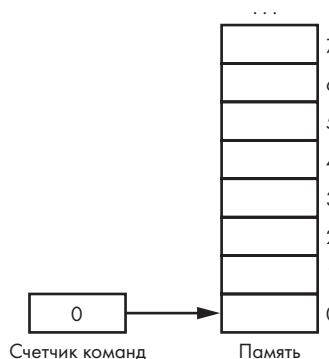


Рис. 4.12. Счетчик команд

¹ Глагол «execute» также переводится как « казнить ». — Примеч. пер.

Счетчик команд содержит адрес памяти. Другими словами, он указывает, или *ссылается*, на место в памяти. Исполнительное устройство выбирает инструкцию из места, на которое указывает счетчик команд. Существуют специальные инструкции, которые изменяют значение этого счетчика, — вскоре мы их увидим. Если ни одна из таких инструкций не выполняется, счетчик команд *увеличивается* (к нему добавляется размер одной инструкции) после выполнения каждой инструкции, так что следующая инструкция будет поступать из следующей ячейки памяти. Обратите внимание, что при включении питания ЦП получают некоторое начальное значение счетчика программ, обычно 0. Счетчик, который мы видели на рис. 3.17, имеет входы для поддержки всех этих функций.

Все это работает как охота за сокровищами. Компьютер переходит в определенное место в памяти и находит записку. Он читает эту записку, в которой говорится, что нужно что-то сделать, а затем переходит в другое место, чтобы получить следующую записку, и т. д.

Набор инструкций

Записки, которые компьютеры находят в памяти во время поиска сокровищ, называются *инструкциями*. В этом разделе рассказывается о том, что в них содержится.

Инструкции

Чтобы увидеть, какие инструкции можно найти в ЦП и как для них подбираются битовые комбинации, предположим, что компьютер использует 16-битные инструкции.

Попробуем разделить инструкцию на четыре поля — код операции, адреса для двух операндов и результат, — как показано на рис. 4.13.

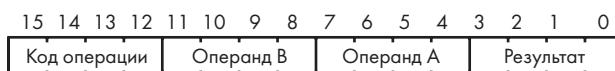


Рис. 4.13. Схема трехадресной инструкции

Это выглядит хорошей идеей, но на практике не все так гладко. Почему? Потому что у нас есть место только для 4 бит адреса для каждого из операндов и результата. Имея 16 адресов, довольно сложно адресовать полезный объем памяти. Можно увеличить размер инструкции, но даже если перейти к 64-битным инструкциям, получится всего 20 бит адреса, занимающих только один мебибайт памяти. Современные компьютеры насчитывают гигабайты памяти.

Другой подход — это повтор трюка с адресацией DRAM, который мы видели на рис. 3.23. Можно иметь *регистр расширения адреса* и загружать его старшими битами адреса с помощью отдельной инструкции. Этот метод использовала Intel, чтобы ее 32-битные компьютеры получали доступ к более чем 4 ГиБ памяти. Intel назвала это PAE — *физическое расширение адреса* (от physical address extension). Конечно, для загрузки этого регистра требуется дополнительное время и много повторных загрузок регистров, если нам нужна память по обе стороны от границы, созданной таким образом.

Однако есть еще более важная причина, по которой трехадресный формат не работает: он рассчитывает на некую волшебную, несуществующую форму памяти, которая позволяет одновременно обращаться к трем разным адресам. Все три блока памяти на рис. 4.14 представляют собой одно и то же устройство памяти; не существует трех адресных шин и трех шин данных.

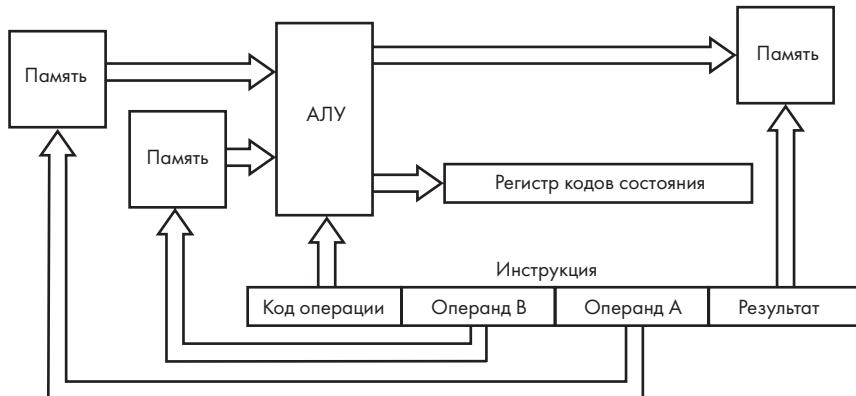


Рис. 4.14. Неработоспособная компьютерная архитектура

Эта архитектура будет работать, если один регистр будет хранить содержимое операнда A, а другой — содержимое операнда B. Аппаратному обеспечению нужно в таком случае сделать следующее:

1. Загрузить инструкцию из памяти, используя адрес из счетчика команд.
2. Загрузить регистр операнда A, используя адрес из части инструкции для операнда A.
3. Загрузить регистр операнда B, используя адрес из части инструкции для операнда B.
4. Сохранить результат в памяти, используя адрес из части инструкции для результата.

Получается слишком сложное решение. Если каждый из этих шагов займет один тактовый цикл, потребуются четыре цикла, чтобы что-то сделать. Следует

учесть тот факт, что можно получить доступ только к одной области памяти за раз, и соответствующим образом спроектировать набор инструкций. Больше битов адреса будет доступно, если адресовать лишь что-то одно за раз.

Это можно сделать, добавив еще один дом на регистровую улицу. Назовем этот регистр *аккумулятором*, или для краткости регистром А, и он будет хранить результат, полученный от АЛУ. Вместо того чтобы выполнять операцию между двумя ячейками памяти, будем делать это между одной из ячеек памяти и аккумулятором. Конечно, нужно будет добавить инструкцию *сохранения*, которая помещает содержимое аккумулятора в ячейку памяти. Итак, теперь можно расставить инструкции, как показано на рис. 4.15.

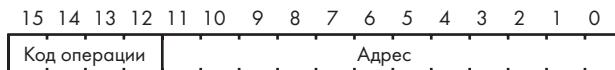


Рис. 4.15. Схема одноадресной инструкции

Это дает больше битов адреса, но для работы требуется еще больше инструкций. Раньше мы использовали инструкцию, в которой говорилось:

$$C = A + B.$$

Вместо этого появились три инструкции:

$$\text{Аккумулятор} = A.$$

$$\text{Аккумулятор} = \text{Аккумулятор} + B.$$

$$C = \text{Аккумулятор}.$$

Вы могли заметить, что мы просто заменили одну инструкцию тремя и в результате противоречим самим себе, потому что это фактически увеличило размер инструкции. Так и есть для этого простого случая, но в целом мы все-таки получаем некоторую выгоду. Допустим, нужно вычислить такую сумму:

$$D = A + B + C.$$

Это невозможно сделать с помощью одной инструкции, даже если она получит доступ к трем адресам, потому что теперь нам требуются четыре адреса. Мы бы выполнили сложение таким образом:

$$\text{Промежуточный результат} = A + B.$$

$$D = \text{Промежуточный результат} + C.$$

Если учитывать 12-битный адрес, нам понадобятся 40-битные инструкции для обработки трех адресов и кода операции. Также нам нужны две из этих

инструкций — 80 бит — для вычисления D . Если использовать одноадресную версию инструкций, достаточно четырех инструкций и 64 бит.

$$\text{Аккумулятор} = A.$$

$$\text{Аккумулятор} = \text{Аккумулятор} + B.$$

$$\text{Аккумулятор} = \text{Аккумулятор} + C.$$

$$D = \text{Аккумулятор}.$$

Режимы адресации

С помощью аккумулятора нам удалось получить 12 бит адреса. Адресовать 4096 байт намного лучше, чем 16, но и этого недостаточно. Этот способ адресации памяти известен как *прямая адресация*, что означает просто адрес, указанный в инструкции.

Можно увеличить объем памяти, добавив *косвенную адресацию*. В этом случае мы получаем адрес из ячейки памяти, содержащейся в инструкции, а не из самой инструкции непосредственно. Например, предположим, что ячейка памяти 12 содержит значение 4321, а ячейка памяти 4321 содержит 345. Если бы мы использовали прямую адресацию, то получили бы 4321 из ячейки 12, а при косвенной адресации — 345, содержимое ячейки 4321.

Это подходит для работы с памятью, но иногда нам просто нужно получить постоянные числа. Например, чтобы сосчитать до 10, нам нужен способ загрузить это число. Мы можем сделать это с помощью еще одного режима адресации, называемого *немедленной адресацией*. При этом адрес обрабатывается просто как число, поэтому в предыдущем примере при загрузке 12 в немедленном режиме мы получим 12. На рис. 4.16 сравниваются эти режимы адресации.

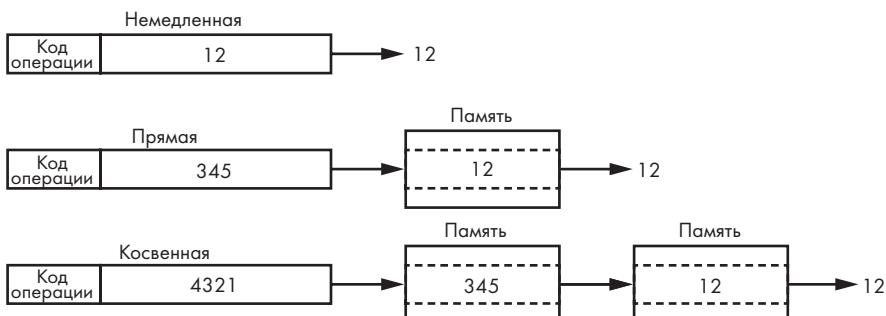


Рис. 4.16. Режимы адресации

По рисунку понятно, что немедленная адресация быстрее, чем прямая, для которой нужен повторный доступ к памяти, а косвенная происходит еще дольше, так как требует дополнительного доступа к памяти.

Инструкции кода состояния

Нашему процессору все еще кое-чего не хватает — например, инструкций, которые работают с кодами состояния. Мы видели, что эти коды устанавливаются путем сложения, вычитания и сравнения. Но нам нужны способы задать для них известные значения и получить новые. Для этого нам пригодятся дополнительные инструкции `ccs` (копирует содержимое регистра кода состояния в аккумулятор) и `acc` (копирует содержимое аккумулятора в регистр кода состояния).

Ветвление

Теперь у нас есть инструкции, которые могут делать все, что угодно, но мы можем только выполнять весь список инструкций от начала до конца. Толку немногого. Нам бы пригодились программы, которые принимают решения и выбирают части кода для выполнения. Они бы получали инструкции, меняющие значение счетчика команд. Такие инструкции называются инструкциями *ветвления* и вызывают загрузку счетчика команд с новым адресом. Ветвление само по себе не полезнее возможности выполнить список инструкций. Однако оно выполняется не всегда — инструкции проверяют коды состояний и выполняются только в том случае, если условия истинны. В противном случае счетчик команд обычно увеличивается на единицу, и следующей выполняется инструкция, идущая за инструкцией ветвления. Инструкциям ветвления нужно дополнительно несколько битов для хранения условия, как показано в табл. 4.2.

Таблица 4.2. Условия инструкции ветвления

Код	Мнемокод	Описание
000	<code>bra</code>	Всегда выполняется
001	<code>bov</code>	Выполняется, если установлен бит кода состояния O (переполнение)
010	<code>beq</code>	Выполняется, если установлен бит кода состояния Z (нуль)
011	<code>bne</code>	Выполняется, если сброшен бит кода состояния Z
100	<code>blt</code>	Выполняется, если установлен бит кода состояния N (отрицательный), а Z сброшен
101	<code>ble</code>	Выполняется, если установлены N или Z
110	<code>bgt</code>	Выполняется, если N и Z сброшены
111	<code>bge</code>	Выполняется, если N сброшен или установлен Z

Иногда требуется явно изменить содержимое счетчика команд. Для этого есть две специальные инструкции: `rcs` (копирует текущее значение счетчика команд в аккумулятор) и `arc` (копирует содержимое аккумулятора в счетчик команд).

Итоговый набор инструкций

Соберем все эти функции в итоговый набор инструкций, как показано на рис. 4.17.

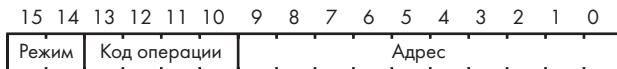


Рис. 4.17. Окончательная схема инструкции

Мы рассмотрели три *режима адресации*, а это значит, что нам нужно 2 бита для выбора режима. Неиспользуемая четвертая битовая комбинация применяется для операций, не связанных с памятью.

Режим адресации и код операции преобразуются в инструкции, как видно из табл. 4.3.

Таблица 4.3. Режимы адресации и коды операций

Код операции	Адресация			
	Прямая (00)	Косвенная (01)	Немедленная (10)	Не указан (11)
0000	load	load	load	
0001	and	and	and	set
0010	or	or	ore	not
0011	xor	xor	xor	neg
0100	add	add	add	shl
0101	sub	sub	sub	shr
0110	cmp	cmp	cmp	acc
0111	store	store	cca	
1000	bra	bra	bra	apc
1001	bov	bov	bov	pca
1010	beq	beq	beq	
1011	bne	bne	bne	
1100	blt	blt	blt	
1101	ble	bge	ble	
1110	bgt	bgt	bgt	
1111	bge	bge	bge	

Обратите внимание, что условия ветвления объединены с кодами операций. Коды операций для режима адресации 3 используются для операций, в которых задействован только аккумулятор. Побочным эффектом полной реализации является то, что коды операций не совсем соответствуют АЛУ, показанному в табл. 4.1. В этом нет ничего необычного — просто требуется дополнительная логика.

Инструкции сдвига влево и вправо используют незанятые биты для подсчета количества сдвигаемых позиций, как показано на рис. 4.18.

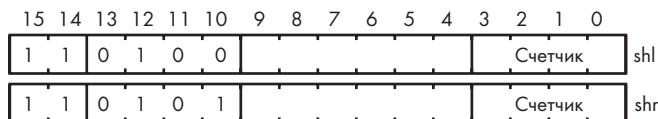


Рис. 4.18. Схема инструкции сдвига

Теперь можно фактически указать компьютеру что-то сделать, написав *программу*, которая представляет собой просто список инструкций, выполняющих некоторую задачу. Вычислим все числа Фибоначчи (итальянский математик, 1175–1250) до 200. Числа Фибоначчи — интересная вещь; количество лепестков на цветках, например, — это тоже числа Фибоначчи. Первые два числа Фибоначчи — это 0 и 1. Следующее число в последовательности получается путем сложения двух предыдущих: 0, 1, 1, 2, 3, 5, 8, 13 и т. д. Процесс подсчета показан на рис. 4.19.

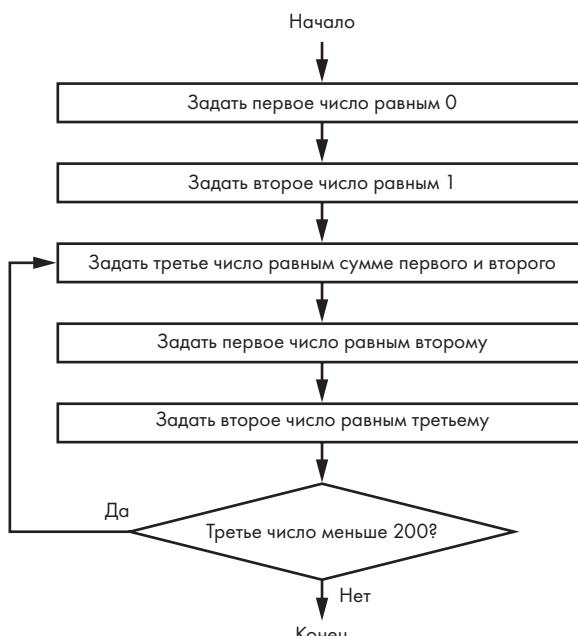


Рис. 4.19. Блок-схема программы для вычисления последовательности Фибоначчи

Короткая программа, представленная в табл. 4.4, реализует этот процесс. Столбец с инструкциями разделен на поля, как показано на рис. 4.17. Адреса в комментариях представляют собой десятичные числа.

Таблица 4.4. Программа на машинном языке для вычисления последовательности Фибоначчи

Адрес	Инструкция	Описание
0000	10 0000 0000000000	Очистить аккумулятор (немедленно загрузить 0)
0001	00 0111 0001100100	Сохранить аккумулятор (0) в ячейке памяти 100
0010	10 0000 0000000001	Загрузить 1 в аккумулятор (немедленно загрузить 1)
0011	00 0111 0001100101	Сохранить аккумулятор (1) в ячейке памяти 101
0100	00 0000 0001100100	Загрузить аккумулятор из ячейки памяти 100
0101	10 0100 0001100101	Добавить содержимое ячейки памяти 101 в аккумулятор
0110	00 0111 0001100110	Сохранить аккумулятор в ячейке памяти 102
0111	00 0000 0001100101	Загрузить аккумулятор из ячейки памяти 101
1000	00 0111 0001100100	Сохранить его в ячейке памяти 100
1001	00 0000 0001100110	Загрузить аккумулятор из ячейки памяти 102
1010	00 0111 0001100101	Сохранить его в ячейке памяти 101
1011	10 0110 0011001000	Сравнить содержимое аккумулятора с числом 200
1100	00 0111 0000000100	Получить новое число, если последнее было меньше 200, путем перехода к адресу 4 (0100)

Окончательный проект

Объединим все, чему мы научились, в настоящий компьютер. Нам понадобится немного «клея», чтобы все заработало.

Регистр команд

Можно подумать, что компьютер просто выполняет программу Фибоначчи по одной инструкции за раз, но это не так. За кулисами происходит еще кое-что. Что нужно компьютеру для выполнения инструкции? Конечный автомат выполняет действия в два этапа, показанные на рис. 4.20.



Рис. 4.20. Цикл получения-выполнения

Первое, что нужно сделать, — извлечь инструкцию из памяти. Только получив инструкцию, можно перейти к ее выполнению.

Выполнение инструкций обычно включает доступ к памяти. Это означает, что нужно хранить инструкцию где-то рядом, пока память используется для другой задачи. На рис. 4.21 добавим в ЦП *регистр команд* для хранения текущей инструкции.



Рис. 4.21. Добавление регистра команд

Передача данных и управляющие сигналы

Мы подошли к сложной части. Нам нужен способ передать содержимое счетчика команд в адресную шину памяти и данные памяти в регистр команд. Можно выполнить аналогичное упражнение, чтобы определить все связи, необходимые для реализации полного набора команд, как подробно описано в табл. 4.4. В итоге получим рис. 4.22, который, вероятно, сбивает с толку. Но на самом деле он просто объединяет все, что мы видели раньше: несколько регистров, селекторов, АЛУ и буферный элемент с тремя состояниями.

Несмотря на то что схема выглядит довольно сложно, обратите внимание на ее сходство с картой. Да, эта схема даже проще, чем карта города. Селектор адреса — это просто трехсторонний перекресток, а селектор данных — четырехсторонний. От адресной шины и шины данных отходят соединения для таких инструментов, как устройства ввода/вывода, которые мы обсудим в главе 6.

Единственный новый компонент — это *регистр косвенных адресов*. Он необходим для хранения косвенных адресов, извлеченных из памяти, аналогично тому, как регистр команд хранит извлеченные из памяти команды.

Для простоты на рис. 4.22 опущен системный генератор тактовых сигналов, который используется для всех регистров и памяти. В случае с обычным регистром просто предположим, что регистр загружается по следующему тактовому сигналу, если генератор включен. Точно так же счетчик команд и память выполняют то, что их управляющие сигналы говорят им делать на каждом такте. Все остальные компоненты, такие как селекторы, являются чисто комбинаторными и не используют генераторы сигналов.

Управление движением

Разобрав все вводы и выводы, перейдем к созданию блока управления движением. Рассмотрим несколько примеров его работы.

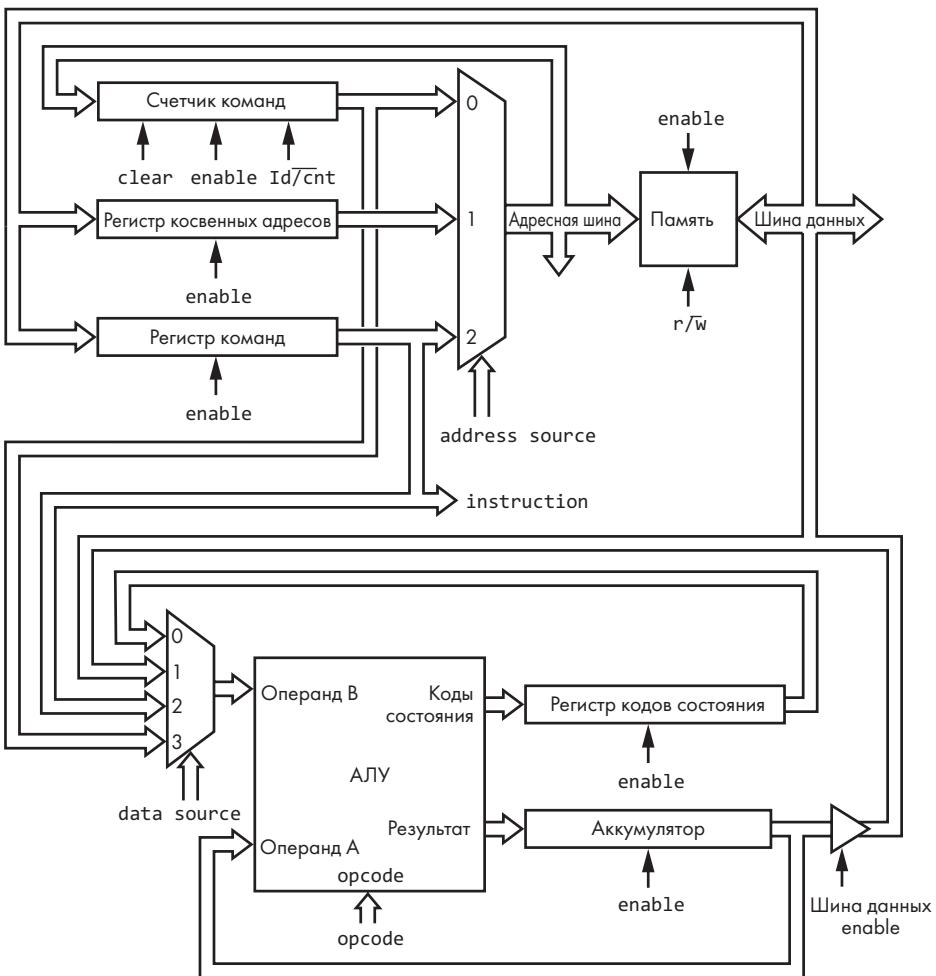


Рис. 4.22. Передача данных и управляющие сигналы (clear — сброс, enable — включение, Id/cnt — загрузка/счетчик, r/w — чтение/запись, address source — источник адресов, data source — источник данных, instruction — команда, opcode — код операции)

Выборка данных — общая для всех инструкций. Речь идет о следующих сигналах:

- address source должен быть установлен для выбора счетчика команд;
- память должна быть активирована, а сигнал r/w должен быть установлен на чтение (1);
- регистр команд должен быть включен.

В следующем примере мы сохраним содержимое аккумулятора по адресу памяти, на который указывает адрес из инструкции, — то есть используем косвенную адресацию. Выборка работает так же, как раньше.

Получение косвенного адреса из памяти:

- `address source` должен быть настроен на выбор регистра инструкции, который получает адресную часть инструкции;
- память включена, а сигнал `r/w` установлен на чтение (1);
- включен регистр косвенных адресов.

Сохранение аккумулятора в этом адресе:

- `address source` должен быть настроен на выбор регистра косвенных адресов;
- шина данных должна быть `enable`;
- память включена, а сигнал `r/w` установлен на запись (0);
- счетчик команд увеличивается.

Поскольку выборка и выполнение инструкций выполняются в несколько шагов, нам нужен счетчик для их отслеживания. Содержимое счетчика, а также части кода операции и режима в команде — все, что нам нужно для генерации всех управляющих сигналов. Нам понадобятся два бита счетчика, потому что для выполнения самых сложных инструкций необходимы три состояния, как показано на рис. 4.23.

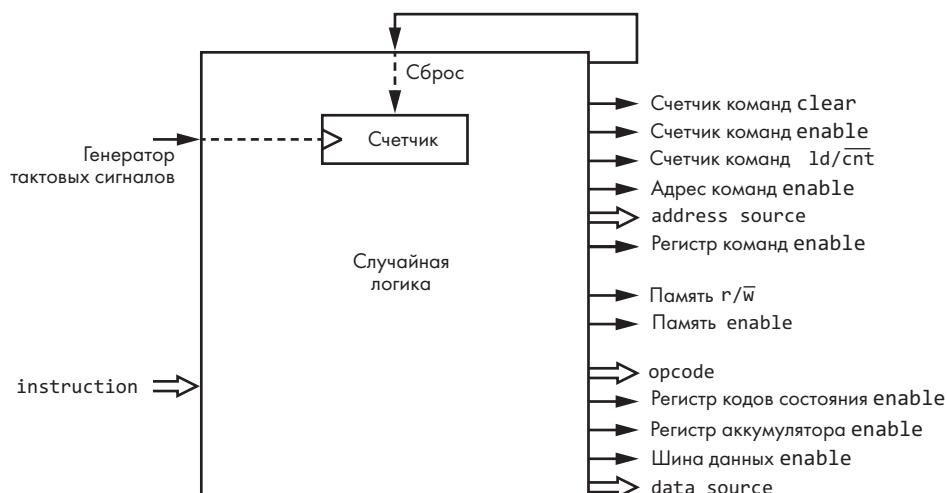


Рис. 4.23. Управление движением со случайной логикой

Это большая коробка, полная так называемой *случайной логики*. Все логические схемы, рассмотренные ранее, следуют определенному шаблону. Функциональные блоки, такие как селекторы и регистры, собираются из более простых блоков. Иногда, например при реализации блока управления движением, имеется набор входов, которые необходимо сопоставить с набором выходов для выполнения нерегулярной задачи. Схема выглядит как крысиное гнездо соединений — потому она и называется «случайной».

Однако существует еще один способ реализовать блок управления движением. Вместо случайной логики можно использовать кусок памяти. Адрес будет формироваться из выходов счетчика и части кода операции и режима в команде, как показано на рис. 4.24.



Рис. 4.24. Управление движением на основе памяти

Каждое 19-битное слово в памяти расположено, как показано на рис. 4.25.

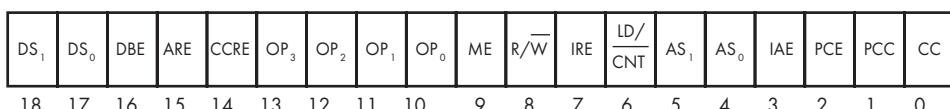


Рис. 4.25. Схема микрокодового слова

Это может показаться странным. С одной стороны, это просто еще один конечный автомат, реализованный с использованием памяти вместо случайной логики. С другой стороны, он выглядит как простой компьютер. Обе интерпретации верны. Это конечный автомат, потому что компьютеры и есть конечные автоматы. Но это еще и компьютер, потому что его можно запрограммировать.

Такой тип реализации называется *микрокодированием*, а содержимое памяти называется *микрокодом*. Да, мы используем маленький компьютер как часть большого.

Разберем часть *микрокоманд*, показанную на рис. 4.26, — она подходит для реализации рассмотренных ранее примеров.

	DS'	DS^o	DBE	ARE	$CCRE$	Op^3	Op^2	Op^1	Op^0	ME	R/W	IRF	LD/CNT	AS'	AS^o	IAE	PCF	PCC	CC
Хранение	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1
Косвенная адресация	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	0
Получение	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0

Рис. 4.26. Пример микрокода

Как и следовало ожидать, хорошей идеей легко злоупотребить. Существуют машины с *нанокодированным* блоком, который реализует *микрокодированный* блок, который, в свою очередь, реализует набор команд.

Использование ПЗУ для памяти микрокода имеет определенный смысл, потому что в противном случае нам пришлось бы хранить копию микрокода в другом месте и для загрузки микрокода потребовалось бы дополнительное аппаратное обеспечение. Однако бывают ситуации, когда использование ОЗУ или сочетания ПЗУ и ОЗУ оправданно. Некоторые процессоры Intel имеют записываемый микрокод, который можно изменять, чтобы исправлять ошибки. Некоторые машины, например из серии HP-2100, имели *перезаписываемое управляющее запоминающее устройство*, которое представляло собой ОЗУ микрокода — его можно было использовать для расширения базового набора команд.

Даже если машины имеют перезаписываемый микрокод, доступ к нему для пользователей обычно закрыт. Производители не заинтересованы в том, чтобы пользователи зависели от этого микрокода при написании своих приложений, так как в этом случае производителям будет сложно вносить в него изменения. Кроме того, микрокод с ошибками может повредить машину — например, одновременно *включить* и память, и шину данных в ЦП, соединив вместе каскадные выходы таким образом, что транзисторы перегорят.

Наборы команд RISC и CISC

Проектировщики писали для компьютеров команды, казавшиеся полезными, но приводившие к созданию довольно сложных машин. В 1980-х годах американские ученые-компьютерщики Дэвид Паттерсон (David Patterson) из Беркли и Джон Хеннесси (John Hennessey) из Стэнфорда провели статистический анализ программ и обнаружили, что многие сложные команды использовались редко. Они

первыми разработали машины, содержащие только часто используемые программы инструкции; менее используемые были удалены и заменены комбинациями других команд. Такие устройства были названы *RISC-машинами – компьютерами с сокращенным набором команд* (от reduced instruction set computer). Старые разработки получили название *CISC-машин – компьютеров со сложным набором команд* (от complicated instruction set computer).

Одной из отличительных черт RISC-машин является то, что они имеют *архитектуру типа «загрузка – сохранение»*. Это означает, что в них используется две категории инструкций: одна для доступа к памяти, а вторая – для всего остального.

Конечно, со временем использование компьютеров вышло на новый уровень. Первоначальные статистические исследования Паттерсона и Хеннесси были проведены до того, как компьютеры стали широко использоваться для таких вещей, как прослушивание аудио и просмотр видео. Статистика по более новым программам побуждает разработчиков добавлять новые инструкции для RISC-машин. Современные RISC-машины на самом деле сложнее прежних CISC-машин.

Одной из машин CISC, оказавших большое влияние на индустрию, была PDP-11 от Digital Equipment Corporation. В ней было восемь универсальных регистров вместо единственного аккумулятора, который мы использовали в примерах. Эти регистры могут использоваться для косвенной адресации. Кроме того, поддерживаются режимы *автоинкремента* и *автодекремента* для увеличения или уменьшения значений в регистрах до или после использования. Это позволило создать несколько очень эффективных программ. Например, предположим, что нужно скопировать n байт памяти, начиная с адреса источника, в память, начиная с адреса назначения. Можно поместить адрес источника в регистр 0, адрес назначения в регистр 1 и счетчик в регистр 2. Мы не будем рассматривать все используемые биты, потому что нет никакой реальной необходимости изучать полный набор инструкций PDP-11. Таблица 4.5 показывает, что делают эти команды.

Таблица 4.5. Программа копирования памяти PDP-11

Адрес	Описание
0	Скопировать содержимое ячейки памяти, адрес которой находится в регистре 0, в ячейку памяти, адрес которой находится в регистре 1, затем добавить 1 к каждому регистру
1	Вычесть 1 из содержимого регистра 2, сравнить результат с 0
2	Перейти к ячейке 0, если результат не был равен 0

Почему мы вообще должны об этом задумываться? Язык программирования C, являющийся продолжением языка B (который был продолжением BCPL), был разработан на PDP-11. Использование *указателей* в языке C, абстракции

косвенной адресации более высокого уровня, в сочетании с функциями из B, такими как операторы автоинкремента и автодекремента, хорошо сопоставляются с архитектурой PDP-11. Стал очень популярным и повлиял на проекты многих других языков, включая C++, Java и JavaScript.

Графические процессоры

Вы, наверное, слышали о *графических процессорах*, или *GPU* (от *graphics processing unit*). По большей части их изучение выходит за рамки данной книги, но я вкратце о них расскажу.

Графика — это раскраска картины по номерам в масштабе. Нет ничего необычного в том, чтобы нарисовать 8 миллионов цветных пятен и раскрашивать их 60 раз в секунду, если вам нужно получить качественное видео. Однако для этого необходимо около полумиллиарда обращений к памяти в секунду.

Работа с графикой — это специализированная задача, которая не требует всех функций универсального процессора. Ее легко можно распараллелить: рисование нескольких точек за раз может улучшить производительность.

Графические процессоры имеют две особенности. Во-первых, они включают большое количество простых процессоров. Во-вторых, у них гораздо более широкие шины памяти, чем у обычных процессоров, что означает, что они получают доступ к памяти намного быстрее. Графические процессоры используют пожарный шланг вместо садового.

Со временем графические процессоры приобрели более универсальные функции. Были продуманы варианты стандартных языков программирования для работы с графическими процессорами — теперь они используются для определенных классов приложений, которым доступны все преимущества своей архитектуры. На момент написания этой книги графических процессоров на рынке не хватало, потому что их раскупили для майнинга биткоинов.

Выводы

В этой главе мы создали настоящий компьютер, используя строительные блоки, представленные в предыдущих главах. Несмотря на простоту схем, машину, которую мы разработали, действительно можно построить и запрограммировать. Однако в ней отсутствуют некоторые элементы, используемые в реальных компьютерах, — стеки и оборудование для управления памятью. Мы рассмотрим их в главе 5.

5

Архитектура компьютера



В главе 4 был рассмотрен проект простой компьютерной системы, и мы обсудили, как ЦП взаимодействует с памятью и устройствами ввода/вывода благодаря адресным шинам и шинам данных. Однако это не конец истории. За прошедшие годы компьютеры стали работать быстрее благодаря множеству усовершенствований. При этом они стали потреблять меньше энергии, а программировать на них стало проще. Эти улучшения привели к значительному усложнению архитектуры.

Архитектура компьютера означает расположение различных компонентов в компьютере, а не то, есть ли у него дорические или ионические колонны либо индивидуальный оттенок бежевого, как тот, который создал некий американский предприниматель Стив Джобс (1955–2011) для оригинального компьютера Macintosh. За прошедшие годы было испробовано множество различных архитектур. Что-то сработало, а что-то нет. Кроме того, эта тема широко освещалась во многих книгах.

В этой главе основное внимание уделяется архитектурным улучшениям, связанным с памятью. Микрофотография современного микропроцессора показывает, что основная часть площади микросхемы отведена под обработку памяти. Это настолько важно, что заслуживает отдельной главы. Мы также коснемся некоторых других различий в архитектурах, таких как проектирование набора команд, дополнительных регистров, управления питанием и более сложных исполнительных устройств. И мы обсудим поддержку *многозадачности* — возможности запускать несколько программ одновременно (хотя бы иллюзорной).

Выполнение нескольких программ подразумевает существование некой управляющей программы — *операционной системы* (ОС), которая контролирует их выполнение.

Основные архитектурные элементы

Две наиболее распространенные архитектуры — это архитектура *фон Неймана* (названная в честь гениального венгеро-американского ученого Джона фон Неймана, 1903–1957) и *гарвардская* (названная в честь компьютера Harvard Mark I, который, конечно же, был машиной с гарвардской архитектурой). Мы уже видели все их части по отдельности; на рис. 5.1 показано, как они организованы.

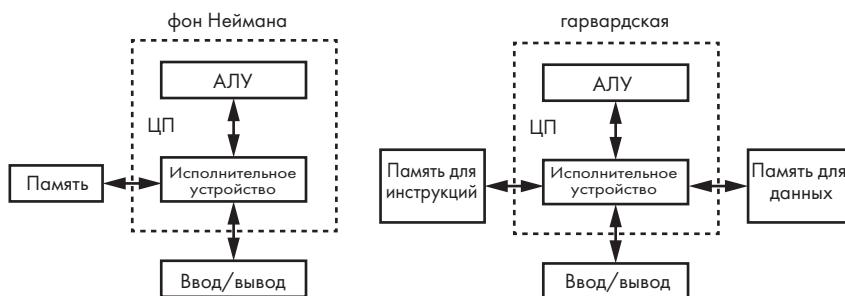


Рис. 5.1. Архитектуры фон Неймана и гарвардская

Обратите внимание, что единственное различие между ними — это способ организации памяти. При прочих равных архитектура фон Неймана немного медленнее, потому что она не может получать доступ к инструкциям и данным одновременно, поскольку использует только одну шину памяти. Гарвардская архитектура не имеет такого ограничения, но требует дополнительного аппаратного обеспечения для второй шины памяти.

Ядра процессора

Обе архитектуры на рис. 5.1 имеют один ЦП, который, как мы видели в главе 4, представляет собой комбинацию АЛУ, регистров и исполнительного устройства. *Многопроцессорные* системы с несколькими процессорами появились в 1980-х годах как способ получить более высокую производительность, чем можно было бы достичь с помощью одного процессора. Однако, как оказалось, это не так просто. Разделение одной программы так, чтобы ее можно было *распараллелить* для использования нескольких процессоров, — в целом до сих пор не решенная задача, хотя некоторые варианты решений иногда работают хорошо, например для определенных типов сложной математики. Тем не менее распараллеливание полезно, если запускается более одной программы одновременно. Оно

весьма пригодилось на заре создания графических рабочих станций, поскольку система X Window была настолько ресурсоемкой, что для ее запуска требовался отдельный процессор.

Уменьшение геометрических размеров процессоров снижает затраты. Микросхемы изготавливаются на кремниевых пластинах, а уменьшение размеров схем означает, что на одной пластине помещается больше микросхем. Раньше более высокая производительность достигалась за счет быстродействия процессора, что означало увеличение тактовой частоты. Но более быстрые машины требовали большей мощности, что в сочетании с меньшими геометрическими размерами давало больше тепла на единицу площади. Процессоры достигли *предела мощности* около 2000 года — удельную мощность невозможно было увеличить без превышения температуры плавления.

В некотором роде спасение было найдено в меньших геометрических размерах. Изменилось определение ЦП; то, что мы раньше называли ЦП, теперь называется *ядром процессора*. *Многоядерные* процессоры стали обычным явлением. Существуют даже системы с несколькими многоядерными процессорами, в основном в центрах обработки данных.

Микропроцессоры и микрокомпьютеры

Еще одно независимое архитектурное отличие заключается в механической упаковке. На рис. 5.1 показаны процессоры, подключенные к памяти и вводу/выводу. Когда память и ввод/вывод не находятся в том же физическом корпусе, что и ядра процессора, мы называем это *микропроцессором*. Если же все встроено в одну схему, мы используем термин *микрокомпьютер*. Это не совсем четко определенные термины, и в их использовании есть некоторые пробелы. Некоторые считают микрокомпьютер компьютерной системой, построенной на базе микропроцессора, и используют термин «микроконтроллер» для обозначения того, что я только что определил как микрокомпьютер.

Микрокомпьютеры, как правило, менее мощны, чем микропроцессоры, потому что такие вещи, как внутренняя память, занимают много места. В этой главе мы не будем уделять много внимания микрокомпьютерам, потому что у них нет сложных проблем с памятью. Однако, как только вы научитесь программировать, стоит взять в руки что-нибудь вроде Arduino — небольшого компьютера с гарвардской архитектурой на базе микросхемы микрокомпьютера Atmel AVR. Arduino отлично подходят для создания всевозможных игрушек и прочего.

Подводя итог: микропроцессоры обычно являются компонентами более крупных систем, а микрокомпьютеры обычно применяются в чем-то более тривиальном — например, в посудомоечных машинах.

Есть еще один вариант — *интегральные*, или *встроенные*, *системы* (system on a chip, SoC). Приемлемое, но опять же нечеткое определение: SoC — это более

сложный микрокомпьютер. Вместо того чтобы иметь относительно простой интегральный ввод/вывод, SoC может включать в себя, например, интерфейс Wi-Fi. SoC используются в таких устройствах, как сотовые телефоны. Существуют даже SoC, которые включают программируемые пользователем вентильные матрицы (ППВМ), которые можно дополнительно настроить.

Процедуры, подпрограммы и функции

Многие инженеры страдают странной разновидностью лени. Если они не хотят совершать некое действие, они вложат всю свою энергию в создание того, что выполнит задачу за них, даже если для этого потребуется больше работы, чем для самого дела. Программисты хотят избежать повторного написания одного и того же фрагмента кода. На это есть веские причины, помимо лени. Например, код без повторов занимает меньше места, и, если в коде есть ошибка, ее нужно исправить только один раз.

Функция (или *процедура*, или *подпрограмма*) является основой повторного использования кода. Для вас все эти термины означают одно и то же; это просто региональные языковые различия. Мы будем использовать понятие «*функция*», потому что оно больше всего похоже на то, что вы, возможно, изучали на уроках математики.

Конструкции в большинстве языков программирования похожи. Например, в JavaScript можно написать код, показанный в листинге 5.1.

Листинг 5.1. Пример JavaScript-функции

```
function
cube(x)
{
    return (x * x * x);
}
```

Этот код создает функцию `cube`, которая принимает единственный параметр с именем `x` и возвращает его значение в кубе. На клавиатуре нет символа умножения (`*`), поэтому во многих языках программирования вместо умножения используется `*`.

Теперь напишем фрагмент программы, как в листинге 5.2.

Листинг 5.2. Пример вызова JavaScript-функции

```
y = cube(3);
```

Приятная особенность функции заключается в том, что мы можем выполнять, или *вызывать*, функцию `cube` несколько раз без необходимости писать ее снова. Мы можем вычислить `cube(4) + cube(6)`, не повторяя код возвведения числа

в куб. Это простой пример, но подумайте, насколько удобна эта возможность для более сложных фрагментов кода.

Как это работает? Нам нужен способ запустить код функции, а затем вернуться к месту ее вызова. Чтобы вернуться, нам нужно знать, откуда мы пришли, — эта информация хранится в счетчике команд (который мы рассматривали на рис. 4.12 на с. 145). Таблица 5.1 показывает, как выполнить вызов функции, используя пример набора инструкций из раздела «Набор инструкций» на с. 146.

Таблица 5.1. Выполнение вызова функции

Адрес	Инструкция	Операнд	Комментарии
100	psa		Счетчик команд -> аккумулятор
101	add	5 (немедленная адресация)	Адрес для результата вычисления ($100 + 5 = 105$)
102	store	200 (прямая адресация)	Хранит адрес возврата в памяти
103	load	3 (немедленная адресация)	Помещает число для <code>cube(3)</code> в аккумулятор
104	bra	300 (прямая адресация)	Вызывает функцию <code>cube</code>
105			Продолжает выполнение с этого же места после вызова функции
...			
200			Зарезервированная ячейка в памяти
...			
300	Функция <code>cube</code>
...			Напоминание о функции <code>cube</code>
310	bra	200 (косвенная адресация)	Ответвление к сохраненному адресу возврата

Что тут происходит? Сначала мы вычисляем адрес, по которому нужно продолжить выполнение после возврата из функции `cube`. Для этого потребуется несколько инструкций; кроме того, нужно загрузить число, которое будет возведено в куб. Это еще пять инструкций, поэтому мы сохраним этот адрес в ячейке памяти 200. Выполнение переходит к функции, а когда функция завершается, выполнение косвенно переходит через ячейку 200, так что мы оказываемся в ячейке 105. Этот процесс выполняется, как показано на рис. 5.2.

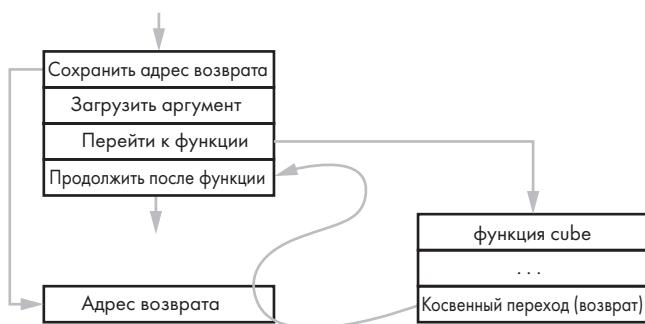


Рис. 5.2. Поток вызова функции

Число действий слишком велико, особенно для повторного выполнения, поэтому многие машины добавляют вспомогательные инструкции. Например, у процессоров ARM есть инструкция «перейти со ссылкой» (Branch with Link, BL), которая объединяет переход к функции с сохранением адреса следующей инструкции.

Стеки

Функции не ограничиваются простыми фрагментами кода, как в только что рассмотренном примере. Функции обычно вызывают другие функции, а те вызывают сами себя.

Подождите-ка, что это сейчас было? Функция вызывает сама себя? Это называется *рекурсией*, и она действительно полезна. Рассмотрим пример. Ваш телефон, вероятно, использует *сжатие JPEG* (Joint Photographic Experts Group) для уменьшения размера файлов фотографий. Чтобы увидеть, как работает сжатие, начнем с квадратного черно-белого изображения на рис. 5.3.

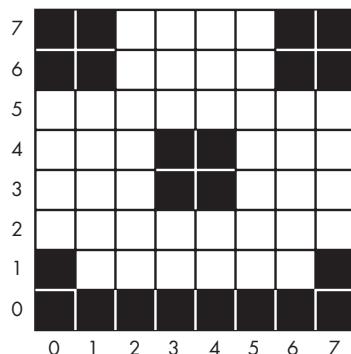


Рис. 5.3. Незатейливый смайлик

Можно решить проблему сжатия, используя *рекурсивное деление*: мы смотрим на изображение и, если оно не одного цвета, делим его на четыре части, затем снова проверяем и так далее, пока части не станут размером в один пиксель.

Листинг 5.3 представляет функцию `subdivide`, которая обрабатывает часть изображения. Она написана на *псевдокоде*, языке программирования, похожем на естественный язык, — он прекрасно подходит для примеров. Функция *принимает* координаты *x* и *y* левого нижнего угла квадрата вместе с размером (нам не нужны и ширина, и высота, поскольку изображение квадратное). «Принимает» — это просто сокращение от того, что в математике называется *передачей аргументов в функцию*.

Листинг 5.3. Функция `subdivide`

```
функция
subdivide(x, y, размер)
{
    ЕСЛИ (размер ≠ 1 И пиксели в квадрате не имеют одинаковый цвет) {
        половина = размер ÷ 2
        subdivide(x, y, half)           нижний левый квадрант
        subdivide(x, y + half, half)    верхний левый квадрант
        subdivide(x + half, y + half, half) верхний правый квадрант
        subdivide(x + half, y, half)    нижний правый квадрант
    }
    ИНАЧЕ {
        сохранить данные о квадрате
    }
}
```

Функция `subdivide` разделяет изображение на блоки одного цвета, начиная с нижнего левого квадранта, затем верхнего левого, верхнего правого и, наконец, нижнего правого. На рис. 5.4 серым цветом показаны объекты, которые необходимо разделить, и объекты одного цвета — черным или белым.

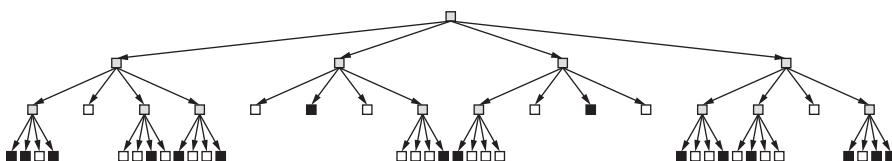


Рис. 5.4. Разделение изображения

То, что мы получили, похоже на то, что компьютерщики называют *деревом*, а математики — *направленным ациклическим графом (НАГ)*. Следите за стрелками. В этой структуре стрелки не идут вверх, поэтому невозможно получить цикл. Узлы, из которых не выходят стрелки, называются *концевыми, или листовыми, узлами*, и они обозначают конец линии, как листья — конец линии на ветке дерева. Если прищуриться и сосчитать их на рис. 5.4, можно увидеть, что сплошных

квадратов всего 40, а на исходном изображении — 64, то есть хранить нужно меньше информации. Это и называется сжатием.

По какой-то причине, вероятно, из-за того что так легче рисовать (или потому что они редко выходят на улицу), компьютерные гики всегда помещают корень дерева наверху и направляют дерево вниз. Этот конкретный вариант называется *квадродеревом*, потому что каждый узел делится на четыре части. Квадродеревья — это *пространственные структуры данных*. Ханан Самет (Hanan Samet) превратил их в дело всей своей жизни и написал несколько отличных книг на эту тему.

Возникает проблема с реализацией функций, показанных в предыдущем разделе. Поскольку имеется только одно место для хранения возвращаемого значения, такие функции не могут вызывать сами себя, потому что это значение будет перезаписано, а путь возврата потерян.

Нам нужно иметь возможность хранить несколько адресов возврата, чтобы рекурсия работала. Также нам необходим способ связать адреса возврата с соответствующими вызовами функций. Посмотрим, можно ли найти закономерность в том, как было разделено изображение. Мы спускались вниз по дереву вдоль каждой ветви и возвращались назад, только когда путь вниз уже был невозможен. Это называется *обходом в глубину*, в противоположность *обходу в ширину*, когда путь начинается с определенной вершины, исследуются все ее соседи, а затем происходит переход к вершинам следующего уровня. Каждый раз при спуске на уровень ниже нужно помнить, откуда мы начали спуск, чтобы вернуться назад. Как только мы вернемся, это место можно больше не хранить в памяти.

Нам нужно что-то вроде приспособлений, которыедерживают стопки тарелок в кафетерии. Вызывая функцию, мы наклеиваем адрес возврата на тарелку и кладем ее в стопку. Возвращаясь, мы достаем эту тарелку. Иначе говоря, это *стек*. Есть еще более умные слова: структура LIFO (last in, first out — «последним пришел — первым ушел»). Объекты *помещаются* в стек и *извлекаются* из него. Попытка поместить объекты в стек, в котором нет места, называется *переполнением стека*. Попытка извлечь что-то из пустого стека — это *исчерпание стека*.

Все эти действия можно реализовать программно. В нашем предыдущем примере вызова функции в табл. 5.1 каждая функция может взять сохраненный адрес возврата и поместить его в стек для последующего извлечения. К счастью, большинство компьютеров имеют аппаратную поддержку стеков, потому что стеки очень важны. Эта поддержка включает в себя *регистры лимитов*, чтобы программе не приходилось постоянно проверять возможное переполнение. В следующем разделе мы поговорим о том, как процессоры обрабатывают исключения, такие как превышение лимитов.

Стеки используются не только для адресов возврата. Наша функция `subdivide` использовала *локальную переменную*, в которой мы один раз вычислили половину размера, а затем использовали ее восемь раз для ускорения программы.

Невозможно просто перезаписывать ее каждый раз при вызове функции. Вместо этого локальные переменные также хранятся в стеке. Это делает каждый вызов функции независимым от вызовов других функций. Набор объектов, хранящихся в стеке для каждого вызова, представляет собой *фрейм стека*. На рис. 5.5 показан пример функции из листинга 5.3.

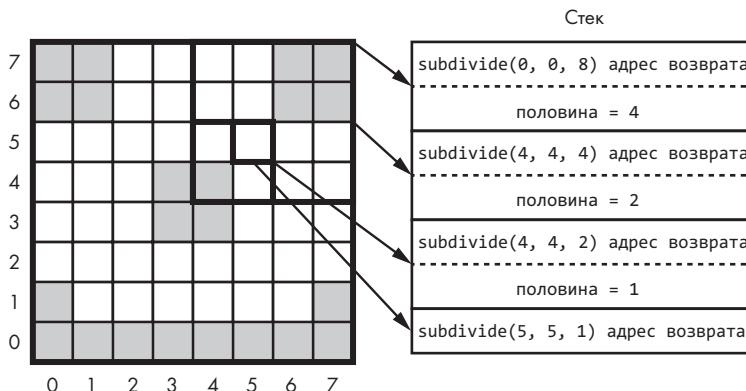


Рис. 5.5. Фреймы стека

Мы идем по пути, обозначенному жирными черными квадратами. Видим, что каждый вызов генерирует новый фрейм стека, который включает как адрес возврата, так и локальную переменную.

Некоторые компьютерные языки, например *forth* и *PostScript*, основаны на стеке (см. врезку «Различные представления уравнений»), как и несколько классических калькуляторов НР.

Стеки не ограничиваются компьютерными языками. Японский язык основан на стеке: существительные помещаются в стек, а глаголы воздействуют на них. Загадочные высказывания магистра Йоды также следуют этой схеме.

РАЗЛИЧНЫЕ ПРЕДСТАВЛЕНИЯ УРАВНЕНИЙ

Существует множество способов упорядочивания операторов и операндов. Вы, вероятно, привыкли заниматься математикой, используя так называемую инфиксную запись. Инфиксная запись помещает операторы между операндами, например $4 + 8$. В инфиксной записи для группировки требуются круглые скобки, например $(1 + 2) \times (3 + 4)$.

Польский логик Ян Лускашевич (Jan Łukasiewicz) изобрел префиксное представление в 1924 году. Оно также известно как польская запись, по национальности ученого. В польской записи оператор ставится перед операндами, например $+ 4 8$. Преимущество польской записи заключается в том, что при такой записи не требуются скобки. Предыдущий инфиксный пример будет записан как $\times + 1 2 + 3 4$.

Американский математик Артур Бёркс (Arthur Burks) предложил обратную польскую запись (reverse Polish notation, RPN), также называемую постфиксным представлением, в 1954 году. RPN помещает оператор после операндов, как в записи $4\ 8\ +$, поэтому предыдущий пример будет выглядеть как $1\ 2\ +\ 3\ 4\ +\ \times$.

RPN легко реализовать с помощью стеков. Операнды помещаются в стек. Операторы извлекают операнды из стека, выполняют операцию, а затем помещают результат обратно в стек.

В калькуляторах HP RPN есть клавиша ввода, которая помещает операнд в стек в неоднозначных ситуациях; иначе никак не определить, что 1 и 2 — это отдельные операнды, а не число 12 . На таком калькуляторе уравнение решается с использованием последовательности клавиш $1\ ENTER\ 2\ +\ 3\ ENTER\ 4\ +\ \times$. Калькулятор инфиксной записи потребует больше нажатий на клавиши.

Пример уравнения будет выглядеть так: $1\ 2\ add\ 3\ 4\ add\ mul$ в языке программирования PostScript. Никакого специального ENTER не требуется, потому что вместо него используется пробел.

Прерывания

Представьте, что вы на кухне замешиваете тесто для шоколадного печенья. Вы готовите по рецепту, который представляет собой программу для поваров. Вы одни дома, поэтому вам нужно знать, не подошел ли кто-то к двери. Изобразим вашу работу с помощью **блок-схемы**, которая представляет собой тип диаграммы, используемый для передачи хода различных процессов. Пример — на рис. 5.6.

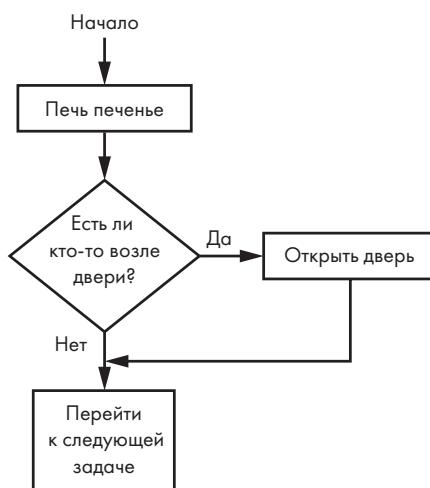


Рис. 5.6. Выпечка печенья № 1

Это может сработать, если ваш гость очень терпелив. Но допустим, что вам доставили посылку, за которую нужно расписаться. Курьер не будет ждать 45 минут, если только не почувствует запах печенья и не понадеется на угощение. Попробуем другую схему, как на рис. 5.7.

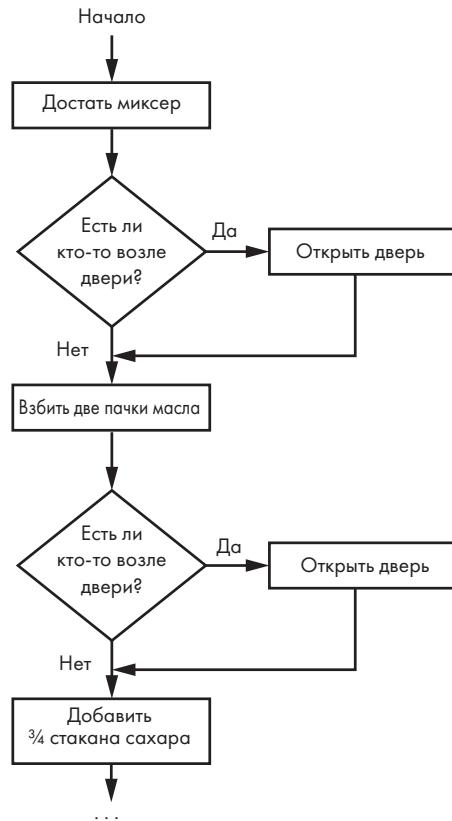


Рис. 5.7. Приготовление печенья № 2

Данный метод называется *опросом*. Он работает, но не очень хорошо. У вас меньше шансов пропустить курьера, но вы тратите слишком много времени на походы к двери.

Можно разделить каждую из задач по приготовлению на более мелкие подзадачи и проверять дверь между ними. Это повысит шансы на получение посылки, но в какой-то момент вы потратите больше времени на походы к двери, чем на готовку.

Это частая и важная проблема, у которой, действительно, нет программного решения. Невозможно заставить схему хорошо работать, изменив структуру программы. Нужно придумать способ *прервать* запущенную программу, чтобы

она могла отреагировать на что-то внешнее, требующее внимания. Пришло время добавить некоторые аппаратные функции в исполнительное устройство.

Практически каждый современный процессор включает в себя блок *прерываний*. Обычно он имеет контакты (пины) или электрические соединения, которые прерывают выполнение при правильном перемещении. *Пин* (pin) — это разговорный термин, обозначающий какое-либо соединение с микросхемой электрическим способом. Раньше в микросхемах были детали, похожие на булавки (pin — «булавка»), но по мере того, как устройства и инструменты становились меньше, появилось много других вариантов. Многие процессорные микросхемы, особенно микрокомпьютеры, содержат *встроенные периферийные устройства* (внутрипроцессорные устройства ввода/вывода), которые имеют внутреннее подключение к системе прерывания.

Объясню, как это работает. Периферийное устройство, требующее внимания, генерирует *запрос на прерывание*. Процессор (обычно) завершает инструкцию, выполняемую в данный момент. Затем он приостанавливает текущую программу и переключается на совершенно другую программу, называемую *обработчиком прерываний*. Обработчик прерываний делает все, что ему нужно, и основная программа продолжает работу с того места, где она остановилась. Обработчики прерываний — это функции.

Обработчиком прерываний в процессе приготовления печенья является дверной звонок. Вы с удовольствием готовите печенье, пока вас не прервут звонок в дверь, хотя, если вас постоянно прерывают опросчики, это раздражает. Следует учесть несколько моментов. Во-первых, *время реакции* на прерывание. Если вы долго болтаете с курьером, печенье может подгореть; вам необходимо убедиться, что вы можете своевременно обрабатывать прерывания. Во-вторых, нужен способ сохранить *состояние* перед ответом на прерывание, чтобы вернуться к готовке после его *обработки*. Например, если в прерванной программе что-то было в регистре, обработчик прерывания должен сохранить содержимое этого регистра, а затем восстановить его перед возвратом в основную программу.

Система прерывания использует стек, чтобы сохранить место в прерванной программе. Задача обработчика прерывания — записать все, что ему может понадобиться. Таким образом, обработчик может сэкономить абсолютный минимум, необходимый для быстрой работы.

Как компьютер узнает, где найти обработчик прерывания? Обычно существует набор зарезервированных адресов памяти для векторов прерываний, по одному для каждого поддерживаемого прерывания. *Вектор прерывания* — это просто указатель, адрес ячейки памяти. Это похоже на вектор в математике или физике — на стрелку, которая говорит: «Иди отсюда туда». Когда происходит прерывание, компьютер ищет этот адрес и передает ему управление.

Многие устройства включают векторы прерываний для исключений, в том числе для переполнения стека и использования недопустимого адреса, например

адреса, выходящего за пределы физической памяти. Переадресация исключений обработчику прерывания часто позволяет ему исправить проблемы, чтобы программа продолжила работу.

Как правило, существуют всевозможные специальные средства управления прерываниями, например способы включения и выключения определенных прерываний. Часто используется *маска*, чтобы сказать что-то вроде «не прерывай, пока дверца духовки открыта». На машинах с несколькими типами прерываний зачастую указывается порядок *приоритетов*, так что самые важные события обрабатываются в первую очередь. Это означает, что обработчики прерываний с более низким приоритетом могут быть прерваны самостоительно. Большинство устройств имеют один или несколько встроенных *таймеров*, которые можно настроить для генерации прерываний.

Операционные системы, описанные в следующем разделе, часто не позволяют получить доступ к *физическим* (аппаратным) прерываниям для большинства программ. Они предоставляют взамен *виртуальную* или программную систему прерывания. Например, операционная система UNIX имеет *сигнальный* механизм. Многие современные системы называют этот механизм *событиями*.

Относительная адресация

Что нужно для одновременного запуска нескольких программ?

Для начала нам понадобится программа-диспетчер, которая знает, как переключаться между программами. Мы будем называть ее операционной системой или *ядром* операционной системы. Чтобы отличать ОС и программы, которые она контролирует, назовем ОС *системной* программой, а все остальное — *пользовательскими* программами, или *процессами*. Простая ОС работает примерно так, как показано на рис. 5.8.

Здесь ОС использует таймер, чтобы указать, когда нужно переключаться между пользовательскими программами. Этот метод планирования называется *квантованием времени*, потому что он дает каждой программе отрезок времени для выполнения. *Состояние* или *контекст* пользовательской программы относится к содержимому регистров и любой памяти, которую программа использует, включая стек.

Это работает, но довольно медленно. На загрузку программы нужно время. Мы бы ускорили работу, если бы программы можно было загружать в память, насколько позволяет пространство, и хранить их там, как показано на рис. 5.9.

В этом примере пользовательские программы загружаются в память одна за другой. Но подождите, как это может работать? Как объяснялось в разделе «Режимы адресации» на с. 149, наш компьютер-образец использовал *абсолютную адресацию*, это означает, что адреса в инструкциях относятся к определенным

ячейкам памяти. Запустить программу, ожидающую адрес 1000, по другому адресу, например 2000, не получится.

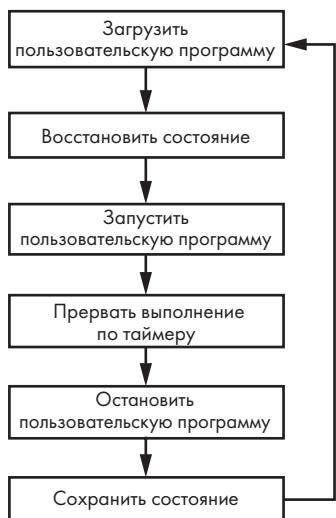


Рис. 5.8. Простая операционная система

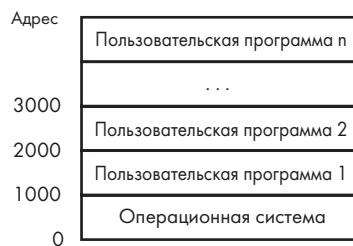


Рис. 5.9. Несколько программ в памяти

Некоторые компьютеры решают эту проблему, добавляя *индексный регистр* (рис. 5.10) — регистр, содержимое которого добавляется к адресам для формирования *действительных адресов*. Если пользовательская программа ожидает запуска по адресу 1000, ОС может передать в индексный регистр адрес 2000, прежде чем запускать ее по адресу 3000.

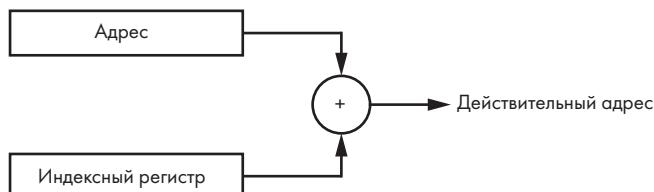


Рис. 5.10. Индексный регистр

Еще один способ решить проблему — использование *относительной адресации*. Вместо того чтобы располагать адреса в инструкциях относительно 0 (начала памяти в большинстве устройств), можно помещать их относительно адресов их инструкций. Вернитесь к табл. 4.4 на с. 153. Обратите внимание, что вторая инструкция содержит адрес 100 (110100 в двоичном формате). В случае с относительной адресацией мы используем адрес +99, поскольку инструкция

находится по адресу 1, а адрес 100 находится на расстоянии 99. Аналогично последняя инструкция — это ответвление к адресу 4, который станет ответвлением с адресом -8 в относительной адресации. Подобные вещи кажутся кошмарными в двоичном формате, но современные языковые инструменты делают всю арифметику за нас. Относительная адресация позволяет *перемещать* программу в любое место в памяти.

Блок управления памятью

Многозадачность перестала быть роскошью сегодня, когда все устройства подключены к интернету, потому что задачи обмена данными постоянно выполняются в *фоновом режиме*, то есть в дополнение к тому, что делает пользователь. Немного помогают регистры индексов и относительная адресация, но этого недостаточно. Что произойдет, если в какой-то программе возникнут ошибки? Например, что если ошибка в пользовательской программе 2 (рис. 5.9) заставит ее перезаписать что-то в пользовательской программе 1 или, что еще хуже, в ОС? Что, если кто-то намеренно написал программу, чтобы шпионить за другими программами, работающими в системе, или изменять их? Вообще желательно изолировать каждую программу, чтобы сделать эти сценарии невозможными. С этой целью большинство микропроцессоров теперь включают в себя **блок управления памятью** (memory management unit, MMU). MMU — пример очень сложного аппаратного обеспечения.

Системы с MMU различают *виртуальные* и *физические* адреса. MMU преобразует виртуальные адреса, задействованные программами, в физические адреса, используемые памятью, как показано на рис. 5.11.



Рис. 5.11. Преобразование адреса в MMU

Чем это отличается от индексного регистра? Блоки управления памятью используют не всю ширину адреса. Здесь происходит разделение виртуального адреса на две части. Нижняя часть идентична физическому адресу. Верхняя часть преобразовывается через часть ОЗУ, называемую *таблицей переадресации страниц*, пример которой представлен на рис. 5.12.

В этом примере память разбита на 256-байтные *страницы*. Содержимое таблицы переадресации страниц определяет фактическое расположение каждой страницы в физической памяти. Это позволяет взять программу, которая ожидает запуска с адреса 1000, и поместить ее в адрес 2000 или другой, если она выровнена по *границе страницы*. И хотя виртуальное адресное пространство

кажется программе непрерывным, его не нужно отображать на непрерывные страницы физической памяти. Можно даже переместить программу в другое место в физической памяти во время ее работы. Мы можем предоставить одной или нескольким взаимодействующим программам общую память, отобразив части их виртуального адресного пространства на одну и ту же физическую память. Обратите внимание, что содержимое таблицы переадресации страниц становится частью контекста программы.

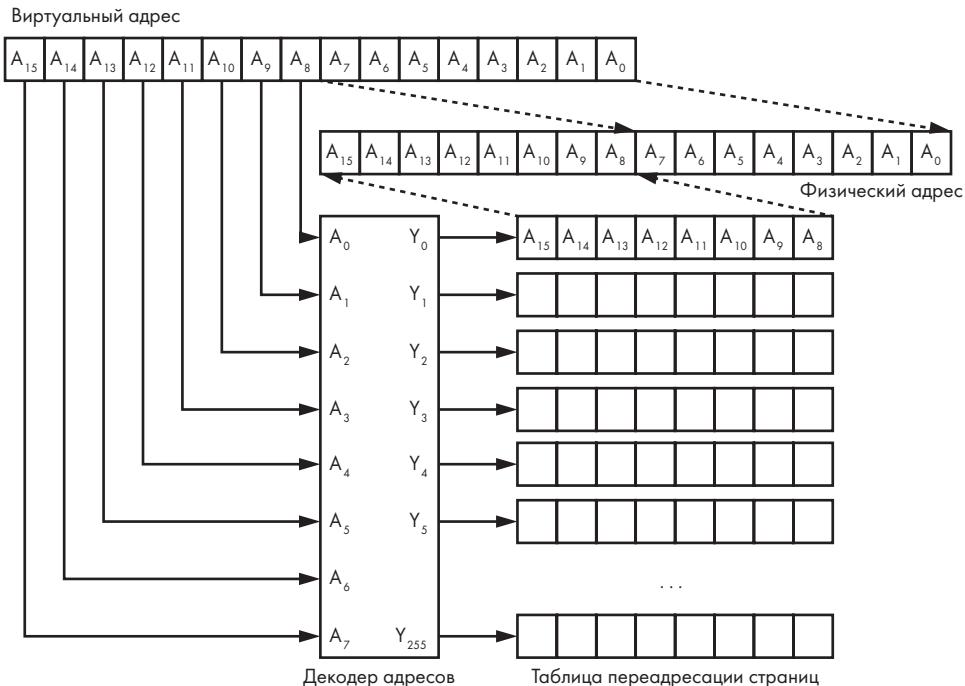


Рис. 5.12. Простая таблица переадресации страниц для 16-битного компьютера

Вы могли заметить, что таблица переадресации страниц выглядит как фрагмент памяти. И вы будете правы. Наверняка вы ждете, что я скажу, что это непросто. Так и есть. В нашем примере используются 16-битные адреса. Что будет в случае с современными компьютерами с 64-битными адресами? Если мы разделим адрес пополам, нам понадобится 4 ГиБ таблицы переадресации страниц, и размер страницы также будет равен 4 ГиБ, что не очень хорошо, поскольку это больше, чем есть во многих системах. Можно уменьшить размер страницы, но это увеличит размер самой таблицы переадресации. С этим нужно что-то делать.

MMU в современном процессоре имеет ограниченный размер таблицы переадресации страниц. Полный набор записей таблицы хранится в основной памяти

или на диске, если память заканчивается. MMU загружает подмножество записей таблицы переадресации страниц в свою таблицу по мере необходимости.

Некоторые проекты MMU добавляют в свои таблицы переадресации страниц дополнительные управляющие биты — например, *бит, не выполняющий никаких действий*. Когда такой бит помещается на страницу, ЦП не будет выполнять инструкции с этой конкретной страницы. Это не позволяет программам загружать свои собственные данные, потому что это представляет собой угрозу безопасности. Другой общий управляющий бит делает страницы *доступными только для чтения*.

MMU генерируют исключения «*отказ страницы*», когда программа пытается получить доступ к адресу, не сопоставленному с физической памятью. Это полезно, например, в случае переполнения стека. Вместо того чтобы прерывать выполняющуюся программу, ОС может использовать MMU для отображения некоторой дополнительной памяти, чтобы увеличить пространство стека, а затем возобновить выполнение пользовательской программы.

MMU практически сглаживают различия в архитектуры фон Неймана и гарвардской. Такие системы имеют единую шину, как в архитектуре фон Неймана, но могут предоставлять отдельную память для команд и данных.

Виртуальная память

Операционные системы управляют распределением ограниченных аппаратных ресурсов между конкурирующими программами. Например, на рис. 5.8 мы видели, как ОС управляет доступом к самому процессору. Память также является управляемым ресурсом. Операционные системы используют MMU для представления *виртуальной памяти* пользовательским программам.

Ранее мы видели, что MMU может связывать виртуальные адреса программы с физической памятью. Но виртуальная память полезна не только для этого. Механизм отказа страницы позволяет программам думать, что у них может быть столько памяти, сколько им понадобится, даже если это превышает ее физический объем. Что произойдет, если запрошеннная память больше доступного объема? ОС перемещает содержимое страниц памяти, которые в настоящее время не нужны, в более крупное, но более медленное запоминающее устройство, чаще всего на диск. Когда программа пытается получить доступ к *выгруженной* памяти, ОС делает все, что ей нужно, чтобы освободить место, а затем копирует запрошенную страницу обратно. Этот процесс известен как *подкачка по требованию*. На рис. 5.13 показана система виртуальной памяти с одной выгруженной страницей.

При подкачке сильно страдает производительность системы, но это все же лучше, чем невозможность запустить программу из-за нехватки памяти. Системы виртуальной памяти используют ряд уловок, чтобы минимизировать снижение производительности. Одна из них — это алгоритм *удаления наилучше давно*

использованных элементов (least recently used, LRU), отслеживающий доступ к страницам. Самые востребованные страницы хранятся в физической памяти, а наименее часто используемые выгружаются.

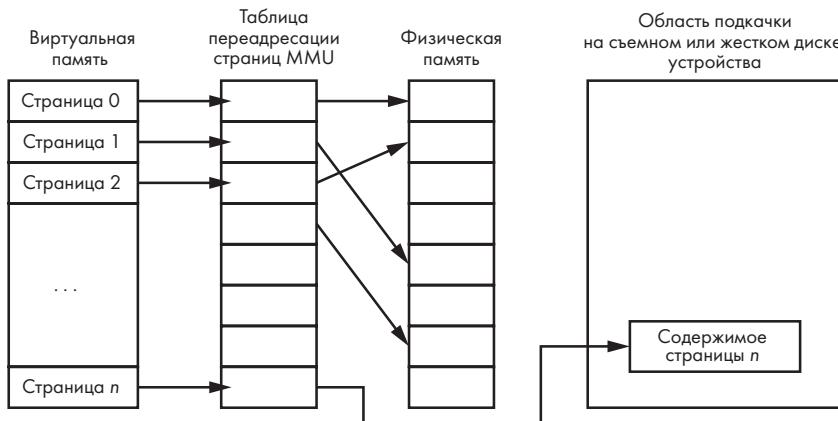


Рис. 5.13. Виртуальная память

Пространство системы и пользователя

Системы многозадачности создают для каждого процесса иллюзию, что это единственная программа, запущенная на компьютере. MMU помогают развить эту иллюзию, предоставляя каждому процессу собственное пространство адресов. Но эту иллюзию трудно поддерживать, если дело касается устройств ввода/вывода. Например, ОС использует таймер, чтобы напомнить самой себе, когда переключаться между программами, показанными на рис. 5.8. ОС решает установить таймер для генерации прерывания один раз в секунду, но, если одна из пользовательских программ изменит его значение на прерывание один раз в час, все пойдет не так, как ожидалось. Точно так же MMU не обеспечит серьезной изоляции между программами, если какая-либо пользовательская программа может изменить собственную конфигурацию.

Многие процессоры включают дополнительное аппаратное обеспечение, которое решает эту проблему. В регистре есть бит, который указывает, находится ли компьютер в *системном* или *пользовательском* режиме. Некоторые инструкции, например те, которые имеют дело с вводом/выводом, считаются привилегированными и могут выполняться только в системном режиме. Специальные команды, называемые *ловушками* или *системными вызовами*, позволяют программам пользовательского режима делать запросы к программам системного режима, то есть к операционной системе.

Такое ранжирование имеет несколько преимуществ. Во-первых, оно защищает ОС от действий пользовательских программ, а эти программы — друг от друга.

Во-вторых, поскольку программы пользователя не имеют доступа к определенным вещам, таким как MMU, операционная система может управлять выделением ресурсов для них. Системное пространство — это место, где обрабатываются аппаратные исключения.

Любые программы, которые вы создаете для телефона, ноутбука или настольного компьютера, будут запускаться в пользовательском пространстве. Прежде чем вносить изменения в программы, работающие в системном пространстве, нужно стать действительно хорошим программистом.

Иерархия памяти и производительность

Когда-то процессоры и память работали с одинаковой скоростью и на земле царил мир. Однако скорость процессоров постоянно увеличивалась, и, хотя память тоже становилась быстрее, она не могла утнаться за ними. Архитекторы придумали всевозможные уловки, чтобы эти быстрые процессоры не сидели без дела в ожидании памяти.

Виртуальная память и подкачка вводят понятие *иерархии памяти*. Хотя для пользовательской программы вся память выглядит одинаково, то, что происходит за кулисами, сильно влияет на производительность системы. Или, перефразируя Джорджа Оруэлла, все обращения к памяти равны, но некоторые равнее.

Компьютеры быстры. Они могут выполнять миллиарды инструкций в секунду. Но вряд ли мы могли что-то сделать, если бы ЦП приходилось ожидать этих инструкций или данных, которые нужно извлечь или сохранить.

Мы выяснили, что процессоры включают в себя очень быструю и дорогостоящую память, называемую регистрами. Ранние компьютеры имели лишь несколько регистров, тогда как некоторые современные машины содержат их сотни. Но в целом соотношение регистров к памяти стало меньше. Процессоры обмениваются данными с *основной памятью*, обычно с DRAM, которая практически в десять раз медленнее процессора. Запоминающие устройства, такие как съемные и жесткие диски, имеют скорость в одну миллионную от скорости процессора.

Пришло время провести аналогию с едой, любезно предоставленную моим другом Клемом. Регистры похожи на холодильник: в них не так много места, но все содержимое можно достать быстро. Основная память похожа на продуктовый магазин: в ней намного больше места для хранения, но требуется время, чтобы добраться туда. Запоминающее устройство похоже на склад: места еще больше, но находится он еще дальше.

Расширим эту аналогию. Вы часто открываете холодильник ради какого-то одного продукта. Отправляясь в магазин, вы наполняете продуктами несколько пакетов. Склад доставляет продукты в магазин целыми грузовыми автомобилями.

Компьютеры действуют так же. Небольшие блоки данных перемещаются между процессором и основной памятью. Большие блоки перемещаются между основной памятью и диском. Загляните в «The Paging Game» Джекфа Берримана (Jeff Berryman) — он придумал забавное объяснение того, как все это работает.

Опустив множество подробностей, предположим, что процессор работает примерно в 10 раз быстрее, чем основная память. Это приводит к тому, что в ожидании памяти тратится много времени, поэтому было добавлено дополнительное оборудование (более быстрая встроенная память) для «кладовки», или *кэша*. Кэш намного меньше магазина, но гораздо быстрее при работе на полной скорости процессора.

Как заполнить кладовку продуктами из магазина? Еще в разделе «Оперативная память» на с. 125 мы увидели, что DRAM лучше работает при обращении к столбцам, чем к строкам. Исследуя, как работают программы, вы заметите, что они обращаются к последовательным ячейкам памяти, если не используется ветвление. При этом изрядное количество данных, используемых программой, имеет тенденцию скапливаться вместе. Это явление применяется для повышения производительности системы. Аппаратное обеспечение *контроллера памяти* ЦП заполняет кэш из последовательных столбцов в строке, потому что чаще всего требуются данные из последовательных расположений. Вместо одной коробки хлопьев мы кладем в пакеты сразу несколько и приносим их домой. Используя самый высокоскоростной режим доступа к памяти, процессоры обычно идут впереди, даже когда происходит промах кэша, вызванный непоследовательным доступом. *Промах кэша* (*cache miss*) — это не участница конкурса красоты мисс Кэш; это состояние, когда ЦП ищет в кэше что-то, чего там нет, и пытается извлечь это «ничего» из памяти. Аналогично, *попадание в кэш* (*cache hit*) — это когда ЦП находит в кэше то, что ищет. Должны же быть и хорошие термины.

Существует несколько уровней кэш-памяти, и они становятся больше и медленнее по мере удаления от ЦП (даже если они находятся на одной микросхеме). Они называются кэшами L1, L2 и L3, где L означает уровень (от level). Это как запасной морозильник в гараже и на складе. При этом существуют диспетчеры, отвечающие за работу службы контроля авиаперелетов. Есть множество логических схем, задача которых — упаковывать и распаковывать пакеты с продуктами, коробки и грузовики разных размеров, чтобы все это работало. На самом деле все это занимает значительную часть чипа. Иерархия памяти показана на рис. 5.14.

Дополнительные сложные настройки еще больше повысили производительность. Компьютеры включают в себя схему *предсказателя переходов*, которая угадывает результат выполнения инструкций условного ветвления, так что правильные данные могут быть *предварительно загружены* из памяти и в кэш, готовые к работе. Есть даже схема для обработки *внеочередного исполнения*. Это позволяет процессору выполнять инструкции в наиболее эффективном порядке, даже если это не тот порядок, в котором они встречаются в программе.

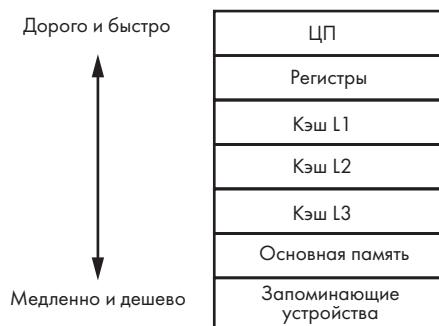


Рис. 5.14. Иерархия памяти

Поддержание *когерентности кэша* — особенно серьезная проблема. Представьте систему, состоящую из двух процессорных микросхем, каждая с четырьмя ядрами. Одно из этих ядер записывает данные в область памяти — точнее, в кэш, откуда они в конечном итоге попадут в память. Как другое ядро или процессор узнает, что пришла нужная версия данных из этой области памяти? Самый простой способ называется *сквозной записью*, это означает, что записи поступают непосредственно в память и не кэшируются. Но это уменьшает многие преимущества кэширования, поэтому существует много дополнительного аппаратного обеспечения для управления кэшем — его обсуждение выходит за рамки данной книги.

Сопроцессоры

Ядро процессора — довольно сложный элемент схемы. Вы можете освободить ядра процессора для общих вычислений, разделив общие операции по более простым частям оборудования, называемым *сопроцессорами*. Раньше сопроцессоры существовали только потому, что на одной микросхеме не хватало места для всех элементов. Например, были представлены сопроцессоры для вычислений с плавающей точкой, если на основном процессоре нельзя было уместить соответствующие аппаратные средства. Сегодня существуют встроенные сопроцессоры под многие задачи, включая специализированную обработку графики.

В этой главе мы говорили о загрузке программ в память для запуска, это обычно означает, что программы поступают из некоторой медленной и дешевой памяти, такой как дисковый накопитель. И мы увидели, что системы виртуальной памяти могут читать и записывать информацию на диски как часть подкачки. Из раздела «Блочные устройства» на с. 129 мы узнали, что диски не имеют байтовой адресации — они передают блоки размером 512 или 4096 байт. Это означает, что нужно копировать много информации между основной памятью и диском, поскольку никаких других вычислений не требуется. Копирование данных из одного места в другое — одна из самых больших затрат процессорного времени. Некоторые сопроцессоры ничего не делают, кроме перемещения данных. Они

называются *блоками прямого доступа к памяти* (direct memory access, DMA). Их можно настроить для выполнения таких операций, как «переместить много всего отсюда туда и сообщить, когда операция будет завершена». ЦП перекладывают большую часть рутинной работы на блоки прямого доступа к памяти, что делает ЦП свободным для выполнения более полезных операций.

Организация данных в памяти

Из программы в табл. 4.4 мы узнали, что память используется не только для инструкций, но и для данных. В нашем случае это *статические* данные, что означает, что необходимый объем памяти уже известен при написании программы. Ранее в этой главе мы видели, что программы также используют память для стеков. Эти области данных необходимо расположить в памяти так, чтобы между ними не возникало конфликтов.

На рис. 5.15 показана типичная организация устройств с архитектурой фон Неймана и гарвардской без MMU. Как видно, единственное отличие состоит в том, что на машинах с гарвардской архитектурой инструкции находятся в отдельной памяти.

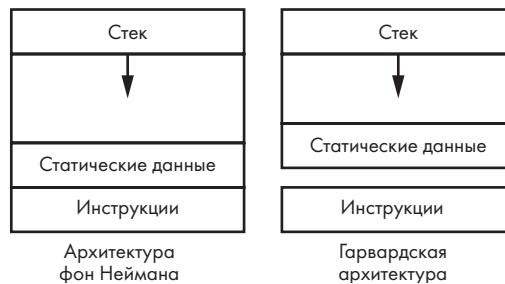


Рис. 5.15. Организация памяти

Существует еще один способ использования памяти программами. Большинству программ приходится иметь дело с *динамическими* данными, размер которых неизвестен до запуска программы. Например, система обмена мгновенными сообщениями не знает заранее, сколько сообщений ей нужно хранить или сколько памяти потребуется для каждого сообщения. Динамические данные обычно помещаются в память над статической областью, называемую *кучей*, как показано на рис. 5.16. Куча увеличивается по мере того, как требуется больше места для динамических данных, в то время как стек растет по направлению вниз. Важно убедиться, что стек и куча не пересекаются. Бывают незначительные



Рис. 5.16.
Организация памяти
с кучей

вариации; некоторые процессоры резервируют адреса памяти в начале или в конце памяти для векторов прерываний и регистров, управляющих периферийными устройствами ввода/вывода.

Такое расположение памяти можно встретить при использовании микрокомпьютеров, поскольку они обычно не имеют блоков MMU. Когда задействованы MMU, инструкции, данные и стек связываются с разными страницами физической памяти, размер которых можно регулировать по мере необходимости. Но та же структура памяти используется для виртуальной памяти, предоставленной программам.

Запуск программ

Вы видели, что компьютерные программы состоят из множества частей. В этом разделе вы узнаете, как все они объединяются.

Ранее я говорил, что программисты используют функции для повторного применения кода. И это еще не всё. Существует множество функций, которые полезны для более чем одной программы, например функции сравнения двух текстовых строк. Хорошо было бы иметь возможность использовать эти сторонние функции, вместо того чтобы каждый раз писать собственные. Один из способов сделать это — сгруппировать связанные функции в *библиотеки*. Доступно большое количество библиотек для всего на свете, от обработки строк до сложных математических вычислений и декодирования MP3.

Помимо библиотек, нестандартные программы обычно собираются по частям. Можно поместить всю программу в один файл, однако есть несколько веских причин этого не делать. Главная из них заключается в том, что это упрощает одновременную работу нескольких человек над одной и той же программой.

Но разбиение программ на части означает, что нужен какой-то способ *связать* или соединить все разрозненные части. Для этого каждая часть программы обрабатывается в промежуточном формате, предназначенному для этой цели, а затем запускается специальная программа-компоновщик, которая устанавливает все соединения. Многие промежуточные форматы файлов были созданы и улучшены по мере разработки программ. *Формат исполняемых и связываемых файлов* (Executable and Linkable Format, ELF) в настоящее время — самый популярный вариант. Этот формат включает разделы, похожие на объявления о поиске товаров. В разделе продаж можно найти что-то вроде объявления, в котором говорится: «У меня есть функция с именем `cube`». Точно так же можно увидеть объявление «Ищу переменную с именем `date`» в разделе «Требуется».

Компоновщик (linker) — это программа, которая *связывает между собой* все объявления, в результате чего получается программа, которая действительно может быть запущена. Но, конечно, при этом возникают сложности

с производительностью. Раньше вы относились к библиотекам как к одному из файлов, наполненных функциями, и связывали их с остальной частью программы. Это называлось *статическим связыванием*. Однако примерно в 1980-х годах люди заметили, что многие программы используют одни и те же библиотеки, что было прекрасным свидетельством ценности библиотек. Но они увеличивали размер каждой программы, которая их использовала, и появилось много копий одних и тех же библиотек, использующих ценную память. Это привело к созданию *общих библиотек*. Можно использовать MMU, чтобы несколько программ могли обращаться к одной копии библиотеки, как показано на рис. 5.17.

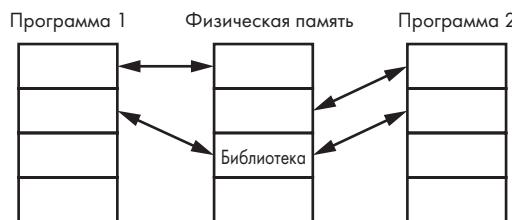


Рис. 5.17. Общая библиотека

Имейте в виду, что инструкции из такой библиотеки являются общими для программ, которые к ней обращаются. Необходимо проектировать библиотечные функции так, чтобы они использовали кучу и стек вызывающих программ.

У программ есть *точка входа* — адрес первой инструкции в программе. Эта инструкция не первая исполняемая при запуске программы, как это ни парадоксально. Когда все части программы связываются в *исполнимый файл*, включается дополнительная *библиотека среды выполнения* (runtime library). Код в ней запускается до достижения точки входа.

Библиотека среды выполнения отвечает за настройку памяти, что означает создание стека и кучи. Она также устанавливает начальные значения для элементов в области статических данных. Эти значения хранятся в исполняемом файле и должны быть скопированы в статические данные после получения памяти из системы.

Такая библиотека выполняет гораздо больше функций, кроме описанных выше, особенно для сложных языков. К счастью, сейчас вам не нужно изучать это подробно.

Мощность запоминающих устройств

До сих пор мы подходили к памяти с точки зрения производительности. Однако нужно учитывать кое-что еще — перемещение данных в памяти требует

некоторых *усилий*. Для настольных компьютеров это не проблема, но не нужно забывать о мобильных устройствах. И хотя время автономной работы не является проблемой в центрах обработки данных, например в тех, которые используются крупными интернет-компаниями, использование дополнительной мощности на тысячах машин в сумме влияет на систему в целом.

Сбалансировать потребление мощности и производительность непросто. Имейте в виду оба этих параметра при написании кода.

Выводы

Вы узнали, что работать с памятью не так просто, как можно было подумать после прочтения главы 4. Вы почувствовали, насколько дополнительные сложности влияют на простые процессоры и использование памяти. Теперь вы получили довольно полное представление о том, что можно найти в современном компьютере, за исключением ввода/вывода, о котором речь пойдет в главе 6.

6

Разбор связей



Компьютеры работают не просто так. Они принимают входные данные из различных источников, выполняют вычисления и производят выходные данные, которые используются огромным количеством устройств. Компьютеры могут общаться с людьми, разговаривать друг с другом или управлять предприятиями. Рассмотрим их работу подробнее.

Я кратко упомянул ввод и вывод (input/output, I/O) в разделе «Ввод и вывод» на с. 139, имея в виду ввод и вывод данных из ядра процессора. Сделать это не так уж и сложно; все, что нам нужно, — несколько *триггеров-защелок* (см. «Триггеры-защелки» на с. 113) для вывода и *буферов с тремя состояниями* (см. рис. 2.38) для ввода. Раньше считалось, что каждый компонент устройства ввода/вывода будет подключен к какому-либо биту на триггере-защелке или буфере, а компьютер будет выступать в роли кукловода, отвечающего за артикуляцию всех конечностей.

Снижение стоимости процессора изменило это. Многие сложные в прошлом устройства ввода/вывода теперь имеют собственные микропроцессоры. Например, можно купить трехосный акселерометр или датчик температуры с хорошим цифровым выходом всего за несколько долларов. Мы не будем обсуждать подобные устройства, потому что они не интересны с точки зрения программирования — интерфейс просто читает и записывает байты, как описано в спецификации устройства. Слишком простые вещи мы не учитываем. Вы можете написать код для устройства со встроенным процессором. Если вы возьметесь за разработку очередной расчески с выходом в интернет, то, скорее всего, у вас волосы встанут дыбом от сложности ее алгоритма управления.

В этой главе рассматриваются методы взаимодействия с некоторыми устройствами ввода/вывода, которые все еще интересны с точки зрения программирования. Также мы разберем *дискретизацию* — именно с ее помощью реальные аналоговые данные преобразуются в цифровую форму, используемую компьютерами, и наоборот.

Низкоуровневый ввод/вывод

Простейшие формы ввода/вывода включают соединение чего-либо с битами, которые могут быть прочитаны и записаны центральным процессором. Благодаря активному использованию эти формы начали развиваться в более сложные устройства. В этом разделе рассматриваются несколько примеров.

Порты ввода/вывода

Самый простой способ заставить компьютер с чем-то разговаривать — это подключить его к *порту ввода/вывода*. Например, компания Atmel (является частью компании Microchip) производит семейство небольших микроконтроллеров AVR. Они включают в себя большое количество встроенных устройств ввода/вывода. На рис. 6.1 мы подключаем некоторые устройства к *порту B*.

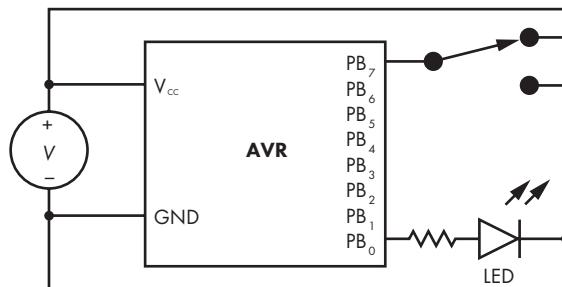


Рис. 6.1. Светодиод и переключатель, подключенные к порту B

Переключатель на рис. 6.1 уже должен быть вам знаком по главе 2. *LED* — это аббревиатура от английского *light-emitting diode* — светодиод. *Диод* — это полупроводниковое устройство, которое работает как турникет в парке развлечений: он пропускает электричество только в одном направлении, обозначенном полой стрелкой. У светодиодов есть приятный побочный эффект — они светятся.

Обратите внимание на резистор, включенный последовательно со светодиодом. Он предназначен для ограничения силы тока, протекающего через светодиод, чтобы ни он сам, ни PB_0 не сгорели. Номинал резистора можно рассчитать, используя закон Ома, введенный в главе 2. Допустим, V составляет 5 вольт. Одна из характеристик кремниевых бутербродов, о которых шла речь в разделе

«Транзисторы» на с. 94, заключается в том, что падение напряжения на одном из них составляет 0.7 вольта. В техническом описании микроконтроллера AVR указано, что выходное напряжение для логической 1, когда V составляет 5 вольт, равно 4.2 вольта. Мы хотим ограничить ток до 10 мА (0.01 А), потому что этого ожидает светодиод; AVR может потреблять 20 мА. Закон Ома гласит, что сопротивление — это напряжение, деленное на ток, поэтому $(4.2 - 0.7) \div 0.01 = 350$ Ом. Как видите, PB_7 может переключаться между напряжениями логического 0 и 1. Когда PB_0 установлен в 0, электричество не проходит. Электрический ток течет через светодиод, когда PB_0 установлен в 1, что заставляет диод светиться. Убедитесь, что вы прочитали техническое описание светодиода или любого другого используемого компонента, потому что некоторые характеристики, такие как падение напряжения, могут отличаться.

Порт В настраивается и управляется тремя регистрами, как показано на рис. 6.2. *DDRB* (data direction register), регистр направления передачи данных, определяет, является ли каждый вывод входом или выходом. *PORTB* — это триггер-защелка, которая хранит выходные данные. *PINB* считывает значения на выводах *PORTB*.

7	6	5	4	3	2	1	0	
DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
7	6	5	4	3	2	1	0	
PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
7	6	5	4	3	2	1	0	
PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB

Рис. 6.2. Регистры PORTB в AVR

Может показаться, что все действительно сложно, но, как видно на рис. 6.3, это просто еще одна схема стандартных строительных блоков: демультиплексоров, триггеров и буферов с тремя состояниями.

DDRB — это регистр направления передачи данных для порта В. Запись 1 в любой бит превращает связанный вывод в выход; если установлено значение 0 — во вход. *PORTB* — это выходная часть порта. Запись 0 или 1 в любой бит задает соответствующему выходу низкий или высокий логический уровень. Чтение *PINB* обеспечивает состояние связанных выводов, поэтому, если выводы 6 и 0 подтянуты вверх, а остальные — вниз, будет прочитано значение 01000001, или 0x41.

Как видите, вводить и выводить данные из микроконтроллера довольно просто. Можно прочитать положение переключателя, посмотрев на $PINB_7$ в регистре *PINB*. Включить и выключить светодиод можно, записав данные в $PORTB_0$ в регистре *PORTB*. Чтобы развлечь себя и своих друзей, можно написать простую программу, которая будет мигать светодиодом.

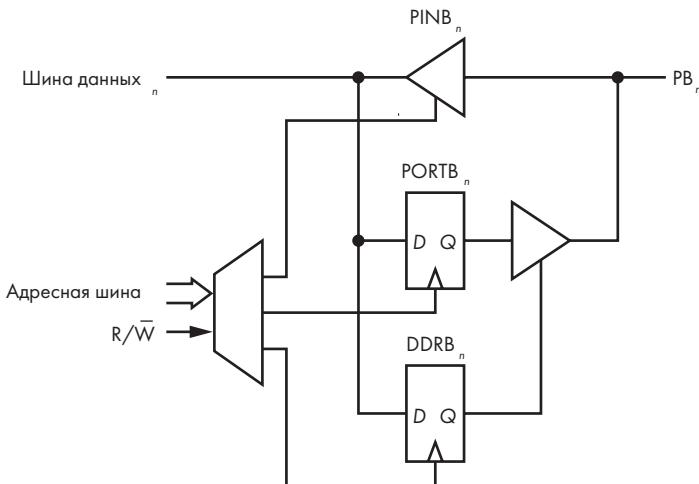


Рис. 6.3. Конструкция порта B в микроконтроллере AVR

Нажми на кнопку

На многих устройствах есть кнопки или переключатели. Из-за их конструкции компьютеру не так-то просто их прочитать, как может показаться. Простая кнопка состоит из пары электрических контактов и металлической части, которая соединяет их при нажатии кнопки. Взгляните на схему на рис. 6.4.

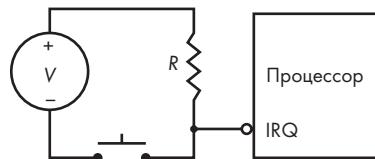


Рис. 6.4. Простая схема кнопки

R — это то, что называется *подтягивающим резистором*, как мы видели ранее на рис. 2.37. Когда кнопка не нажата, резистор подтягивает напряжение на входе *запроса прерывания процессора* (interrupt request, IRQ) до напряжения, подаваемого от источника питания V , что делает его логической 1. Когда кнопка нажата, резистор ограничивает ток, протекающий в цепи от источника питания, чтобы он не вышел из строя, — тогда на входе IRQ будет установлен логический 0.

Это выглядит просто, пока не взглянуть на рис. 6.5. Можно подумать, что после нажатия и отпускания кнопки сигнал IRQ будет выглядеть как на картинке слева, но на самом деле он больше похож на сигнал справа.

Что происходит? Когда металл, соединенный с кнопкой, соприкасается с контактами, он *дребезжит* и на короткое время теряет с ними связь. Он может

дребезжать некоторое время, прежде чем успокоится. Поскольку мы соединили кнопку с генерирующим прерывание входом процессора, одно нажатие кнопки может привести к нескольким прерываниям. Наверняка это не то, чего вы ожидали, и *дребезг* кнопки нужно устранить. (Можно приобрести кнопки без дребезга, но они обычно дороже.)



Рис. 6.5. Дребезг контактов кнопки

Простой способ устранения дребезга — настроить обработчик прерывания на таймер, а затем проверить состояние кнопки после истечения таймера, как показано на рис. 6.6. Здесь есть два варианта: установить таймер при первом прерывании или менять запущенный таймер на новый при каждом прерывании.

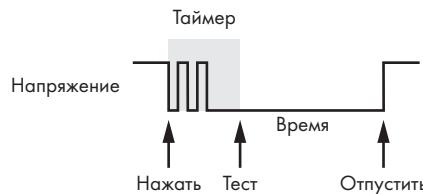


Рис. 6.6. Таймер для устранения дребезга кнопки

Это рабочий, но не самый лучший подход. Трудно выбрать значение таймера, потому что время дребезга кнопки может впоследствии измениться из-за механического износа. Наверняка у вас даже был ненавистный будильник, в котором кнопки были изношены до такой степени, что установить время было сложно. Кроме того, на большинстве устройств имеется более одной кнопки, и маловероятно, что у процессора достаточно входов прерываний, чтобы их можно было перемещать. Можно построить схему для разделения прерываний, но лучше всего сделать это бесплатно программным способом. В большинстве систем существует таймер, который может генерировать периодические прерывания. Можно использовать это прерывание для устранения дребезга кнопки.

Предположим, что у нас есть восемь кнопок, подключенных к порту ввода/вывода, как на рис. 6.1, и что состояние порта ввода/вывода доступно в переменной с именем `INB` — 8-битном `unsigned char`. Можно построить *фильтр с конечной импульсной характеристикой*, или КИХ (FIR, finite impulse

response) из массива `filter`, как показано на рис. 6.7. КИХ — это очередь; на каждом такте таймера мы отбрасываем самый старый элемент и переходим на новый. Мы выполняем операцию ИЛИ над всеми элементами массива, чтобы сформировать текущее состояние (`current`) как часть очереди из двух элементов; `current` перемещается в предыдущее значение (`previous`), прежде чем мы вычисляем новое `current`. Все, что нужно сделать, — выполнить операцию XOR для состояний `current` и `previous`, чтобы узнать, какие кнопки изменили состояние.

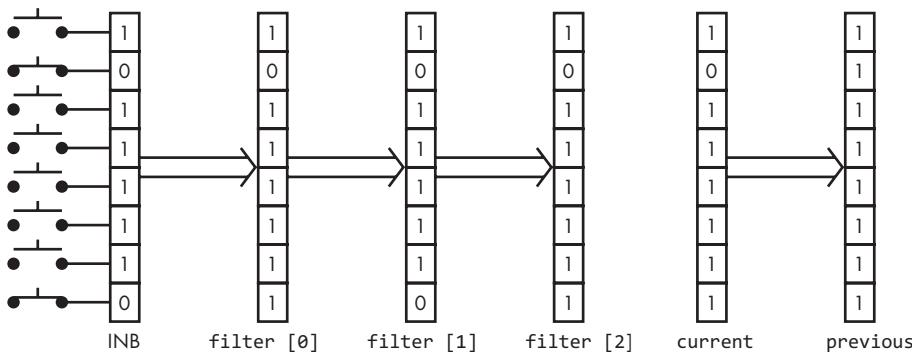


Рис. 6.7. Устранение дребезга кнопки с помощью КИХ-фильтра

Все это очень просто переводится в код, как показано на языке программирования C в листинге 6.1.

Листинг 6.1. Устранение дребезга кнопки с помощью КИХ-фильтра

```
unsigned char filter[FILTER_SIZE];
unsigned char changed;
unsigned char current;
unsigned char previous;

previous = current;
current = 0;

for (int i = FILTER_SIZE - 1; i > 0; i--) {
    filter[i] = filter[i - 1];
    current |= filter[i];
}

filter[0] = INB;
current |= filter[0];
changed = current ^ previous;
```

`FILTER_SIZE` — количество элементов в фильтре, выбор которого зависит от уровня шума кнопок и частоты прерываний таймера.

Да будет свет

Во многих приборах есть дисплей. Я не имею в виду компьютерные экраны — скорее речь о будильниках и посудомоечных машинах. Часто они имеют несколько световых индикаторов и иногда несколько простых числовых дисплеев.

Распространенным типом простых индикаторов является семисегментный дисплей, показанный на рис. 6.8. Эти дисплеи имеют семь светодиодов, расположенных в виде восьмерки, а иногда и дополнительную десятичную точку.

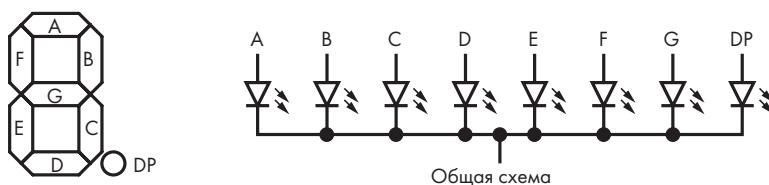


Рис. 6.8. Семисегментный дисплей

Для восьми светодиодов на дисплее требуется 16 электрических соединений (контактов). Но обычно они строятся не так; на одном конце каждого светодиода есть вывод, а на другом — общее соединение. Поскольку нам нужно управлять только одним концом, чтобы включить или выключить светодиод, это общее соединение позволяет сэкономить на выводах, что снижает стоимость дисплея. На рис. 6.8 показан *дисплей с общим катодом*, в котором все катоды связаны вместе, а каждый анод имеет свои собственные выводы.

Можно просто подключить аноды к выходным контактам на процессоре, а катоды — к заземлению либо отрицательному полюсу источника напряжения или источника питания. Высокий логический уровень (1) на выводе микроконтроллера включит соответствующий светодиод. На практике большинство процессоров не обеспечивают достаточный ток для работы, поэтому используется дополнительная схема драйвера. Часто используются выходы с открытым коллектором (показаны на рис. 2.36).

Программное обеспечение для управления одним из этих дисплеев довольно простое. Все, что нам нужно, — таблица, которая отображает числа (и, возможно, буквы) в соответствующих освещаемых сегментах. Но неудивительно, что при этом возникают осложнения. Единственный дисплей редко встречается; например, у будильника их четыре. Хотя можно подключить каждый дисплей к собственному порту ввода/вывода, маловероятно, что их будет так много. Решение состоит в том, чтобы *мультиплексировать* дисплеи, подключив аноды к порту А и катоды к порту В, как показано на рис. 6.9.

Аноды дисплея подключены параллельно; все сегменты А соединены между собой, все сегменты В соединены между собой и т. д. Выходы общих катодов

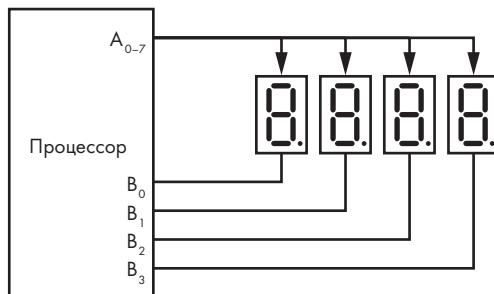


Рис. 6.9. Мультиплексные дисплеи

каждого дисплея подключаются к собственному выходу микроконтроллера. Сегмент дисплея может загореться только в том случае, если на его анод подана 1, а на катод — 0. Вы можете спросить, почему, например, сегменты А и В не загорятся, если А будет равно 1, а В равно 0. Помните, что буква *D* в аббревиатуре LED означает диод, а диоды — это улицы с односторонним движением тока.

Дисплеи работают благодаря *инерционности зрительного восприятия*. Не обязательно держать дисплей постоянно включенным, чтобы воспринимать его как включенный. Наши глаза и мозг будут считать, что он горит, если он включен всего на 1/24 секунды. Тот же эффект используется в фильмах и видеороликах. Все, что нам нужно сделать, — выбрать, какой дисплей включен, установив соответствующий катодный вывод на 0, а аноды сегмента — на то, что мы хотим отобразить¹. Мы можем переключать отображение в обработчике прерывания таймера, аналогичном тому, который мы использовали в предыдущем примере с кнопкой.

Свет, камера, мотор...

Обычно устройства включают в себя как кнопки, так и дисплеи. Как оказалось, можно сэкономить пару выводов, мультиплексируя кнопки и дисплеи. Допустим, мы имеем 12-кнопочную клавиатуру телефонного типа в дополнение к четырем дисплеям, как показано на рис. 6.10.

Что мы получили благодаря всей этой сложности? Теперь можно использовать только три дополнительных вывода для двенадцати кнопок. Все кнопки подтягиваются к логической единице с помощью подтягивающих резисторов. Нажатие кнопки не имеет никакого эффекта, если не выбран ни один дисплей, потому что все выходы В также равны 1. Если выбран крайний левый дисплей, B_0 имеет низкий уровень, и нажатие любой кнопки в верхнем ряду приведет к понижению уровня соответствующего входа С и т. д. Поскольку дисплей и кнопки

¹ Этот принцип работы семисегментного дисплея называется динамической индикацией. — Примеч. науч. ред.

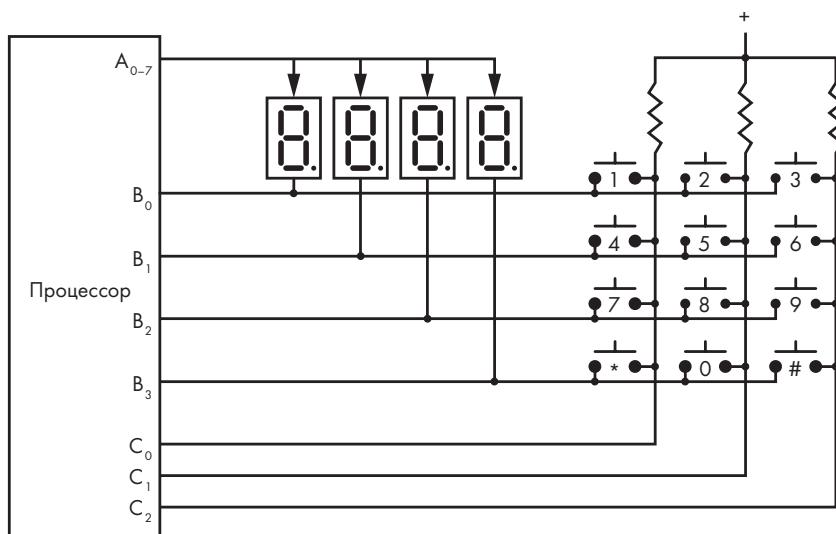


Рис. 6.10. Мультиплексированные кнопки и дисплеи

сканируются одним и тем же набором сигналов, код, выполняющий сканирование, может быть объединен в обработчике прерывания таймера.

Обратите внимание, что на рис. 6.10 представлена упрощенная схема. На практике контакты В должны быть устройствами с открытым коллектором или *открытым стоком* (см. «Варианты выходов» на с. 101); в противном случае, если были нажаты две кнопки в разных строках, но в одних и тех же столбцах, мы бы соединили 1 с 0, что могло бы повредить электронные компоненты. Однако обычно это так не реализуется, поскольку вышеупомянутая схема драйвера дисплея делает это за нас.

Можно узнать, спроектировано ли устройство так, как показано на рис. 6.10, нажав одновременно несколько кнопок и наблюдая за дисплеями. Дисплеи будут выглядеть странно. Подумайте почему.

Светлые идеи

Ваш будильник может иметь функцию регулировки яркости дисплея. Как это устроено? Можно изменять *режим работы* дисплея согласно рис. 6.11.

Каждый дисплей светится четверть времени, как показано в левой части рис. 6.11. Правая часть демонстрирует, что каждый дисплей светится только одну восьмую времени; ни один из дисплеев не горит половину времени. В результате дисплеи справа кажутся примерно вдвое тусклее дисплеев слева. «Яркость» связана со средним временем работы дисплея. Обратите внимание, что зависимость между режимом работы и воспринимаемой яркостью вряд ли будет линейной.

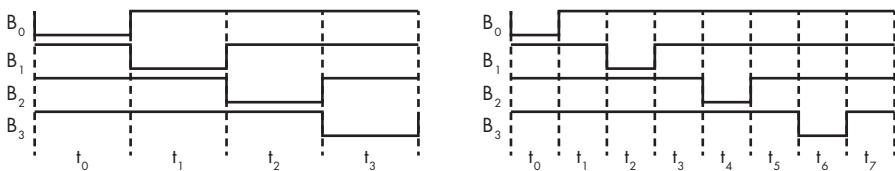


Рис. 6.11. Режим работы

2ⁿ оттенка серого

Обычная задача датчика — определить положение врачающегося вала, например двигателей, колес и ручек. Можно определить положение, используя переключатели на валу или черные и белые точки, которые можно прочитать с помощью фотодатчика. Какой бы подход мы ни выбрали, каждое положение вала будет закодировано как двоичное число. Кодировщик может выглядеть как на рис. 6.12, если нас интересуют восемь различных позиций. Если белые секторы равны 0, а черные секторы — 1, то видим, как будет читаться значение позиции. Радиальные линии не являются частью кодировщика; они нужны только для того, чтобы диаграмму было легче понять.

Как обычно, схема кажется простой, но это не так. В этом случае проблема заключается в механических допусках. Обратите внимание, что даже при идеально выровненном кодировщике у нас все равно будут проблемы, связанные с различиями в задержке распространения в схеме,читывающей каждый бит. Что произойдет, если кодировщик выровнен не идеально, как на рис. 6.13?

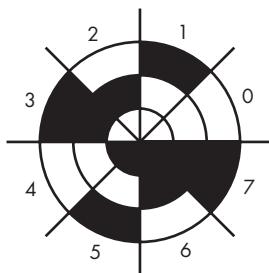


Рис. 6.12. Двоичный датчик угла поворота

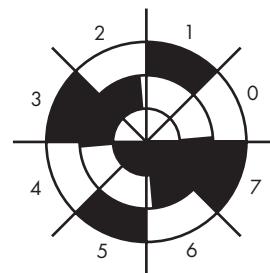


Рис. 6.13. Погрешность центровки датчика угла поворота

Вместо того чтобы прочитать ожидаемое 01234567, мы получаем 201023645467. Американский физик Фрэнк Грей (Frank Gray) (1887–1969) из Bell Telephone Laboratories рассмотрел эту проблему и придумал способ кодирования, в котором для каждой позиции изменяется только значение одного бита. Для 3-битного кодировщика, который мы рассматривали, одноименный код Грея равен 000, 001, 011, 010, 110, 111, 101, 100. Код можно легко преобразовать в двоичный с помощью небольшой таблицы. На рис. 6.14 показана версия датчика с кодом Грея.

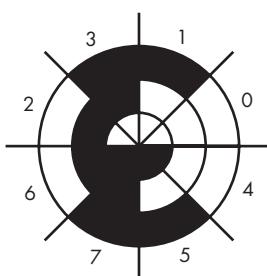


Рис. 6.14. Датчик угла поворота с кодом Грея

Квадратура

В 2-битных кодах Грея есть уловка, которую можно использовать, когда не обязательно знать абсолютное положение чего-либо, но нужно знать, когда положение изменяется и в каком направлении. Некоторые ручки на приборной панели автомобиля, например регулятор громкости стереосистемы, скорее всего, будут работать таким образом. Можно это проверить, если поворот ручки при выключенном зажигании не дает никакого эффекта после запуска двигателя. Метод называется *квадратурным кодированием*, потому что он использует четыре состояния. Двухбитный код Грея повторяется несколько раз. Например, существуют дешевые квадратурные кодировщики, которые работают до 1/4096 оборота. Для квадратуры нужны всего два датчика, по одному на каждый бит. Абсолютный кодировщик на 4096 позиций потребует 12 датчиков.

Квадратурный сигнал показан на рис. 6.15.

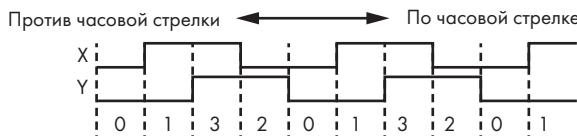


Рис. 6.15. Квадратурный сигнал

Как видите, когда вал вращается по часовой стрелке, получается последовательность 0132; против часовой стрелки — 2310. Можно сформировать 4-битное число из текущей и предыдущей позиций. Это число указывает направление вращения, как показано в табл. 6.1.

Обратите внимание, что это конечный автомат, где комбинированное значение является состоянием.

Что получится, если взять пару квадратурных энкодеров, расставить их под углом 90 градусов друг к другу и воткнуть в середину резиновый шарик? Компьютерная мышь.

Таблица 6.1. Обнаружение квадратурного вращения

Текущее значение	Предыдущее значение	Комбинированное значение	Расшифровка
00	00	0	Неверные данные
00	01	1	По часовой стрелке
00	10	2	Против часовой стрелки
00	11	3	Неверные данные
01	00	4	Против часовой стрелки
01	01	5	Неверные данные
01	10	6	Неверные данные
01	11	7	По часовой стрелке
10	00	8	По часовой стрелке
10	01	9	Неверные данные
10	10	а	Неверные данные
10	11	б	Против часовой стрелки
11	00	с	Неверные данные
11	01	д	Против часовой стрелки
11	10	е	По часовой стрелке
11	11	ф	Неверные данные

Параллельная связь

Параллельная связь — это расширение ситуации, которую мы видели ранее, включая и выключая светодиоды. Мы могли подключить восемь светодиодов к порту В и мигать кодами символов ASCII. *Параллельное соединение* означает, что у нас есть провод для каждого компонента и мы можем управлять ими всеми одновременно.

На старых компьютерах может быть *параллельный порт* IEEE 1284. Такие порты обычно использовались для принтеров и сканеров до появления *универсальной последовательной шины* (Universal Serial Bus, USB). И да, на параллельном порте было восемь строк данных, поэтому можно было отправлять коды символов ASCII.

В связи со всем этим возникает проблема: как узнать, достоверны ли данные? Допустим, вы отправляете символы ABC. Как узнать, какой символ появится следующим? Отследить это не получится, потому что последовательность

может измениться на *AAAC*. Один из способов — получить еще один сигнал прерывания для проверки состояния. В IEEE 1284 для этой цели предусмотрен *стробирующий* сигнал. На рис. 6.16 данные в битах с 0-го по 7-й действительны всякий раз, когда стробирующий импульс имеет низкий уровень или равен 0.

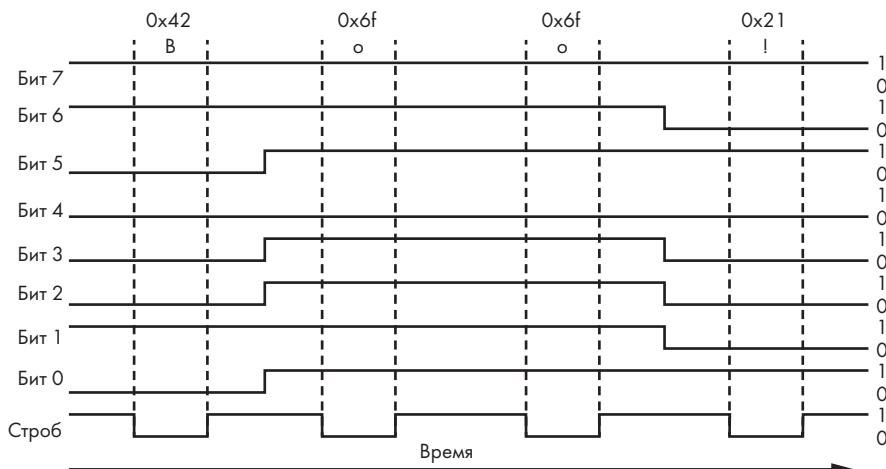


Рис. 6.16. Синхронизация передачи параллельных данных с помощью стробирующих импульсов

Еще один параллельный интерфейс, который практически ушел на второй план, — это IDE. Он использовался для передачи данных на старых дисковых накопителях.

Параллельные интерфейсы дороги, потому что для них требуется много выводов I/O, выводов разъемов и проводов. Параллельный порт имел 25-контактный разъем и большой толстый кабель. У IDE было 40 проводов. Существует предел скорости передачи сигнала по проводу, и при его превышении потребуется еще больше проводов.

Последовательная связь

Предпочтительно иметь возможность устанавливать соединение, используя как можно меньше проводов, потому что они стоят денег, особенно когда речь идет о больших расстояниях. Два провода — это минимальное необходимое количество, потому что, как вы узнали в главе 2, нам нужен путь для обратного электрического сигнала. Этот обратный путь опущен на диаграммах для простоты.

Как отправить восемь сигналов по одному проводу? Подсказка есть на диаграмме времени на рис. 6.16. Несмотря на то что каждый бит находится на своем собственном проводе, символы разнесены во времени. Биты мы тоже можем разнести по времени.

Я говорил о регистрах сдвига в разделе «Сдвиг» на с. 143. На передающей стороне стробирующий, или *тактовый*, сигнал сдвигает все биты на одну позицию и отправляет бит, оказавшийся в конце, в линию. На принимающей стороне генератор тактовых сигналов сдвигает все биты на одну позицию и переводит состояние канала связи в новую освобожденную позицию, как показано на рис. 6.17.

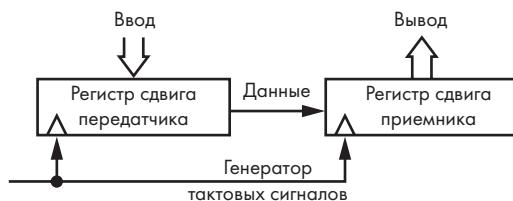


Рис. 6.17. Последовательная связь с использованием регистров сдвига

Можно использовать счетчик, чтобы получить уведомление, когда дойдем до 8 бит, и тогда обработать значение. Этот подход требует двух проводов, а не одного, и он часто приводит к ошибкам. При этом нужно, чтобы передатчик и приемник были *синхронизированы* (in sync, что не имеет ничего общего с одноименной поп-группой). Достаточно пропустить один тактовый сигнал, и все пойдет прахом. Можно добавить третий провод для сообщения о новом символе, но наша цель — минимизировать количество проводов.

Давным-давно (в начале XX века) телеграф скрестили с пишущей машинкой, что привело к появлению *телетайпа* — телеграфного буквопечатающего аппарата, который позволял печатать символы на удаленной печатной машинке. Изначально телетайпы использовались для передачи информации о фондовом рынке по телеграфным проводам.

Данные отправлялись по последовательному *протоколу* (набору правил), который работал с использованием только одного провода в дополнение к обратному проводу. В этом протоколе замечательно то, что он работал как таймер на соревнованиях по плаванию. Все участники запускают свои личные таймеры при срабатывании стартового пистолета, и они находятся достаточно близко друг к другу, чтобы это сработало. Рисунок 6.18 иллюстрирует работу протокола.

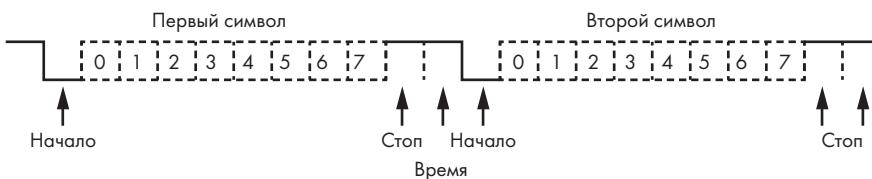


Рис. 6.18. Передача данных с помощью меток и пробелов

Линия здесь находится в состоянии 1, или *высоком*, когда ничего не происходит. Высокое состояние называется *меткой*, а низкое — *пробелом*. Это именование появилось из-за того, что раннее телеграфное оборудование либо делало отметку, либо оставляло пустое место на полосе бумаги. Переход из высокого уровня в низкий, отмеченный как «Начало» на рис. 6.18, работает как стартовый пистолет и называется *стартовым битом*. После стартового бита отправляются 8 бит данных. Символ заканчивается парой старших *стоповых битов*. Каждому биту отводится одинаковое количество времени. Могут возникнуть ошибки синхронизации, но все, что нужно сделать передатчику в таком случае, — остановиться на *время отправки одного символа*, и приемник синхронизируется заново. Мы разделяем время так, чтобы получить слот для каждого бита, а затем мультиплексируем данные по одному проводу. Этот метод, называемый *мультиплексированием с временным разделением*, может быть реализован с использованием селектора (см. «Построение селекторов» на с. 108) вместо регистра сдвига. Кстати, скорость в битах в секунду известна как скорость в бодах, названная в честь французского инженера Эмиля Бодо (Émile Baudot) (1845–1903).

Телетайпы были потрясающими машинами. В них не было никакой электроники, и они работали за счет вращения вала двигателя. Электромагнит отпускал вал, когда появлялся начальный бит, чтобы тот мог вращаться. В каждом месте вращения для смены положения бита перемещались всевозможные кулачки, рычаги и толкатели, и в конечном итоге металлический символ дотрагивался до окрашенной ленты, а затем и до листа бумаги. Вы знали, что поступило новое сообщение, если аппарат начинал грохотать. Клавиатура работала аналогично. Нажатие клавиши запускало вращение вала, который перемещал электрический контакт, в зависимости от того, какие клавиши были нажаты, для генерации кода ASCII.

Еще один крутой трюк, называемый *полудуплексным* соединением, заключается в том, что передатчик и приемник используют один и тот же провод. Одновременно говорить можно только по одному из них, иначе в результате получится тарабарщина. Вот почему радисты произносили что-то вроде «прием». Вы знаете все о полудуплексной связи, если когда-либо пользовались радио. Если несколько передатчиков активны одновременно, возникает *коллизия* — искажение данных. *Полнодуплексное* соединение — это два провода, по одному на каждое направление.

Все схемы для реализации этого принципа в итоге стали доступны в одной интегральной микросхеме под названием *UART*, что означает Universal Asynchronous Receiver-Transmitter — *универсальный асинхронный приемник-передатчик*. UART также можно реализовать в программном обеспечении, используя так называемый *бит-бэндинг* (bit-banging).

Стандарт под названием *RS-232* определил уровни напряжения, используемые для меток и пробелов на старых последовательных портах, а также множество

дополнительных сигналов управления. Сейчас он в значительной степени заменен USB, хотя вариант под названием *RS-485*, который использует дифференциальную передачу сигналов (см. рис. 2.32) для большей помехоустойчивости, используется в промышленных средах. Параллельный интерфейс IDE для дисков был заменен последовательным эквивалентом *SATA*. Электроника сейчас достаточно быстрая, чтобы выполнять операции последовательно, хотя раньше они происходили параллельно. К тому же провода остаются дорогими. В мире не хватает меди, которая добывается как полезное ископаемое и используется в качестве проводника. Основной источник меди сейчас — переработка существующих медных изделий. Чипы в основном состоят из кремния, который в большом количестве содержится в песке.

Существует ряд последовательных интерфейсов, предназначенных для подключения периферийных устройств к небольшим микрокомпьютерам. К ним относятся *SPI*, *I2C*, *TWI* и *OneWire*.

Поймать волну

Один из минусов передачи данных с метками и пробелами заключается в том, что она не подходит для очень больших расстояний. Данный подход не работает при использовании телефонных линий по причинам, выходящим за рамки этой книги. Это очень важно, потому что после замены телеграфа более совершенными технологиями связь на дальних расстояниях стала возможна только с помощью телефона или радио. Проблема передачи данных с метками и пробелами решается с помощью того же метода, который заставляет работать радио.

На Вселенную влияют самые разные волны. Они бывают морскими, звуковыми, световыми, также существуют микроволны и множество других разновидностей. Волна-основа — это *синусоида*. Все другие формы волн представляют собой комбинации синусоидальных волн. Вы получите синусоидальную волну, построив график зависимости высоты точки на окружности от угла, — нечто похожее на рис. 6.19.

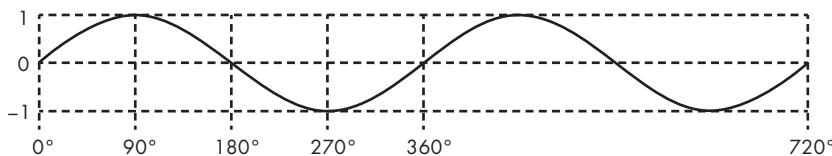


Рис. 6.19. Синусоидальная волна

Высота синусоиды — это *амплитуда*. Число пересечений с нулем в одном направлении в секунду — это *частота*, измеряемая в *герцах*, названных по

имени немецкого физика Генриха Герца (1857–1894). Герц сокращенно обозначается как Гц и является синонимом *циклов в секунду*. Расстояние между двумя пересечениями с нулем в одном направлении — это *длина волны*. Они связаны следующим образом:

$$\lambda = \frac{v}{f}$$

В этом уравнении λ — длина волны в метрах, f — частота в герцах, а v — скорость волны в среде, в которой она движется. Радиоволны движутся со скоростью света. Чем выше частота, тем короче длина волны. В качестве ориентира — частота ноты до первой октавы составляет около 261 Гц.

Если вы поразмышляете об этом еще немного, то поймете, что свойства разных волн отличаются. Звуковые волны не распространяются очень далеко, их останавливает вакуум, но они огибают препятствия на пути. Световые волны проходят очень длинный путь, но их останавливают стены. Некоторые частоты радиоволн проходят сквозь стены, а другие — нет. Между ними много различий.

Пришло время серфить. Найдем нужную волну и покатаемся. Назовем эту волну *несущей* и будем *модулировать* или изменять ее в зависимости от нужного сигнала, например *формы волны* с метками и пробелами.

AT&T представила набор данных Bell 103A в начале 1960-х годов. Он обеспечивал полнодуплексную связь на колоссальной скорости 300 бод по телефонной линии с использованием четырех звуковых частот; каждый конец соединения получил свою пару сигналов с метками и пробелами. Это называется *частотной манипуляцией* (frequency shift keying, FSK), потому что частота сдвигается вместе с метками и пробелами (рис. 6.20).

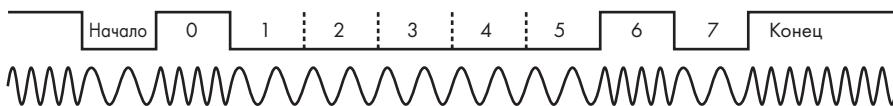


Рис. 6.20. Частотная манипуляция — буква А в ASCII

Принимающая сторона должна превратить звук обратно в метки и пробелы, что называется *демодуляцией* в противоположность модуляции. Устройства, которые это делают, называются *модемами*. Станные шумы, которые вы слышите, когда кто-то выходит в интернет по дилл-апу или отправляет факс в дурацком фильме, — это частоты, используемые модемами.

Универсальная последовательная шина

Стандарт USB не так интересен, но о нем стоит упомянуть, потому что он очень распространен. В нем больше несовместимых и сложных в использовании

разъемов, чем в любом другом стандарте, и, возможно, он более важен для зарядки устройств, чем для передачи данных.

USB заменил многие громоздкие разъемы на компьютерах, бывшие в обиходе в середине 1990-х годов, такие как PS/2, RS-232 и параллельные порты с одним четырехпроводным разъемом. Мы использовали два провода питания и витую пару для передачи данных с применением дифференциальной сигнализации. USB повторяет модель, с которой мы столкнемся в ближайшем будущем: «не могу остановиться на достигнутом», так что теперь USB Type-C имеет до 24 проводов, что совсем немного по сравнению со старым параллельным портом.

USB все же не так прост. Существует *контроллер*, который отвечает за обмен данными с *конечными точками*. Передача данных структурирована; это не просто копание в непонятной информации. Используется обычная техника: данные передаются *пакетами*, которые похожи на пакеты, отправляемые по почте. Пакеты содержат *заголовок* и необязательную *полезную нагрузку*. Заголовок — это, по сути, информация, которую можно найти на посылке: откуда она пришла, куда идет, класс почтовых отправлений и т. д. Полезная нагрузка — это содержимое пакета.

USB обрабатывает аудио и видео посредством *изохронной передачи*. Конечная точка может запросить зарезервировать определенную часть *полосы пропускания* (скорости передачи данных), что дает гарантию передачи данных. Контроллер отклоняет запрос, если пропускной способности недостаточно.

Сети

Трудно составить четкое представление о современном мире сетевых технологий, не зная их истоков. Меня сводит с ума, когда моя дочь говорит: «Wi-Fi не работает» или «Интернет не работает», потому что это не одно и то же. Попытки объяснить ей это наталкиваются на фирменное подростковое закатывание глаз и взмах волос.

Для описания сетей используются две общие классификации. *Локальная сеть* (local area network, LAN) — это сеть, охватывающая небольшое пространство, например дом или офис. *Глобальная сеть* (wide area network, WAN) охватывает большую географическую область. Эти термины несколько расплывчаты, поскольку нет точного определения *малых* и *больших* областей.

Первоначальное представление сети — это телеграфная сеть, которая превратилась в сеть телефонную. Я еще не веду речь о компьютерной сети, потому что компьютеров в то время не существовало. Первоначальная телефонная сеть была сетью *с коммутацией каналов*. Во время звонка между двумя сторонами их провода фактически соединялись вместе, образуя цепь. Выражение «*с коммутацией*» означало, что это соединение существовало только во время разговора. После завершения вызова можно было создавать новые цепи.

За некоторыми исключениями, такими как все еще существующие стационарные телефоны, телефонная система теперь представляет собой *сеть с коммутацией пакетов*. Я упомянул пакеты в последнем разделе. Обмен данными делится на пакеты, которые включают адреса отправителя и получателя. Пакеты могут совместно использовать провода с мультиплексированием с временным разделением (описано ранее в разделе «Последовательная связь» на с. 198), что позволяет более эффективно использовать каналы; это стало возможным, когда количество данных, которые можно было передать по проводам, стало больше, чем нужно только для голоса.

Одна из первых компьютерных сетей входила в состав *Semi-Automatic Ground Environment (SAGE)*, системы обороны времен холодной войны. Она использовала модемы в телефонной сети для связи между объектами.

Многие организации начали экспериментировать с локальными сетями в конце 1960-х годов. Например, моя лаборатория в Bell разрабатывала графические терминалы, которые были подключены к компьютеру Honeywell DDP-516 нашего отдела через локальную сеть, называемую *кольцом*. В то время периферийные устройства, такие как ленточные накопители и принтеры, были очень дорогими, и у большинства отделов не было своей собственной периферии. Но все нужные устройства были доступны в главном вычислительном центре. Наш компьютер был подключен к модему, и когда ему требовалось что-то, чего у него не было, он просто звонил в компьютерный центр. Фактически это была глобальная сеть. Мы могли не только посыпать документы на печать, но и отправлять программы для запуска — и компьютерный центр соединялся с нашей машиной для передачи результатов.

Аналогичная деятельность происходила во многих исследовательских лабораториях и компаниях. Было изобретено много разных локальных сетей. Однако все они существовали в собственных вселенных и не могли разговаривать друг с другом. Модемы и телефонные линии были основой для глобальной связи.

Набор компьютерных программ, разработанных в Bell Labs, под названием *UUCP* (от *UNIX-to-UNIX copy*) был выпущен для внешнего мира в 1979 году. UUCP позволял компьютерам звонить друг другу для передачи данных или удаленного запуска программ. Он лег в основу первых систем электронной почты и новостей, таких как *USENET*. Эти системы представляли собой интересную лазейку. Если вы хотите отправить данные внутри страны, они будут перескакивать с компьютера на компьютер, пока не дойдут до места назначения. Это обычно позволяло избежать платы за междугороднюю телефонную связь.

Между тем ARPA (Advanced Research Projects Agency — Управление перспективных исследований Министерства обороны США) финансировало разработку ARPANET, глобальной сети с коммутацией пакетов. ARPANET превратилась в интернет в 1990-х годах. Сегодня большинство людей воспринимают интернет как должное и, как и моя дочь, вероятно, думают, что это синоним сети.

Но его настоящая природа указана прямо в названии: сокращение «*интер*» — от *inter* — «взаимодействие», «*нет*» — от *net* — «сеть». Интернет — это сеть сетей, глобальная сеть, которая соединяет локальные сети.

Современные локальные сети

Многие другие вещи, которые мы считаем само собой разумеющимися в наши дни, были изобретены в исследовательском центре Хорох в Пало-Альто (Palo Alto Research Center, PARC) в середине 1970-х годов. Например, американский инженер-электротехник по имени Боб Меткалф (Bob Metcalfe) изобрел *Ethernet*, который представляет собой локальную сеть, потому что не рассчитан на большие расстояния.

ПРИМЕЧАНИЕ

Более подробную информацию об истории PARC можно найти в книге Адель Голдберг (Adele Goldberg) «A History of Personal Workstations» (Addison-Wesley, 1988).

Изначально Ethernet был полудуплексной системой. Каждый компьютер был подключен к одному проводу. У каждого компьютерного сетевого интерфейса был уникальный 48-битный адрес, называемый *адресом управления доступом к среде*, или MAC-адресом (Media Access Control, MAC), и он все еще актуален сегодня. Данные организованы в пакеты, называемые *фреймами*, размером около 1500 байт. Фреймы имеют *заголовок*, который включает адрес отправителя, адрес получателя и некоторые проверки ошибок (например, проверки циклическим избыточным кодом, или CRC, как описано в разделе «Обнаружение и исправление ошибок» на с. 133) вместе с полезными данными.

Обычно один компьютер говорит, а другие слушают. Компьютеры, не совпадающие с MAC-адресом получателя, игнорируют данные. Каждая машина слушала, что происходило, и не передавала данные, если это делал кто-то другой. Когда компьютеры начинали передавать данные одновременно, коллизия приводила к искажению пакетов — та же ситуация, что и при коллизиях полудуплексных систем. Одним из больших нововведений Меткалфа стал принцип «*случайный откат и повторный запуск*». Машина, которая пыталась «поговорить», ожидала в течение случайного времени, а затем пыталась повторно отправить пакеты.

Ethernet все еще используется сегодня, хотя и не в полудуплексной версии. Теперь компьютеры подключены к *маршрутизаторам*, которые отслеживают, какая машина в каком соединении находится, и направляют пакеты в нужные места. При этом коллизии больше не возникают. Wi-Fi — это, по сути, версия Ethernet, в которой вместо проводов используется радио. Bluetooth — еще одна популярная система LAN. Представьте его как версию USB, которая сменила провода на радио.

Интернет

Как вам теперь известно, интернет на самом деле не физическая сеть; это набор многоуровневых протоколов. Он разработан таким образом, что нижние уровни, определяющие физическую сеть, могут быть заменены, не затрагивая верхних уровней. Такая организация позволяет интернету работать при помощи проводов, радио, оптоволоконных кабелей и любых новых технологий.

TCP/IP

Протокол управления передачей/интернет-протокол (Transmission Control Protocol/Internet Protocol, TCP/IP) — это пара протоколов, на которых построен интернет. IP получает пакеты для передачи из одного места в другое. Эти пакеты, называемые *дейтаграммами*, похожи на телеграммы для компьютеров. Как и в случае с настоящими телеграммами, отправитель не знает, когда адресат получил сообщение и пришло ли оно ему вообще. TCP накладывается поверх IP и обеспечивает надежную доставку пакетов. Это довольно сложная работа, потому что большие сообщения охватывают множество пакетов, которые, возможно, поступают не по порядку, поскольку могли идти по разным маршрутам, — это похоже на заказ нескольких вещей и их отправку в разных коробках. Коробки могут не прибыть в один и тот же день, или их могут доставлять разные перевозчики.

IP-адреса

Каждый компьютер в интернете имеет уникальный адрес, известный как *IP-адрес*. В отличие от MAC-адресов IP-адреса, не привязаны к оборудованию и могут изменяться. Система IP-адресов — это иерархическая система, в которой кто-то выдает блоки адресов кому-то, кто также выдает блоки адресов, и т. д., пока блок адресов не передаст к тому, кто предоставит компьютеру собственный адрес.

Интернет в основном работает по *IPv4*, IP версия 4, которая использует 32-битный адрес. Адреса представлены в *октетной* записи *xxx.xxx.xxx.xxx*, где каждый *xxx* — это 8 из 32 бит, записанных в десятичном формате. Это более 4 миллиардов адресов, но и этого недостаточно. Теперь, когда у всех есть адреса настольных компьютеров, ноутбуков, планшетов, мобильных телефонов и других устройств, свободных адресов не осталось. Следовательно, мир постепенно переходит на *IPv6*, работающий с 128-битными адресами.

Система доменных имен

Как вас найти, если ваш адрес изменится? Этим занимается *система доменных имен* (Domain Name System, DNS), которая похожа на телефонную книгу — для тех, кто помнит, что это такое. DNS сопоставляет имена с адресами. Она знает, что сайт *whitehouse.gov* имеет IP-адрес 23.1.225.229, в то время когда я пишу эти

строки. Это что-то вроде адресной книги в телефоне, которую нужно постоянно обновлять; DNS позаботится обо всем при переезде.

Всемирная паутина

Многие другие протоколы построены на основе TCP/IP, например *Simple Mail Transfer Protocol* (SMTP), обеспечивающий работу электронной почты. Одним из наиболее часто используемых протоколов является *HTTP* – сокращение от *HyperText Transfer Protocol*, который используется для веб-страниц, наряду с *HTTPS*, где S означает «безопасный» (от secure).

Гипертекст – это просто текст со ссылками. Американский инженер Вэнвар Буш (Vannevar Bush) (1890–1974) придумал его в 1945 году. Тим Бернерс-Ли (Tim Berners-Lee), ученый из CERN (European Organization for Nuclear Research), изобрел Всемирную паутину, чтобы физики могли обмениваться информацией.

Стандарт HTTP определяет, как *веб-браузеры* взаимодействуют с *веб-серверами*. Веб-браузеры – это то, что вы используете для просмотра веб-страниц. Веб-серверы отправляют эти страницы по запросу. Веб-страницы обнаруживаются и выбираются по *унифицированному указателю ресурсов* (Uniform Resource Locator, URL), адресу веб-сайта в адресной строке браузера. Это способ поиска нужной информации, включающий доменное имя компьютера в интернете и описание того, где найти информацию на компьютере.

Веб-страницы обычно создаются на *HTML* (сокращение от *HyperText Markup Language* – «язык гипертекстовой разметки»), наиболее распространенном языке для их написания. К HTML с течением времени прилипло множество всего, и теперь это довольно адская смесь. Подробнее об этом в главе 9.

Аналоговые устройства в цифровом мире

Компьютеры используются во многих развлекательных устройствах, от аудиоплееров до телевизоров. Возможно, вы заметили, что цифровые фотографии выглядят не очень хорошо, если их увеличить выше определенного предела. Наше восприятие звука и света непрерывно, но компьютеры не могут хранить непрерывные данные. Данные нужно дискретизировать, что означает необходимость снимать показания в определенные моменты во времени и/или в пространстве. Затем из этих выборок требуется восстановить аналоговый (непрерывный) сигнал для воспроизведения.

Дискретизация не нова. Даже во времена немого кино сцена снималась со скоростью около 16 кадров в секунду. Существует целая область науки под названием *дискретная математика*, которая занимается дискретизацией. Дискретно, разумеется.

ПРИМЕЧАНИЕ

В интернете есть хорошее видео под названием Episode 1: A Digital Media Primer for Geeks. Оно представляет собой понятное введение в дискретизацию. Есть и вторая серия — она тоже неплоха, но может немного запутать. Хотя все сказанное там технически верно, все описанные принципы относятся только к моно-, а не к стерео-сигналам. Ведущий подразумевает, что они подходят для стерео, но это не так.

Мы говорили о различиях между аналоговыми и цифровыми сигналами еще в главе 2. Эта книга посвящена цифровым компьютерам, и многие реальные приложения требуют, чтобы компьютеры генерировали или интерпретировали аналоговые сигналы или делали и то и другое. В следующих разделах обсуждается, как это происходит.

Цифро-аналоговое преобразование

Как сгенерировать аналоговое напряжение на основе цифрового числа? Быстрый и правильный ответ: с помощью цифро-аналогового преобразователя. Как его построить?

Вернемся к рис. 6.1, где представлен светодиод, подключенный к порту ввода/вывода. На рис. 6.21 светодиод подключается к каждому из восьми контактов порта B.

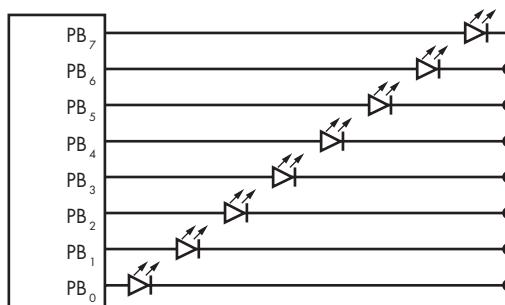


Рис. 6.21. Цифро-аналоговый преобразователь с использованием светодиодов

Теперь можно генерировать девять различных уровней света — от восьми выключенных светодиодов до восьми включенных. Но 9 уровней из 8 бит — не очень хорошее использование битов; из 8 бит мы должны получить 256 уровней. Как? Так же, как и с числами. На рис. 6.22 один светодиод подключается к биту 0, два — к биту 1, четыре — к биту 2 и т. д.

Это целая куча светодиодов. Повесьте эту схему на воздушный шар, чтобы получился светодиодный дирижабль (LED zeppelin). Двигаясь дальше, вы поймете, что это отражение двоичного представления чисел. Бит 1 дает в два раза больше света, чем бит 0, бит 2 — в четыре раза больше и т. д.

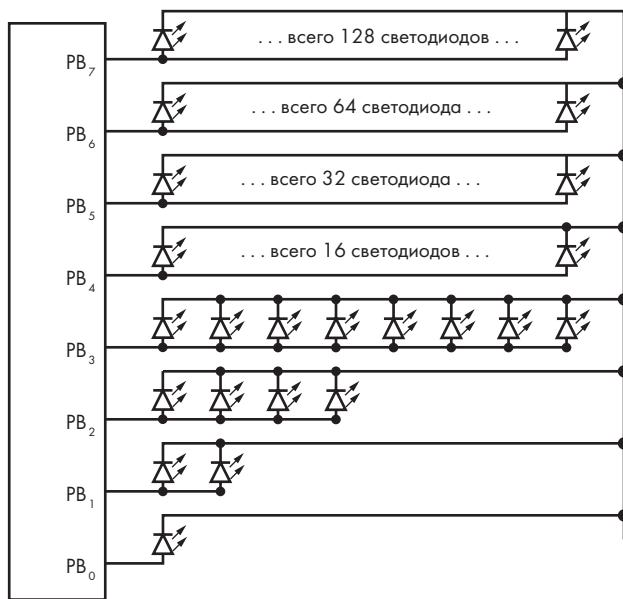


Рис. 6.22. Улучшенный цифро-аналоговый преобразователь с использованием светодиодов

Мы использовали пример со светодиодами, чтобы осветить работу цифро-аналогового преобразователя. Настоящий цифро-аналоговый преобразователь (ЦАП) вырабатывает напряжение вместо света. Термин «разрешение» в общих чертак используется для описания числа «шагов», которое может произвести ЦАП. Я говорю «в общих чертак», потому что обычно говорят, что ЦАП имеет, например, разрешение 10 бит, но на самом деле это означает, что он имеет разрешение «1 часть из 2^{10} ». Чтобы дать полностью правильное определение, разрешение — это максимальное напряжение, которое ЦАП может производить, деленное на количество шагов. Например, если 10-битный ЦАП может выдавать максимум 5 В, то он имеет разрешение примерно 0.005 В¹.

На рис. 6.23 показано условное обозначение, используемое для ЦАП.

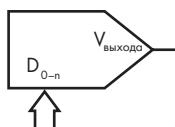


Рис. 6.23. Условное обозначение ЦАП

¹ 5/1024 В. — Примеч. науч. ред.

С помощью ЦАП можно генерировать аналоговые сигналы. Именно так работают аудиоплееры и музыкальные синтезаторы. Все, что нужно делать, — регулярно менять входы ЦАП. Например, с помощью 8-битного ЦАП, подключенного к порту B, можно генерировать сигнал пилообразной формы, показанный на рис. 6.24.



Рис. 6.24. Синтезированный сигнал пилообразной формы

Для более сложных сигналов устройства обычно включают память, куда можно записывать данные, которые затемчитываются дополнительными схемами. Это обеспечивает постоянную скорость передачи данных, не зависящую от того, чем в этот момент занимается ЦП. Типичный способ реализации этого — создание конфигурации *FIFO* («first in, first out» — «первым пришел — первым ушел»), как показано на рис. 6.25. Обратите внимание, что FIFO — это то же самое, что и очередь в программировании.

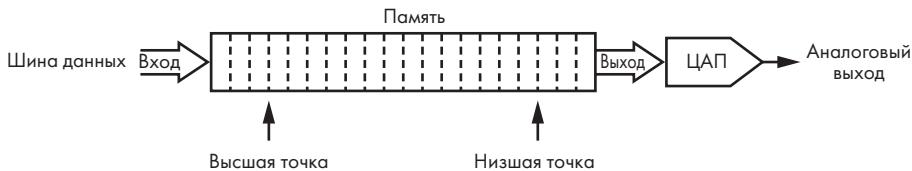


Рис. 6.25. FIFO с высшими и низшими точками

С памятью FIFO связаны два триггера: *высшая* и *низшая точки*, которые заимствуют свою терминологию из приливов и отливов (high-water mark — высшая отметка прилива, low-water mark — низшая отметка отлива). Низшая точка вызывает прерывание, когда очередь FIFO почти пустая; высшая срабатывает, когда очередь почти заполнена. Таким образом, программное обеспечение более высокого уровня может сохранять память заполненной для непрерывного вывода. Хотя это не совсем FIFO, потому что новые данные смешиваются со старыми. Именно так работают водонапорные башни; когда вода опускается ниже низшей отметки уровня воды, включается насос для наполнения бака; при достижении высшей отметки насос отключается. FIFO действительно удобны для объединения вещей, работающих с разной скоростью.

Аналого-цифровое преобразование

Аналого-цифровое преобразование — это обратный процесс, выполняется с помощью аналого-цифрового преобразователя, или АЦП, который сложнее ЦАП.

Первая проблема, с которой сталкивается АЦП, — как заставить аналоговый сигнал оставаться неподвижным, потому что невозможно измерить колеблющийся сигнал. (Вы понимаете, о чём речь, если когда-нибудь пытались измерить температуру у маленького ребенка.) На рис. 6.26 требуется взять *выборку* формы входного сигнала — больше одной, если нужно, чтобы оцифрованная версия напоминала аналоговый оригинал. Это делается с помощью схемы, называемой *«выборка и хранение»*, которая является аналоговым эквивалентом цифрового триггера-защелки (см. «Триггеры-защелки» на с. 113).



Рис. 6.26. Выборка и хранение

Когда мы принимаем выборку, замыкая переключатель, текущее значение аналогового сигнала сохраняется в хранилище. Имея стабильный сигнал в хранилище, мы можем его измерить, чтобы сгенерировать цифровое значение. Нам нужно что-то, что сравнивает сигнал с порогом, подобным тому, что мы видели в правой половине рис. 2.7 в главе 2. К счастью, аналоговая схема, называемая *компаратором*, может определить, когда одно напряжение больше другого. Это похоже на логический вентиль, за исключением того, что можно задать значение порога. Схематический символ компаратора показан на рис. 6.27.

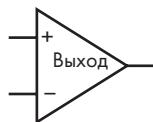


Рис. 6.27. Аналоговый компаратор

Выход равен 1, если сигнал на входе «+» больше или равен сигналу на входе «-».

Можно использовать стек компараторов с разными *опорными напряжениями* на входах — для создания *быстродействующего преобразователя*, как показано на рис. 6.28.

Эта система называется быстродействующим преобразователем, потому что выдает результаты быстро, в мгновение ока. Как вы можете видеть, выходы следующие: 00000000 для напряжения менее 0.125 В, 000000001 для напряжения от 0.125 до 0.250 В, 00000011 для напряжения от 0.250 до 0.375 В и т. д. Это работает, но с той же проблемой, что и у ЦАП на рис. 6.25: неэффективным использованием битов. Быстродействующие преобразователи также относительно дороги из-за большого количества компараторов, но они прекрасно подходят, когда требуется экстремальная скорость. Как создать более дешевый АЦП, который лучше использует биты?

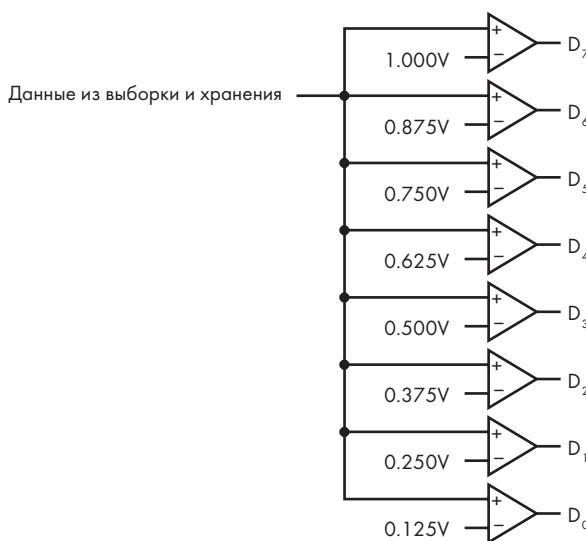


Рис. 6.28. Быстродействующий преобразователь

В быстродействующем преобразователе применялся набор фиксированных опорных напряжений, по одному на каждый компаратор. Можно использовать один компаратор при регулируемом опорном напряжении. Как его получить? При помощи ЦАП!

На рис. 6.29 видно, что компаратор используется для проверки значения выборки в хранилище по сравнению со значением ЦАП. После сброса счетчик ведет отсчет, до тех пор пока значение ЦАП не достигнет значения выборки, — затем счетчик отключается и выдает результаты. Таким образом, счетчик содержит оцифрованное значение образца.

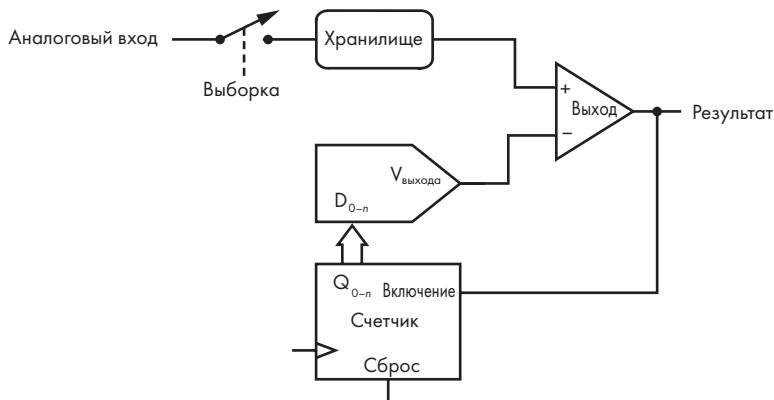


Рис. 6.29. Аналого-цифровой преобразователь

Увидеть, как это работает, можно на рис. 6.30. Аналоговый сигнал колеблется, но выходной сигнал хранилища остается стабильным после получения выборки. Затем счетчик очищается и ведет отсчет, до тех пор пока выходной сигнал ЦАП не достигнет значения выборки. После чего счетчик остановится, и мы получим результаты.

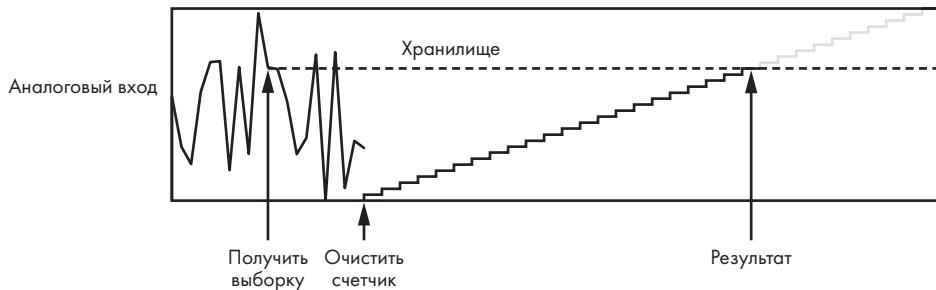


Рис. 6.30. АЦП в работе

Этот АЦП называется *преобразователем сравнения с пилообразным сигналом*, из-за того что выходной сигнал ЦАП генерирует пилообразное изменение. Одна из проблем, связанных с таким преобразователем, заключается в том, что его работа может занять много времени, поскольку время преобразования является линейной функцией значения дискретизированного сигнала. Если дискретизированный сигнал имеет максимальное значение и у нас есть n -битный АЦП, преобразование может занять 2^n тактов.

Один из способов обойти это ограничение — использовать преобразователь *последовательного приближения*, который выполняет *двоичный поиск* аппаратно, как показано на рис. 6.31.



Рис. 6.31. АЦП последовательного приближения в работе

Первый генератор тактовых сигналов устанавливает ЦАП на половину полного диапазона. Поскольку это меньше дискретизированного сигнала, он

настраивается на четверть диапазона. Это чересчур много, поэтому теперь его нужно уменьшить на одну восьмую диапазона, что, в свою очередь, слишком мало, поэтому его увеличивают на одну шестнадцатую полного диапазона — и это дает результат. В худшем случае требуется $\log_2 n$ тактов. Это действительно упростило задачу.

Термин «разрешение» используется для АЦП аналогично тому, как он применяется для ЦАП. Условное обозначение представлено на рис. 6.32.



Рис. 6.32. Условное обозначение АЦП

Цифровое аудио

Аудио включает в себя *дискретизацию* в одном измерении, то есть вычисление *амплитуды* или высоты сигнала в определенные моменты времени. Посмотрите на синусоидальный сигнал на рис. 6.33. У нас есть прямоугольный сигнал с некоторой *частотой дискретизации*, и мы записываем высоту сигнала на каждом *нарастающем фронте* с помощью аналого-цифрового преобразователя.

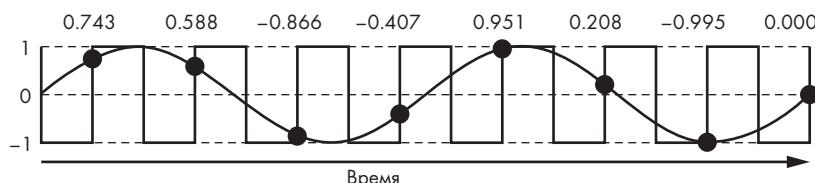


Рис. 6.33. Дискретизация синусоидальной волны

Получив набор образцов, мы должны иметь возможность восстановить исходный сигнал, подав их на ЦАП. Давайте попробуем это сделать, как показано на рис. 6.34.

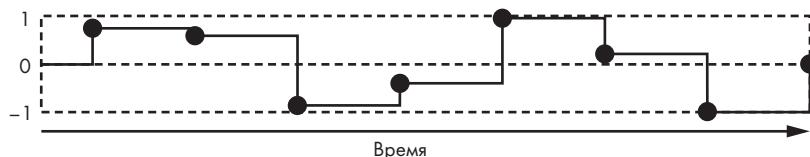


Рис. 6.34. Восстановленная синусоида из выборок

Ой, получилось ужасное искажение. Похоже, нам понадобится намного больше выборок, чтобы улучшить результат, как на рис. 6.35.

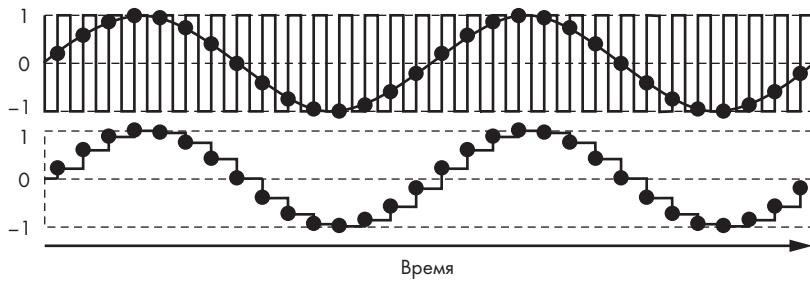


Рис. 6.35. Высокочастотная выборка и реконструкция

Однако нам это не нужно. Выборки и реконструкции на рис. 6.33 и 6.34 на самом деле достаточно. Я объясню почему, но имейте в виду: впереди тяжелая теория.

Синусоидальную волну относительно легко описать, как упоминалось в разделе «Поймать волну» на с. 201. Но нам нужен способ представить более сложные формы волн, такие как на рис. 6.31.

На графиках показана зависимость амплитуды от времени, но можно построить его по-другому. Взгляните на партитуру на рис. 6.36.

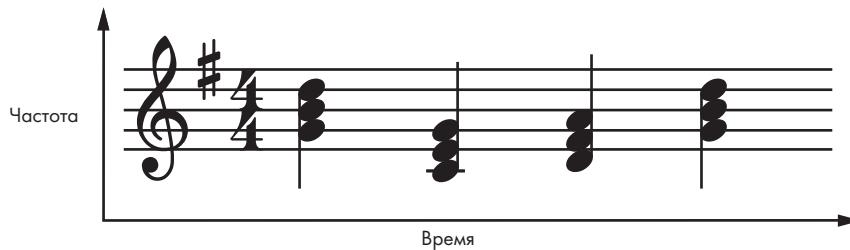


Рис. 6.36. Музыкальная партитура

Видим, что ноты зависят от времени, но, помимо этого, происходит что-то еще. В каждый момент времени играются не просто ноты, а *аккорды*, состоящие из нескольких нот. Посмотрим на первый аккорд с нотами G_4 (400 Гц), B_4 (494 Гц) и D_5 (587 Гц). Представьте, что мы играем аккорд на синтезаторе, который может генерировать синусоидальные волны для нот. На рис. 6.37 видно, что, хотя каждая нота является синусоидальной волной, сам аккорд представляет собой более сложную форму волны — сумму трех нот. Оказывается, любую форму волны можно представить как взвешенную (умноженную на некоторый масштабный коэффициент) сумму набора синусоидальных волн. Например, если прямоугольная волна на рис. 6.33 имеет частоту f , ее можно представить как сумму синусоидальных волн:

$$\frac{\sin(f)}{1} + \frac{\sin(3f)}{3} + \frac{\sin(5f)}{5} + \frac{\sin(7f)}{7} + \dots$$

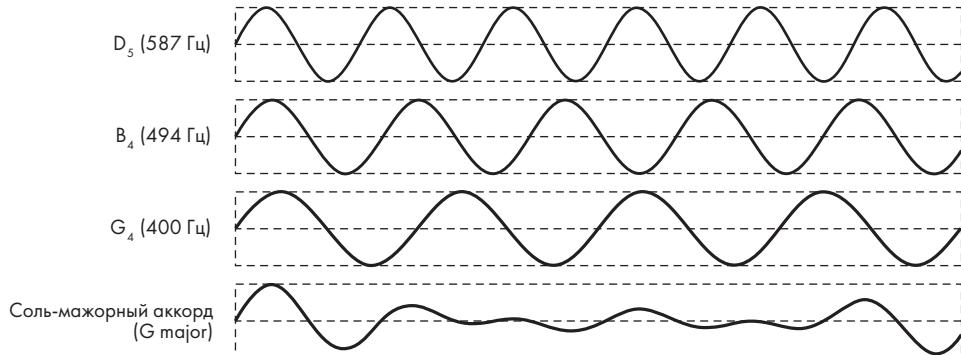


Рис. 6.37. Форма волны соль-мажорного аккорда

Если у вас хороший слух, вы можете прослушать такой аккорд и выделить составляющие ноты. Немузыкальным людям приходится полагаться на математическую акробатику под названием «преобразование Фурье», изобретенное французским математиком и физиком Жан-Батистом Жозефом Фурье (1768–1830), который также открыл парниковый эффект. Все графики, которые мы видели в этом разделе, показывают зависимость амплитуды от времени. Преобразование Фурье позволяет построить график зависимости амплитуды от частоты. Благодаря этому можно взглянуть на зависимость с другой стороны. Преобразование Фурье для соль-мажорного аккорда будет выглядеть, как на рис. 6.38.



Рис. 6.38. График преобразования Фурье для соль-мажорного аккорда

Вы, наверное, видели подобное раньше, даже не подозревая об этом. Во многих медиаплеерах добавлены приятные глазу анализаторы спектра, отображающие громкость в различных частотных диапазонах с использованием преобразования Фурье. Анализаторы спектра возникли как сложное электронное оборудование. Теперь их можно реализовать на компьютерах с помощью алгоритма *быстрого преобразования Фурье* (БПФ). Одно из самых крутых применений анализа Фурье – орган Hammond B-3.

ОРГАН ХАММОНДА В-3

Hammond B-3 — удивительный пример применения теории электромагнитных волн и анализа Фурье. Он работает так: двигатель приводит в движение вал, на котором смонтировано 91 «фоническое колесо». С каждым фоническим колесом связан звукосниматель, аналогичный тому, что используется на электрогитарах, который генерирует определенную частоту, определяемую неровностями на тональных колесах. Поскольку все фонические колеса установлены на одном валу, они не могут разладиться по отношению друг к другу.

Нажатие клавиши на B-3 не просто генерирует частоту, созданную фоническим колесом. Существуют девять восьмипозиционных «тяг», которые используются для смешивания сигнала, производимого «основным» тоном (игрой нотой), с сигналами других фонических колес. Тяги устанавливают уровень субоктавы, пятой, основной, 8-й, 12-й, 15-й, 17-й, 19-й и 22-й гармоник.

Создаваемый звук представляет собой взвешенную сумму этих девяти сигналов, установленную с помощью регуляторов, аналогично тому, как мы составили соль-мажорный аккорд на рис. 6.37.

Еще одна особенность многих медиаплееров — *графический эквалайзер*, позволяющий настроить звук по собственному вкусу. Графический эквалайзер — это набор настраиваемых *фильтров*, устройств, выделяющих или исключающих определенные частоты. Они похожи на передаточные функции, которые мы видели в разделе «Цифровые устройства в аналоговом мире» на с. 80, но применяются для частоты, а не для напряжения или света. Существуют два основных типа фильтров: *фильтр низких частот*, который пропускает все, что ниже определенной частоты, и *фильтр верхних частот*, который пропускает все, что выше определенной частоты. Их можно комбинировать для создания полосовых фильтров, которые включают все, что находится между низкими и высокими частотами, или *режекторных фильтров*, исключающих определенную частоту. На рис. 6.39 видно, что края фильтра нечеткие; они *спадают*. Идеальных фильтров не существует. Обратите внимание на то, что защита от дребезга кнопок на рис. 6.7 представляет собой фильтр низких частот.

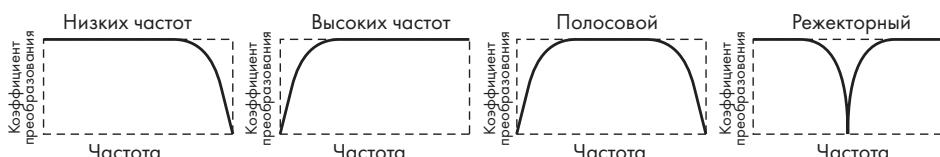


Рис. 6.39. Фильтры

Можно, например, применить фильтр нижних частот к соль-мажорному аккорду, как показано на рис. 6.40. Применение фильтра эффективно умножает кривые; фильтр регулирует уровень звука на разных частотах.

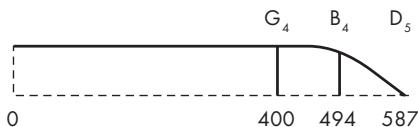


Рис. 6.40. График преобразования Фурье с фильтрацией нижних частот соль-мажорного аккорда

Как вы понимаете, фильтрованный аккорд звучит уже не так. B_4 стал немного тише, а D_5 практически не слышно.

Почему все это важно? На рис. 6.41 показано преобразование Фурье восстановленной синусоидальной волны из рис. 6.34. Я не уточнял детали на этом рисунке, поэтому предположим, что это синусоидальная волна 400 Гц, дискретизированная с частотой 3 кГц.

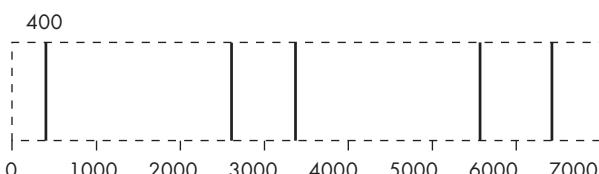


Рис. 6.41. График преобразования Фурье восстановленной синусоидальной волны

Обратите внимание, что ось X уходит в бесконечность с частотами, кратными частоте дискретизации, плюс или минус частота дискретизированного сигнала.

Что произойдет, если взять восстановленную синусоидальную волну и применить фильтр нижних частот, как показано на рис. 6.42?

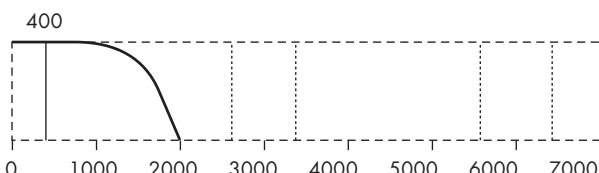


Рис. 6.42. График преобразования Фурье восстановленной синусоидальной волны с фильтром нижних частот

Все искажения исчезают; осталась только синусоидальная волна 400 Гц. Похоже, что выборка работает, если применить соответствующую фильтрацию. Но как выбрать частоту дискретизации и фильтр?

Гарри Найквист (Harry Nyquist) (1889–1976), шведский инженер-электронщик, придумал теорему, согласно которой, если необходимо точно захватить сигнал, выборку следует проводить с частотой, по крайней мере вдвое превышающей максимальную. Это хорошая теория, но, поскольку электроника не следует идеальной математике, она помогает производить выборку быстрее, чем необходимо для получения хорошо звучащего результата. Диапазон человеческого слуха составляет от 20 до 20 000 Гц.

Исходя из всего этого, мы должны иметь возможность захватывать все, что слышим, с частотой дискретизации 40 кГц. Что, если мы случайно получим звук с частотой 21 кГц, который *не дискретизирован* согласно теореме Найквиста? В этом случае мы получаем *свертку*, или *наложение*, спектров. Представьте, что частота дискретизации представляет собой зеркало, в котором отражается любая информация, превышающая эту частоту. Рассмотрев еще раз рис. 6.41, видим, что в частоте дискретизации есть *помехи* с плюс или минус дискретной частотой. Поскольку частота дискретизации намного больше, чем дискретная частота, эти помехи находятся далеко друг от друга. Входной сигнал с частотой 21 кГц, выбранный на частоте 40 кГц, будет подвержен помехам на частоте 19 кГц ($40 - 21$). Этот ошибочный сигнал называется *наложением*. Мы не получаем то, что предоставили. Перед дискретизацией необходимо применить фильтр низких частот, чтобы избежать наложения спектров.

Компакт-диски принимают 16-битные выборки с частотой 44 100 Гц – разумеется, умноженной на 2, потому что это стерео. Это дает чуть больше 175 Кбайт в секунду – довольно много данных. Некоторые стандартные частоты дискретизации звука составляют 44,1, 48, 96 и 192 кГц. Зачем проводить выборку с более высокой частотой, если это приведет к получению гораздо большего количества данных, а Найквист говорит, что в этом нет необходимости?

Частоту и амплитуду сигнала, дискретизированного около частоты Найквиста, можно восстановить, а вот *фазу* – нет. Еще один новый термин! Представьте фазу как небольшой сдвиг во времени. Вы можете видеть на рис. 6.43, что более толстый сигнал *отстает* (а не *опережает*) от более тонкого сигнала на 45 градусов, из-за чего он появляется немного позже по времени.

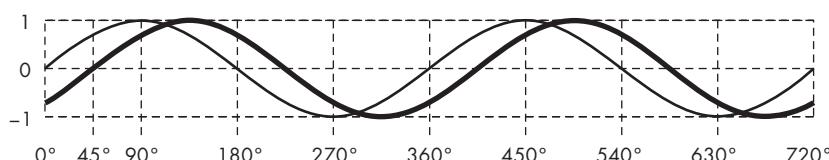


Рис. 6.43. Разность фаз сигналов

Разве это важно? Ну, вообще нет, если мы не говорим о стерео. *Разность фаз* вызывает временную задержку между сигналом, достигающим левого и правого уха, — благодаря этому можно определить, где в пространстве находится источник звука, как показано на рис. 6.44.

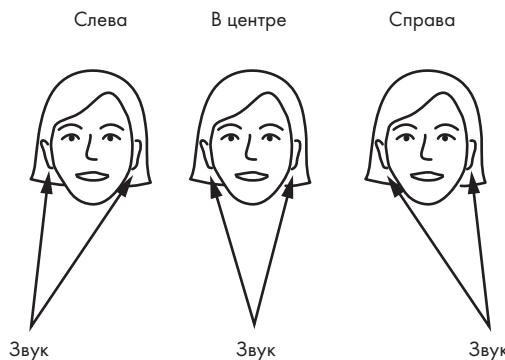


Рис. 6.44. Разница фаз в реальной жизни

ДИСКРЕТИЗАЦИЯ И ФИЛЬТРАЦИЯ ДЛЯ ЧМ-СТЕРЕО

ЧМ-стерео — интересное применение дискретизации и фильтрации. Это также отличный пример того, как новые функциональные возможности были встроены в систему, которая никогда не была для того предназначена, с обратной совместимостью, а это означает, что и старая система по-прежнему работала.

Вернувшись к рис. 6.20, видим, как биты могут использоваться для модуляции частоты. ЧМ означает «частотная модуляция» (FM, frequency modulation). FM-радио работает путем модуляции несущей частоты аналоговым сигналом вместо цифрового.

Несущие частоты FM-радиостанций распределяются каждые 100 кГц. Вы видели на рис. 6.41, что выборка генерирует дополнительные частоты до бесконечности; то же самое происходит с модуляцией. В результате к модулированному сигналу должен применяться фильтр нижних частот, иначе возникнут помехи на других станциях. Вы видели спад фильтра на рис. 6.39. Чем круче спад, тем сильнее фильтр искажает фазу, что отрицательно сказывается на звуке. Это показано в части радиоспектра на рис. 6.45.

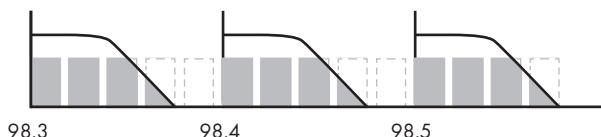


Рис. 6.45. Радиоспектр

До стерео звуковая информация в монофоническом ЧМ-сигнале занимала место примерно на 15 кГц выше несущей частоты. Приемник удалял несущую частоту, в результате чего получался исходный звук. Этую характеристику нужно было сохранить при переходе на стерео; в противном случае перестали бы работать все существующие приемники.

На рис. 6.46 представлен обзор того, как работает ЧМ-стерео. Прямоугольный сигнал 38 кГц используется для попаренной выборки левого и правого каналов. Генерируется контрольный сигнал с частотой 19 кГц, который синхронизируется с прямоугольным сигналом дискретизации. Контрольный сигнал смещивается на низком уровне, который трудно услышать через музыку, и объединяется с выборками для создания транслируемого составного сигнала.

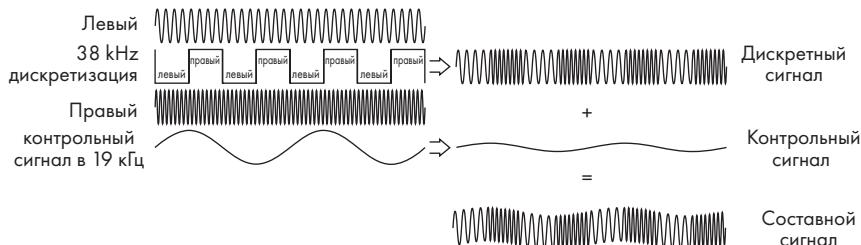


Рис. 6.46. Генерация ЧМ-сигнала

Посмотрим на результат анализа Фурье на рис. 6.47 и увидим, что первый набор частот слева представляет собой сумму левого и правого каналов — именно то, что нужно для монозвука. Для старых приемников это не проблема. Следующий набор частот — это разница между левым и правым каналами, которую не уловил бы старый моноприемник. Однако стереоприемник может использовать простую арифметику для разделения левого и правого каналов, производящих стереозвук.

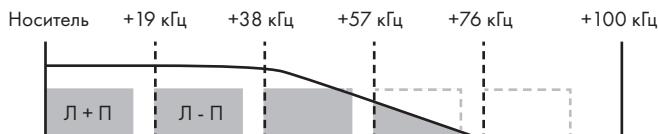


Рис. 6.47. ЧМ-стереоспектр

Мы лучше распознаем высокие частоты, потому что они имеют более короткие длины волн по сравнению с толщиной человеческой головы. Если бы голова была настолько узкой, что уши находились бы в одном месте, временной задержки не было бы. Большеголовые люди лучше воспринимают стерео! Это одна из причин, почему можно обойтись одним сабвуфером: нельзя точно сказать,

откуда исходит звук, потому что длина волны настолько велика по сравнению с толщиной головы, что разницу фаз невозможно обнаружить.

Когда вы слушаете стереозвук, разница фаз между звуками, выходящими из динамиков, создает *изображение*, способность «видеть», где музыканты находятся в пространстве. Изображение «мутное» без точной фазы. Таким образом, причиной более высоких частот дискретизации является лучшее воспроизведение фазового и стереоизображения. Вы можете никогда этого не заметить, если для прослушивания музыки всегда будете использовать дешевые наушники на сотовом телефоне.

Ранее я упоминал, что звук включает в себя большой объем данных. Было бы неплохо иметь возможность сжимать эти данные, чтобы они занимали меньше места. Есть два класса сжатия: *без потерь* и *с потерями*. Сжатие без потерь сохраняет все исходные данные. В результате объекты можно сжимать только примерно до половины их первоначального размера. Самым популярным сегодня сжатием без потерь является *FLAC*, сокращение от *Free Lossless Audio Codec*. *Кодек* (*codec*) — это устройство кодирования-декодирования, похожее на модем, которое знает, как переводить что-либо из одной системы кодирования в другую.

MP3, *AAC*, *Ogg* и им подобные — кодеки сжатия с некоторыми потерями точности. Они работают на психоакустических принципах. Люди, изучавшие работу уха и мозга, решили, что есть определенные вещи, которые услышать невозможно, например что-то тихое, происходящее сразу после громкого удара в барабан. Кодеки работают, удаляя эти звуки, что дает гораздо лучшую степень сжатия, чем в случае *FLAC*. Вот только уши не у всех одинаковы. Я считаю, что *MP3* звучит ужасно.

Цифровые изображения

Визуальные изображения сложнее аудио, поскольку подразумевают дискретизацию двумерного пространства. Цифровые изображения представлены в виде прямоугольных массивов элементов изображения, или *пикселей*. Каждый пиксель цветного изображения представляет собой триаду из красного, зеленого и синего цветов. Стандартные дисплеи, доступные сегодня, имеют по 8 бит красного, зеленого и синего цветов. Обычное представление изображено на рис. 1.20.

Компьютерные дисплеи используют цветовую систему *сложения*, которая воспроизводит практически любой цвет, комбинируя (или добавляя, отсюда и название) различное количество красного, зеленого и синего *основных цветов*. Это отличается от *субтрактивной* цветовой системы, используемой для печати, которая создает цвета путем смешивания разного количества голубого, пурпурного и желтого основных цветов.

Дискретизация изображения сродни помещению москитной сетки поверх изображения и записи цвета в каждом квадрате. Это несколько сложнее из-за *поточечной выборки*, что означает, что записывается не весь квадрат, а только

точка в центре каждого из них. На рис. 6.48 показана дискретизация изображения с трех экранов с разным разрешением.

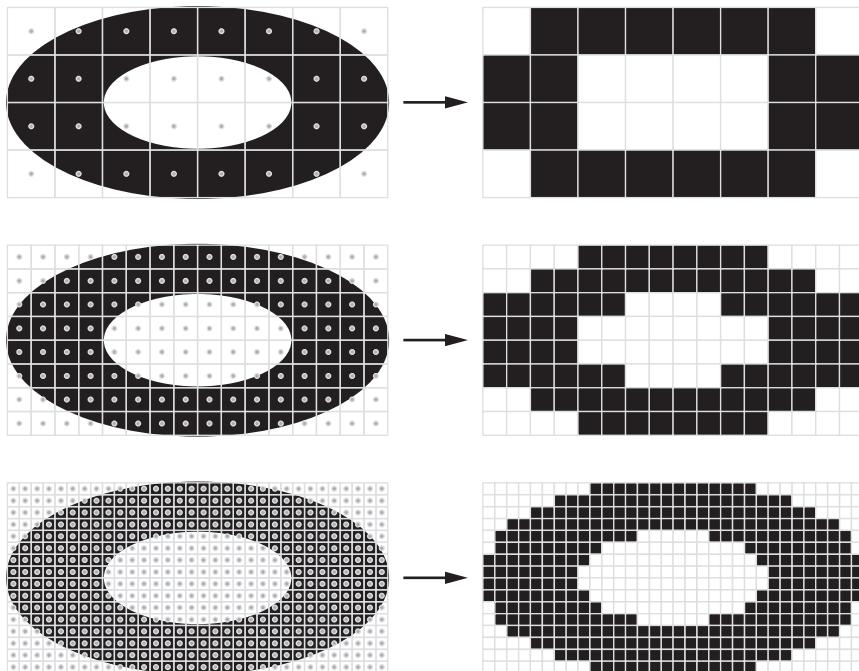


Рис. 6.48. Дискретизация изображения с разным разрешением

Видим, что дискретизированное изображение выглядит лучше на экранах с более высоким разрешением, но, конечно, это значительно увеличивает объем данных. Однако даже при наличии экрана с высоким разрешением мы все равно получаем неровные края. Это происходит из-за недостаточной выборки и наложения спектров согласно Найквисту, хотя сопутствующая математика слишком сложна для этой книги. Как и в случае со звуком, иногда помогает фильтрация. Один из способов фильтрации — *супердискретизация*, или взятие нескольких выборок на квадрат и их усреднение, как показано на рис. 6.49.

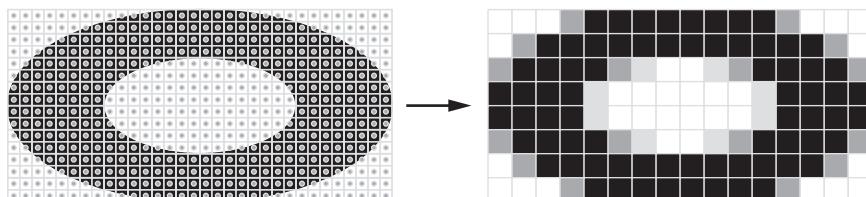


Рис. 6.49. Супердискретизация

При увеличении картинка неприглядная, но если вы отойдете подальше, то заметите, что все не так уж плохо. Если задуматься, супердискретизация эквивалентна увеличению частоты дискретизации, что мы уже рассматривали для звука на рис. 6.35.

Изображения становятся все больше и занимают много места. Неизвестно, останется ли когда-нибудь место для хранения всех фотографий и видео с котиками в мире. Как и в случае со звуком, нужно, чтобы изображения занимали меньше места, чтобы как можно больше разместить их в том же объеме памяти и быстрее передавать их по сети. Это снова решается путем сжатия.

Самый распространенный формат сжатия изображений сейчас — *JPEG*, стандарт, созданный группой экспертов Joint Photographic Experts Group. В его основе лежит сложная математика, которую я здесь не буду описывать. Грубо говоря, работа JPEG заключается в том, что он ищет соседние пиксели, которые довольно близки по цвету, и сохраняет описание этой области вместо отдельных пикселей, которые он содержит. Возможно, у вас есть камера с настройкой качества изображения; этот параметр регулирует определение «довольно близки по цвету». Это цветная версия примера из раздела «Стеки» на с. 166.

JPEG использует знания о человеческом восприятии аналогично аудиокодекам с потерями. Например, он использует тот факт, что наш мозг более чувствителен к изменениям яркости, чем к изменениям цвета.

Видео

Видео, еще один шаг вперед в многомерном пространстве, — это последовательность двумерных изображений, дискретизированных через равные промежутки времени. Временной интервал — функция зрительной системы человека. Старые фильмы обходились частотой 24 кадра в секунду (к/с); средний человек сегодня вполне доволен 48 кадрами в секунду.

Дискретизация видео не сильно отличается от дискретизации изображений, за исключением того, что различные помехи визуально раздражают, и поэтому их необходимо минимизировать. Проблема в том, что расположенные по краям помехи дискретизации, которые мы видели на рис. 6.48, не стоят на месте во время движения объектов.

Чтобы лучше понять это, взгляните на рис. 6.50, где показана диагональная линия, которая со временем перемещается слева направо. Перемещается только доля пикселя за кадр, а это означает, что выборка не всегда одинакова. Это все еще похоже на приближение линии, но каждая из линий представляет собой отдельное приближение. Это заставляет края «плавать», что визуально мешает. Фильтрация с использованием супердискретизации — один из способов уменьшить такие неприятные визуальные помехи.

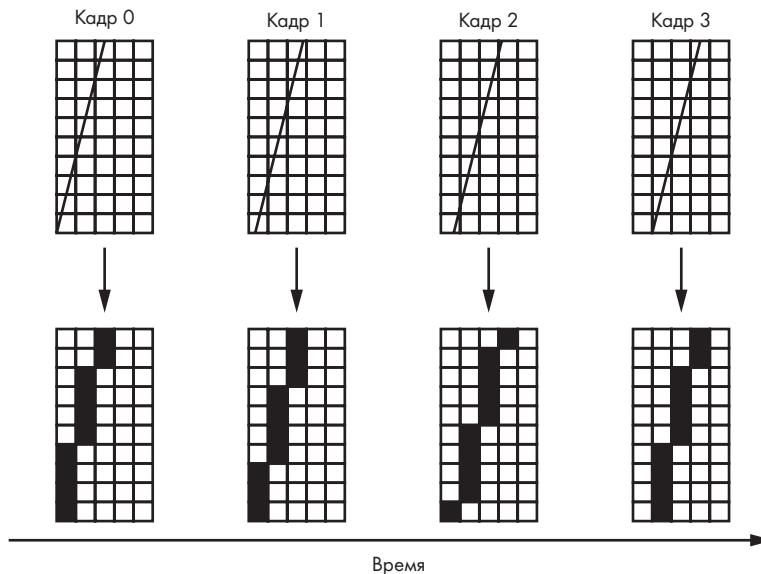


Рис. 6.50. Пример «плавания» краев

Видео производит намного больше данных, чем изображения или аудио. Видео в формате UHD имеет разрешение 3840×2160 пикселей. Умножьте это на 3 байта на пиксель и 60 кадров в секунду, и вы получите колоссальные 1 492 992 000 байт в секунду! Очевидно, что сжатие очень важно для видео.

Ключ к сжатию видео — наблюдение о том, что только часть изображения обычно меняется от кадра к кадру. Посмотрите на рис. 6.51, на котором мистер Сигма идет забрать посылку. Как видите, очень малая часть изображения меняется между кадрами. Можно хранить или передавать намного меньше данных, если использовать только данные из области изменений. Этот метод называется *сжатием движения*.



Рис. 6.51. Межкадровое движение

Один из минусов представления видео как набора изменений исходного изображения заключается в том, что иногда данные могут искажаться. Вероятно, вы видели раздражающие помехи на цифровом телевидении или при воспроизведении поврежденного видеодиска.

Нам нужен способ восстановить данные. Это достигается путем регулярного включения в них *ключевых кадров*. Ключевой кадр — это полное изображение. Даже если ошибки накапливаются из-за поврежденных измененных данных, данные восстанавливаются при обнаружении следующего ключевого кадра.

Алгоритмы обнаружения различий между кадрами сложны и требуют больших вычислительных ресурсов. Новые стандарты сжатия, такие как MPEG4, включают поддержку *многослойной обработки*, в которой используется тот факт, что большая часть видео теперь генерируется компьютером. Многослойная обработка работает так же, как старая нарисованная вручную анимация на целлулоидных пленках, которую мы обсуждали в главе 1, где объекты, нарисованные на прозрачных пленках, перемещались по неподвижному фоновому изображению.

Устройства взаимодействия с человеком

Компьютеры очень похожи на подростков с мобильными телефонами. Они проводят большую часть времени, обмениваясь сообщениями, но иногда и находят минутку поговорить с людьми. В этом разделе рассказывается о том, как компьютеры взаимодействуют с людьми.

Терминалы

Не так давно клавиатура, мышь и дисплей или сенсорный экран, к которым вы так привыкли, были невообразимой роскошью.

Было время, когда мы взаимодействовали с компьютером, записывая программу или данные на бумаге, используя специальные формы для кодирования. Затем данные нужно было передавать тому, кто пользуется клавищным перфоратором, чтобы превратить формы в стопку перфокарт (см. рис. 3.25). Мы брали эти карты, стараясь не уронить, и передавали оператору компьютера, который вставлял их в устройство для чтения карт. Оно, в свою очередь, считывало их в компьютер и запускало программу. Мы использовали этот подход, известный как *пакетная обработка*, потому что компьютеры были очень медленными и дорогими, что делало компьютерное время действительно ценным — пока ваши карты перфорировались, выполнялась чья-то программа.

Компьютеры стали быстрее, меньше и дешевле. К концу 1960-х годов в вашей компании или отделе вполне мог появиться небольшой компьютер. Маленький,

примерно как автоФургон. Компьютерное время стало чуть менее дефицитным. Произошло очевидное: люди начали подключать компьютеры к *телетайпам*. Телетайпы назывались *терминалами*, потому что они находились в конце линии (от terminal — конечный пункт). Особенno популярная модель, Teletype ASR-33, включала в себя клавиатуру, принтер, перфоратор для бумажной ленты (рис. 3.26) и устройство для чтения бумажной ленты. Бумажная лента была эквивалентом карты памяти USB. ASR-33 был прекрасен — он позволял печатать аж *10 символов в секунду!* Термин TTY все еще используется нами как сокращение от телетайпа.

Системы с *разделением времени* были изобретены для того, чтобы занять эти небольшие компьютеры. Да, такие системы действительно походили на аренду жилья на время отпуска. Вы считаете место своим, и это действительно так, пока вы занимаете его, но в другое время им пользуются другие люди.

В системе с разделением времени есть программа *операционной системы* (ОС), которая запускается на компьютере. Программа ОС похожа на агента по бронированию для аренды жилья на курорте. Его задача — выделить каждому пользователю различные ресурсы компьютера. Когда подходит ваша очередь использовать машину, программы другого пользователя выгружаются на диск, а ваша программа загружается в память и какое-то время работает. Все это происходит достаточно быстро — можно подумать, что у вас есть самый настоящий компьютер, по крайней мере до тех пор, пока очередь немного не увеличится. В какой-то момент память начнет *переполняться*, поскольку операционная система станет тратить больше времени на процессы загрузки и выгрузки, чем на запуск пользовательских программ.

Пробуксовка (thrashing) сильно замедлила работу систем с разделением времени, когда пользователей стало слишком много. Программисты начали работать поздно ночью, потому что они могли забирать компьютеры себе, после того как все уйдут домой.

Системы с разделением времени являются *многозадачными*, поскольку компьютер создает иллюзию, что он делает более одного дела одновременно. Внезапно к одной машине стали подключать множество терминалов. Так появилось понятие «пользователь», чтобы машины могли определять, что кому принадлежит.

Время шло, появлялись лучшие версии телетайпов, и каждое их поколение становилось быстрее и тише. Но они все еще печатали что-то на бумаге (то есть создавали *бумажную копию*) и годились только для текста. В модели телетайпа 37 были добавлены греческие символы, чтобы ученые могли печатать математические уравнения. Терминалы IBM Selectric имели сменяемые *печатные шары*, которые позволяли пользователю изменять шрифты. Существовал шрифт с точками в разных позициях, который позволял рисовать графики.

Графические терминалы

Отказ от терминалов с бумажными копиями был обусловлен многими причинами — скоростью, надежностью, шумом и т. п. Экраны существовали пока лишь для таких вещей, как радары и телевидение, — пришло время заставить их работать с компьютерами. Это происходило медленно из-за эволюции электроники. Память была слишком медленной и дорогой. Графические терминалы изначально создавались на основе разновидности вакуумной лампы (см. «Вакуумные лампы» на с. 93), называемой *электронно-лучевой трубкой* (ЭЛТ). Внутренняя часть стекла покрывается химическим люминофором, который светится при ударе электронов. Имея более одной решетки или *отражательной пластины*, можно рисовать изображения на люминофоре. Это похоже на очень талантливого игрока с битой, который может поразить мячом любую цель.

На самом деле, существуют два способа заставить этот дисплей работать. Версия с отражающей пластиной, называемая *электростатическим отклонением*, использует тот же принцип, что и опасное статическое электричество. Другой вариант — версия с электромагнитом, называемая *электромагнитным отклонением*. В любом случае биты необходимо преобразовать в напряжения, что является еще одним применением строительного блока ЦАП.

Сегодня ЭЛТ — это в основном пережиток прошлого, который был заменен *жидкокристаллическим дисплеем* (liquid crystal display, LCD). Жидкие кристаллы — это вещества, которые изменяют свои светопропускающие свойства при подаче электрического тока. Типичный дисплей с плоским экраном очень похож на ЭЛТ в том, что в каждой точке раstra есть три капли жидкого кристалла с красным, зеленым и синим фильтрами и светом, который поступает сзади. Мы по-прежнему говорим о жидкокристаллических дисплеях как об электронно-лучевых трубках, но на самом деле ЭЛТ давно канули в Лету. ЖК-дисплеи распространились повсеместно и заменили ЭЛТ в большинстве приложений; благодаря ЖК-дисплеям стало возможным появление сотовых телефонов, ноутбуков и телевизоров с плоским экраном.

Первые экранные терминалы назывались *стеклянными телетайпами*, потому что они могли отображать только текст. Эти терминалы отображали 24 строки по 80 символов в каждой, всего 1920 символов. Поскольку символ помещался в байт, всего получалось два доступных в то время кибайта памяти. Со временем были добавлены дополнительные функции, такие как редактирование на экране и перемещение курсора, которые в конечном итоге были стандартизированы как часть ANSI X3.64.

Векторная графика

ЭЛТ по принципу работы очень похожи на миллиметровую бумагу. Электронный луч перемещается в некоторую точку в зависимости от напряжений по

оси X и Y . Также есть ось Z , которая определяет яркость. Ранние модели не учитывали цвета, поэтому это были черно-белые, или *полутоновые*, дисплеи. Количество координат на дюйм называется разрешением.

Векторная графика – это рисование линий, или *векторов*. Можно создать картину, рисуя набор направленных линий. Тонкие стрелки на рис. 6.52 нарисованы полностью с выключенной яркостью.

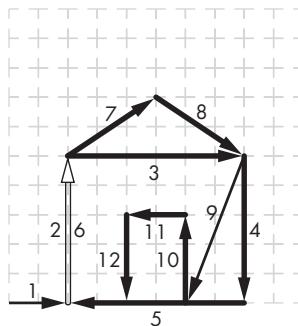


Рис. 6.52. Дом в векторной графике

Белая стрелка с черным контуром рисуется дважды: один раз с включенной яркостью, а затем снова с выключенной. Если дважды провести одну и ту же линию с включенной яркостью, она станет вдвое ярче, чего мы не хотим делать только потому, что меняем положение.

Дом на рис. 6.52 нарисован по *таблице отображения*, которая представляет собой список инструкций для рисования, как на рис. 6.53.

1. Переместитесь в $(2, 0)$
2. Нарисуйте линию до $(2, 5)$
3. Нарисуйте линию до $(7, 5)$
4. Нарисуйте линию до $(7, 0)$
5. Нарисуйте линию до $(2, 0)$
6. Переместитесь в $(2, 5)$
7. Нарисуйте линию до $(8, 7)$
8. Нарисуйте линию до $(7, 5)$
9. Переместитесь в $(6, 0)$
10. Нарисуйте линию до $(5, 3)$
11. Нарисуйте линию до $(4, 3)$
12. Нарисуйте линию до $(4, 0)$
13. Начните заново с шага 1

Рис. 6.53. Таблица отображения

Обратите внимание на последнюю инструкцию. Мы начинаем сначала, потому что изображение на экране довольно быстро тускнеет. Это работает только из-за

постоянства люминофора ЭЛТ — свойства, благодаря которому он остается гореть после удаления луча, — и медленной реакции человеческого глаза. Нужно повторять одни и те же действия, чтобы изображение оставалось на экране.

Однако в этой инструкции есть еще кое-что. Вокруг нас много излучения в 60 Гц, потому что в Америке это частота электроэнергии переменного тока (в некоторых других странах¹ она составляет 50 Гц). Несмотря на все попытки экранировать излучение, оно все равно влияет на дисплеи и заставляет их колебаться. Таким образом, графические терминалы, такие как GLANCE G, разработанные в Bell Telephone Laboratories, имели команду «перезапуск на шаге 1, после того как в следующий раз линия питания пересечет 0 в направлении от плюса к минусу». Это синхронизировало рисунок с интерференцией, так что он всегда колебался в соответствии с ней и, следовательно, был незаметен.

На рисование изображения требовалось время, и неприятным побочным эффектом было то, что все шло хорошо до тех пор, пока список отображения не стал настолько длинным, что его нельзя было нарисовать за одну шестидесятую долю секунды. Изображение стало сильно мерцать, если его можно было нарисовать только раз в тридцатую долю секунды.

Компания Tektronix нашла интересное решение проблемы мерцания, которое получило название *запоминающая трубка*. Это был электронный аналог игрушки «Волшебный экран». С его помощью можно было рисовать очень сложные изображения, но их нужно было встряхнуть электронным способом, чтобы стереть. Было очень сложно рисовать цельные изображения на GLANCE G, потому что для этого требовалось огромное количество векторов и в итоге появлялось мерцание. Запоминающие трубы могли обрабатывать сплошные изображения, поскольку количество векторов не ограничивалось, но центры сплошных областей изображения начинали исчезать. На GLANCE G можно было стереть одну строку, удалив ее из таблицы отображения. Запоминающая трубка этого не позволяла. При стирании экран испускал ярко-зеленую вспышку, которую наверняка вспомнят многие программисты-старожилы.

Растровая графика

Растровая графика — полная противоположность векторной. Именно так изначально работало телевидение. Растр представляет собой непрерывно нарисованный узор, как показано на рис. 6.54.

Растр начинается в верхнем левом углу и проходит по всему экрану. Затем *горизонтальный обратный ход* приводит к началу следующей линии. Наконец, *вертикальный обратный ход* возвращает картинку к началу после рисования последней линии.

¹ В том числе в России. — Примеч. науч. ред.

Это очень похоже на аналогию со стартовым пистолетом, которую я приводил ранее при обсуждении последовательной связи. Когда растр работает, все, что вам нужно сделать, — изменить яркость в нужное время, чтобы получить желаемое изображение, как показано на рис. 6.55.

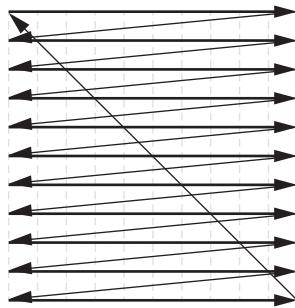


Рис. 6.54. Растр

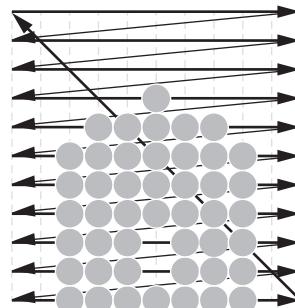


Рис. 6.55. Дом в растровой графике

Я также использовал аналогию с москитной сеткой в разделе «Цифровые изображения» на с. 222. Растворое изображение — это *настоящая* сетка, а это значит, что рисовать между точками не получится. Это может привести к появлению неприятных визуальных помех, например к тому, что крыша будет выглядеть неправильно. Это связано с тем, что разрешение типичного растрового дисплея довольно низкое — порядка 100 точек на дюйм. Низкое разрешение приводит к заниженной выборке и искажению, аналогичному тому, что мы видели для цифровых изображений. В настящее время используется достаточная вычислительная мощность, чтобы постоянно применять *сглаживание* — например, с помощью супердискретизации.

Растворое сканирование используется в таких устройствах, как факсы, лазерные принтеры и сканеры. Поднимите крышку сканера и посмотрите, как он работает. Только не забудьте надеть солнцезащитные очки. Когда у принтеров было больше движущихся частей и они были громче, люди придумали, как воспроизводить на них *растровую музыку*, тщательно подбрав символы для печати.

Растворые дисплеи не применяют таблицы отображения, хотя те по-прежнему используются за растворыми дисплеями. Как мы увидим позже, веб-страницы представляют собой таблицы отображения. В графическом языке OpenGL есть таблицы отображения, и поддержка языка часто включена в графическое оборудование. Монокромные дисплеи используют часть памяти с 1 битом для каждой позиции раstra. В свое время это был огромный объем памяти; теперь он не имеет такого значения. Конечно, эта память может очень быстро разрастись. Для растворого дисплея, который может отображать 256 различных уровней серого, вам потребуется 8 бит памяти для каждой позиции раstra.

Цвет был открыт в Стране Оз и быстро перешел на экран. Монохромные или полутоновые дисплеи были простыми: все, что вам нужно было сделать, — это покрыть внутреннюю часть экрана слоем люминофора. Для цветных дисплеев требовались три точки разного цвета в каждом месте раstra — красный, зеленый и синий — и три электронных луча, которые могли пройти через эти точки с большой точностью. Это означало, что для обычного дисплея нужно было в три раза больше памяти.

Клавиатура и мышь

Терминалы позволяют вводить данные в дополнение к дисплею, который данные отображает. Хорошо известные вам терминалы — это клавиатура и мышь, сенсорная панель на ноутбуке и сенсорный экран на телефоне и планшете.

Клавиатуры довольно просты. Это просто набор переключателей с некоторой логикой. Распространенный способ создания клавиатуры — поместить ключевые переключатели в сетку, мультиплексируя их, как показано на рис. 6.10. Мощность последовательно подается на строки сетки, и значения столбцов считаются.

Мышь в том виде, в каком мы ее знаем, была изобретена американским инженером Дугласом Энгельбартом (Douglas Engelbart) (1925–2013) в Стэнфордском исследовательском институте. Я упоминал в разделе «Квадратура» на с. 196, что можно сделать мышь, используя пару квадратурных кодировщиков, по одному для направлений X и Y .

Существует множество технологий для сенсорных панелей и экранов. Основное их отличие состоит в том, что сенсорные экраны должны быть прозрачными, чтобы дисплей был виден. Сенсорные устройства — это устройства для сканирования строк и столбцов, подобно клавиатурам, но в гораздо более мелком масштабе.

Выводы

В этой главе вы узнали о системе прерываний, которая позволяет процессорам эффективно обрабатывать ввод/вывод. Мы говорили о том, как работают различные типы устройств ввода/вывода и как они взаимодействуют с компьютерами. Мы также обсудили сложную область дискретизации аналоговых данных, чтобы их можно было обрабатывать с помощью цифровых компьютеров. Теперь вы знаете достаточно о том, как работают компьютеры, поэтому, начиная со следующей главы, мы рассмотрим взаимосвязь между аппаратным и программным обеспечением, чтобы научиться писать ПО, которое хорошо работает на аппаратном обеспечении.

7

Организация данных



Вы могли заметить, что я становлюсь немного одержим, когда речь заходит о работе с памятью. Из главы 3 вы узнали, что порядок доступа к устройствам памяти, таким как DRAM, флеш-память и дисковые накопители, влияет на их скорость. А из главы 5 — что производительность также зависит от того, присутствуют ли нужные данные в кэш-памяти. Учитывайте эти характеристики системы памяти при организации данных — и вы повысите производительность. В этой главе мы рассмотрим ряд *структур данных*, или стандартных способов организации данных. Многие из них созданы для поддержки эффективного использования различных типов памяти. Выбор структуры данных часто связан с компромиссом между пространством и временем, когда больше памяти используется для ускорения определенных операций. (Обратите внимание, что структуры данных более высокого уровня предоставляются языками программирования, а не компьютерным оборудованием.)

Выражение «*локальность ссылок*» бытует как профессиональный жаргонизм, но резюмирует большую часть того, о чем идет речь в этой главе. Оно означает: «Храните нужные данные под рукой, а данные, которые вам вскоре понадобятся, — еще ближе».

Базовые типы данных

В языках программирования существует множество базовых типов данных. Эти типы определяются двумя характеристиками: размером (количеством

битов) и интерпретацией типа (знаковый, беззнаковый, с плавающей точкой, символьный, указатель, логический). На рис. 7.1 показаны типы данных, доступные на типичном современном компьютере в языке программирования С. Различные реализации С на одном компьютере, а также разные языки, такие как Pascal или Java, могут по-разному представлять эти типы данных. Некоторые языковые среды включают средства, позволяющие программисту запрашивать порядок байтов (см. рис. 4.4 на с. 139), количество битов в байте и многое другое.

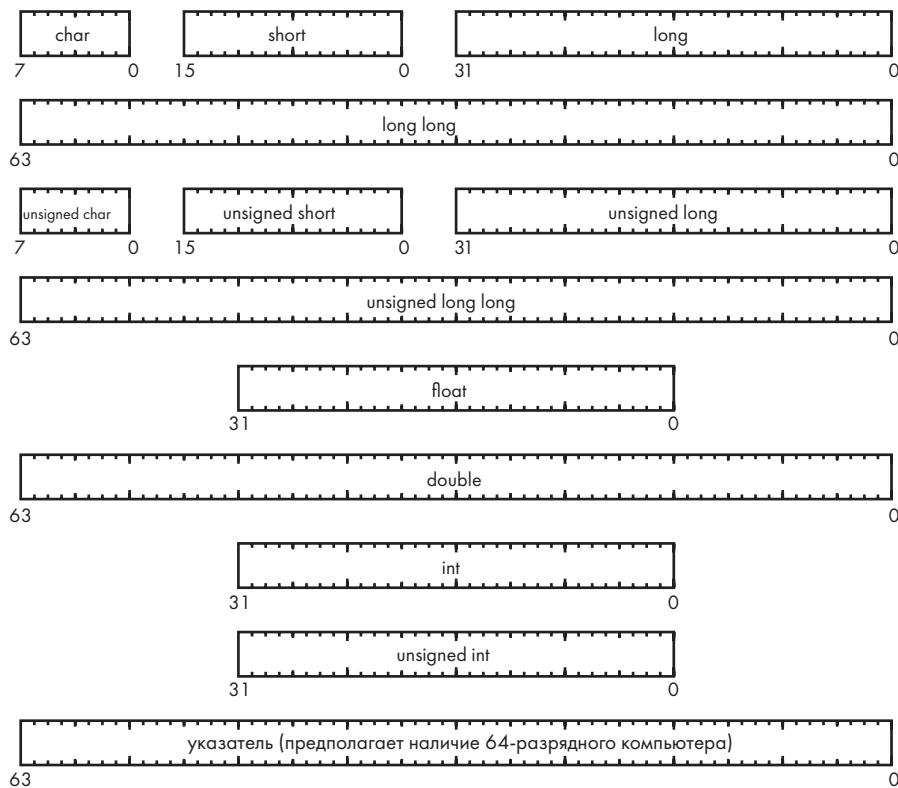


Рис. 7.1. Типичные базовые типы данных языка С

Мы уже рассматривали все эти типы данных в главе 1, кроме указателя; единственная разница здесь в том, что мы используем для них названия языка С.

Американский инженер Гарольд Лоусон (Harold Lawson) изобрел указатель для языка PL/I (Programming Language One) в 1964 году. Указатель — это просто целое число без знака некоторого архитектурно-зависимого размера, но оно интерпретируется как адрес памяти. Его можно сравнить с адресом дома — число обозначает не сам дом, но адрес, по которому его можно найти. Мы уже

видели, как это работает — подобно косвенной адресации из раздела «Режимы адресации» на с. 149. Нулевой, или NULL-указатель, обычно не является допустимым адресом памяти.

Указатели стали популярными благодаря языку С. В некоторых языках вместо указателей реализованы более абстрактные *ссылки*. Они были призваны решить проблемы, возникающие из-за неправильного использования указателя, и я затрону этот вопрос позже в данной главе. Указатели, как правило, совпадают по размеру с величиной типичного для данного устройства слова, поэтому к ним можно получить доступ за один цикл.

Достижения в технологии производства микросхем стимулировали разработку большого количества новых вычислительных машин в 1980-х годах, включая переход от 16-битных компьютеров к 32-битным. В 1970-х и начале 1980-х большинство программистов очень бесцеремонно относились к использованию указателей в коде; например, предполагалось, что указатели и целые числа имеют одинаковый размер, и потому их использовали взаимозаменяя. Такой код при переносе на новые устройства часто ломался, и его было трудно отладить. Это привело к появлению двух независимых подходов к исправлению. Во-первых, гораздо больше внимания стали уделять вопросам переносимости. Это принесло плоды; проблемы с переносимостью и указателями сегодня встречаются не так часто. Во-вторых, были разработаны языки без указателей, например Java. В некоторых случаях такой подход оказался оправданным, но не всегда.

Массивы

Типы данных, рассмотренные в предыдущем разделе, просты; их можно сравнить с частными домами. В языках программирования также поддерживаются *массивы*, которые вместо этого можно сравнить с многоквартирными зданиями. У многоквартирных зданий есть адрес, а у отдельных квартир — номера. Программисты называют номер устройства *индексом* (начиная с 0, в отличие от большинства номеров квартир), а отдельные квартиры — *элементами* массива. Типичные компьютерные строительные нормы и правила требуют, чтобы все квартиры в доме были идентичными. На рис. 7.2 показано здание, которое содержит десять 16-битных квартир в языке С.

Каждый прямоугольник на рис. 7.2 представляет собой байт. Таким образом, в этом массиве из 16-битных элементов каждый элемент занимает два 8-битных байта. Нижний индекс элемента обозначает индекс массива.

Посмотреть на элементы массива по-другому можно через призму относительной адресации (см. «Относительная адресация» на с. 173). Каждый элемент представляет собой смещение от адреса 0-го элемента, или *базового адреса*. Таким образом, элемент₁ находится на расстоянии двух байтов от элемента₀.

Массив →

0	1	элемент ₀
2	3	элемент ₁
4	5	элемент ₂
6	7	элемент ₃
8	9	элемент ₄
10	11	элемент ₅
12	13	элемент ₆
14	15	элемент ₇
16	17	элемент ₈
18	19	элемент ₉

Рис. 7.2. Десятиэлементный массив 16-битных чисел

Массив на рис. 7.2 представляет собой *одномерный* массив — уродливое однэтажное здание с квартирами, соединенными общим коридором. Языки программирования также поддерживают *многомерные* массивы — например, четырехэтажное здание с трехбайтными квартирами. Для его представления понадобится двумерный массив с двумя индексами: один — для номера этажа, а другой — для номера квартиры на этом этаже. Мы даже можем создавать трехмерные здания с указателями корпуса, этажа и квартиры; четырехмерные постройки с четырьмя индексами и т. д.

Важно понимать, как многомерные массивы размещаются в памяти. Допустим, мы подсовываем листовки под каждую дверь в многоквартирном доме 4×3 . Можно сделать это двумя способами. Первый — начать с этажа 0 и подложить листовку в квартиру 0, затем перейти на этаж 1 и подложить листовку в квартиру 0 и т. д. Или можно начать с этажа 0 и подсунуть листовки под каждую дверь на этом этаже, затем сделать то же самое на этаже 1 и т. д. Это как раз то, что называется *локальностью ссылок*. Второй подход (перебор всех дверей на одном этаже) представляет лучшую локальность ссылок — такой путь гораздо проще пройти. Он показан на рис. 7.3, где числа в скобках — это адреса относительно начала массива.

Индекс столбца перемещается между соседними столбцами, тогда как индекс строки перемещается между строками, которые находятся дальше друг от друга в адресном пространстве.

Массив →

элемент ₀	элемент _{0,0} (0)	элемент _{0,1} (1)	элемент _{0,2} (2)
элемент ₁	элемент _{1,0} (3)	элемент _{1,1} (4)	элемент _{1,2} (5)
элемент ₂	элемент _{2,0} (6)	элемент _{2,1} (7)	элемент _{2,2} (8)
элемент ₃	элемент _{3,0} (9)	элемент _{3,1} (10)	элемент _{3,2} (11)

Рис. 7.3. Схема двумерного массива

Этот подход применим и на более высоких измерениях. Если бы у нас был комплекс из пяти четырехэтажных зданий, по три квартиры на этаже, рис. 7.3 повторился бы пять раз, по одному для каждого здания. В адресном пространстве соседние здания дальше друг от друга, чем соседние строки, которые, в свою очередь, дальше друг от друга, чем соседние столбцы.

Возвращаясь к рис. 7.2, подумайте, что произойдет, если попытаться получить доступ к элементу₁₀. Некоторые языки программирования, такие как Pascal, проверяют, находится ли индекс массива в границах массива, но многие другие (включая C) этого не делают. Без проверки элемент₁₀ попадет в байты 20 и 21 относительно начала массива. Это может привести к сбою программы, если по этому адресу нет доступной памяти. Отсутствие проверки — это также прореха в безопасности, которая может привести к непреднамеренному доступу к данным, хранящимся вне массива. Ваша работа как программиста — всегда оставаться в рамках массива, если язык программирования не делает этого самостоятельно.

Битовые матрицы

Мы уже узнали, как создавать массивы из базовых типов данных, но иногда подходящего базового типа данных не существует. Например, Санта должен отличать непослушных детей от хороших, а детей огромное множество. Два значения означают, что нам нужен только 1 бит на одного ребенка. Можно использовать байт для каждого значения, но это менее эффективно, потому что приведет к глобальному потеплению на Северном полюсе и плохим новостям для Снеговика Фрости¹, поскольку таяние снегов считается уже существующим условием и не учитывается примером. Что нам действительно нужно, так это массив битов, или *битовая матрица*.

Битовые матрицы легко создавать. Предположим, что мы хотим отслеживать 35 бит. Мы знаем, что массива памяти из пяти 8-битных байт будет достаточно, как показано на рис. 7.4.

bits ₀	7	6	5	4	3	2	1	0
bits ₁	15	14	13	12	11	10	9	8
bits ₂	23	22	21	20	19	18	17	16
bits ₃	31	30	29	28	27	26	25	24
bits ₄						34	33	32

Рис. 7.4. Массив как битовая матрица

¹ «Снеговик Фрости» («Frosty the Snowman») — американский рождественский короткометражный мультфильм, снятый в 1969 году. — Примеч. ред.

Для битовых матриц доступны четыре основные операции: задать бит, сбросить бит (задать его значение равным 0), проверить, задан ли бит, и проверить, сброшен ли бит.

Можно использовать целочисленное деление, чтобы найти байт, содержащий определенный бит, — для этого используется деление на 8. Сделать это быстро можно на устройствах циклического сдвига (см. раздел «Сдвиг» на с. 143), сдвинув желаемый номер бита вправо на 3. Например, бит номер 17 будет сдвинут в третий байт, потому что $17 \div 8$ равно 2 в целочисленном делении, а байт 2 — это третий байт, отсчитываемый от 0.

Следующий шаг — создание маски для битовой позиции. Подобно своему физическому аналогу, *маска* представляет собой битовый узор с отверстиями, через которые «можно смотреть». Начинаем с операции И для желаемого номера бита с маской 0x07, чтобы получить три младших бита; для 17 это 00010001 И 00000111, что дает 00000001, или позицию бита 1. Затем сдвигаем 1 влево на эту величину, получая маску 00000010 — это позиция бита 17 в байте 2.

Используя индекс массива и битовую маску, легко выполнить следующие операции:

Задать бит	$\text{биты}_{\text{индекс}} = \text{биты}_{\text{индекс}} \text{ ИЛИ } \text{маска}$
Сбросить бит	$\text{биты}_{\text{индекс}} = \text{биты}_{\text{индекс}} \text{ И } (\text{НЕ маска})$
Проверить, задан ли бит	$(\text{биты}_{\text{индекс}} \text{ И маска}) \neq 0$
Проверить, сброшен ли бит	$(\text{биты}_{\text{индекс}} \text{ И маска}) = 0$

Есть еще одно полезное применение битовых матриц: проверка того, доступны или заняты ресурсы. Если заданный бит представляет занятый ресурс, можно просканировать массив в поисках байта, который состоит не только из единиц. Это позволяет проверять восемь бит одновременно. Конечно, прежде чем обнаружить чистый бит, нужно найти байт, который его содержит, но это намного эффективнее, чем проверять каждый бит по отдельности. Обратите внимание, что в подобных случаях более эффективно использовать массив с самым большим базовым типом данных, например `unsigned long` в языке С, вместо массива байтов.

Строки

Вы узнали о кодировании символов в разделе «Представление текста» на с. 63. Последовательность символов, например в этом предложении, называется *строкой*.

Как и в случае с массивами, часто нужно знать длину строки, чтобы иметь возможность работать с ней. Обычно недостаточно просто создать массив для

каждой строки, потому что многие программы работают со строковыми данными переменной длины; большие массивы часто используются, когда длина строки заранее неизвестна. Поскольку размер массива не связан с длиной строки, нужен другой метод для отслеживания ее длины. Самый удобный способ сделать это — связать длину строки с ее данными.

Один из подходов — хранить длину в самой строке, например в первом байте. Это работает, но ограничивает длину строки 255 символами, что недостаточно для многих областей применения. Для поддержки более длинных строк можно использовать больше байтов, но в определенный момент количество оверхедов (байтов длины строки) превысит длину самой строки. Кроме того, поскольку строки представлены байтами, они могут иметь любое выравнивание, но для многобайтового счетчика строки должны быть выровнены по его границам.

Язык С использует другой подход, заимствованный из псевдоинструкции `.ASCII` языка ассемблера PDP-11, в которой нет специального типа данных для строк, как в некоторых языках. Он просто использует одномерные массивы байтов; байтовый тип данных в С — `char` именно потому, что строки представляют собой массивы символов. Но есть одна хитрость: С не хранит длину строки. Вместо этого он добавляет дополнительный байт в конец массива символов для хранения символа NUL-терминатора. С использует символ NUL в ASCII (см. табл. 1.11), имеющий значение 0, в качестве *терминатора строки*. Другими словами, NUL-терминатор используется для обозначения конца строки. Это работает как для ASCII, так и для UTF-8. Пример показан на рис. 7.5.

0	1	2	3	4	5	6
с	h	е	е	s	е	NUL

Рис. 7.5. Хранение строки с ограничением в С

Как видите, С использует 7 байт памяти для строки, хотя она состоит всего из шести символов, потому что для ограничителя требуется дополнительный байт.

Символ NUL хорош в качестве ограничителя, потому что большинство компьютеров имеют инструкцию, которая проверяет, равно ли значение строки 0. Любой другой символ потребует дополнительных инструкций для загрузки значения, которое мы будем проверять.

У использования терминатора строки вместо явной длины есть свои преимущества и недостатки. С одной стороны, хранилище компактно, что важно, и, по сути, не требует оверхедов для таких операций, как «выводить каждый символ, пока не будет достигнут конец строки». Однако для получения длины строки нам понадобится просканировать всю строку до конца, считая символы. Кроме того, при таком подходе в строке не может использоваться символ NUL.

Составные типы данных

Однокомнатные квартиры — это, конечно, неплохо, но чаще люди предпочитают что-нибудь получше, например апартаменты. В большинстве современных языков есть способы спроектировать «апартаменты» на свой вкус, называемые структурами. В качестве комнат в «апартаментах» выступают *элементы структур*.

Допустим, мы пишем программу «Календарь», которая содержит список (массив) событий с датами и временем их начала и окончания. Делая это на языке С, каждый из объектов, соответствующий дню, месяцу, часам, минутам и секундам, мы поместили бы в отдельный контейнер типа `unsigned char`, но для года следовало бы использовать контейнер типа `unsigned short`. На рис. 7.6 показана структура для размещения даты и времени.

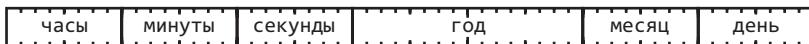


Рис. 7.6. Структура даты и времени

Замечу, что размещать дату и время подобным способом отнюдь не обязательно. Можно задействовать неструктурированные массивы часов, минут и т. д. Но использовать массивы со структурами «дата — время» гораздо удобнее. Такие программы воспринимаются гораздо легче. Британский ученый в области компьютерных наук Питер Ландин (Peter Landin) в 1964 году ввел термин «*синтаксический сахар*». Под сахаром подразумевались подобные структуры, которые как бы делали программы «слаще». Конечно, тут есть место жарким философским дебатам: то, что один считает подсластителем, для другого может оказаться необходимой функциональностью. Многие утверждают, что синтаксический сахар сводится к таким вещам, как замена `a = a + 1` на `a += 1` или `a++`, и лишь немногие — что массивы структур являются «сахаром» для наборов массивов. Оглядываясь назад, мы понимаем, что первое утверждение спорно: `a += 1` и `a++` были введены, поскольку компиляторам, которые в прошлом не были столь совершенны, было проще преобразовывать их в машинный язык. Вместе с тем, только появившись, структуры были сладче по сравнению с ранее использовавшимися массивами. Сейчас значение структур увеличилось, так как они широко используются в разработке программ.

Принципы работы с базовыми типами данных применимы и к составным типам данных наподобие структуры «дата — время». Рисунок 7.7 объединяет две структуры «дата — время» с небольшим массивом, хранящим строки с именем события. Это позволяет отразить в календаре всю информацию о событии.

Структуры часто занимают больше места, чем можно ожидать. Расположение данных в памяти с выравниванием по естественным границам и без него обсуждалось в разделе «Память» на с. 136. Допустим, мы разместили структуру



Рис. 7.7. Структура организации данных календаря

даты и времени в памяти 32-разрядного компьютера, как показано на рис. 4.2 на с. 138. Язык сохраняет элементы структуры в порядке, указанном программистом. Порядок может иметь значение. Но язык также должен соблюдать выравнивание (рис. 4.3 на с. 138). Язык не может поместить год в четвертый и пятый байты, как показано на рис. 7.7, потому что при этом произойдет пересечение границ. Средства языков программирования решают эту проблему, автоматически выделяя *дополнительное место* для элемента структуры. Фактическое использование памяти структурой будет выглядеть, как показано на рис. 7.8.

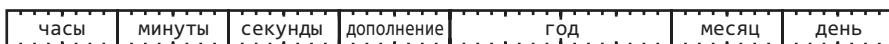


Рис. 7.8. Структура для даты и времени с дополнением

Можно перегруппировать элементы структуры, чтобы в итоге получить 7-байтовую структуру и избежать выделения дополнительного байта. Когда вы снова объедините пару таких элементов, языковые инструменты, скорее всего, расширят структуру до 8 байт.

Отмечу, что это выдуманный пример. Не обязательно размещать даты и время именно таким образом. Стандартным во многих системах подходом, пришедшим из UNIX и существующим с момента начала «эпохи UNIX» — 1 января 1970 года, является использование 32-битного числа для представления числа секунд. Возможности 32-битного числа для представления числа секунд будут исчерпаны в 2038 году, но во многих системах разработчики заранее подготовились, расширив это число до 64 бит.

На рис. 1.21 показано использование четырех 8-битных контейнеров для представления цвета с учетом прозрачности. Составная структура отлично справится с этой задачей, но ее использование не всегда оправданно с точки зрения доступа к данным. Например, если требуется скопировать значение цвета, то лучше скопировать все 32 бита сразу, а не делать четыре копирования по 8 бит за раз. Помочь в этом случае может другой тип структур.

Существуют не только «апартаменты», рассмотренные в предыдущем разделе, но и «офисы с передвижными перегородками». Такие офисы в языке С называются *объединениями*. Объединение позволяет упорядочивать представление данных в определенном сегменте памяти. Разница между структурой и объединением

в том, что элементы, входящие в структуру, занимают память, а элементы, входящие в объединение, делят эту память между собой. На рис. 7.9 для создания объединения структура RGBa совмещена с беззнаковым длинным числом.



Рис. 7.9. Объединение для пикселя

Используя объединение и синтаксис языка C, можно установить значение `pixel.color`, равное 0x12345678, следовательно, `pixel.components.red` будет равно 0x12, `pixel.components.green` будет равно 0x34 и т. д.

Односвязные списки

Массивы — это наиболее эффективный способ хранения списков. Они содержат только сами данные (`data`), не требуя дополнительной информации для их идентификации. С произвольными объемами данных дело обстоит иначе, потому что при создании недостаточно большого массива, при его заполнении придется создавать новый массив большего размера и копировать в него данные из старого. А чрезмерно большой массив займет лишний объем памяти, который не будет использоваться. К тому же без копирования не обойтись при вставке элемента в середину массива или его удалении.

Если точное количество объектов неизвестно, лучшим решением будут *связанные списки*. Односвязные списки с использованием структур показаны на рис. 7.10.

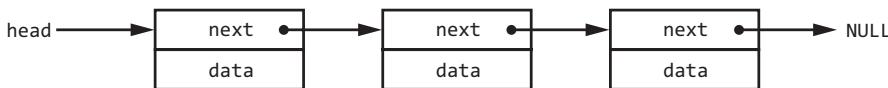


Рис. 7.10. Односвязный список

Обратите внимание, что `next` — это указатель, содержащий адрес следующего элемента в списке. Первый элемент в списке известен как `head`; последний элемент — `tail`. Можно распознать окончание списка, потому что указатель `next` перед концом списка содержит значение, которое не может соответствовать элементу списка. Обычно это значение `NULL`.

В отличие от списка, показанного на рис. 7.10, все элементы массива расположены в памяти последовательно. Элементы списка могут находиться в любом месте памяти, их организация показана на рис. 7.11.

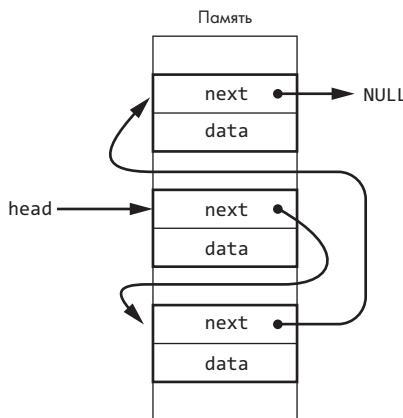


Рис. 7.11. Односвязный список в памяти

Добавить элемент в список легко; нужно просто поместить его перед первым элементом списка, как показано на рис. 7.12.

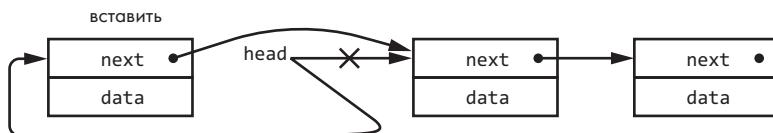


Рис. 7.12. Вставка элемента в односвязный список

Удалить элемент немного сложнее. Для этого необходимо, чтобы указатель **next** из элемента перед удаляемым направлял на следующий за удаляемым элемент, как показано на рис. 7.13.

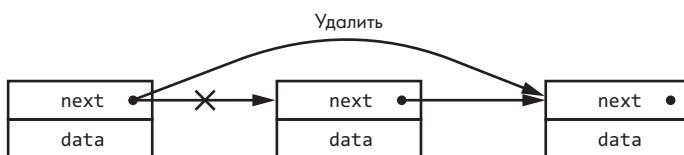


Рис. 7.13. Удаление элемента из односвязного списка

Один из способов удаления связан с использованием пары указателей и показан на рис. 7.14.

Указатель **current** перемещается по списку в поисках элемента для удаления. Указатель **previous** позволяет поместить указатель **next** перед удаляемым элементом списка. Для обозначения элемента используется точка (.), то есть **current.next** означает следующий элемент текущего узла.

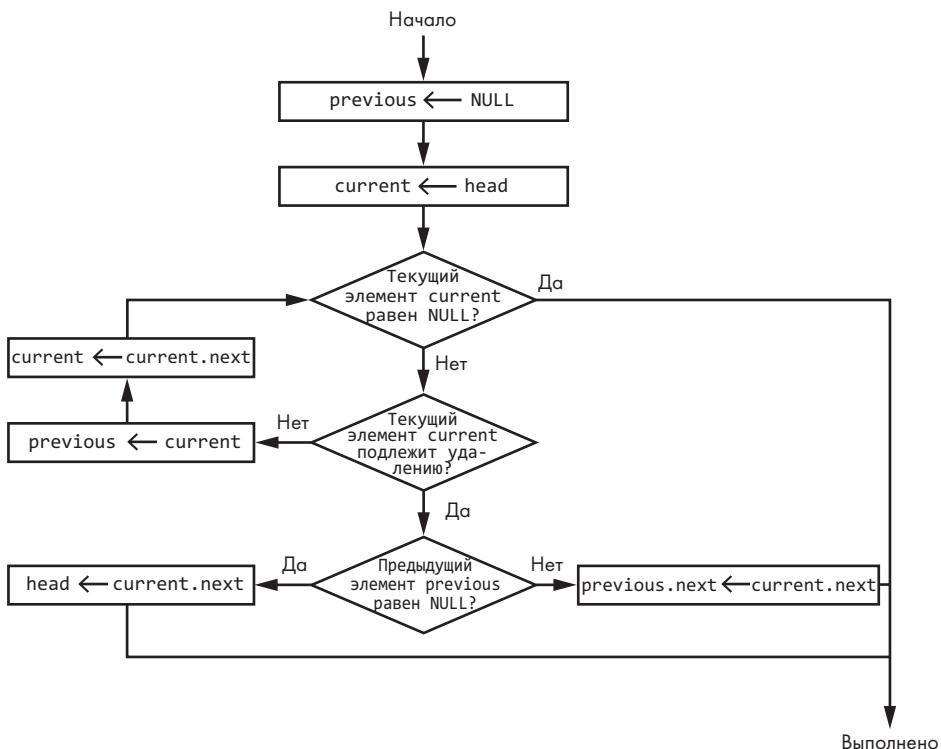


Рис. 7.14. Удаление элемента односвязного списка с помощью пары указателей

ПРИМЕЧАНИЕ

Рисунок 7.14 не идеальный пример; хотя, если честно, при работе над этим разделом мне встретились в Сети и намного худшие алгоритмы. Проблема с этим кодом в том, что необходимость специальной проверки начального элемента списка его усложняет.

Алгоритм на рис. 7.15 показывает, как использование *косвенной адресации второго порядка* устраниет потребность в сложной логике и делает код проще.

Рассмотрим работу этого алгоритма подробнее. Взгляните на рис. 7.16. Индексы показывают, как изменяется значение `current` по мере выполнения алгоритма.

Шаги, показанные на рис. 7.16, сложны для понимания, поэтому рассмотрим их детально:

1. Мы начинаем с записи в `current0` адреса `head`, вследствие чего `current1` указывает на `head`. Это означает, что `current` указывает на `head`, который указывает на элемент списка A.

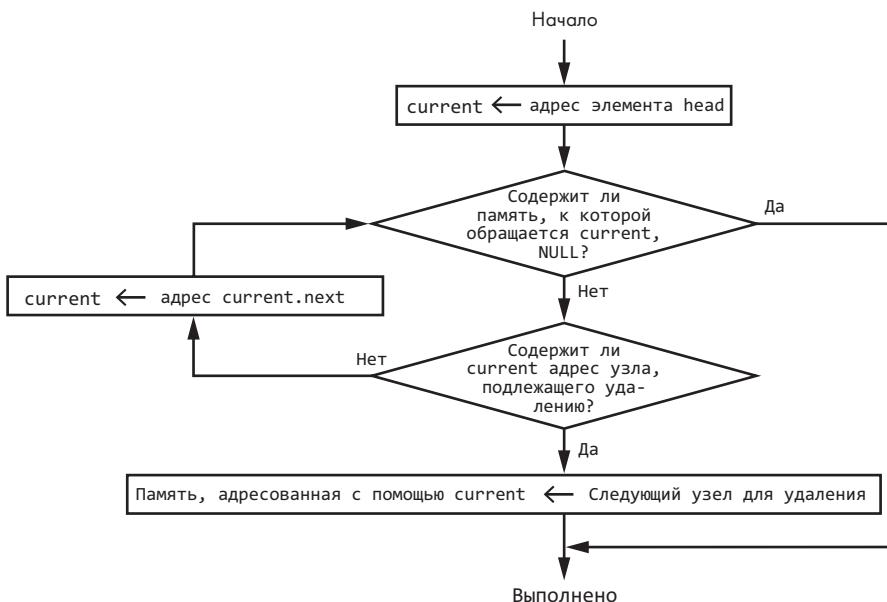


Рис. 7.15. Удаление элемента односвязного списка с использованием косвенной адресации

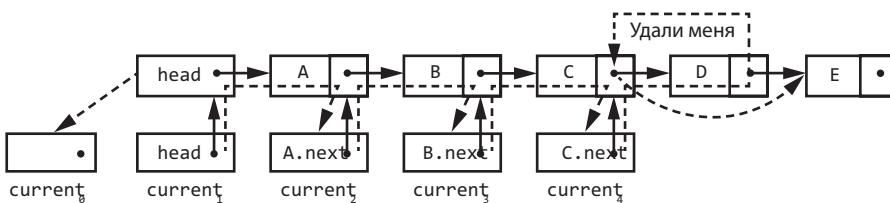


Рис. 7.16. Удаление элемента односвязного списка в действии

2. Мы не ищем элемент A, поэтому двигаемся дальше.
3. Как показано пунктирной стрелкой, мы записываем в `current` адрес из указателя `next` в элементе, на который `current` указывает в данный момент. Поскольку `current1` указывает на `head`, который указывает на элемент A, `current2` в конечном итоге указывает на `A.next`.
4. Это все еще не тот элемент, который мы хотим удалить, поэтому мы повторяем операцию, в результате чего `current3` ссылается на `B.next`.
5. Это все еще не тот элемент, который мы хотим удалить, поэтому мы повторяем операцию, при этом `current4` будет ссылаться на `C.next`.
6. `C.next` указывает на элемент D, который мы хотим удалить. По тонкой пунктирной стрелке мы проходим путь `current` от `C.next` до D и заменя-

ем `C.next` значением `D.next`. Поскольку `D.next` указывает на элемент `E`, `C.next` теперь указывает на `E`, как показано жирной пунктирной стрелкой, удаляя `D` из списка.

Можно изменить подготовительный алгоритм, добавив переходы к середине списка. Это полезно в случае необходимости упорядочивания по дате, имени или другим критериям.

Я упоминал, что второй алгоритм позволяет получить более качественный код. Сравним два варианта, записанные на языке программирования С. Чтобы увидеть разницу между листингом 7.1 и листингом 7.2, понимания кода не требуется.

Листинг 7.1. Код языка С для удаления элемента односвязного списка с помощью пары указателей

```
struct node {
    struct node *next;
    // data
};

struct node *head;
struct node *node_to_delete;
struct node *current;
struct node *previous;

previous = (struct node *)0;
current = head;

while (current != (struct node *)0) {
    if (current == node_to_delete) {
        if (previous == (struct node *)0)
            head = current->next;
        else
            previous->next = current->next;
        break;
    }
    else {
        previous = current;
        current = current->next;
    }
}
```

Листинг 7.2. Код языка С для удаления односвязного списка с использованием косвенной адресации второго порядка

```
struct node {
    struct node *next;
    // data
};

struct node *head;
struct node *node_to_delete;
struct node **current;
```

```
for (current = &head; *current != (struct node *)0; current = &(*current)->next)
    if (*current == node_to_delete) {
        *current = node_to_delete->next;
        break;
    }
}
```

Можно заметить, что код с косвенной адресацией в листинге 7.2 намного проще, чем код с парой указателей в листинге 7.1.

Динамическое выделение памяти

При рассмотрении вставки элемента в связанный список был опущен один важный момент. Я показал, как вставить новый узел, но не сказал, откуда взялась память для этого узла.

На рис. 5.16 мы видели, что место в памяти для данных программы начинается с раздела для статически выделенных данных, за которым следует куча, которую создает библиотека среды выполнения для этой программы. Это вся память, доступная программе (за исключением стека и векторов прерываний) на машинах без модулей управления памятью (MMU). Так как использование всего объема памяти не имеет смысла, на машинах с MMU библиотека среды выполнения запрашивает выделение объема памяти, который, по ее мнению, необходим программе. *Разрыв* (break) означает конец участка памяти, выделенной программе. Существуют некоторые системные вызовы, которые увеличивают или уменьшают объем доступной памяти.

Память для переменных, таких как массивы, является статической; то есть ей назначается адрес, который не меняется. Узлы списка, наоборот, являются динамическими; они поступают в ячейки и уходят из ячеек памяти по мере необходимости. Мы берем память для них из кучи.

Программе нужен способ управления кучей. Ей нужно знать, какая часть памяти используется, а какая является доступной для записи. Для этого существуют библиотечные функции, так что вам не придется писать собственные. В языке C это функции `malloc` и `free`. Посмотрим, как они применяются.

Один из способов применения `malloc` заключается в использовании структуры односвязного списка. Куча разделена на блоки, каждый из которых имеет размер (`size`) и указатель на следующий блок (`next`), как показано на рис. 7.17.

Изначально вся куча представляет один блок. Когда программа запрашивает выделение памяти, `malloc` ищет блок, в котором достаточно места, возвращает вызывающему объекту указатель на блок и корректирует размер блока, учитывая выделенную для программы память. Когда программа освобождает

память с помощью функции `free`, она просто помещает блок обратно в список свободных блоков.

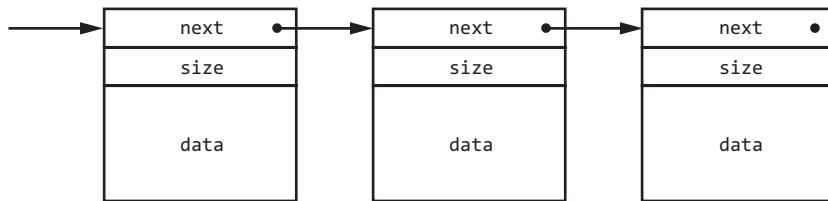


Рис. 7.17. Использование `malloc` для управления кучей

Время от времени `malloc` сканирует список в поиске свободных блоков, расположенных по соседству, и объединяет их в блок большего размера. Так как выделение памяти требует прохождения по элементам в списке в поисках достаточно большого блока, это можно осуществить при выделении памяти (вызове `malloc`). Со временем память может стать *фрагментированной*, что означает отсутствие достаточного по объему блока памяти при наличии множества небольших блоков. В системах с MMU место разрыва можно корректировать, что дает возможность использовать больший объем памяти при необходимости.

Этот подход сопряжен со значительными оверхедами: `next` и `size` добавляют 16 байт к каждому блоку на 64-разрядной машине.

Освобождение нераспределенной памяти — частая ошибка неопытных программистов. Другая ошибка — учет памяти, которая уже освобождена. Как показано на рис. 7.17, записывая данные за пределы выделенной памяти, можно исказить значения `size` и `next`. Это чревато сбоями, когда в дальнейшем потребуется использовать информацию из этих полей.

Один из побочных эффектов технологического прогресса — наличие у небольших вычислительных машин гораздо большего объема оперативной памяти, чем требуется программе. В таких случаях лучше распределить память статически. Это сократит оверхеды и позволит избежать ошибок, связанных с выделением памяти.

Более эффективное выделение памяти

Часто встречаются связанные списки, содержащие текстовые строки. Предположим, у нас есть связанный список, в котором узел содержит указатель (`pointer`) на строку (`string`), как показано на рис. 7.18.

Мы должны выделить память не только для каждого узла, но и для строки, прикрепленной к узлу. Оверхеды на `malloc` могут быть значительными, особенно на 64-разрядной машине, где будет использоваться 16 байт оверхедов

для 16-байтного узла, а затем еще 16 байт для указания строки. Взгляните на 4-байтную строку с `cat`, показанную на рис. 7.18.

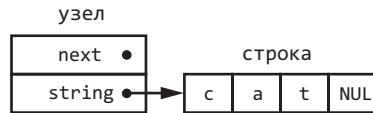


Рис. 7.18. Узел списка со строкой

Можно сократить оверхеды, упаковав узел и строку в один контейнер. Вместо выделения места для узла, а затем строки можно выделить пространство для узла и строки, сложив их объемы и добавив дополнительное пространство, которое может потребоваться для корректного размещения в памяти. Узлы имеют переменный размер, и это нормально. Этот способ вдвое сокращает оверхеды. Результат со строкой `cat` показан на рис. 7.19.



Рис. 7.19. Более эффективное выделение памяти

Этот подход эффективнее и при удалении узлов. В первом варианте для удаления данных потребуются два вызова функции `free`: один — для строки, а другой — для узла. Во втором случае требуется лишь один вызов функции `free`.

Сборка мусора

При применении динамического управления памятью могут возникнуть две проблемы, которые являются результатом неаккуратного использования указателей. Напомню, что указатель — это просто число, представляющее адрес в памяти. Но не все числа являются допустимыми адресами. Указание на несуществующую память или память, которая не соответствует заданным процессором правилам выравнивания данных, может вызвать ошибку обработки и сбой программы.

Возможно, вы изучаете язык программирования, такой как Java или JavaScript, который не имеет указателей, но поддерживает динамическое выделение памяти без использования аналогов `malloc` и `free`. Вместо этого названные языки

реализуют *сборку мусора* — метод, изобретенный в 1959 году американским специалистом по computer science и когнитивистом Джоном Маккарти (John McCarthy) (1927–2011) для языка программирования LISP. Сборка мусора пережила второе рождение, став средством для решения проблемы неправильного использования указателей.

Такие языки, как Java, используют ссылки вместо указателей. *Ссылки* — это абстракции для указателей, обеспечивающие большую часть той же функциональности без фактического предоставления адресов в памяти.

Языки с автоматической сборкой мусора часто имеют в своем арсенале оператор `new`, который создает элементы и выделяет для них память (этот оператор также имеется в языках, не использующих автоматическую сборку мусора, таких как C++). Для удаления элемента не существует определенного оператора. Вместо этого языковая среда выполнения отслеживает использование переменных и автоматически удаляет те, которые, по ее мнению, больше не используются. Способов для этого много, и один из них заключается в подсчете ссылок на переменные и удалении переменных в случае отсутствия ссылок на них.

Сборка мусора — это компромисс; он имеет и свои недостатки. Одна из проблем сборки мусора аналогична проблеме обновления LSI-11 (см. «Оперативная память» на с. 125). Суть ее в том, что программист не имеет достаточного контроля над сборкой мусора, которая может запуститься, даже несмотря на то, что программе нужно сделать что-то более важное. Кроме того, программы, как правило, занимают много памяти, потому что легче оставлять ненужные ссылки, а не удалять их. Такой подход препятствует восстановлению памяти и вызывает замедление работы программы. Однако сбоя программы, который наступил бы из-за некорректных указателей, в данном случае не происходит. Несмотря на преимущество решения проблем с указателями, проблему отслеживания ненужных ссылок порой устраниить труднее.

Двусвязные списки

Так как при удалении элемента для обработки указателя в односвязном списке нужно найти элемент, предшествующий удаляемому, функция `delete` может быть довольно медленной. Если список большой, поиск может выполняться очень долго. К счастью, существует другой тип списка, который решает эту проблему за счет использования дополнительной памяти.

Двусвязный список содержит ссылку не только на следующий элемент, но и на предыдущий, как показано на рис. 7.20. Это удваивает оверхеды для каждого узла, но устраняет необходимость перемещения по списку для функции `delete`. Это позволяет достичь компромисса между количеством необходимой памяти и временем выполнения.

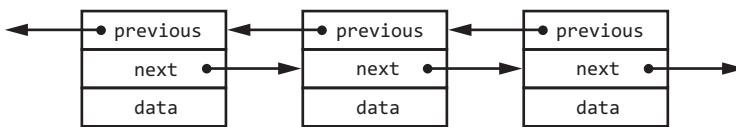


Рис. 7.20. Двусвязные списки

Преимущество двусвязного списка в том, что можно вставлять и удалять элемент в любом месте списка без просмотра ненужных элементов. На рис. 7.21 показано, как происходит добавление нового узла после элемента А.

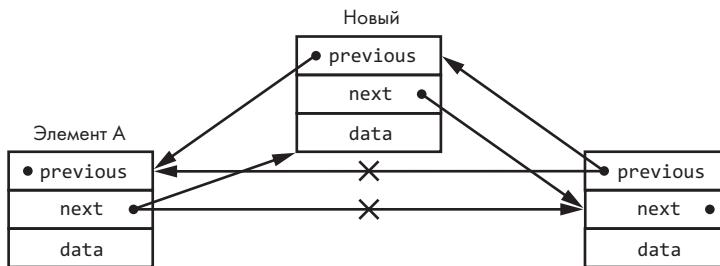


Рис. 7.21. Вставка в двусвязном списке

На рис. 7.22 показано, что удалить элемент так же просто.

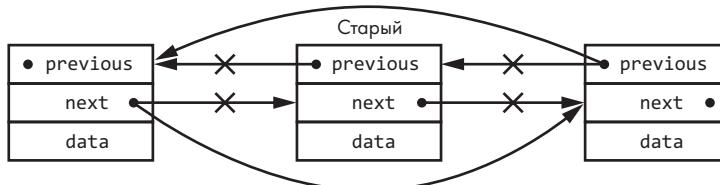


Рис. 7.22. Удаление в двусвязном списке

Как видите, при выполнении этих операций не нужно просматривать все элементы списка.

Иерархические структуры данных

До сих пор мы рассматривали только *линейные* структуры данных. Они отлично подходят для многих приложений, но в какой-то момент их линейность может стать проблемой. Хранить данные — лишь половина дела; необходимо еще и эффективно их извлекать. Допустим, у нас есть перечень элементов, хранящихся в связанным списке. Возможно, нам придется пройти весь список, чтобы найти

искомый элемент; для списка длиной n может потребоваться n просмотров. Это не критично для небольшого списка, но для больших значений n подобный подход неэффективен.

Ранее мы рассмотрели использование указателей для соединения узлов в списки. Мы не ограничены количеством указателей — только воображением и объемом памяти. Можно расположить узлы иерархически, как в примере на рис. 5.4.

Простейшей иерархической структурой данных является *двоичное дерево* — «двоичное» не из-за двоичных чисел, а потому, что узел может соединяться с двумя другими узлами. Создадим узел, содержащий число (*number*), как показано на рис. 7.23.

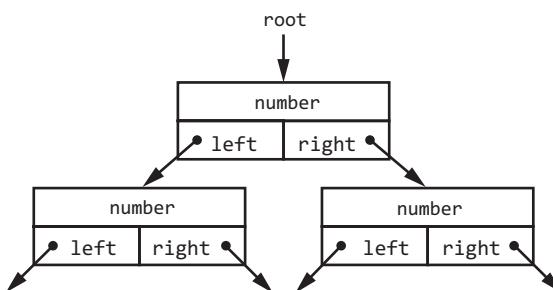


Рис. 7.23. Узлы двоичного дерева, содержащие числа (left — левый, right — правый)

Корень (root) для древовидной структуры — это аналог элемента *head* из связанных списков.

Поиграем в лото и запишем числа в форме двоичного дерева по мере их выпадения. Затем посмотрим номера и узнаем, какие числа выпали во время игры. На рис. 7.24 показан алгоритм, который добавляет число в дерево. Он работает аналогично удалению узла в односвязном списке, в том смысле что он основан на косвенной адресации.

Посмотрим на алгоритм в действии, вставив числа 8, 6, 9, 4 и 5. Когда мы вставляем 8, к корню root ничего еще не прикреплено, поэтому прикрепляем это число. Когда мы вставляем 6, корень уже занят, поэтому сравниваем число 6 со значением в корневом узле; затем, поскольку 6 меньше 8, выбираем левую сторону. Она свободна, поэтому прикрепляем там новый узел. 9 находится справа от 8, 4 — слева от 6 и т. д., как показано на рис. 7.25.

Хотя в этой структуре содержится пять узлов, но, проверив в худшем случае всего три узла, мы найдем нужный. Это гораздо эффективнее связанных списков, где может потребоваться проверить все пять узлов. Искать элемент в двоичном дереве, как показано на рис. 7.26, достаточно легко. В указателях нет необходимости, так как дерево менять не нужно.

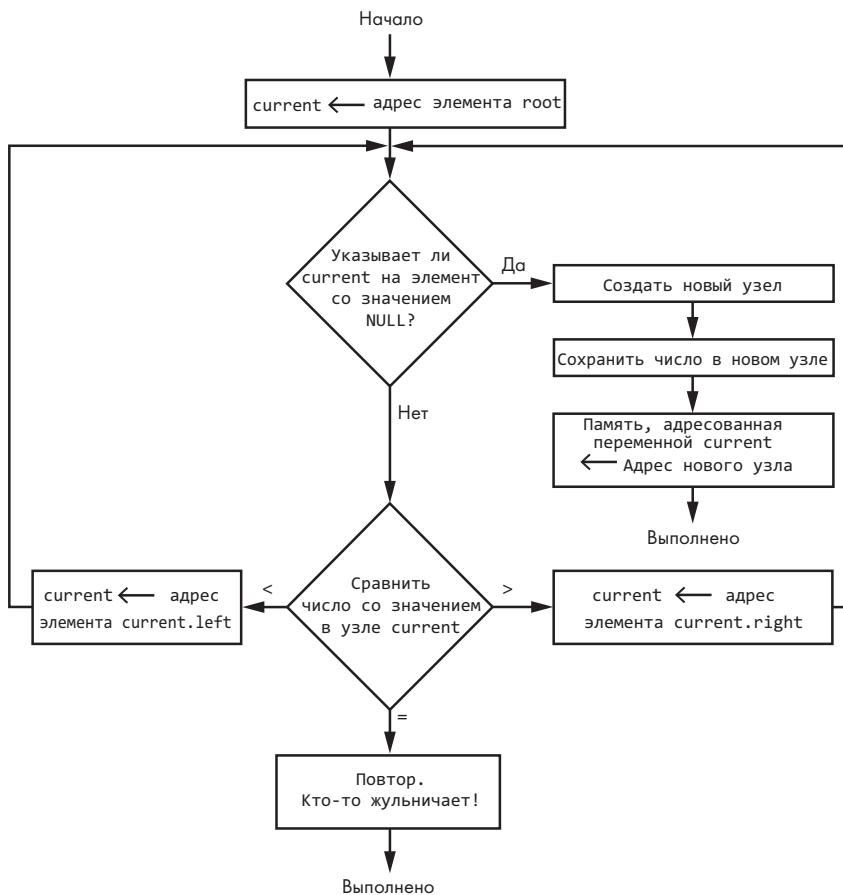


Рис. 7.24. Алгоритм вставки в двоичное дерево

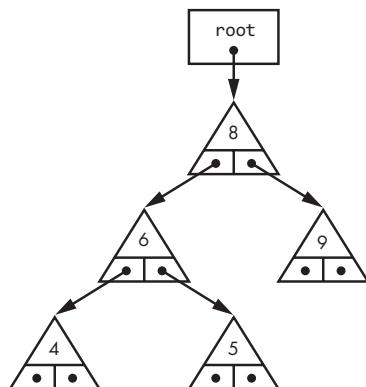


Рис. 7.25. Двоичное дерево

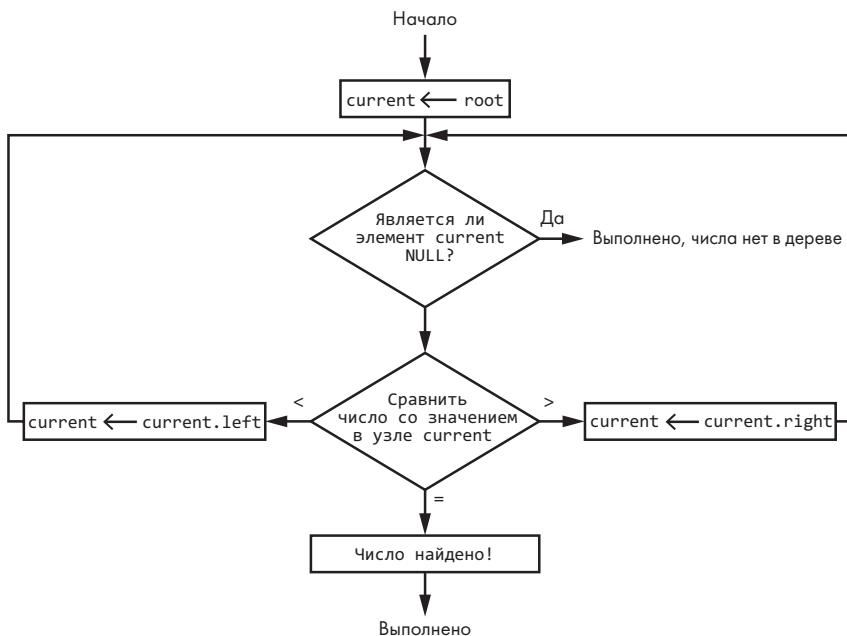


Рис. 7.26. Алгоритм просмотра двоичного дерева

Примечательно, что распределение чисел по ветвям зависит от порядка вставки. На рис. 7.27 показано, что произойдет, если вставить числа по порядку: 4, 5, 6, 8 и 9.

Этот вырожденный случай очень похож на односвязный список. Мы не только теряем преимущества двоичного дерева — из-за неиспользуемых левых указателей снижается эффективность использования памяти. Лучше, чтобы дерево выглядело, как показано на рис. 7.28 справа.

Поиск в двоичном дереве зависит от глубины дерева; если элемент находится на n уровнях ниже корня, то для его поиска требуется n проверок. Для этого нужно только $\log_2 n$ запросов в случае сбалансированного двоичного дерева, а не n запросов, как в связанным списке. Можно утверждать, что в худшем случае придется просмотреть 1024 узла в связанным списке, содержащем 1024 узла, но нужно будет проверить только 10 узлов в сбалансированном двоичном дереве.

Существует множество алгоритмов балансировки деревьев, которые я не буду здесь подробно описывать. Чтобы сбалансировать дерево, требуется время, так что необходим компромисс между скоростью алгоритма, временем вставки/поиска и временем балансировки. Алгоритмы балансировки деревьев требуют затрат вычислительной мощности, а некоторые — еще и выделения дополнительной памяти для хранения. Однако использование дополнительной памяти довольно скоро, по мере увеличения размера дерева, оправдывает себя, поскольку $\log_2 n$ становится намного меньше n .

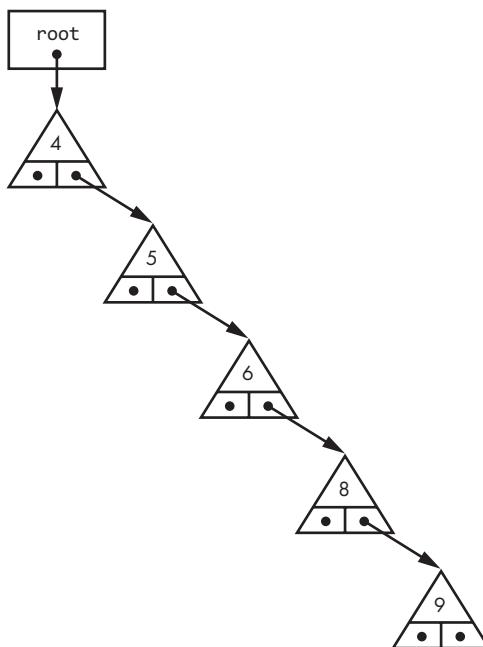


Рис. 7.27. Плохо сбалансированное двоичное дерево



Рис. 7.28. Несбалансированное и сбалансированное двоичные деревья

Хранение данных на дисковых устройствах

О дисководах мы говорили еще в разделе «Блочные устройства» на с. 129. Рассмотрим их подробнее, чтобы понять особенности организации данных при использовании этих устройств. Осторожно: впереди множество указателей!

Я упоминал, что основной единицей на диске является *блок*. Последовательно расположенные блоки называются *кластерами*. Данные можно хранить в кластерах, занимающих смежные секторы на дорожке. Так поступают, когда требуется очень высокая производительность. В остальных случаях это не очень хороший способ. К тому же количество данных может превышать объем, размещаемый на одной дорожке. Поэтому данные хранятся в любых доступных секторах. Иллюзию непрерывного хранения обеспечивает драйвер устройства операционной системы. Теперь ситуация выглядит знакомой, но есть нюанс: вместо одного

блока для хранения объекта нам нужно найти достаточное количество блоков фиксированного размера и разделить объект между ними.

Связанные списки — не самое лучшее решение для учета свободных и используемых блоков диска, потому что последовательный анализ элементов происходит слишком медленно. На диске 8 ТиБ содержится почти 2 миллиарда блоков, и при наихудшем сценарии доступны 250 блоков в секунду, то есть просмотр всего диска займет более 15 лет, что делает этот способ непрактичным. Звучит действительно удручающе, но имейте в виду, что это 1 МиБ данных в секунду.

Когда мы управляем данными в основной памяти, достаточно ссыльаться на них с помощью указателя. Но указатели являются временными объектами, а так как диски используются для долгосрочного хранения данных, нужно что-то более постоянное. Вы уже видели ответ: *имена файлов*. Необходимо каким-то образом сохранить их на диск и связать их с блоками, используемыми для хранения данных самих файлов.

Один из способов управления этими операциями восходит к — совершенно верно — UNIX. Ряд блоков выделяется в качестве *айнодов* (inode) — название, образованное соединением сокращения слова «индекс» (index) и слова «узел» (node); таким образом, айноды являются индексными узлами. Айнод содержит сведения о файле, такие как информация о владельце, размере и разрешениях на доступ, а также индексы блоков, которые содержат данные файла, как показано на рис. 7.29.

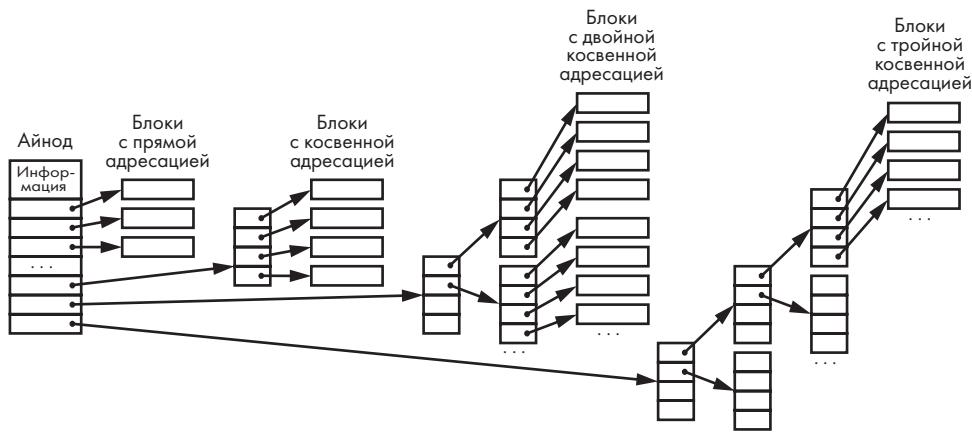


Рис. 7.29. Структура данных в файловой системе

Выглядит довольно сложно, но только на первый взгляд. Айнод обычно имеет 12 **прямых указателей на блоки** (на самом деле это не указатели, а просто индексы блоков), которые поддерживают файлы длиной до $4096 \times 12 = 49\,152$ байта. Этого достаточно для большинства файлов. Если файл больше, то используются

блоки *косвенной адресации*. Предполагая использование 32-разрядных индексов (хотя скоро они должны быть 64-разрядными), можно в одном айноде записать индексы 1024 блоков косвенной адресации, каждый из которых занимает по 4 байта, помещенных в один блок, добавляя еще 4 МиБ к максимальному размеру файла. Если этого недостаточно, 4 ГиБ доступны при использовании блоков с *двойной косвенной адресацией* и, наконец, еще 4 ТиБ — через использование блоков с *тройной косвенной адресацией*.

Часть информации, хранящейся в айноде, показывает, что содержат блоки: сведения о *каталоге* или другие данные. При помощи каталога устанавливается связь между именами файлов и индексами, которые соответствуют файлам. Одна из приятных особенностей работы UNIX состоит в том, что каталог на самом деле — просто файл другого типа. Это означает, что он может ссылаться на другие каталоги, что и дает знакомые *иерархические файловые системы* с древовидной структурой.

Все это кажется очень похожим на произвольное дерево. Так и есть, но лишь до определенного момента. Одной из особенностей иерархического расположения является возможность для нескольких айнодов ссылаться на одни и те же блоки с помощью *ссылок*. Ссылки позволяют одному и тому же файлу находиться в нескольких каталогах. Оказывается, очень удобно также ссылаться на каталоги. Для этого были изобретены *символические ссылки*. Но символические ссылки могут привести к зацикливанию в графе файловой системы, поэтому нужен специальный код программы для обнаружения и предотвращения бесконечного цикла. В любом случае эта сложная иерархическая структура отслеживает используемые блоки, но еще не хватает эффективного способа отслеживать *свободное пространство*.

Один из способов отследить свободное пространство — использовать битовую карту (см. «Битовые матрицы» на с. 237) с 1 битом для каждого блока на диске. Битовая карта может быть довольно большой: дисковому накопителю объемом 8 Тбайт потребуется почти 2 миллиарда бит, на что уйдет около 256 Мбайт. Однако этот способ все-таки эффективен. Тратится менее чем 0.01 % от общего объема дискового пространства, и не обязательно это должно быть в памяти одновременно.

Работа с битовыми картами довольно проста и эффективна, особенно если они хранятся в 64-битных словах. Предположив, что 1 указывает на используемый блок, а 0 указывает на свободный блок, можно легко искать слова, которые состоят не только из единиц, чтобы находить свободные блоки.

Но есть проблема: граф файловой системы и битовая карта могут рассинхронизироваться. Например, может произойти сбой питания во время записи данных на диск. В «темные времена», когда на компьютерах были переключатели и мигающие лампочки, пришлось бы восстанавливать поврежденную файловую систему, вводя номера айнодов с помощью этих самых переключателей. Необходимость

в этом отпала с приходом программы `fsck`. Она следует по графу файловой системы и сопоставляет его с данными о свободных блоках. Это лучше, чем было, но по мере увеличения дисков обработка занимает все больше и больше времени. Новые архитектуры файловой системы с ведением журнала повышают эффективность восстановления поврежденных данных.

Базы данных

Двоичные деревья — отличный способ хранения данных в памяти, но они не так хороши для хранения огромных объемов данных, которые не помещаются в память. Отчасти это связано с тем, что узлы дерева, как правило, небольшие и поэтому плохо справляются с хранением данных о секторах диска.

База данных — это просто набор данных, организованных определенным образом. *Система управления базами данных (СУБД)* — это программа, позволяющая хранить информацию в базе данных и извлекать ее оттуда. СУБД обычно включает в себя ряд интерфейсов, расположенных поверх базового механизма хранения.

Базы данных являются примером применения *B-дерева* — структуры данных, изобретенной немецким ученым в области компьютерных наук Рудольфом Байером (Rudolf Bayer) и его американским коллегой Эдом Маккрайтом (Ed McCreight) в «Боинге» в 1971 году. В-дерево — это сбалансированное, но не двоичное дерево. Оно требует немного больше места, чем сбалансированное двоичное дерево, но работает лучше, особенно когда данные хранятся на диске. Это еще один пример того, как понимание архитектуры памяти позволяет более эффективно ее использовать.

Допустим, у нас есть сбалансированное двоичное дерево имен, упорядоченных по алфавиту. Оно будет выглядеть примерно так, как показано на рис. 7.30¹.

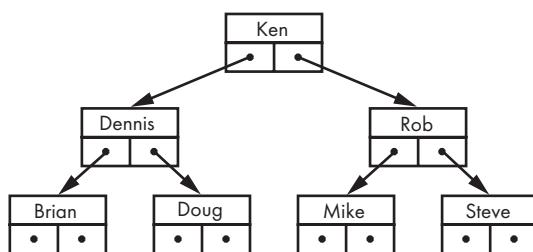


Рис. 7.30. Сбалансированное двоичное дерево

¹ Здесь и далее имена и фамилии приведены без перевода, так как пример основан на английском алфавите. — Примеч. ред.

Узел В-дерева имеет намного больше ответвлений (дочерних элементов) по сравнению с узлами двоичного дерева. Количество ответвлений выбирается так, чтобы узел точно вписывался в дисковый блок, как показано на рис. 7.31.

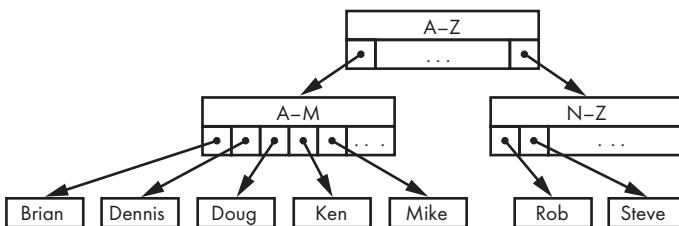


Рис. 7.31. В-дерево

Как видите, внутренние узлы сбалансированы, что обеспечивает предсказуемое время поиска. На рис. 7.31 есть неиспользуемые дочерние ссылки, которые занимают много места. Когда дочерние ссылки заканчиваются, дерево легко перебалансировать, просто изменив диапазон, охватываемый узлом. Например, если у узла А-М закончились дочерние узлы, его можно разделить на узлы А-Г и Н-М. Алфавит не лучший пример, так как оптимальным подходом в подобных случаях является разделение на части, кратные двум. Число же элементов в этом примере нечетное.

Большее количество ссылок в узле означает меньше узлов. Крупные узлы не вызывают проблем, так как они не превышают размера дискового блока, который может быть обработан как единое целое. Немного пустого пространства из-за неиспользуемых дочерних ссылок остается, но это разумный компромисс.

Индексы

Упорядочивание данных позволяет организовать эффективный доступ к ним. Однако нам часто требуется доступ к данным, упорядоченным более чем по одному критерию, например имена и фамилии или имена и любимые группы.

На рис. 7.31 показаны узлы, упорядоченные по именам. Эти узлы часто называют *первичным индексом*. Но индексов может быть более одного, как показано на рис. 7.32. Несколько индексов позволяют эффективно искать объекты различными способами.

Издержки при работе с индексами заключаются в необходимости их обслуживания. Каждый индекс должен обновляться при изменении данных. Если поиск выполняется чаще модификации, эти издержки оправданы.

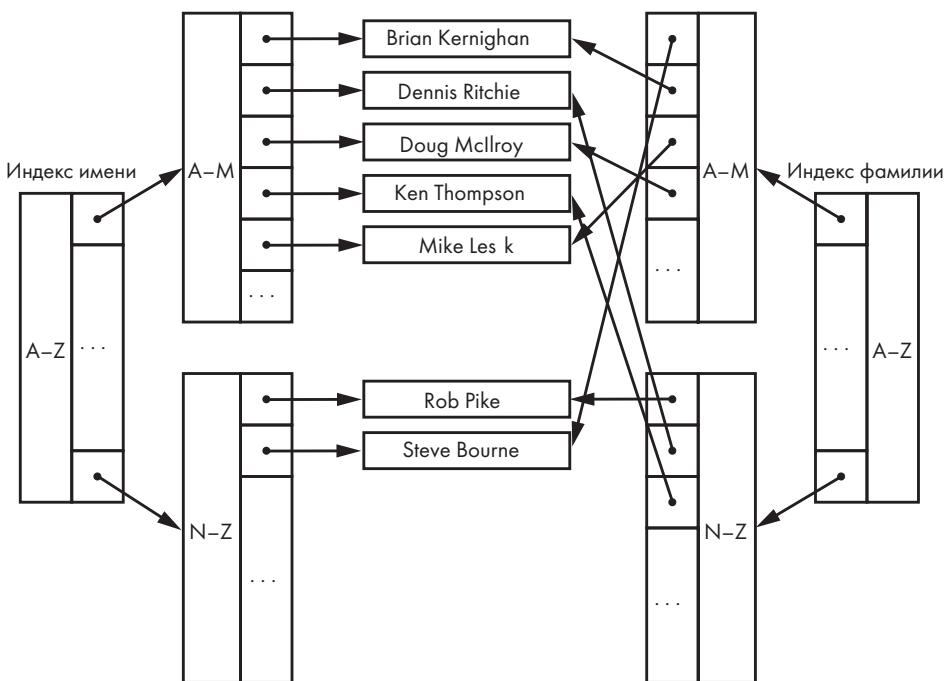


Рис. 7.32. Множественные индексы

Перемещение данных

Ранее я упоминал, что использование массивов вместо связанных списков вынуждает копировать данные при необходимости увеличения массива. Копирование нужно, чтобы перемещать таблицы страниц в ММУ и из него, битовые карты на диск и с диска и т. д. Программы тратят много времени на перемещение данных из одного места в другое, поэтому важно делать это эффективно.

Начнем с полумеры: установим значение длины `length` всех блоков памяти равным 0, как показано на рис. 7.33.

Этот алгоритм работает, но не очень эффективно. Предполагая, что длительность выполнения каждого блока на рис. 7.33 одинакова, мы тратим больше времени на вспомогательные записи, чем на обнуление ячеек памяти. Применение *развертывания цикла* может сделать процесс более эффективным, что и показано на рис. 7.34. Предположив, что `length` — четное число, развернем цикл так, чтобы теперь больше времени тратилось на обнуление и меньше — на другие операции.

Универсальное средство решения подобных проблем, к счастью, имеется. Во время работы в Lucasfilm канадский программист Том Дафф (Tom Duff) изобрел *метод Даффа* для ускорения копирования данных; рис. 7.35 показывает

вариант обнуления памяти. Этот подход работает только в том случае, если `length` больше нуля.

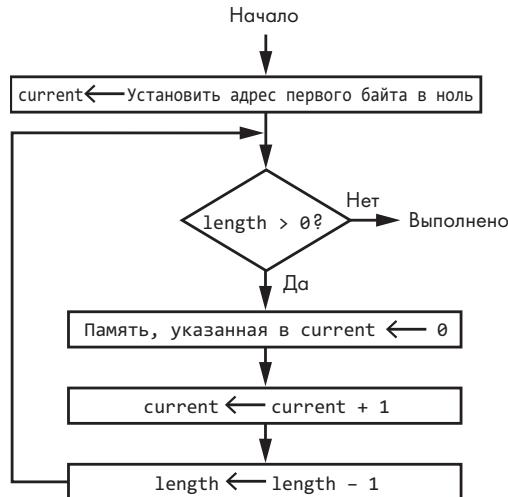


Рис. 7.33. Обнуление блока памяти

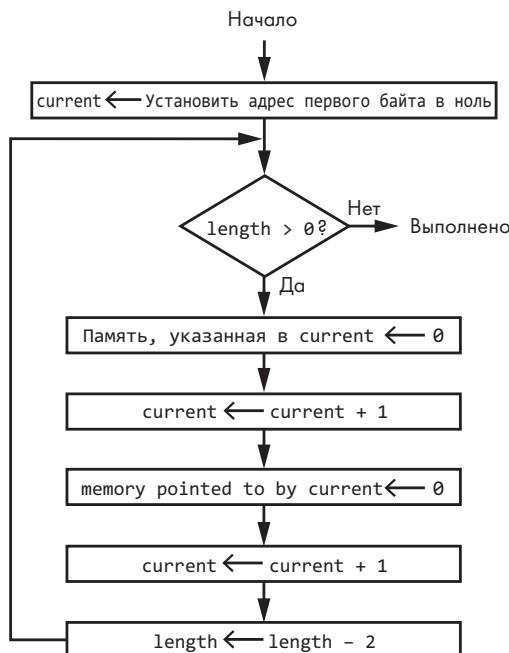


Рис. 7.34. Обнуление блока памяти с развертыванием цикла

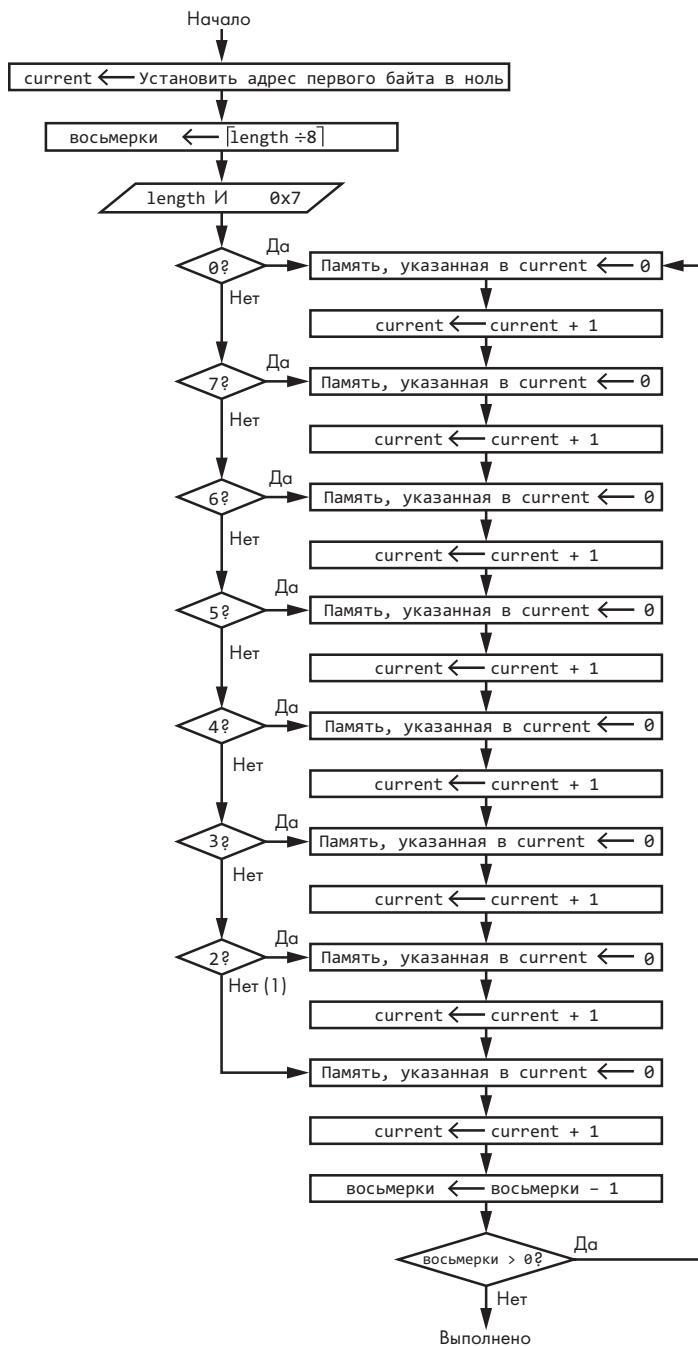


Рис. 7.35. Обнуление блока памяти с помощью модифицированного метода Даффа

Метод Даффа разворачивает цикл восемь раз и перемещается в середину, чтобы обработать все оставшиеся байты. Может возникнуть соблазн развернуть цикл еще раз, но необходимо соблюдать размер кода, потому что его размещение в кэше команд требует большой скорости.

На рисунке в части цикла видно, что соотношение времени восстановления памяти к времени вспомогательных операций значительно увеличилось. Хотя начальная настройка и ветвление в нужном месте цикла выглядят сложными, на самом деле это не так. Для этого не требуется множество условных ветвей, а всего лишь несколько операций с адресами:

1. Задать маску для всего, кроме трех младших битов, используя значение 0x7.
2. Вычесть результат из 8.
3. Задать маску для всего, кроме трех младших битов, используя значение 0x7.
4. Умножить на количество байтов между инструкциями по обнулению.
5. Добавить адрес первой инструкции по обнулению.
6. Ветвление по этому адресу.

Другой способ повысить эффективность — принять, что на 64-разрядной машине 8 байт могут быть обнулены одновременно. Конечно, для обработки оставшихся байтов в начале и в конце требуется дополнительный код. Необходимо использовать алгоритм рис. 7.36 без цикла на восьмерках для начала и конца. В середине обнуляем столько 8-байтовых блоков, сколько возможно.

Процесс усложняется, если вместо установления значения для блока копировать блок. Ведь, скорее всего, источник и место назначения не будут иметь одинаковое разбиение на байты. Следует проверять, что и источник, и приемник выровнены по машинным словам. Несоответствие между разбиением на слова — обычное дело.

Копирование имеет еще одну сложность. Она заключается в том, что копирование обычно используется для перемещения данных в области памяти. Например, у нас может быть буфер со словами, разделенными пробелами, в котором мы хотим прочитать первое слово и сжать все остальное, чтобы оставить место для дополнительных данных. Нужно быть осторожными при копировании данных в перекрывающихся областях; иногда приходится копировать данные обратно, чтобы избежать их перезаписи.

Интересен пример раннего терминала растровой графики (см. «Растровая графика» на с. 230) под названием *blit*, разработанный канадским программистом Робом Пайком (Rob Pike) в Bell Telephone Laboratories в начале 1980-х годов. В то время еще не было практики создания индивидуальных интегральных микросхем для выполнения подобных задач. Исходная и целевая области

памяти могли перекрываться, например в случае перетаскивания окна, и данные могли иметь любое выравнивание. Производительность была очень важна, потому что процессоры были не очень быстры по сравнению с сегодняшними; blit использовал Motorola 68000. MMU отсутствовал, поэтому Пайк написал код, который анализировал источник и место назначения и сходу генерировал оптимальный код для максимально быстрого копирования. Я создал аналог для системы, использующей Motorola 68020. Это позволило добиться еще более высокой производительности, ведь у 68020 был кэш команд, в который идеально помещался сгенерированный код, так что не нужно было постоянно обращаться к памяти с инструкциями. Следует отметить, что это послужило предвестником JIT-метода (метода «точно в срок», just-in-time), используемого на многих виртуальных машинах, включая Java.

Векторный ввод/вывод

Эффективное копирование данных важно для производительности системы, но полное исключение копирования — лучший вариант. Большое количество данных проходит через операционную систему в пользовательские программы и из них, и эти данные часто расположены в памяти отдельными фрагментами.

Предположим, что мы генерируем некие аудиоданные в формате mp3 и хотим записать их на аудиоустройство. Как и многие форматы файлов, mp3-файлы состоят из нескольких *фреймов*, каждый из которых содержит *заголовок*, за которым следуют данные. Типичный аудиофайл содержит несколько фреймов, которые во многих случаях имеют одинаковые заголовки, как показано на рис. 7.36.



Рис. 7.36. Макет фрейма mp3

Можно построить каждый фрейм, скопировав все данные в буфер. Но когда потребуется записать эти данные на аудиоустройство, придется копировать их еще раз. В качестве альтернативы можно записать каждую часть каждого фрейма отдельно, но это увеличит затраты времени и мощности на переключение контекста и в случае записи только части фрейма чревато проблемами с аудиоустройством.

Гораздо эффективнее просто передать системе набор указателей на каждый фрагмент фрейма и позволить собирать фреймы вместе в соответствии с порядком

их написания, как показано на рис. 7.37. При этом необходимо обеспечить поддержку системных вызовов (`readv`, `writev`).

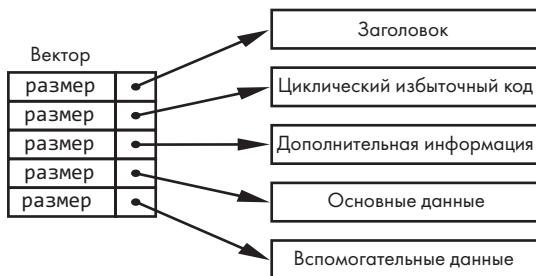


Рис. 7.37. Сбор данных

Идея состоит в том, чтобы передать вектор размеров и указателей на данные операционной системе, которая затем собирает их по порядку. Существуют версии как для чтения, так и для записи: запись известна как *сбор* (gathering), потому что данные собираются из многих мест, в то время как чтение известно как *рассеивание* (scattering), потому что данные распределены по многим местам. Вся концепция называется *рассеивание/сбор* (scatter/gather).

Рассеивание/сбор стали мейнстримом благодаря сетевому коду Беркли, на котором основана работа интернета. Я упоминал в разделе «TCP/IP» на с. 206, что IP-данные отправляются в пакетах, а TCP отвечает за передачу пакетов целиком и в правильном порядке. Пакеты, поступающие от конечной точки связи (таковой она может быть для вас, а для меня это сокет), собираются в непрерывный поток для предоставления их пользовательским программам.

Подводные камни объектно-ориентированного программирования

Поскольку вы учитесь программировать, возможно, вы изучаете *объектно-ориентированный* язык, такой как Java, C++, Python или JavaScript. Объектно-ориентированное программирование — отличная методология. Однако при неправильном подходе она может привести к проблемам с производительностью.

Объектно-ориентированное программирование получило серьезное развитие с языка C++. C++ — интересный пример. Изначально он был построен поверх языка C, что позволяет увидеть, как он работает.

Объекты имеют *методы*, эквивалентные функциям, и *свойства*, эквивалентные данным. Все необходимое для объекта может быть собрано в единую структуру данных. Использовать средства языка C для создания типов и указателей,

особенно указателей на функции, несомненно, хорошо. Структура объекта в языке С может выглядеть примерно так, как показано на рис. 7.38.

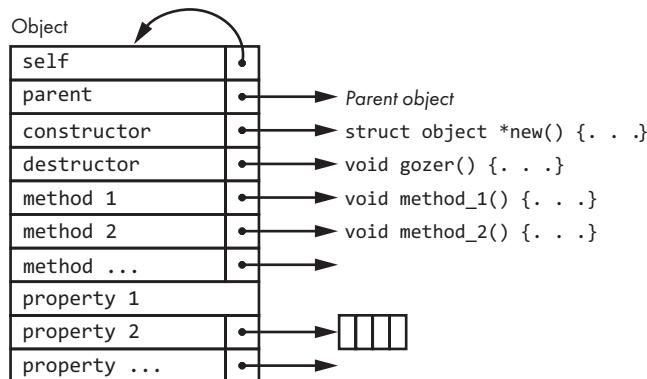


Рис. 7.38. Структура объекта в языке С (object — объект, self — собственно объект, parent — родитель, constructor — конструктор, destructor — деструктор, method — метод, property — свойство)

Некоторые свойства, такие как свойства с целочисленными значениями (Property 1), содержатся в самой структуре объекта, в то время как другие требуют дополнительной памяти (Property 2), на которую ссылается структура объекта.

Очевидно, что эта структура может стать довольно большой, особенно если существует множество методов. Можно решить эту проблему, выделив методы в отдельную структуру, как показано на рис. 7.39. Это еще один компромисс времени и памяти.

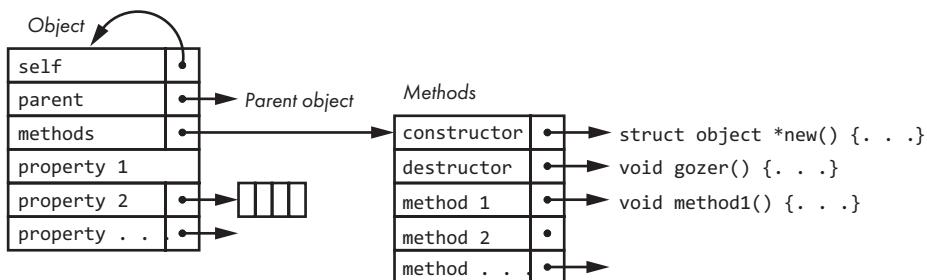


Рис. 7.39. Отдельная структура методов

Программисты использовали такой подход к объектно-ориентированному программированию задолго до того, как датский программист Бьорн Страуструп

(Bjarne Stroustrup) изобрел C++. Изначально C++ был оболочкой вокруг C, которая делала возможным подобные преобразования.

Почему это важно? Идеологи объектно-ориентированного программирования считают, что объекты способны решить любые задачи. Но, как видно на предыдущих рисунках, с объектами связано определенное количество вспомогательных данных. Объекты вынуждены использовать собственные методы вместо глобально доступных функций и в результате упаковываются не так плотно, как чистые типы данных. Поэтому когда производительность имеет первостепенное значение, лучше придерживаться классических массивов.

Сортировка

Сортировка данных нужна по множеству причин. Иногда отсортированные результаты, например имена в алфавитном порядке, легче искать. Очень часто хранение данных в отсортированном виде ускоряет поиск за счет сокращения количества обращений к памяти.

Я не буду углубляться в алгоритмы сортировки, потому что это довольно серьезная тема, широко освещенная в литературе. Существует множество хороших функций сортировки, так что вряд ли вам придется писать собственные — если только в качестве домашнего задания. Но есть несколько важных моментов, которые следует иметь в виду.

Один из них в том, что, если размер сортируемых объектов больше размера указателей на эти объекты, сортировку следует проводить, перемещая не сами данные, а указатели на них.

Кроме того, существует соглашение о сортировке. Дерево для игры в лото позволяло принимать решения на основе арифметического сравнения — было ли одно число меньше, равно или больше другого. Этот метод уходит корнями к языку программирования FORTRAN, созданному в 1956 году и включавшему инструкцию, показанную в листинге 7.3.

Листинг 7.3. Арифметическая операция IF (ЕСЛИ) в языке FORTRAN
IF (expression) branch1, branch2, branch3

Оператор IF оценивает выражение и переходит к branch1, если результат меньше нуля, branch2, если он равен нулю, и branch3, если он больше нуля. Ветви похожи на описанные в разделе «Ветвление» на с. 150.

Сортировка чисел является простой операцией. Хорошо бы применить эту же методологию к сортировке других объектов. На рис. 7.10 видно, что узел списка может содержать произвольные данные. То же самое верно для древовидных и других структур данных.

В версии UNIX III была представлена библиотечная функция `qsort`, которая реализовала классический *алгоритм быстрой сортировки*. Интересная особенность `qsort` заключалась в том, что функция знала, как сортировать объекты, но не умела их сравнивать. Из-за этого она использовала указатели на функции языка C; при вызове `qsort` со списком объектов для сортировки также вызывалась функция сравнения, которая возвращала `<0`, `0` или `>0` для значений «меньше», «равно» или «больше», как в арифметическом операторе FORTRAN `IF`. Этот подход позволил применять `qsort` произвольно. Например, если узел содержал как имя, так и возраст, дополнительная функция могла сравнивать элементы сначала по возрасту, а затем по имени. Таким образом, `qsort` выдавала результаты, упорядоченные по возрасту, а затем по имени. Данный подход работал хорошо и был скопирован во многих других системах.

С учетом этого была разработана стандартная функция сравнения строк библиотеки C `strcmp`. Она возвращает значение меньше, равное или больше нуля. Этот подход также был взят за основу в других системах.

Версия `strcmp`, изначально предназначенная для ASCII, просто проходила по строкам, выявляя отличия одного символа от другого. Она продолжала работать, если значение было равно нулю, и возвращала 0, если был достигнут конец строк. В противном случае возвращался результат вычитания.

Все это эффективно, если данные сортируются для распределения в дереве, и неэффективно, если данные сортируются для размещения в алфавитном порядке. Этот подход работал во времена ASCII. Из табл. 1.10 видно, что числовой и алфавитный порядок одинаковы. Но метод перестает работать при поддержке других локалей. Побочным эффектом поддержки других языков, проявившимся на более поздней стадии, является то, что только символы ASCII находятся в *сопоставимом порядке* сортировки или, иначе, в порядке, отражающем специфику языка.

Например, какое значение присвоить немецкой букве β — «острой» букве *S* (*Eszett* или *scharfes S*)? Ее значение в Юникоде равно 0x00DF. Из-за этого слово *Straße* будет расположено после слова *Strasse* в случае сравнения строк. Но на самом деле это разные представления одного и того же слова. β эквивалентно *ss*. Сравнение строк, учитывающее языковой стандарт, воспринимало бы эти слова как равные.

Создание хешей

Все методы поиска, которые мы встречали до сих пор, включали повторную проверку при анализе структуры данных. Есть еще один подход, называемый *хешированием*, который в определенных обстоятельствах работает лучше. Хеширование применяется во множестве случаев. Здесь мы говорим о хранении и извлечении данных из памяти процессора, а не о хранении на жестком диске.

Общая концепция заключается в применении некоторой *хеш-функции* к ключам поиска, которые путем этой хеш-функции равномерно распределяются по памяти. Если хеш-функцию легко вычислить и она превращает ключ в последовательность символов в уникальном месте в памяти, то это обеспечивает быстрый поиск ключа. Однако следует учитывать некоторые практические реалии.

Каждый «пласт» представляет хранилище для объекта, связанного с ключом. Хеш-функция должна выдавать значения, которые помещаются в памяти. Также она не должна распределять объекты по слишком большому объему памяти. В противном случае производительность пострадает из-за использования чрезмерного объема памяти и отсутствия места ссылки. Из-за отсутствия предварительных данных о ключах невозможно создать идеальную хеш-функцию.

Один из способов стабилизировать хранилище — использовать хеш-функцию, которая сопоставляет ключи и определенные индексы массива. Массив называется *хеш-таблицей*, пример которой представлен на рис. 7.40. Элементы массива называются *сегментами*.



Рис. 7.40. Хеширование

Что делает хеш-функцию хорошей? Она должна легко вычисляться и равномерно распределять ключи по сегментам. Простая хеш-функция, приемлемо работающая для текста, всего лишь суммирует значения символов. Этого недостаточно, потому что в результате суммирования может получиться индекс, выходящий за пределы хеш-таблицы. Но эту проблему легко решить, сделав индекс суммой по модулю размера хеш-таблицы. Посмотрим, как это происходит. Возьмем размер таблицы, равный 11; лучше использовать простые числа для размеров таблиц, потому что часть суммы, кратная 2, попадает в разные сегменты, улучшая распределение результатов.

Допустим, у нас есть приложение для песен, сыгранных на концертах нашей любимой группы. Возможно, в нем хранится дата последнего воспроизведения. Мы просто будем использовать первое слово в названии каждой песни.

Как видно на рис. 7.41, мы начинаем с «Hell in a bucket» — в данном случае используем сегмент 4. Далее идет «Touch» в сегменте 9, за которым следует «Scarlet» в сегменте 3. Но когда мы добираемся до «Alligator», у нас возникает проблема, потому что значение хеш-функции такое же, как и для «Scarlet». Это называется *коллизией*.

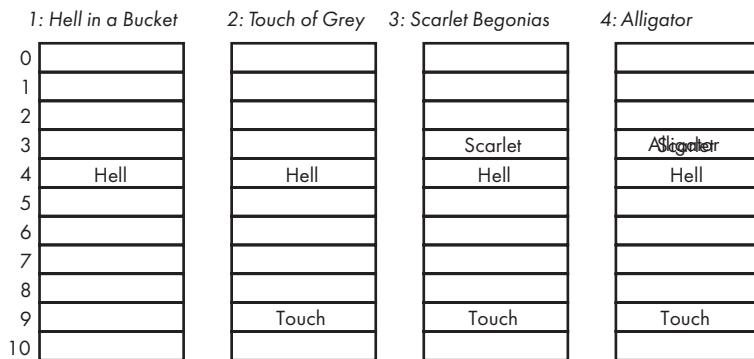


Рис. 7.41. Коллизия хеш-функции

Решим эту проблему, заменив сегменты *хеш-цепочками*, которые в простейшей форме представляют односвязные списки, как показано на рис. 7.42.

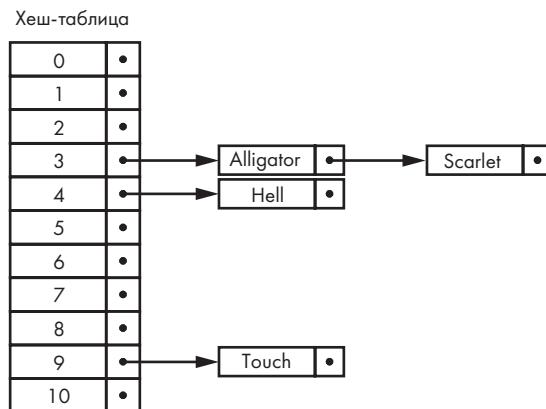


Рис. 7.42. Цепочки хешей

Вариантов управления хеш-цепочкой множество. Можно просто вставить коллизии в начало цепочки, как показано на рис. 7.42. Это быстро. Но поиск может замедляться по мере удлинения цепочек, поэтому можно выполнить сортировку вставками. Сортировка занимает больше времени, но исключает прохождение цепочки до конца, если искомый элемент найден. Существует также множество методов обработки коллизий — например, устранение цепочек хешей и использование некоторого алгоритма для поиска пустого места в таблице.

Не зная заранее ожидаемого количества символов, трудно подобрать оптимальный размер хеш-таблицы. Можно отслеживать длину цепочки и увеличивать хеш-таблицу, если цепочки становятся слишком длинными. Эта операция может быть дорогостоящей, но окупится по причине нечастого выполнения.

Существует множество вариантов хеш-функций. Святой грааль хеш-функций – это *идеальный хеш*, который сопоставляет каждый ключ с уникальным сегментом. Практически невозможно создать идеальную хеш-функцию, если все ключи не известны заранее, но математики придумали функции гораздо лучше тех, что используются в этом примере.

Эффективность и производительность

Создание эффективных алгоритмов поиска стоило множества усилий. Большая часть этой работы была проделана в эпоху, когда компьютеры были дорогими. Производительность и эффективность были взаимосвязаны.

Стоимость электроники упала настолько, что практически к любой покупке вам добавят горсть каких-нибудь светодиодов в виде бонуса. Производительность и эффективность больше не имеют тесной взаимосвязи; иногда более высокой производительности можно добиться за счет использования менее эффективных алгоритмов на большем количестве процессоров, а не за счет использования более эффективного алгоритма при меньшем количестве процессоров.

Такое разделение эффективности и производительности используется при *сегментировании* базы данных, также называемом *горизонтальным разделением*. Сегментирование включает в себя разделение базы данных на несколько сегментов, каждый из которых находится на отдельном компьютере, как показано на рис. 7.43.

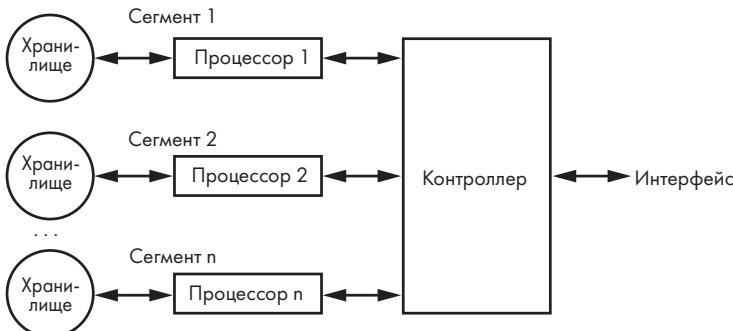


Рис. 7.43. Сегментирование базы данных

Операции с базой данных, запрошенные через интерфейс, отправляются всем сегментам, а результаты собираются контроллером. Этот метод повышает производительность, ведь операции распределяются между несколькими обработчиками.

Один из вариантов сегментирования называется *MapReduce*. По сути, он позволяет предоставлять код контроллеру для сборки промежуточных результатов.

Благодаря этому при выполнении таких операций, как «подсчитать количество учащихся во всех математических классах», не нужно запрашивать список учащихся перед подсчетом.

Базы данных – не единственное применение подхода с несколькими обработчиками. Интересен пример DES (стандарта шифрования данных) Фонда электронных инноваций, созданного в 1998 году; подробнее см. в книге «Cracking DES» (O'Reilly, 1998). Была сконструирована машина, которая использовала 1856 специальных чипов, каждый из которых подбирал ключи к зашифрованным данным. Любые «интересные» результаты направлялись контролеру для дальнейшего анализа. Эта машина могла тестировать 90 миллиардов ключей в секунду.

Выводы

В этой главе вы познакомились с рядом способов организации данных, позволяющих использовать преимущества аппаратной части компьютера. В следующей главе мы увидим, как программы преобразуются в понятные компьютеру формы.

8

Обработка языка



Только истинным безумцам может прийти в голову написать компьютерную программу. Так было всегда, но современные языки *программирования* хотя бы облегчают эту задачу.

В этой главе мы узнаем, как появились языки программирования. Вы начнете понимать, что происходит с кодом по мере его обработки. Также вы узнаете, как написанный код превращается в нечто, что компьютер может выполнить, — в *машинный язык*.

Язык ассемблера

Мы рассматривали реализацию программы на машинном языке для вычисления последовательности Фибоначчи в табл. 4.4 на с. 153. Как вы понимаете, подбирать все битовые комбинации для инструкций довольно неудобно. Программисты привыкли все упрощать, потому они и изобрели новый способ написания компьютерных программ — *язык ассемблера*.

С помощью этого языка можно было делать удивительные вещи. Он позволял программистам использовать *мнемоники* для инструкций, вместо того чтобы запоминать все битовые комбинации. В нем появились имена, или *ярлыки*, адресов. А еще он позволял программистам добавлять *комментарии*, чтобы другим людям было проще читать и понимать программу.

Программа под названием *ассемблер* может читать исходный код на языке ассемблера и извлекать из него *машинный код*, заполняя значения ярлыков

(символы) по мере чтения. Это особенно удобно, поскольку таким образом можно предотвратить глупые ошибки, которые неизбежно появляются при перемещении кода.

В листинге 8.1 показано, как может выглядеть программа Фибоначчи (см. табл. 4.4) на гипотетическом ассемблерном языке из главы 4.

Листинг 8.1. Программа на ассемблерном языке для вычисления последовательности Фибоначчи

```

load  #0          ; установить 0 первым числом в последовательности
store first
load  #1          ; установить 1 вторым числом в последовательности
store second
again: load  first   ; сложить первое и второе числа, чтобы получить
           add   second   ; следующее число в последовательности
           store next
                   ; какие-нибудь действия над числом
           load  second   ; установить второе число на место первого
           store first
           load  next     ; установить следующее число на место второго
           store second
           cmp   #200     ; результат получен?
           ble  again    ; нет, повторить
first:  bss   1     ; где хранится первое число
second: bss   1     ; где хранится второе число
next:   bss   1     ; где хранится следующее число

```

Псевдоинструкция `bss` (что означает *блок, начинающийся с символа, block started by symbol*) резервирует участок памяти — в нашем случае один адрес, — не помещая ничего внутри. Псевдоинструкции не соотносятся с инструкциями машинного языка напрямую, а представляют собой инструкции для ассемблера. Видим, что с языком ассемблера работать проще, чем с машинным, — но все еще утомительно.

В далекие времена нужные для работы программы приходилось загружать в компьютер самостоятельно. К моменту появления первого компьютера ассемблер еще *не был* изобретен, поэтому программисты были вынуждены создавать его буквально на ходу, собирая информацию по крупицам. Первый ассемблер был довольно примитивным, но он хотя бы работал, и его можно было улучшить — было с чем работать.

Термин *самозагрузка* со временем прижился, и сейчас его часто сокращают до просто *загрузки*. Загрузка компьютера чаще всего подразумевает запуск маленькой программы, которая запускает программу побольше, а та, в свою очередь, запускает новую, еще большую. В первые компьютеры программу загрузки приходилось вводить вручную, используя переключатели и датчики на передней панели.

ЯЗЫКИ ВЫСОКОГО УРОВНЯ

Ассемблер стал большим подспорьем для программистов. Однако выполнение простых задач все еще требовало значительных усилий. Нужен был способ использовать меньше слов для описания более сложных задач. В книге «*Мифический человеко-месяц, или Как создаются программные системы*» Фредерика Брукса (Fred Brooks) (Addison-Wesley), написанной в 1975 году, утверждается, что в среднем программист может написать от 3 до 10 документированных и отлаженных строк кода в день. Если бы одной строкой можно было описать более сложные задачи, мы делали бы свою работу гораздо быстрее.

Пришло время представить *высокоуровневые языки*, которые оперируют более высокими уровнями абстракции, чем ассемблер. Исходный код в высокоуровневых языках пропускается через программу под названием *компилятор*, которая переводит, или компилирует, его в машинный язык, известный как *объектный код*.

Сегодня насчитываются тысячи высокоуровневых языков, предназначенных для решения как общих, так и специфических задач. Один из первых высокоуровневых языков известен под названием FORTRAN, что расшифровывается как «транслятор формул» (formula translator). Его можно использовать, чтобы легко написать программу, которая вычисляет значения выражений вроде $y = m \times x + b$. В листинге 8.2 показано, как программа вычисления последовательности Фибоначчи будет выглядеть на языке FORTRAN.

Листинг 8.2. Программа для вычисления последовательности Фибоначчи на языке FORTRAN

```
C УСТАНОВИТЬ ДВА ПЕРВЫХ ЧИСЛА В ПОСЛЕДОВАТЕЛЬНОСТИ КАК I И J
  I=0
  J=1
C ПОЛУЧИТЬ СЛЕДУЮЩЕЕ ЧИСЛО В ПОСЛЕДОВАТЕЛЬНОСТИ
5  K=I+J
C ДЕЙСТВИЯ С ЧИСЛОМ
C СДВИНУТЬ ЧИСЛА I И J В ПОСЛЕДОВАТЕЛЬНОСТИ
  I=J
  J=K
C ПОВТОРИТЬ, ЕСЛИ ПОСЛЕДНЕЕ ЧИСЛО МЕНЬШЕ 200
  IF (J .LT. 200) GOTO 5
C ГОТОВО
```

Выглядит проще, чем ассемблер, не правда ли? Обратите внимание, что строчки, начинающиеся с буквы *C*, обозначают комментарии. Здесь используются и метки, но теперь они должны обозначать численные значения. Также обратите внимание, что не требуется явно определять нужное количество памяти — она магическим образом резервируется при объявлении *переменных*, таких как *I* и *J*. Благодаря FORTRAN получилось кое-что интересное (или ужасное, тут уж как посмотреть) — и примерно этим мы и занимаемся по сей день. Любое имя

переменной, начинающиеся с букв I, J, K, L, M или N , обозначало целое число — таким же образом описывались доказательства в математике. Переменные, начинающиеся с любой другой буквы, обозначали числа с плавающей точкой, или вещественные (REAL), как они назывались в языке FORTRAN. Поколения программистов, работавших с FORTRAN, все еще используют i, j, k, l, m и n или их заглавные эквиваленты для обозначения целочисленных переменных.

Громоздкий язык FORTRAN в свое время использовался для таких же громоздких компьютеров. С появлением меньших и не таких дорогих устройств (для них можно было выделить комнату поменьше) пришли и новые языки программирования. Многие из этих языков, например BASIC (сокращение от Beginner's All-purpose Symbolic Instruction Code — универсальный код символьских инструкций для начинающих), представляли собой вариации на тему FORTRAN. У всех этих языков была одна и та же проблема — с увеличением сложности программы становилось все труднее разобраться в запутанной сети номеров строк и инструкций GOTO. Изначально стройная система нумерации меток распадалась при малейшей попытке внести изменения. Многие программисты использовали метки с шагом 10 или 100, чтобы впоследствии промежутки между ними можно было заполнить, но и это не всегда помогало.

Структурное программирование

Языки вроде FORTRAN или BASIC называются *неструктурированными*, потому что они не имеют заданной структуры для организации меток и инструкций GOTO. В реальности нельзя построить дом, просто бросив на землю кучу досок, чего не скажешь о языке FORTRAN. Я имею в виду оригинальную версию FORTRAN — в результате эволюции язык приобрел некоторые черты структурного программирования, и он до сих пор остается одним из самых востребованных языков программирования для научных вычислений.

Структурные языки программирования были разработаны, чтобы избавиться от проблемы *спагетти-кода* — ужасного нагромождения инструкций GOTO. Некоторые из них зашли слишком далеко — например, в языке Pascal вы не встретите ничего похожего на GOTO, из-за чего его можно использовать разве что для обучения элементарному структурному программированию. Если честно, то он и был создан именно для этого. Язык C, преемник созданного Кеном Томпсоном (Ken Thompson) языка B, был разработан Деннисом Ритчи (Dennis Ritchie) в Bell Telephone Laboratories. Сегодня этот pragматичный язык остается одним из самых популярных в мире, и многие из более новых языков программирования — в том числе C++, Java, PHP, Python и JavaScript — используют приемы C.

В листинге 8.3 показан вариант программы Фибоначчи на языке JavaScript. Обратите внимание, что в этом фрагменте кода отсутствует явное ветвление.

Листинг 8.3. Программа на языке JavaScript для вычисления последовательности Фибоначчи

```
var first; // первое число
var second; // второе число
var next; // следующее число в последовательности

first = 0;
second = 1;

while ((next = first + second) < 200) {
    // действия с числом
    first = second;
    second = next;
}
```

Операторы внутри фигурных скобок {} выполняются до тех пор, пока условие while в скобках истинно. Выполнение продолжается после }, когда это условие становится ложным. Поток управления стал чище, что делает программу более понятной.

Лексический анализ

Разберемся, что включает в себя понятие «обработка языка». Начать стоит с *лексического анализа* — перевода *символов* (букв) в *лексемы* (слова).

В упрощенном виде лексический анализ можно представить как разделение языка на два типа лексем: слова и разделители. Например, в соответствии с описанными выше правилами выражение *лекс лютор*¹ (автор всех злодейских языков программирования) состоит из двух лексем-слов (*лекс* и *лютор*) и одной лексемы-разделителя (пробела). На рис. 8.1 показан простой алгоритм разделения вводимых символов по типам лексем.

Просто получить лексемы недостаточно — нужно классифицировать их, поскольку в реальной жизни в языках используется множество типов лексем (имена, числа, операторы). В языках, как и в математике, есть операторы и операнды, а операнды, в свою очередь, могут быть переменными и постоянными (числами). Из-за допущений во многих языках задача еще больше усложняется — например, разделители могут быть не указаны явно, как в примере с A+B (подразумевается A + B — обратите внимание на пробелы). Эти две формы записи эквивалентны, но в первом случае мы не видим явного указания разделителей.

На удивление трудно классифицировать численные константы, даже если не задумываться о разнице между восьмеричными, шестнадцатеричными, целыми числами и числами с плавающей точкой. Изобразим схематично несколько

¹ Лекс Лютор — суперзлодей, заклятый враг Супермена. — Примеч. ред.

правильных вариантов записи числа с плавающей точкой. Оно может быть задано как `1.`, `.1`, `1.2`, `+1.2`, `-.1`, `1e5`, `1e-5` и `1.2E5`, что показано на рис. 8.2.

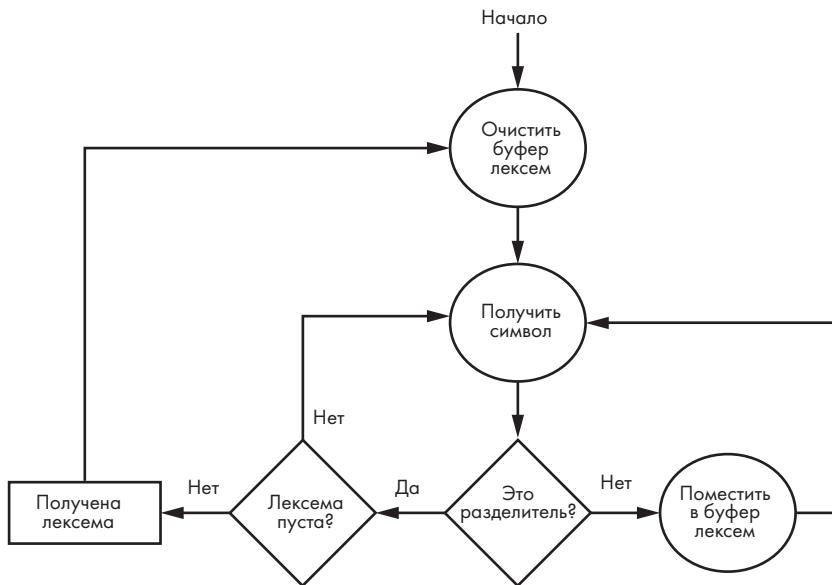


Рис. 8.1. Простой лексический анализ

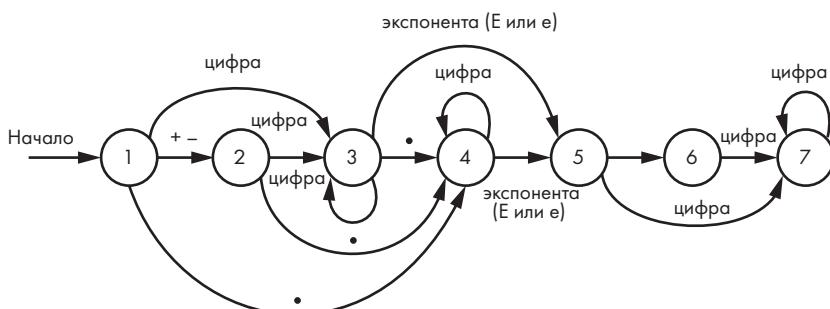


Рис. 8.2. Схематическое изображение числа с плавающей точкой

Начнем с кружка, обозначенного цифрой 1. Знак `+` или `-` приводит нас к кружку под номером 2, а с помощью символа `.` мы перейдем к кружку 4. После кружка 2 мы можем перейти к кружку 4 (если следующий символ — это `.`) или к кружку 3 (если следующий символ — цифра). В кружках под номерами 3 и 4 находятся цифры. Классификация завершится, если для следующего символа не встретится подходящей стрелки перехода. Например, ввод пробела после любого кружка означает конец записи. Однако если после кружка 2 не ввести цифру или

десятичную точку, после кружка 6 — цифру, а после кружка 5 — знак или цифру, это будет ошибка, потому что мы не получим правильный вариант записи числа с плавающей точкой. Для упрощения ошибочные пути не включены в схему.

Все это очень напоминает поиск сокровищ по пиратской карте. Переходя по стрелкам, мы перемещаемся в новые локации. Если вместо допустимого символа ввести, к примеру, букву Я в кружке 1, мы потеряемся и не сможем больше ориентироваться.

Схему на рис. 8.2 можно рассматривать как спецификацию для записи чисел с плавающей точкой — можно даже написать программу, реализующую заданный алгоритм. Есть и другие, более формальные способы написания спецификаций — например, нормальная форма Бэкуса — Наура.

ФОРМА БЭКУСА — НАУРА

Форма Бэкуса — Наура (БНФ) восходит к работам индийского знатока санскрита Панини (Pāṇini) (прим. V в. до н. э.). БНФ получила свое название от фамилии американского программиста Джона Бэкуса (John Backus) (1924–2007) — он также изобрел язык FORTRAN — и датского программиста Петера Наура (Peter Naur) (1928–2016). БНФ представляет собой формальный способ спецификации языков. В этой книге мы не будем углубляться в детали, но о БНФ стоит знать, потому что она используется в документах RFC («запрос на комментарии», *request for comments*), определяющих, помимо прочего, интернет-протоколы. Ниже представлена БНФ для числа с плавающей точкой:

```

<цифра>           ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<цифры>          ::= <цифра> | <цифры> <цифра>
<знак экспоненты> ::= "e" | "E"
<знак>             ::= "+" | "-"
<необязательный знак> ::= <знак> | ""
<экспонента>       ::= <знак экспоненты> <необязательный знак> <цифры>
<необязательная экспонента> ::= <экспонента> | ""

<мантийса>        ::= <цифры> | <цифры> "." | "." <цифры> | <цифры> "." <цифры>
<десятичная точка> ::= <необязательный знак> <мантийса> <необязательная экспонента>

```

Выражения слева от знака «`::=`» можно заменить на выражения справа от него.

Знак «`|`» обозначает выбор между выражениями, а все, что указано в кавычках, обозначает литералы — то есть то, что должно быть записано в буквальном виде.

Конечные автоматы

Учитывая сложность записи чисел, можно представить, что для извлечения языковых лексем из входных данных потребуется огромное множество специального кода. Однако, судя по рис. 8.2, существует более простой подход — создание

конечного автомата, содержащего набор состояний и причин для перехода из одного состояния в другое, — именно это и изображено на рис. 8.2. Если собрать все данные воедино, получим табл. 8.1.

Таблица 8.1. Таблица состояний для чисел с плавающей точкой

Входные данные	Состояние						
	1	2	3	4	5	6	7
0	3	3	3	4	7	7	7
1	3	3	3	4	7	7	7
2	3	3	3	4	7	7	7
3	3	3	3	4	7	7	7
4	3	3	3	4	7	7	7
5	3	3	3	4	7	7	7
6	3	3	3	4	7	7	7
7	3	3	3	4	7	7	7
8	3	3	3	4	7	7	7
9	3	3	3	4	7	7	7
е	Ошибка	Ошибка	5	5	Ошибка	Ошибка	Конец ввода
Е	Ошибка	Ошибка	5	5	Ошибка	Ошибка	Конец ввода
+	2	Ошибка	Конец ввода	Конец ввода	6	Ошибка	Конец ввода
-	2	Ошибка	Конец ввода	Конец ввода	6	Ошибка	Конец ввода
.	4	4	4		Ошибка	Ошибка	Конец ввода
Другое	Ошибка	Ошибка	Конец ввода	Конец ввода	Ошибка	Ошибка	Конец ввода

В соответствии с таблицей из состояния 1 при вводе цифры мы перейдем в состояние 3, ввод е или Е приведет к состоянию 5, ввод знака «+» или «-» — к состоянию 2, ввод знака «.» — к состоянию 4, а ввод любого другого значения — к состоянию ошибки.

Благодаря конечным автоматам можно классифицировать входные данные при помощи простого фрагмента кода, как показано в листинге 8.4. Для этого заменим состояния **ошибка** и **конец ввода** из табл. 8.1 на 0 и -1 соответственно. Для любых значений, не представленных в таблице, достаточно вывести строку **другое**.

Листинг 8.4. Применение конечного автомата

```
state = 1;
while (state > 0)
    state = state_table[state][next_character];
```

Этот же подход можно использовать и для любых других типов лексем. Для этого вместо единственного значения конец ввода вводятся значения всех необходимых типов лексем.

Регулярные выражения

В случае со сложными языками построить без ошибок таблицу наподобие 8.1 из схемы на рис. 8.2 очень непросто, поэтому были придуманы языки для спецификации языков. В 1956 году американский математик Стивен Коул Клини (Stephen Cole Kleene) (1909–1994) представил математическое основание этого подхода. А в 1968 году Кен Томпсон впервые использовал его в компьютерном текстовом редакторе. В 1974-м он же создал команду `grep` для UNIX (сокращение от *globally search a regular expression and print* — глобальный поиск и вывод регулярного выражения). Так появился ныне повсеместно используемый термин *регулярное выражение*. Регулярные выражения представляют собой самостоятельные языки — на сегодняшний день в мире насчитывается несколько языков регулярных выражений, несовместимых между собой. Регулярные выражения легли в основу механизма *поиска по шаблону*. На рис. 8.3 показан пример регулярного выражения, выполняющего поиск по шаблону для числа с плавающей точкой.



Рис. 8.3. Регулярное выражение для числа с плавающей точкой

На первый взгляд, это кажется настоящей тарабарщиной, но на самом деле здесь действует пара простых правил. Вычисление происходит слева направо, то есть выражение `abc` будет соответствовать шаблону строки символов «`abc`». Если

какая-то часть выражения завершается знаком «?», это означает, что данная часть либо отсутствует вообще, либо присутствует только один раз. Знак «*» означает, что часть выражения может повторяться от нуля до нескольких раз, а знак «+» означает один или несколько повторов части выражения. Набор символов в квадратных скобках представляет собой шаблон, проверяющий, встречается ли в выражении один из этих символов. К примеру, шаблону [abc] соответствуют символы a, b или c. Знак «.» означает любой символ, а для обозначения самого знака «.» в выражении используется экранирование с помощью обратной косой черты (\). Знак «|» означает символ либо справа, либо слева от него. Скобки(), как и в математике, используются для группировки частей выражения.

Читаем выражение слева направо — оно начинается с необязательного знака «плюс» или «минус». За ним следуют цифры или десятичная точка, а за ними — одна или несколько цифр (в случае с числами вроде 1.2 или .2). Или же можно увидеть одну или несколько цифр, необязательную десятичную точку и, что также необязательно, одну или несколько цифр после нее (как в случае с числами 1 и 1.). За всем этим может следовать экспонента, обозначаемая знаком E или e, а за ней — необязательный знак «плюс» или «минус» и одна или несколько цифр. Не так уж и страшно, правда?

Язык регулярных выражений мог быть еще более полезным, если бы автоматически генерировал таблицы состояний помимо обработки входных данных и разделения их на лексемы. И, между прочим, он на самом деле умеет это делать благодаря исследованию Bell Telephone Laboratories. В 1975 году американский физик Майк Леск (Mike Lesk) со стажером Эриком Шмидтом (Eric Schmidt), ныне председателем правления Alphabet, родительской компании Google, написали программу под названием `lex` — сокращение от lexical analyzer — лексический анализатор. Как поют Beatles в песне Penny Lane: «It's a Kleene machine»¹. Позже в проекте GNU была разработана версия программы с открытым исходным кодом — `flex`. Оба этих инструмента делают как раз то, что нам нужно, — запускают программу на основе таблицы состояний, которая выполняет предоставленные пользователем фрагменты, если входные данные совпадают с шаблоном регулярного выражения. Например, фрагмент программы `lex` в листинге 8.5 выводит символы `ah`, если во входных данных встречаются строки `ar` или `er`, а символы `er` — если введенное слово заканчивается на `a`.

Листинг 8.5. Фрагмент бостонской программы lex

```
[ae]r      printf("ah");
a/[ .;!]?] printf("er");
```

Косая черта «/» во втором шаблоне означает «проверять совпадения по шаблону слева от черты, если перед этим встретилась часть выражения справа от черты». Если совпадений не найдено, выводится исходный текст. Данную программу

¹ Отсылка к теореме Клини (Kleene's theorem). — Примеч. науч. ред.

можно использовать, чтобы преобразовать текст на стандартном американском английском в текст с бостонским акцентом. К примеру, если ввести фразу *Park the car in Harvard yard and sit on the sofa*, в результате получим *Pahk the cah in Hahvahd yahd and sit on the sofer*.

`lex` в два счета справляется с классификацией лексем. В листинге 8.6 приведен фрагмент программы `lex`, которая определяет все ранее рассмотренные форматы чисел, имена переменных и некоторые операторы. Вместо простого вывода результата программа возвращает заранее определенные значения для каждого типа лексем. Обратите внимание: некоторые символы имеют специальное значение в `lex`, поэтому их необходимо экранировать с помощью обратной косой черты. В этом случае программа распознает их как литералы.

Листинг 8.6. Классификация лексем при помощи `lex`

```
0[0-7]*
[+-]?[0-9]+
[+-]?(([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*))([Ee][+-]?[0-9]+)?
0x[0-9a-fA-F]+
[A-Za-z][A-Za-z0-9]*
\+
-
\*
\/
=
return (INTEGER);
return (INTEGER);
return (FLOAT);
return (INTEGER);
return (VARIABLE);
return (PLUS);
return (MINUS);
return (TIMES);
return (DIVIDE);
return (EQUALS);
```

В листинге не показано, как именно `lex` получает действительные значения лексем. Обнаружив число, мы должны узнать его значение. То же с именем переменной — обнаружив имя переменной, мы должны знать ее значение.

Учтите, что `lex` подходит не для всех языков. Программист Стивен С. Джонсон (Stephen C. Johnson) (мы еще встретим его в этой книге) объяснил, что `lex` может использоваться для разработки довольно сложных лексических анализаторов, но для некоторых языков (например, FORTRAN), не подходящих под известные теоретические стандарты, лексический анализатор придется создавать вручную.

От слов к предложениям

Выше мы рассмотрели, как последовательности символов превращаются в отдельные слова. Однако язык этим не ограничивается — пришло время научиться соединять слова в предложения в соответствии с правилами грамматики.

Создадим простой калькулятор с четырьмя функциями, взяв за основу лексемы из листинга 8.6. Калькулятор должен считать выражения вроде `1 + 2` корректными, а `1 ++ 2` — нет. Это напоминает уже известный нам поиск по шаблону, не правда ли? И тогда велика вероятность, что кто-то из программистов уже задумывался над подобной задачей.

Этот кто-то — Стивен С. Джонсон. Он, что неудивительно, также работал в Bell Labs. Стивен создал программу *yacc* (сокращение от yet another compiler compiler, «еще один компилятор компилятора») в начале 1970-х. Название недвусмысленно намекает на то, что тогда над похожими задачами уже работало немало программистов. *yacc* до сих пор используется — версия с открытым исходным кодом под названием *bison* доступна в рамках проекта GNU. *yacc* и *bison*, подобно *lex*, генерируют таблицы состояний и код для работы с ними.

Программа *yacc* генерирует *восходящий* синтаксический анализатор (или анализатор типа «сдвиг-свертка») при помощи стека (см. раздел «Стеки»). В данном контексте *сдвиг* означает перемещение лексемы по стеку, а *свертка* — замену найденного по шаблону набора лексем из стека соответствующей лексемой. В листинге 8.7 представлен пример БНФ для калькулятора — он использует значения лексем, полученные в результате работы программы *lex* из листинга 8.6.

Листинг 8.7. Простая БНФ для калькулятора

```

<operator>   ::= PLUS | MINUS | TIMES | DIVIDE
<operand>    ::= INTEGER | FLOAT | VARIABLE
<expression> ::= <operand> | <expression> PLUS <operand>
                  | <expression> MINUS <operand>
                  | <expression> TIMES <operand>
                  | <expression> DIVIDE <operand>
<assignment> ::= <variable> EQUALS <expression>
<statement>  ::= <expression> | <assignment>
<statements> ::= "" | <statements> <statement>
<calculator> ::= <statements>

```

Гораздо проще понять алгоритм «сдвиг-свертка», увидев его в действии. На рис. 8.4 показано, что произойдет, если ввести в калькулятор выражение $4 + 5 - 3$. Из врезки «Различные представления уравнений» на с. 169 вы можете помнить, что для обработки инфиксного представления требуется большая глубина стека, чем для постфиксного. Дело в том, что в случае с инфиксным представлением нужно сдвинуть больше лексем (например, скобок) перед применением свертки.

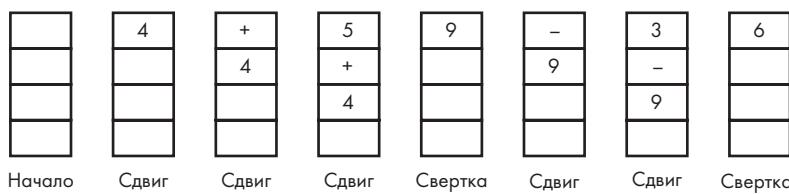


Рис. 8.4. Анализатор типа «сдвиг-свертка» в действии

В листинге 8.8 показан код калькулятора, реализованного с помощью программы *yacc*. Обратите внимание на его сходство с БНФ. Это учебный

пример — полностью рабочий алгоритм включал бы слишком много уточняющих деталей.

Листинг 8.8. Часть yacc-кода для простого калькулятора

```
calculator : statements
;
statements : /* empty */
| statement statements
;
operand : INTEGER
| FLOAT
| VARIABLE
;
expression : expression PLUS operand
| expression MINUS operand
| expression TIMES operand
| expression DIVIDE operand
| operand
;
assignment : VARIABLE EQUALS expression
;
statement : expression
| assignment
;
```

Клуб «Язык дня»

Раньше языки были сложными. В 1977 году два программиста, канадец Альфред Ахо (Alfred Aho) и американец Джейфри Ульман (Jeffrey Ullman) из Bell Labs, опубликовали книгу «Компиляторы. Принципы, технологии и инструментарий». Это была одна из первых книг, выпущенных с использованием системы компьютерной верстки и типографского языка `troff`. Если вкратце, то основной посыл книги можно выразить так: «Языки настолько сложны, что вам придется зарыться в матчасть, выучить теорию и т. д.». Второе издание (1986), созданное совместно с индийским программистом Рави Сети (Ravi Sethi), транслировало совершенно другую идею. Ее можно описать как «мы разобрались в языках и научим этому вас». И это была сущая правда.

Новое издание открыло миру программы `lex` и `yacc`. Оказалось, что уже существует множество языков для самых разных задач — и не только языков программирования. Мне особенно полюбился небольшой язык `chem` (автор — канадский программист Брайан Керниган (Brian Kernighan) из Bell Labs), который умеет выводить химические структурные формулы на основе входных данных вроде

С double bond 0. Схемы в данной книге, между прочим, были созданы с помощью изобразительного языка `pic`, также придуманного Брайаном Керниганом.

Создавать новые языки весело. Как итог, люди начали выпускать новые языки программирования, не имея четкого представления об их истории, что привело к повторению прежних ошибок. Например, способ обработки *пробелов* (промежутков между словами) в языке Ruby вносит ошибку, давно исправленную в ранних версиях языка С. (Помните, что один из классических подходов к ошибкам — объявить, что так и было задумано.)

В результате сегодня доступно огромное множество языков программирования. Некоторые из них не содержат ничего принципиально нового и служат лишь воплощением вкуса их создателей. Стоит обратить внимание разве что на *предметно-ориентированные* языки программирования, в том числе *небольшие языки* вроде `pic` и `chem`, предназначенные для решения локальных задач. Американский программист Джон Бентли (Jon Bentley) выпустил прекрасную колонку под названием *Programming Pearls* о небольших языках в журнале *Communications of the ACM* еще в 1986 году. В 1999 году эти статьи были собраны и изданы под обложкой одноименной книги (Addison-Wesley).

Деревья синтаксического анализа

Ранее я уже упоминал о компиляции языков высокого уровня. Теперь вы узнаете, что высокоуровневые языки можно не только компилировать, но и *интерпретировать*. Выбор действия зависит, скорее, не от проектирования языка, а от его реализации.

Языки компилируются в машинный код, как показано в табл. 4.4 на с. 153. Компилятор принимает исходный код и переводит его на машинный язык для определенного устройства. Большинство компиляторов могут компилировать одну и ту же программу для разных устройств. После компиляции программа готова к запуску.

В свою очередь, интерпретировав язык, вы не получите машинный код для реального устройства (под «реальными устройствами» я имею в виду аппаратное обеспечение). Интерпретированные языки подходят для запуска на *виртуальных машинах* — устройствах, созданных на основе ПО. Они могут иметь собственный машинный язык, но его нельзя назвать набором компьютерных инструкций, реализованных в аппаратном виде. Обратите внимание, что термин *виртуальная машина* в последнее время сильно перегружен — в данном контексте я говорю об абстрактном вычислительном устройстве. Некоторые интерпретированные языки могут быть запущены *интерпретаторами* напрямую, а другие компилируются в так называемые *промежуточные языки* для дальнейшей интерпретации.

В целом скомпилированный код выполняется быстрее, потому что после компиляции он представляет собой машинный язык. Можно сравнить это с переводом книги — после перевода на определенный язык она становится доступной любому, кто его понимает. Интерпретированный язык — это нечто эфемерное. Представьте, что вы читаете книгу вслух, одновременно переводя ее на язык слушателя. Чтобы прочитать эту книгу кому-то другому на другом языке, придется заново ее переводить. Несмотря на сложности, интерпретированные языки позволяют выполнять задачи, которые трудно решить с помощью аппаратного обеспечения. Компьютеры работают достаточно быстро, поэтому во многих случаях можно смириться с неизбежными задержками при использовании интерпретаторов.

На рис. 8.4 изображен калькулятор, работающий напрямую с входными данными. Для калькулятора этого может быть достаточно, но в данном примере мы опускаем шаг, важный для работы компилятора или интерпретатора. Я говорю о создании *дерева синтаксического анализа* с использованием грамматики калькулятора — НАГ-структуры данных (НАГ — направленный ациклический граф). Подобное дерево строится из узлов, как показано на рис. 8.5.



Рис. 8.5. Схема узла дерева синтаксического анализа

Каждый узел включает в себя код, по которому определяется тип данного узла. Существует также массив листьев; интерпретация каждого листа определяется его кодом. Листья представляют собой объединения — они могут включать в себя более одного типа данных. Для именования членов мы используем синтаксис языка C — например, `.i` указывает на интерпретацию листа как целого числа (integer).

Предположим, что существует функция `makenode`, задача которой — создавать новые узлы. Она принимает количество листьев (`leaf`) в качестве первого аргумента и код (`code`) — в качестве второго, а также последовательно переданные значения всех листьев.

Добавим немного кода в листинг 8.8, все еще опуская некоторые незначительные подробности. Также для простоты ограничимся обработкой целых чисел. Новые строки кода отвечают за работу программы при проверке грамматических правил. В `уасс` значения всех элементов с правой стороны обозначаются как `$1`, `$2` и т. д., а `$$` представляет собой результат применения правила. В листинге 8.9 видим расширенную версию `уасс`-калькулятора.

Листинг 8.9. Создание дерева синтаксического анализа для простого калькулятора на основе yacc

```

calculator : statements           { do_something_with($1); }
;
statements : /* empty */
            | statement statements { $$ .n = makenode(2, LIST, $1, $2); }
;
operand   : INTEGER              { $$ = makenode(1, INTEGER, $1); }
            | VARIABLE             { $$ = makenode(1, VARIABLE, $1); }
;
expression: expression PLUS operand { $$ .n = makenode(2, PLUS, $1, $3); }
            | expression MINUS operand { $$ .n = makenode(2, MINUS, $1, $3); }
            | expression TIMES operand { $$ .n = makenode(2, TIMES, $1, $3); }
            | expression DIVIDE operand { $$ .n = makenode(2, DIVIDE, $1, $3); }
            | operand                { $$ = $1; }
;
assignment: VARIABLE EQUALS expression { $$ .n = makenode(2, EQUALS, $1, $3); }
;
statement  : expression           { $$ = $1; }
            | assignment            { $$ = $1; }
;

```

В данном примере все простые правила возвращают соответствующие им значения. Более сложные правила вроде `statements`, `expression` и `assignment` создают узел, добавляют к нему дочерние элементы и возвращают результат. На рис. 8.6 представлен пример обработки некоторых входных данных.

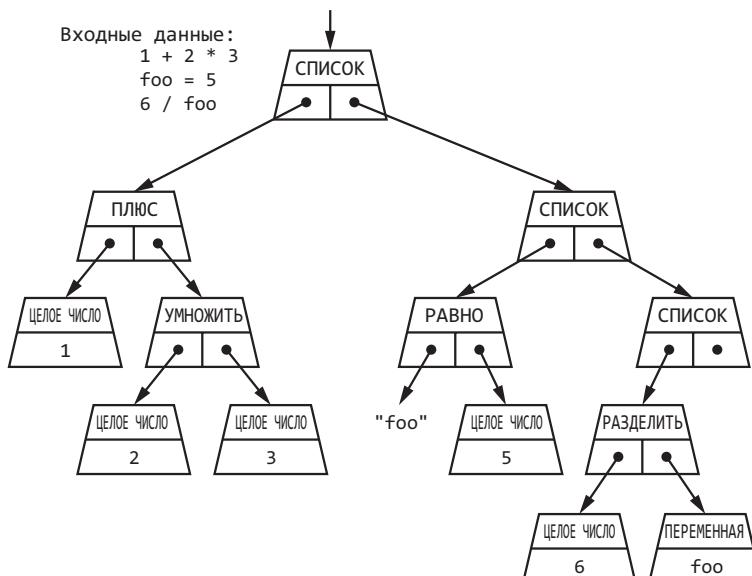


Рис. 8.6. Дерево синтаксического анализа для простого калькулятора

Как видите, код генерирует дерево. В самом его верху используется правило `calculator` (калькулятор) для создания связанного списка выражений из узлов дерева. Все оставшееся дерево состоит из узлов `statement` (выражение), которые содержат `operator` и `operands` (оператор и операнды).

Интерпретаторы

В листинге 8.9 вы могли заметить загадочный вызов функции `do_something_with`, куда передается корневой узел дерева синтаксического анализа. Данная функция заставляет интерпретатор «запустить» дерево, начиная с обхода связанного списка, как показано на рис. 8.7.

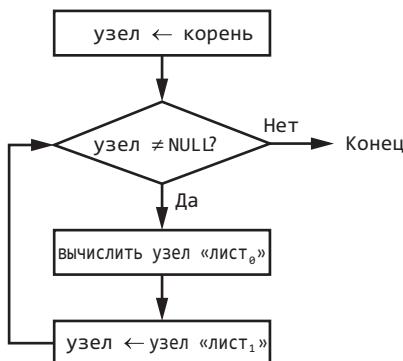


Рис. 8.7. Обход связанного списка дерева синтаксического анализа

Далее мы переходим к вычислению значений в дереве при помощи рекурсивного обхода в глубину. Эта функция изображена на рис. 8.8.

Благодаря тому что каждый узел имеет свой код, легко понять, как действовать. Однако нам потребуется дополнительная функция для хранения имени переменной (символа) и ее значения в *таблице символов* и еще одна функция для поиска значения, связанного с именем переменной. Все это можно реализовать с помощью хеш-таблиц.

Добавив код обхода списка и вычисления в `uass`, мы сможем тут же запустить дерево синтаксического анализа. Как вариант, дерево можно сохранить в файл, чтобы прочитать и запустить его позже. Именно так работают некоторые языки программирования, такие как Java и Python. Во всех отношениях данный код представляет собой набор инструкций на машинном языке, только предназначенный для устройств на основе ПО, а не для аппаратного обеспечения. На каждой используемой машине должна быть предустановлена программа, которая может запустить сохраненное дерево синтаксического анализа. Часто один и тот же исходный код, полученный из интерпретатора, по-разному компилируется и используется для разных задач.

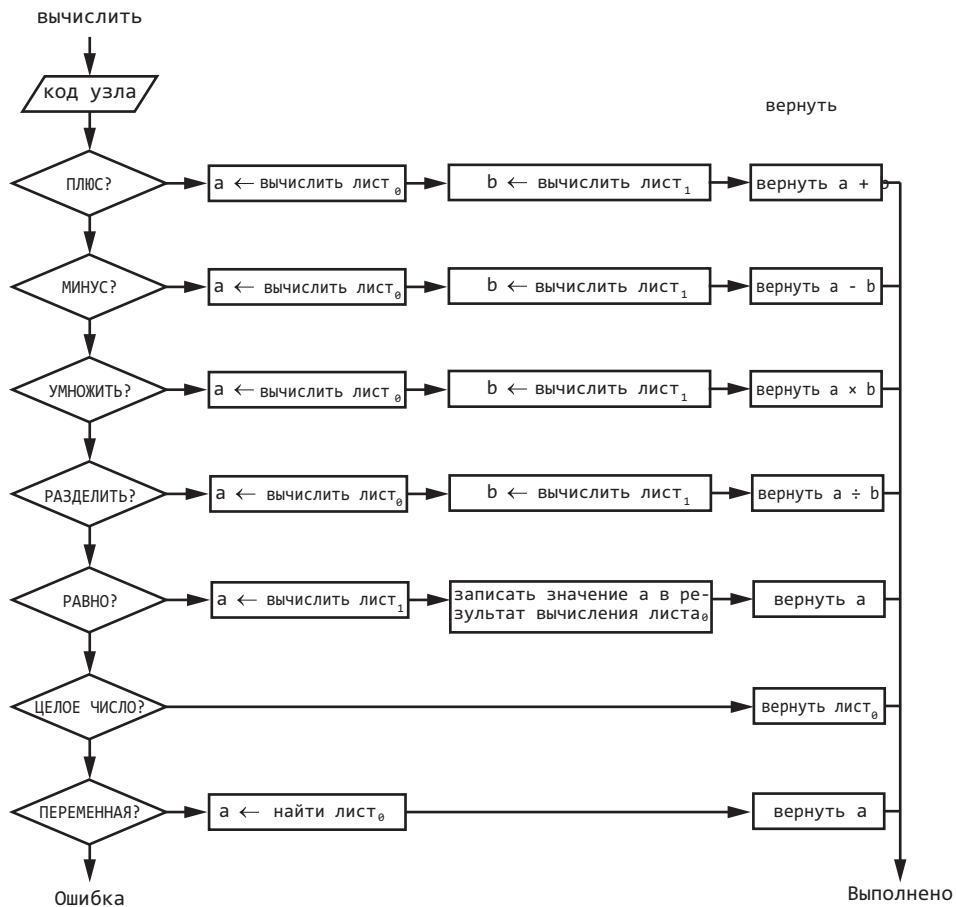


Рис. 8.8. Вычисление значений в дереве синтаксического анализа

На рис. 8.9 представлена структура интерпретатора.

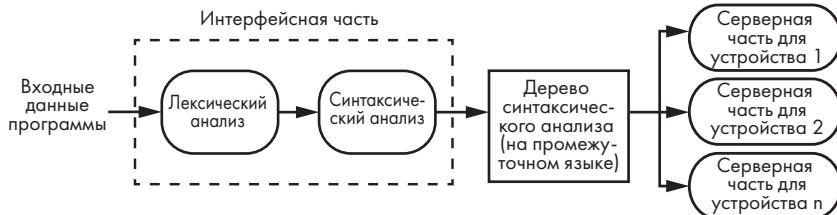


Рис. 8.9. Структура интерпретатора

Интерфейсная часть интерпретатора генерирует дерево синтаксического анализа, представленное с помощью *промежуточного языка*, в то время как для каждого отдельного устройства или среды выполнения может существовать отдельная серверная часть.

Компиляторы

Компиляторы в целом похожи на интерпретаторы, но вместо кода выполнения в серверной части они используют генераторы кода, как показано на рис. 8.10.

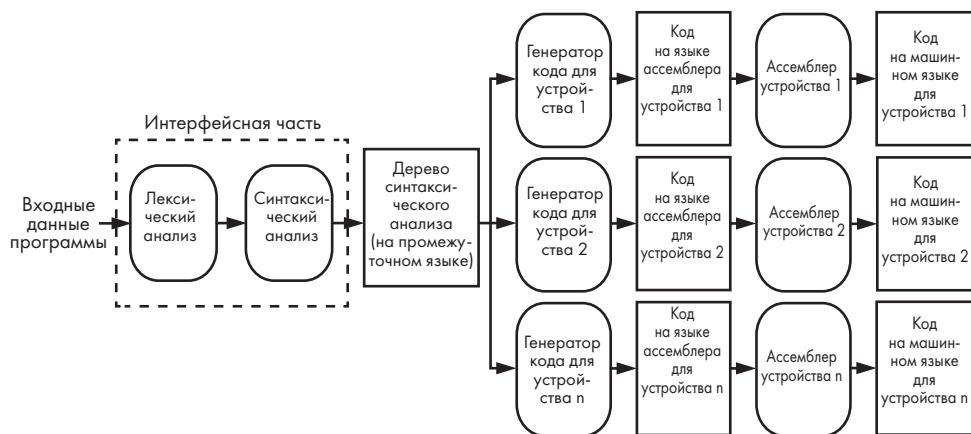


Рис. 8.10. Структура компилятора

Генератор кода создает код на машинном языке для определенного устройства. Инструменты, представленные в некоторых языках (например, в C), могут создавать код на ассемблере (см. «Язык ассемблера») для заданного устройства. Затем код на ассемблере пропускается через ассемблер устройства для получения кода на машинном языке.

Генератор кода работает в точности как рассмотренная ранее комбинация обхода дерева синтаксического анализа и вычисления его значений (см. рис. 8.7 и 8.8). Разница в том, что прямоугольники для вычисления значений дерева на рис. 8.8 заменяются прямоугольниками с генерацией кода на языке ассемблера. На рис. 8.11 изображен упрощенный вариант генератора кода. Выделенные полужирным выражения (например, **add tmp**) — это инструкции на машинном языке для учебного компьютера из главы 4. Обратите внимание, что в этом компьютере нет инструкций для умножения и деления, но для примера достаточно представить, что они есть.

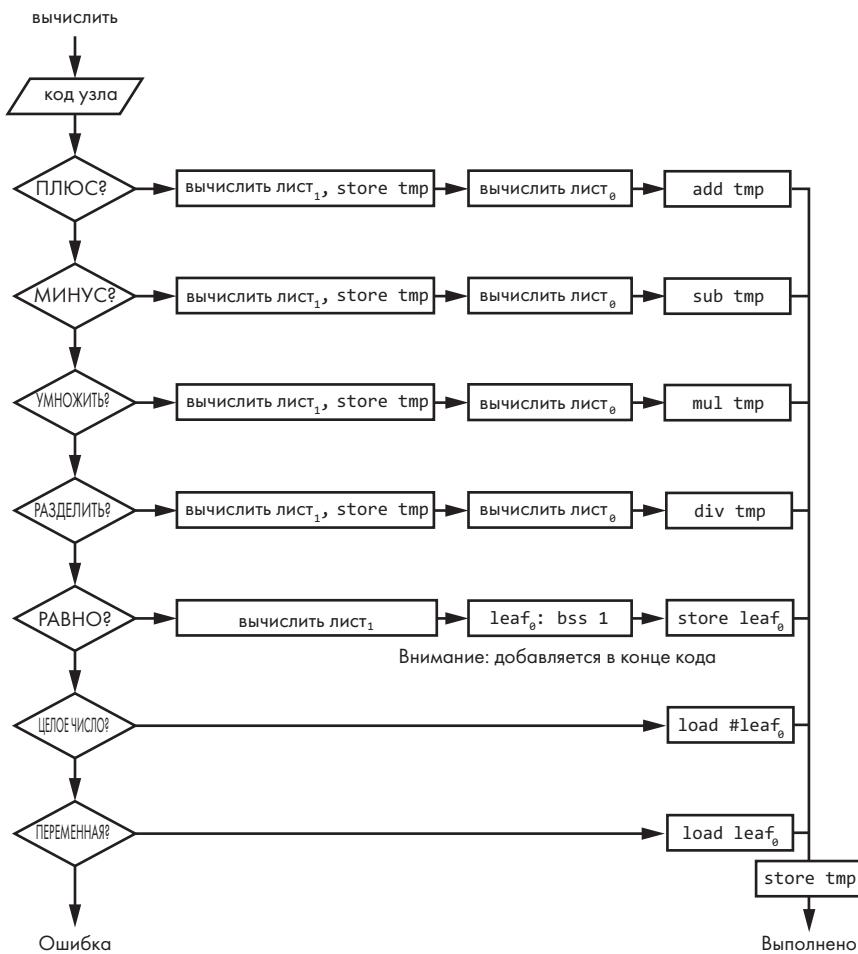


Рис. 8.11. Создание ассемблера из дерева синтаксического анализа

Применив схему с рис. 8.11 к дереву синтаксического анализа на рис. 8.6, получим программу на языке ассемблера, представленную в листинге 8.10.

Листинг 8.10. Вывод на машинном языке, представленный генератором кода

```

load    #3          ; первый элемент списка
store   tmp          ; взять целое число 3
load    #2          ; записать в хранилище
mul    tmp          ; взять целое число 2
store   tmp          ; перемножить значения узлов поддерева
store   #1          ; записать в хранилище
load    tmp          ; взять целое число 1
add    tmp          ; сложить его с результатом умножения 2 и 3

```

```

store  tmp          ; записать в хранилище
        ; второй элемент списка
load   #5           ; взять целое число 5
store  foo          ; сохранить в хранилище для переменной foo
store  tmp          ; записать в хранилище
        ; третий элемент списка
load   foo          ; вызвать содержимое переменной foo
store  tmp          ; записать в хранилище
load   #6           ; взять целое число 6
div    tmp          ; разделить
store  tmp          ; записать в хранилище
tmp:   bss  1        ; ячейка хранилища для временных переменных
foo:   bss  1        ; ячейка хранилища для переменной foo

```

Как видите, в результате получился не лучший код — в нем слишком много загрузок и хранений. Не стоит ожидать многоного от простого вымышленного примера, но и этот код можно улучшить при помощи оптимизаций, рассматриваемых в следующем разделе.

Этот машинный код можно запустить на устройстве, и он выполнится гораздо быстрее, чем его ассемблерный вариант, благодаря небольшому размеру и эффективности.

Оптимизация

Во многих инструментах для работы с языками добавлен дополнительный шаг обработки — *оптимизатор* — между деревом синтаксического анализа и генератором кода. С помощью оптимизатора дерево синтаксического анализа можно преобразовать в более эффективный вариант. Например, оптимизатор может заметить, что все операнды с левой стороны в дереве синтаксического анализа на рис. 8.12 являются константами. В этом случае можно провести вычисления во время компиляции, чтобы позже не тратить драгоценное время выполнения.

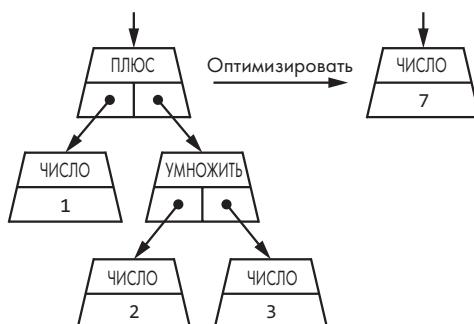


Рис. 8.12. Оптимизация дерева синтаксического анализа

Предыдущий пример не очень информативен, потому что в учебном калькуляторе не предусмотрено условное ветвление. На самом деле, оптимизаторы умеют очень многое. Рассмотрим код из листинга 8.11 (пример на языке C).

Листинг 8.11. Присваивание внутри цикла на языке C

```
for (i = 0; i < 10; i++) {
    x = a + b;
    result[i] = 4 * i + x * x;
}
```

В листинге 8.12 показано, как улучшить данный пример при помощи оптимизатора.

Листинг 8.12. Инвариантная оптимизация цикла

```
x = a + b;
optimizer_created_temporary_variable = x * x;
for (i = 0; i < 10; i++) {
    result[i] = 4 * i + optimizer_created_temporary_variable;
}
```

Код из листинга 8.12 делает то же самое, что и в листинге 8.11, но эффективнее. Оптимизатор определил, что выражение $a + b$ представляет собой *инвариант цикла* (то есть его значение внутри цикла не меняется). Поэтому оптимизатор вынес его из тела цикла, тем самым избавившись от десятикратного повтора вычисления. Кроме этого, выяснилось, что результат вычисления выражения $x * x$ не меняется, и оптимизатор также вынес его из тела цикла.

В листинге 8.13 представлена еще одна возможность под названием *снижение стоимости* — процесс замены дорогих операций более дешевыми, в нашем случае замена умножения сложением.

Листинг 8.13. Пример цикла на языке C со снижением стоимости и инвариантной оптимизацией

```
x = a + b;
optimizer_created_temporary_variable = x * x;
optimizer_created_4_times_i = 0;
for (i = 0; i < 10; i++) {
    result[i] = optimizer_created_4_times_i + optimizer_created_temporary_variable;
    optimizer_created_4_times_i = optimizer_created_4_times_i + 4;
}
```

Снижая стоимость операции, можно воспользоваться преимуществами относительной адресации, чтобы более эффективно вычислять `result[i]`. `result[i]`, согласно рис. 7.2, — это адрес переменной `result` плюс `i`, умноженное на размер элемента массива. Как и в случае с `optimizer_created_4_times_i`, можно начать с адреса `result` и прибавлять к нему размер элемента массива на каждой итерации цикла, вместо того чтобы применять более медленное умножение.

Осторожнее с аппаратной частью!

Оптимизаторы прекрасны, но они могут вызвать неожиданные проблемы с кодом, управляющим аппаратным обеспечением. На рис. 8.14 показана переменная, обозначающая аппаратный регистр, который включает лампочку при задании бита равным 0, как мы видели на рис. 6.1.

Листинг 8.14. Пример кода, который не стоит оптимизировать

```
void
lights_on()
{
    PORTB = 0x01;
    return;
}
```

Выглядит отлично, но что с этим сделает оптимизатор? Он скажет: «Хм, переменная была записана, но ни разу не прочитана, поэтому избавимся от нее». Похожий пример представлен в коде из листинга 8.15, который включает лампочку и проверяет, включена она или нет. Оптимизатор может переписать функцию так, чтобы она просто возвращала 0x01 без записи значения в переменную PORTB.

Листинг 8.15. Еще один пример кода, который не стоит оптимизировать

```
unsigned int
lights_on()
{
    PORTB = 0x01;
    return (PORTB);
}
```

Эти примеры доказывают, что иногда полезно иметь возможность отключить оптимизацию. Обычно с этой целью ПО делят на программы общего назначения и программы для аппаратного обеспечения и применяют оптимизацию только в ПО общего назначения. Кроме того, в некоторых языках программирования существуют механизмы, блокирующие оптимизацию для заданных фрагментов кода. Например, в языке С ключевое слово `volatile` означает, что обозначенная им переменная недоступна для оптимизации.

Выводы

Дочитав до этого места, вы получили представление о том, как работают компьютеры и как они запускают программы. В этой главе вы увидели, как преобразовать программный код для запуска на устройствах, и узнали, как компилировать и интерпретировать программы.

В следующей главе вы встретитесь с боссом интерпретаторов — *веб-браузером* — и языками, которые он интерпретирует.

9

Веб-браузер



Вы, скорее всего, и не задумывались, что веб-браузер, которым вы пользуетесь каждый день, представляет собой *виртуальную машину* — абстрактный компьютер с весьма сложным набором инструкций, созданный исключительно на основе программного обеспечения. Другими словами, он также является примером рассмотренных в предыдущей главе интерпретаторов.

В этой главе вы узнаете, что может делать виртуальная машина. Мы изучим новый язык ввода и поймем, как он интерпретируется браузером. Важно понимать, что браузер — существо чрезвычайно сложное, поэтому мне не удастся описать здесь абсолютно все его возможности.

Браузеры интересно изучать по разным причинам. Они представляют собой, с одной стороны, большие и сложные приложения, а с другой — компьютеры, созданные на основе программного обеспечения, которые можно запрограммировать самостоятельно. В браузерах есть *консоль разработчика*, которую можно использовать для запуска примеров из данной главы. Это поможет в режиме реального времени разобраться, как работает браузер.

Понимание работы веб-браузеров — это первый шаг к изучению проектирования систем, понятия не менее важного, чем само программирование (мы рассмотрим проектирование систем более подробно в главе 15). Повышение интереса к веб-разработке превратило браузер в магнит для новых возможностей. Некоторые новые функции добавились к уже существующим наборам инструкций, другие же стали дублировать имеющуюся функциональность, что отрицательно сказалось на совместимости. В результате мы получили поддержку сразу нескольких наборов инструкций в браузерах. В этой главе я постарался показать, что не вижу

особой ценности в новых возможностях браузеров, которые мало отличаются от уже имеющихся.

Прежде всего наличие множества способов выполнять одни и те же операции означает, что программистам приходится тратить много времени на то, чтобы все это выучить. Кроме того, приходится расходовать силы на то, чтобы выбрать один из существующих подходов к разработке, что также влияет на сложность программ. Статистика в компьютерной индустрии говорит о прямой зависимости между количеством строк кода и числом ошибок. Браузеры часто ломаются. Как мы рассмотрим подробнее в главе 13, сложность исходного кода увеличивает вероятность появления проблем в области безопасности.

Несовместимые способы разработки приводят к ошибкам в программировании. Представьте, что американец сел за руль в Новой Зеландии: элементы управления автомобилем расположены непривычно для него, потому что в Новой Зеландии левостороннее движение. Иностранных, привыкших к правостороннему движению, легко узнать по тому, что только они включают «дворники» при повороте. Хотелось бы избежать подобного в программировании.

По моему мнению, сам факт, что многие веб-стандарты превратились в *живые документы*, сигнализирует о наличии проблем. Если этот термин вам незнаком, то знайте, что я говорю об онлайн-документации, которая постоянно обновляется. Стандарты существуют именно для того, чтобы поддерживать стабильность и согласованность; живые документы же актуальны только в определенный момент времени. Писать код в условиях постоянно меняющихся спецификаций сложно. В этом контексте живые документы удобны только для некоторых своих создателей (потому что документация и связанное с ней ПО могут вообще никогда не дойти до завершающей стадии) и неудобны для большинства потребителей.

Языки разметки

Если бы книга, которую вы держите в руках, была написана на 15 лет раньше, эта глава началась бы с введения в HTML (HyperText Markup Language — язык гипертекстовой разметки). Но я уже намекнул, что браузер очень похож на Большое тихоокеанское мусорное пятно — он постепенно разрастается, потому что к нему прилипает все больше всякой всячины. В этом контексте невозможно не сказать о языках разметки — прежде всего о том, что это и для чего они нужны.

Разметка представляет собой систему аннотирования или добавления меток к тексту — таким образом, чтобы их можно было отличить от самого текста. Метки похожи на язвительные комментарии красными чернилами, оставленные учителем в тетради ученика.

Языки разметки появились задолго до первых компьютеров. Они применялись еще во времена печатных станков — с их помощью авторы и редакторы оставляли

пометки для наборщиков. Эта же полезная задумка пригодилась, когда появилась возможность автоматизировать набор текста с помощью компьютеров. Таким образом, сегодняшние языки разметки — лишь реинкарнация старой идеи.

Языков разметки огромное множество. Например, первый вариант этой книги был написан при помощи языка для набора текстов под названием **troff**. В листинге 9.1 представлен исходный код этого абзаца.

Листинг 9.1. Предыдущий абзац на языке **troff**

```
.PP
```

Языков разметки огромное множество.

Например, первый вариант этой книги был написан при помощи языка для набора текстов под названием **\fCtroff\fP**.

В листинге 9-1 представлен исходный код этого абзаца.

Как видите, практически весь пример кода выше представляет собой обычный текст, за исключением трех элементов разметки. Элемент **\fC** указывает языку **troff**, что текущий шрифт нужно поместить в стек и заменить его шрифтом **C** (Courier). Элемент **\fP** означает, что нужно извлечь шрифт из стека (см. раздел «Стеки» на с. 166) и тем самым вернуться к использованию предыдущего шрифта.

Веб-страницы представляют собой обычные текстовые файлы, подобные примеру с языком **troff**. Для их создания не нужны особенные программы — достаточно текстового редактора. В действительности модные программы для создания веб-страниц только усложняют процессы, поэтому, чтобы упростить себе жизнь, лучше работать с текстовыми редакторами.

Как разметить текст, имея под рукой лишь обычные текстовые символы? Необходимо наделить некоторые символы сверхспособностями — подобно сверхспособностям Супермена, который в обычной жизни работает простым журналистом. Например, в языке **troff** все символы, перед которыми стоят знаки **«.**», **«'»** или **«\»**, имеют сверхспособности.

Компания IBM выпустила собственный язык разметки под названием **GML** (сокращение от *Generalized Markup Language* — обобщенный язык разметки, хотя на самом деле он назван по первым буквам фамилий создателей — Goldfarb (Голдфарб), Mosher (Мошер) и Lorie (Лори)). Этот язык используется для корпоративного издательского инструмента **ISIL**. Позже эта разработка была расширена до языка *Standard Generalized Markup Language (SGML)* (стандартный обобщенный язык разметки), который был принят Международной организацией по стандартизации в 1980-х. SGML был «обобщен» настолько, что неизвестно, смог ли вообще кто-то создать полную рабочую версию стандарта.

Язык *eXtensible Markup Language (XML)* (расширяемый язык разметки) представляет собой практическую разновидность SGML. Позже именно он стал поддерживаться веб-браузерами.

И HTML, и XML были созданы на основе SGML. Они используют часть синтаксиса SGML, но не соответствуют стандарту в полной мере.

XHTML — это подправленная версия HTML, работающая в соответствии с правилами XML.

Унифицированные указатели ресурсов

Первый веб-браузер под названием «Всемирная паутина» (WWW, WorldWideWeb), изобретенный британским инженером и программистом сэром Тимом Бернерсом-Ли в 1990 году, работал достаточно просто, что можно понять по рис. 9.1. Для получения документа с сервера при помощи протокола HTTP в браузере использовались *унифицированные указатели ресурсов* (Uniform Resource Locator (URL)), что мы уже обсуждали в разделе «Всемирная паутина» на с. 207. Сервер посыпал документ браузеру, а тот отображал его на экране. Ранее для создания веб-документов использовался только язык HTML, теперь же для этой цели создано множество самых разных языков.

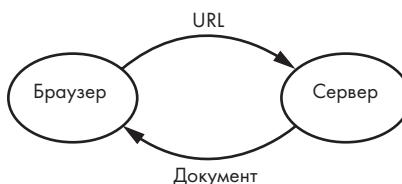


Рис. 9.1. Взаимодействие веб-браузера с веб-сервером

URL — это текстовые строки, имеющие определенную структуру. Сейчас нам достаточно рассмотреть три основных элемента URL, изображенных на рис. 9.2.



Рис. 9.2. Анатомия URL

Протокол определяет механизм коммуникации — например, протокол `https` означает «протокол защищенной передачи гипертекста» (сокращение от HyperText Transfer Protocol (Secure)). *Хост* — это сервер, от которого мы получаем документ. Хост можно обозначить с помощью цифрового интернет-адреса (см. раздел «IP-адреса» на с. 206), но чаще всего для этого используются текстовые доменные имена (см. раздел «Система доменных имен» на с. 206). *Путь* обозначает местоположение нужного нам документа, подобно пути к документу в файловой системе.

В качестве протокола можно указать `file` — в этом случае вместо имени хоста и пути указывается локальное имя файла в системе, где запущен браузер. Другими словами, протокол `file` указывает на файл, расположенный на компьютере.

Число доступных протоколов постоянно растет — например, сегодня можно встретить протоколы `bitcoin` для криптовалют и `tv` для телевизионных трансляций. В целом они подобны, а иногда и идентичны протоколам, рассмотренным ранее в разделе «Всемирная паутина».

HTML-документы

Как я уже говорил, первые веб-страницы представляли собой документы, написанные на языке разметки HTML. В HTML использовался *гипертекст* — текст, в котором содержатся дополнительные ссылки, например на другие веб-страницы. Любители фантастики могут сравнить гипертекст с гиперпространством: вы щелкаете ссылку и — *взух!* — переноситесь в другое место. Гипертекст появился гораздо раньше, чем веб-страницы, но именно в веб-окружении его наконец-то оценили по достоинству.

Рассмотрим простой HTML-документ в листинге 9.2.

Листинг 9.2. Моя первая веб-страница

```
<html>
  <head>
    <title>
      My First Web Page
    </title>
  </head>
  <body>
    This is my first web page.
    <b>
      <big>
        Cool!
      </big>
    </b>
  </body>
</html>
```

Сохраните HTML-код из листинга 9.2 в файл на компьютере и откройте его в браузере. В результате должно получиться как на рис. 9.3.

Видим, что результат на рис. 9.3 не совсем совпадает с текстом из листинга 9.2. Все из-за знака «меньше» (`<`), который обладает сверхспособностями. В нашем случае он означает начало элемента разметки. Элементы представлены парами — для каждого открывающего `<тега>` есть парный закрывающий `</тег>`.



Рис. 9.3. Моя первая веб-страница в браузере

Теги сообщают браузеру, как интерпретировать тот или иной элемент разметки, — они, по сути, являются инструкциями для виртуальной машины. Например, тег `<title>` означает, что его содержимое (текст между открывающим и закрывающим тегами) будет помещено в заголовок веб-страницы. Элементы `` и `<big>` выделяют полужирным слово «Cool» из примера и увеличивают размер шрифта для него. Кроме того, они входят в тег `<body>` на веб-странице.

Как вы уже знаете, знак «<>» обладает сверхспособностями. А что, если нужно вывести его на странице как обычный текст, например в предложении `This is my first web page with a <?` Для этого в HTML существует свой криптонит под названием *ссылка на сущность* — альтернативная форма символа. В нашем случае, чтобы знак «<>» потерял свою сверхспособность, его нужно заменить на последовательность символов `<`. (Здесь мы встречаем еще один символ со сверхспособностями — «&» — его можно заменить на сочетание `&`.) При помощи ссылки на сущность текст примера можно исправить на `This is my first web page with a <`, и тогда он корректно отобразится в браузере.

HTML-элементы не так просты, как кажется на первый взгляд. Существуют примеры тегов, для которых не нужна закрывающая пара, или элементы в формате `<tag/>`, не имеющие содержимого. В XHTML таких случайных исключений практически нет, не считая *атрибутов* — необязательных для использования наборов пар «имя/значение» для тегов, как показано в листинге 9.3.

Листинг 9.3. HTML-элементы с атрибутами

```
<tag name1="value1" name2="value2" ...>  
    содержимое элемента  
</tag>
```

Поведение некоторых имен атрибутов задано по умолчанию; для атрибутов, которые не предопределены, можно добавить произвольные значения. Все значения атрибутов обрабатываются в HTML одинаково, за исключением атрибута `class` — его значение представляет собой список отдельных значений, разделенных пробелами.

Объектная модель документа

Веб-браузеры работают с документами в соответствии с *объектной моделью документа* (Document Object Model, *DOM*). Веб-страницу можно представить как совокупность элементов, включающих в себя другие элементы (это хорошо проиллюстрировано с помощью отступов в листинге 9.2). На рис. 9.4 представлено что-то вроде скрученной *матрёшки* в разрезе (вид сверху) — именно так выглядит структура кода из примера в листинге 9.2.

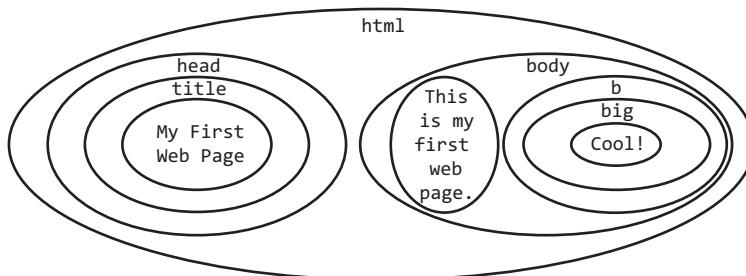


Рис. 9.4. Вложенные элементы в HTML-документе

Возьмем этот рисунок и перевернем его так, чтобы все вложения выпали и распределились сверху вниз, как показано на рис. 9.5.

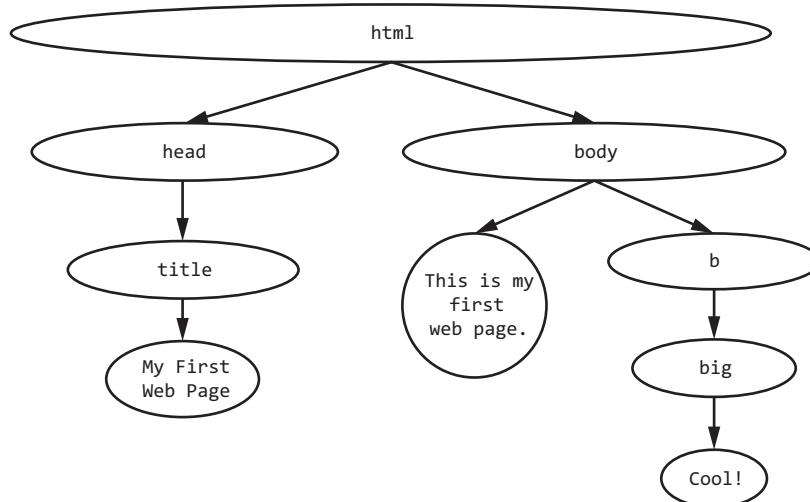


Рис. 9.5. HTML-документ в виде древовидной структуры данных

Узнаете? Это же наш старый друг, направленный ациклический граф (НАГ) из раздела «Стеки» на с. 166. Он же — древовидная структура данных (см. раздел

«Иерархические структуры данных» на с. 251). А еще HTML-разметку можно обработать с помощью приемов из главы 8, превратив ее в дерево синтаксического анализа (см. «Деревья синтаксического анализа» на с. 286).

Словарь древовидных структур данных

Древовидные структуры данных наподобие модели DOM настолько популярны в среде разработки, что успели обзавестись своим словарем. Примеры в табл. 9.1 взяты из рис. 9.5.

Таблица 9.1. Словарь древовидных структур данных

Термин	Определение	Пример
Узел	Элемент дерева	html, head, body
Внутренний узел	Элемент дерева с входящими и исходящими стрелками	title
Лист дерева	Элемент дерева, не имеющий исходящих стрелок	Cool!
Корень	Вершина дерева	html
Родительский узел	Узел со стрелками, прямо указывающими на другой узел	Узел html является родительским узлом для узлов head и body
Дочерний узел	Узел, на который прямо указывают стрелки другого узла	Узлы head и body — дочерние для узла html
Узел-потомок	Узел, на который прямо или косвенно указывают стрелки другого узла	Узел title — потомок узла html
Узел-предок	Узел со стрелками, прямо или косвенно указывающими на другой узел	Узел body — предок узла big
Соседний, или братский, узел	Узел, имеющий того же родителя, что и текущий узел	Узел head — братский для узла body

Узлы дерева определенным образом упорядочены. Например, узел head является первым дочерним узлом html, а узел body — вторым и последним дочерним узлом html.

Интерпретация модели DOM

Как браузер обрабатывает дерево документа? Теоретически можно создать специальное вычислительное аппаратное обеспечение с инструкциями, касающимися HTML-элементов, но никто до сих пор ничего подобного не сделал.

Поэтому компиляция дерева синтаксического анализа для модели DOM в машинный язык сейчас происходит с использованием *обхода в глубину*, как показано на рис. 9.6.

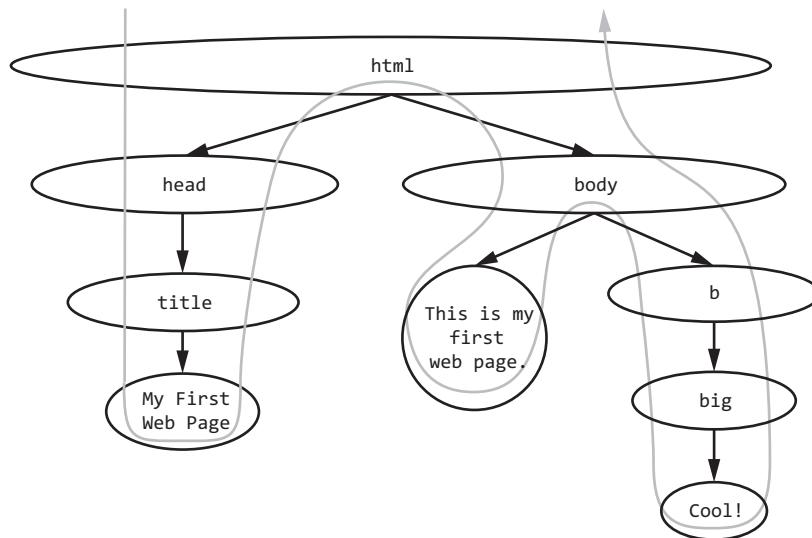


Рис. 9.6. Порядок обхода HTML-документа

Видим, что браузер начинает обход с корня дерева, спускаясь к его первому дочернему узлу, затем к первому дочернему узлу текущего узла, вплоть до первого листа дерева. Затем обрабатывается ближайший узел-предок с его дочерними узлами и т. д., пока не будет обработан каждый узел дерева. Обратите внимание, что порядок обработки соответствует структуре написанного HTML-кода. Обход в глубину представляет собой еще один способ практического применения стеков.

Каскадные таблицы стилей

Первоначальная идея HTML заключалась в том, что авторы веб-страниц позволяли браузеру самостоятельно определять, как отображать исходный текст. Это было не лишено смысла, потому что в те времена не было возможности задать размер окна браузера, разрешение экрана или список доступных цветов и шрифтов.

Растущая популярность веб-разработки стала притягивать внимание маркетологов. Все большее внимание уделялось внешней оболочке, поэтому было придумано множество приемов (в основном благодаря новой спецификации CSS¹) для контроля отображения страниц в браузере. Это полностью противоречило оригинальной идеи HTML, поэтому в результате воцарился хаос.

¹ CSS (Cascading Style Sheets) – каскадные таблицы стилей. – Примеч. науч. ред.

Веб-страницы с HTML-разметкой уже включали в себя информацию о стилях. Например, элемент `font`, предназначенный для выбора шрифта текста, имел атрибут `size` для обозначения размера шрифта. Такой подход работал не всегда, учитывая разнообразие устройств (от настольных ПК до мобильных телефонов), на которых могла отображаться данная страница. *Каскадные таблицы стилей* (Cascading Style Sheets, CSS) отделили стили страницы от ее HTML-разметки — теперь можно было использовать одну и ту же разметку, но разные стили в зависимости от устройства.

На рис. 9.7 показана структура данных, с помощью которой можно представить хранение HTML-элемента в памяти.

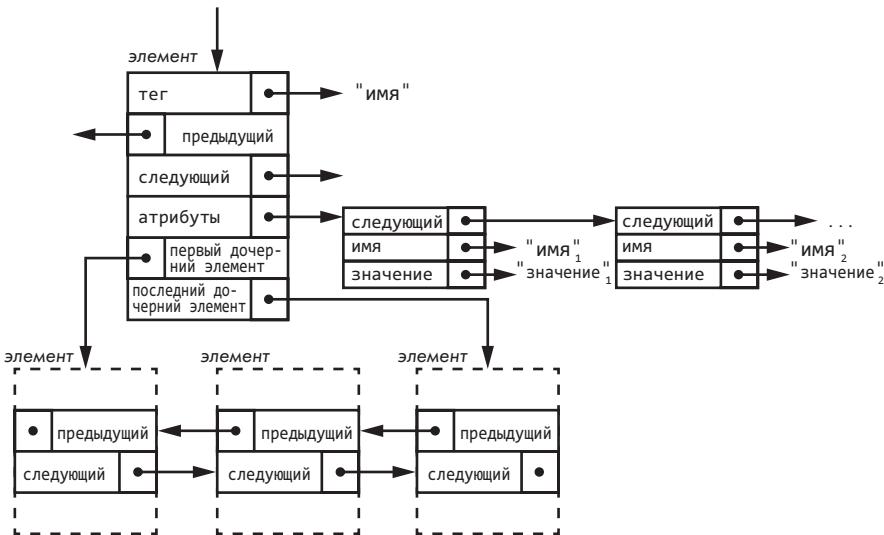


Рис. 9.7. Структура данных для HTML-элемента

Схема может показаться сложной, но она всего лишь объединяет все, что вы уже знаете из этой книги. Она представляет два составных типа данных (см. раздел «Составные типы данных» на с. 240): один — для элементов, а второй — для их атрибутов. Атрибуты организованы в виде односвязного списка (см. «Односвязные списки» на с. 242). Элементы собраны в древовидную структуру (см. «Иерархические структуры данных» на с. 251). Поскольку можно встретить произвольное число дочерних элементов, расположенных в определенном порядке, эти элементы организованы в двусвязный список (см. «Двусвязные списки» на с. 250).

Порядок элементов важен, потому что в CSS для обнаружения элемента в модели DOM используются специальные регулярные выражения (см. раздел «Регулярные выражения» на с. 281) под названием *селекторы*. Поиск селекторов в CSS похож на то, как `yacc` ищет определенные лексемы в стеке. После того

как подходящие элементы будут найдены, каскадные таблицы стилей привяжут к нему заданные атрибуты. Это позволит веб-дизайнеру, к примеру, изменить размер текста в зависимости от используемого устройства или свернуть содержимое бокового меню в выпадающий список на устройствах с небольшими экранами.

Каскадные таблицы стилей внесли некоторую путаницу в терминологию. Они определили огромное количество *свойств* — вроде цвета, размера шрифта и т. п. Как только свойства связываются с определенным элементом в модели DOM, они становятся *атрибутами* этого элемента.

В табл. 9.2 представлены некоторые селекторы CSS. Когда-то их было немного, однако новые селекторы продолжают появляться с пугающей быстротой.

Таблица 9.2. CSS-селекторы

Шаблон	Значение
*	Соответствует любому элементу
E	Соответствует любому элементу типа E (то есть <E>...</E>)
F	Соответствует любому элементу типа F (то есть <F>...</F>)
E F	Соответствует любому элементу F, который является потомком элемента E
E > F	Соответствует любому элементу F, который является дочерним для элемента E
E + F	Соответствует любому элементу F, который является прямым братским, или соседним, элементом для элемента E
E - F	Соответствует любому элементу F, перед которым расположен братский, или соседний, элемент E
E[name]	Соответствует любому элементу E, который имеет атрибут name
E[name=value]	Соответствует любому элементу E, который имеет атрибут name со значением value
E[name~="value"]	Соответствует любому элементу E, чей атрибут name представляет собой список слов, разделенных пробелами, где одно из слов соответствует значению value
E#id	Соответствует любому элементу E, который имеет атрибут ID со значением id
E.class	Соответствует любому элементу E, который имеет атрибут class со значением class
E:first-child	Соответствует любому элементу E, который является первым дочерним элементом для своего родительского элемента
E:last-child	Соответствует любому элементу E, который является последним дочерним элементом для своего родительского элемента

Шаблон	Значение
E:nth-child(n)	Соответствует любому элементу E, который является n-м дочерним элементом для своего родительского элемента
E:empty	Соответствует любому элементу E, не имеющему дочерних элементов
E:link	Соответствует любому элементу E, если тот представляет собой точку привязки гиперссылки (например, <a>)
E:visited	Соответствует любому элементу E, если тот представляет собой точку привязки гиперссылки (например, <a>), по которой был совершен переход
E:hover	Соответствует любому элементу E при наведении на него курсора мыши
E:active	Соответствует любому элементу E при нажатии на него левой кнопкой мыши
E:focus	Соответствует любому элементу E, если он находится в фокусе ввода (то есть может в данный момент времени принимать ввод с клавиатуры)

В HTML существует специальный элемент <link>, который позволяет привязать отдельный файл с CSS к веб-странице. Такой подход предпочтителен, потому что он предусматривает хранение содержимого страницы отдельно от ее стилей. Однако для примеров из данной книги достаточно элемента <style>, который позволяет напрямую добавлять CSS-стили в HTML-документы.

Изменим код веб-страницы из листинга 9.1, добавив элемент <style> со стилями (выделен **полужирным** шрифтом в листинге 9.4).

Листинг 9.4. Веб-страница со встроенными CSS-стилями

```
<html>
  <head>
    <title>
      Моя первая веб-страница
    </title>
    <style>
      body {
        color: blue;
      }
      big {
        color: yellow;
        font-size: 200%;
      }
    </style>
  </head>
  <body>
    Это моя первая веб-страница.
  </body>
</html>
```

```
<b>
  <big>
    Круто!
  </big>
</b>
</body>
</html>
```

В листинге 9.4 представлены два селектора: `body` и `big`. За каждым селектором следуют заключенные в фигурные скобки пары имен и значений свойств, которые отделяются друг от друга точкой с запятой. Имена и значения свойств разделены двоеточиями. Первым делом мы задаем цвет текста всего тела документа (свойство `color`) как `blue`, а текста внутри элемента `<big>` — как `yellow`. Кроме того, для элемента `<big>` мы настраиваем размер шрифта (свойство `font-size`), делая его равным 200 % от обычного размера шрифта. Попробуйте!

Обратите внимание, что в CSS есть свои причуды, о которых при разработке HTML-разметки, как правило, никто не задумывается. В HTML существуют элементы с определенным значением — например, элемент `` для **полужирного** начертания текста и элемент `<i>` для текста, выделенного *курсивом*. Представьте, что фрагмент кода на CSS в листинге 9.5 задает этим элементам противоположные значения.

Листинг 9.5. Меняем местами полужирный текст и текст, выделенный курсивом, с помощью CSS

```
b {
  font-style: italic;
  font-weight: normal;
}
i {
  font-style: normal;
  font-weight: bold;
}
```

CSS во всех смыслах стирает разницу между многими HTML-элементами. Стили некоторых элементов заданы по умолчанию, но если эти стили изменить с помощью CSS, элементы потеряют свое первоначальное назначение.

Изначально CSS предоставлял лишь более гибкий способ добавления атрибутов к элементам. Затем в нем стали появляться новые атрибуты, которые не очень-то вписывались в HTML. В результате какие-то атрибуты можно объявить с помощью как HTML, так и CSS, а какие-то — только с помощью CSS. Профессиональное сообщество склоняется к тому, что старый способ добавления атрибутов должен уйти в прошлое, но важно понимать, что это повлечет сложности в изменении и поддержке существующего кода.

XML и друзья

XML во многом похож на HTML. Однако, подобно SGML, он требует задания *правильного формата* для элементов. Это означает, что для каждого открывающего *<тега>* нужно использовать соответствующий ему закрывающий *</тег>*. Основная разница между HTML и XML заключается в том, что HTML был создан специально для веб-страниц, а XML — это многофункциональный язык разметки, который подойдет для разных приложений.

Многие теги в XML не имеют предопределенного значения — им можно присвоить любое значение, которое подойдет для конкретного случая. Таким образом, для любого приложения можно создавать собственные языки разметки на основе XML. Представьте, например, что вы разрабатываете приложение для учета овощей в саду. Именно для этой цели вы можете создать специальный Язык Разметки Овощей (Vegetable Markup Language, VML), как показано в листинге 9.6.

Листинг 9.6. Пример языка разметки на основе XML

```
<xml>
  <garden>
    <vegetable>
      <name>томат</name>
      <variety>Cherokee Purple</variety>
      <days-until-maturity>80</days-until-maturity>
    </vegetable>
    <vegetable>
      <name>брюквa</name>
      <variety>American Purple Top</variety>
      <days-until-maturity>90</days-until-maturity>
    </vegetable>
    <vegetable>
      <name>брюквa</name>
      <variety>Helenor</variety>
      <days-until-maturity>100</days-until-maturity>
    </vegetable>
    <vegetable>
      <name>брюквa</name>
      <variety>White Ball</variety>
      <days-until-maturity>75</days-until-maturity>
    </vegetable>
    <vegetable>
      <name>брюквa</name>
      <variety>Purple Top White Globe</variety>
      <days-until-maturity>45</days-until-maturity>
    </vegetable>
  </garden>
</xml>
```

Возможность создавать собственные языки разметки и специальные элементы для них порождает конфликты. Представьте, что кто-то в дополнение к VML разработал новый Язык Разметки Рецептов (Recipe Markup Language, RML), где также встречается элемент `<name>`, как показано в листинге 9.7.

Листинг 9.7. Пример конфликта с элементом `name` в языке разметки на основе XML

```
<xml>
  <garden>
    <vegetable>
      <name>томат</name>
      <variety>Cherokee Purple</variety>
      <days-until-maturity>80</days-until-maturity>
      <name>Purple Tomato Salad</name>
    </vegetable>
  </garden>
</xml>
```

Как понять, какой из элементов `<name>` относится к VML, а какой к RML? Понадобится специальный механизм объединения двух языков разметки, чтобы не путать представленные в них элементы `<name>`. Этот механизм в виде префиксов для тегов уже существует и называется *пространством имен*.

Как и все, что связано с браузерами, пространства имен имеют несколько способов задания, но в этой книге я рассмотрю только один из них. Все пространства имен связаны с определенными URL — уникальными, но необязательно действительными. Для связи префикса пространства имен с URL используется атрибут `xmlns` в элементе `<xml>`. В листинге 9.8 представлен пример одновременного использования языков разметки для овощей и рецептов с указанием пространств имен.

Листинг 9.8. Пример языка разметки на основе XML с использованием пространств имен

```
<xml xmlns:vml="http://www.garden.org" xmlns:rml="http://www.recipe.org">
  <vml:garden>
    <vml:vegetable>
      <vml:name>томат</vml:name>
      <vml:variety>Cherokee Purple</vml:variety>
      <vml:days-until-maturity>80</vml:days-until-maturity>
      <rml:name>Purple Tomato Salad</rml:name>
    </vml:vegetable>
  </vml:garden>
</xml>
```

Видим, что теперь элементы из обоих вымышленных языков разметки собраны в одном фрагменте кода и что их можно различить по префиксам. Префикс пространства имен задается произвольно — его выбирает программист, который решает объединить несколько языков разметки. Префикс `rml` необязательно должен соответствовать языку Recipe Markup Language — мы можем использовать

recipe вместо него, а rml оставить для другого RML-языка, например Ridiculous Markup Language.

Для создания пользовательских языков разметки, таких как рассмотренные выше, доступно множество инструментов. Существуют и различные библиотеки языков программирования, которые позволяют создавать деревья синтаксического анализа из XML-документов и работать с ними.

Один из таких инструментов – *Document Type Definition (DTD, определение типа документа)*. Его можно охарактеризовать как метаразметку. DTD – это документ, по формату похожий на XML (только без закрывающих тегов), который определяет доступные элементы в языке разметки. В XML существует механизм, позволяющий XML-документу ссылаться на DTD. Например, можно создать DTD-документ, в котором будет указано, что в элемент `<garden>` можно вставить один или несколько элементов `<vegetable>`, а внутри элемента `<vegetable>` допускаются только элементы `<name>`, `<variety>` и `<days-until-maturity>`. Анализаторы XML могут проверять XML-документы на соответствие правилам DTD. Это полезно, но на самом деле не так уж и важно – например, DTD может убедиться, что элемент `<variety>` присутствует в документе, но не может проверить его правильность.

Язык *XML Path Language (XPath)* предоставляет селекторы для XML-документов на основе, как вы уже, наверное, догадались, еще одного несовместимого с другими языками синтаксиса. Селекторы XPath имеют ту же функциональность, что и селекторы CSS, но задаются при помощи совершенно нового для нас синтаксиса. Сам по себе XPath не обладает особенной ценностью, но он является важной составляющей *расширяемого языка преобразования XML-документов (Xtensible Stylesheet Language Transformations, XSLT)*.

XSLT представляет собой еще один язык на основе XML. В паре с XPath он позволяет создавать фрагменты кода на XML, которые преобразуют XML-документ в новые формы путем поиска и внесения изменений в дереве синтаксического анализа. В листинге 9.9 представлен простой пример, в котором выражение XPath используется для поиска соответствий по овощам из сада и вывода их названий и разновидностей, разделенных пробелами.

Листинг 9.9. Поиск овощей при помощи XSLT и XPath

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/garden/vegetable">
    <xsl:value-of select="variety"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="name"/>
  </xsl:template>
</xsl:stylesheet>
```

Применив XSLT-код из листинга 9.9 к XML из листинга 9.6, получим результаты, представленные в листинге 9.10.

Листинг 9.10. Результаты поиска по овощам

томат Cherokee Purple
брюква American Purple Top
брюква Helenor
брюква White Ball
брюква Purple Top White Globe

Еще один пример показан в листинге 9.1 — выбор только овощей со значением свойства `name`, равным брюква.

Листинг 9.11. Поиск по свойству `name`

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/garden/vegetable[name/text()='брюква']">
    <xsl:value-of select="name"/>
    <xsl:text> </xsl:text>
  </xsl:template>
  <xsl:template match="text()"/>
</xsl:stylesheet>
```

Применив XSLT-код из листинга 9.11 к XML из листинга 9.6, получим результаты, представленные в листинге 9.12.

Листинг 9.12. Результаты поиска по свойству `name`

брюква брюква брюква брюква

Язык XSLT особенно полезен при преобразовании разметки, содержащей произвольные данные, в HTML-код для отображения в браузере.

JavaScript

Наша учебная веб-страница статична — она лишь отображает некий отформатированный текст. Возвращаясь к рис. 9.1, чтобы вывести на ней что-то новое, нужно предоставить веб-серверу новый URL и получить новый документ. Мало того что этот способ слишком медленный, он еще и требует черезсур много ресурсов. Если в поле формы ввести номер телефона, данные отправятся на сервер для проверки правильности ввода, и в случае ошибки мы получим новую страницу с соответствующим сообщением.

В 1993 году Марк Андриссен (Mark Andreesen) создал графический веб-браузер Mosaic, который сразу же подогрел интерес потребителей к интернет-разработке. Андриссен на этом не остановился и перешел в Netscape, где в 1994 году выпустил браузер Netscape Navigator. Чтобы сделать веб-страницы более интерактивными, в 1995 году Netscape представил новый язык программирования — JavaScript. Позже JavaScript был стандартизирован организацией стандартизации Ecma International (бывшей Европейской ассоциацией производителей

компьютеров) под названием ECMA-262. Стандартизированная версия сегодня больше известна как ECMAScript (звучит как название кожной болезни), и ее основное назначение — усложнять жизнь браузерам. JavaScript был создан на основе языков программирования C и Java, последний из которых сам был создан на основе языка C.

С помощью JavaScript веб-страницы могут запускать нужные программы на компьютере, а не на сервере. Такие программы могут вносить изменения в модель DOM и напрямую взаимодействовать с сервером, как показано на рис. 9.8.

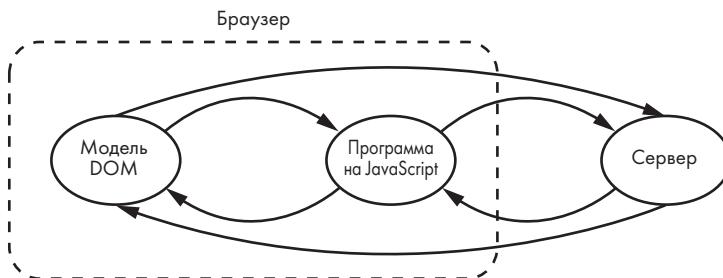


Рис. 9.8. Взаимодействие веб-браузера с JavaScript и веб-сервером

Взаимодействие программы на языке JavaScript и сервера отличается от взаимодействия браузера с сервером, показанного на рис. 9.1. В случае с JavaScript используется так называемый *асинхронный JavaScript и XML* (Asynchronous JavaScript and XML, AJAX). Разберем этот термин по частям. Понятие «*асинхронный*» связано со счетчиками со сквозным переносом, о котором мы говорили в разделе «Счетчики» на с. 119. В случае с браузером это означает, что браузер не управляет временем ответа с сервера и вообще возможностью получить этот ответ. Часть *«JavaScript»* означает, что именно программа на языке JavaScript отвечает за ответы с сервера. Часть *«и XML»*, я думаю, объяснять не нужно. И в заключение, данные с сервера, получаемые JavaScript-программой, изначально кодируются с использованием *XML* вместо HTML.

Код на JavaScript можно добавить в HTML-разметку напрямую при помощи элементов `<script>`. Добавим кое-что к листингу 9.4 (изменения снова выделены полужирным начертанием), получив листинг 9.13.

Листинг 9.13. Веб-страница со встроенным JavaScript-кодом

```
<html>
  <head>
    <title>
      Моя первая веб-страница
    </title>
```

```
<style>
  body {
    color: blue;
  }
  big {
    color: yellow;
    font-size: 200%;
  }
</style>
<script>
  window.onload = function() {
    var big = document.getElementsByTagName('big');
    big[0].style.background = "green";
  }
</script>
</head>
<body>
  Это моя первая веб-страница.
  <b>
    <big>
      Круто!
    </big>
  </b>
</body>
</html>
```

Рассмотрим вкратце, что делает этот код. Часть взаимодействия с браузером заключается в переменной `window.onload`, которая задается как функция — она выполнится при завершении загрузки начальной страницы. Вторая часть взаимодействия — это функция `document.getElementsByTagName`, которая возвращает массив с подходящими элементами, полученными из DOM. В нашем случае вернется единственный элемент `<big>`. Помимо этого, мы получаем возможность изменять свойства элемента — в примере цвет фона элемента меняется на зеленый.

Существует множество специальных функций для управления моделью DOM. Они позволяют не только программно изменять некоторые CSS-стили, но и переопределять само дерево DOM, добавляя или удаляя элементы.

jQuery

С функциями для работы с DOM в браузере есть две проблемы. Во-первых, в разных браузерах эти функции могут вести себя по-разному. Во-вторых, они достаточно громоздки и неудобны для пользователей.

Потому пришло время познакомиться с *jQuery* — библиотекой, созданной американским программистом Джоном Резигом (John Resig) в 2006 году. Библиотека *jQuery* избавляет нас от обеих проблем, описанных выше, — она сама сглаживает

несовместимости браузеров, так что теперь не нужно обращать на них внимание, и предоставляет простой интерфейс для управления моделью DOM.

Библиотека jQuery объединяет селекторы с *действиями*. Для примера, код в листинге 9.14 делает то же, что и код в листинге 9.13, но теперь он более удобен для программистов.

Листинг 9.14. Веб-страница со встроенным кодом на JavaScript и jQuery

```
<html>
  <head>
    <title>
      Моя первая веб-страница
    </title>
    <style>
      body {
        color: blue;
      }
      big {
        color: yellow;
        font-size: 200%;
      }
    </style>
    <script type="text/javascript" src="https://code.jquery.com/
      jquery-3.2.1.min.js"> </script>
    <script>
      $(function() {
        $('big').css('background', 'green');
      });
    </script>
  </head>
  <body>
    Это моя первая веб-страница.
    <b>
      <big>
        Круто!
      </big>
    </b>
  </body>
</html>
```

Первый элемент `<script>` импортирует библиотеку jQuery, а второй непосредственно содержит код для работы с ней. Функция, которую браузер запускает сразу после загрузки документа, содержит единственное jQuery-выражение. Оно состоит из селектора `($('big')')`, похожего на селекторы из табл. 9.2, и действия `(.css('background', 'green'))`, которое нужно применить к выбранным элементам. В нашем примере функция `css` изменяет свойство `background`, задавая его значение равным `green`.

Добавим немного кода на jQuery (листинг 9.15) в рассматриваемую функцию, чтобы сделать ее чуть более интерактивной.

Листинг 9.15. Обработчик событий в jQuery

```
$( 'big' ).click( function() {
    $( 'big' ).before( '<i>Очень</i>' );
    $( 'big' ).css( 'font-size', '500%' );
});
```

Этот простой фрагмент кода добавляет *обработчик событий* к элементу `<big>`. Функция обработчика выполнится, когда пользователь нажмет левой кнопкой мыши на соответствующий элемент. При этом перед элементом `<big>` добавится новый элемент `<i>`, а размер шрифта содержимого элемента `<big>` увеличится.

Пример показывает, что jQuery значительно упрощает работу с моделью DOM при использовании JavaScript. Проследить за изменениями страницы при щелчке мышью по элементу можно из консоли разработчика в браузере.

jQuery сформировала очень интересную и популярную тенденцию — эта библиотека до сих пор широко используется разработчиками. Однако, как это часто случается в веб-разработке, уже появилось множество других JavaScript-библиотек, которые делают то же самое, но в несовместимой с jQuery форме.

SVG

Масштабируемая векторная графика (Scalable Vector Graphics, *SVG*) — этакая белая ворона среди многочисленных браузерных дополнений. Она представляет собой совершенно новый язык для создания красивых графиков и текстов, но при этом абсолютно не совместима с другими инструментами.

В 1982 году Джон Уорнок (John Warnock) и Чак Гешке (Chuck Geschke) создали компанию Adobe Systems и разработали язык программирования PostScript. Уорнок годами работал над самыми заковыристыми идеями в PostScript, и в итоге разработка PostScript стала похожей на приведение чрезмерно сложного SGML к более простому HTML. Эта парочка поймала удачу за хвост, когда Стив Джобс предложил им использовать PostScript для запуска лазерных принтеров. С тех пор LaserWriter, созданный Apple на основе PostScript, стал их визитной карточкой и во многом способствовал популярности компьютерной верстки и успеху Adobe.

PostScript столкнулся с проблемой *переносимости* — получения одних и тех же результатов в разных системах. Чтобы ее решить, был создан *переносимый формат документов* (Portable Document Format, *PDF*) на основе PostScript. SVG так или иначе представляет собой PDF-документы, встроенные в браузер. Конечно, SVG и PDF несовместимы в полной мере — это было бы слишком.

В целом SVG более автоматизирована, чем недавно добавленные холсты, которые мы рассмотрим в следующем разделе. SVG достаточно указать, что нужно сделать, а для холста придется написать целую управляющую программу. Для примера добавьте фрагмент кода из листинга 9.16 в элемент `body` своей

веб-страницы и откройте ее в браузере — ведь каждой веб-странице просто необходим красный пульсирующий круг.

Листинг 9.16. Пульсирующий круг, созданный при помощи SVG

```
<br>
<svg xmlns="http://www.w3.org/2000/svg" width="400" height="400">
  <circle id="c" r="10" cx="200" cy="200" fill="red"/>
  <animate xlink:href="#c" attributeName="r" from="10" to="200" dur="5s"
repeatCount="indefinite"/>
</svg>
```

HTML5

Как я уже упоминал в начале главы, ничто в браузере не создается в одном-единственном варианте. HTML5 представляет собой последнюю (на данный момент) реинкарнацию HTML, в которой, помимо всего прочего, была добавлена куча *семантических* элементов, включая `<header>`, `<footer>` и `<section>`. При использовании по назначению эти элементы должны задавать определенную структуру документов.

В HTML5 добавлен *холст*, который по функциональности схож с SVG — он делает практически все то же самое, но абсолютно другим способом. Одно из основных различий между ними заключается в том, что для управления холстом используется набор новых функций JavaScript, а для SVG — уже имеющиеся функции для внесения изменений в модель DOM. Другими словами, чтобы переписать код в листинге 9.16 с использованием холста, понадобится программа на JavaScript.

Также в HTML5 добавлены элементы `<audio>` и `<video>`, предоставляющие некоторые стандартные механизмы для управления аудио- и видеофайлами.

JSON

Я немного затронул тему AJAX в разделе «JavaScript» на с. 312 и упомянул, что асинхронные данные в формате XML отправляются с сервера в JavaScript-программу в браузере. Четыре раздела назад это еще *можно было* принять за правду, но сейчас вам стоит знать, что букву *X* в аббревиатуре AJAX на самом деле нужно заменить на *J* (от аббревиатуры JSON). Новая аббревиатура AJAJ, конечно же, существует, но программисты по старинке пользуются сокращением AJAX.

JSON расшифровывается как JavaScript Object Notation (нотация объектов JavaScript). JSON представляет собой удобный для восприятия формат *объекта* в JavaScript — одного из составных типов данных. В теории одни данные в этом формате можно легко заменять другими, но из-за проблем с имеющейся спецификацией это возможно, только если придерживаться некоторых негласных правил (например, не использовать определенные символы). Программистам стоит учитывать еще и то, что JSON поддерживает не все типы данных в JavaScript.

В листинге 9.17 описываются создание JavaScript-объекта и его конвертация в формат JSON. Результат, выделенный в коде полужирным, сохраняется в переменную `the_quest`.

Листинг 9.17. JSON и объект «Аргонавты»

```
var argonauts = {};
argonauts.goal = "Золотое руно";
argonauts.sailors = [];
argonauts.sailors[0] = { name: "Акаст", father: "Пелий" };
argonauts.sailors[1] = { name: "Актер", father: "Гиппас" };
argonauts.sailors[2] = { name: "Адмет", father: "Ферет" };
argonauts.sailors[3] = { name: "Амфиарай", father: "Экл" };
argonauts.sailors[4] = { name: "Анкей", father: "Посейдон" };

var the_quest = JSON.stringify(argonauts);

{
  "goal": "Золотое руно",
  "sailors": [
    { "name": "Акаст", "father": "Пелий" },
    { "name": "Актер", "father": "Гиппас" },
    { "name": "Адмет", "father": "Ферет" },
    { "name": "Амфиарай", "father": "Экл" },
    { "name": "Анкей", "father": "Посейдон" }
  ]
}"
```

При использовании JavaScript формат JSON более предпочтителен, чем XML, и не только потому, что объекты JavaScript легко конвертируются в JSON, как показано в листинге 9.17. Сопутствующая функция `eval` в JavaScript может напрямую работать с форматом JSON, потому что он хранит в себе данные как в отдельной JavaScript-программе. Популярность JSON обусловлена тем, что для импортирования и экспортования данных не нужно создавать дополнительный код.

Однако простота использования JSON не должна вводить в заблуждение. Беспечное импортирование данных в формате JSON при помощи функции `eval` может сыграть на руку злоумышленникам, позволив им выполнять в браузере произвольные фрагменты кода. Недавно была добавлена сопутствующая функция `JSON.parse` для безопасного обратного конвертирования данных в формате JSON в JavaScript-объект.

Выводы

В этой главе вы узнали об основных компонентах, из которых складывается работа веб-браузера, — все они представлены на рис. 9.9. Конечно, мы не охватили абсолютно все возможности браузера, которые не особенно важны в контексте данной книги, — например, закладки и историю поисковых запросов.

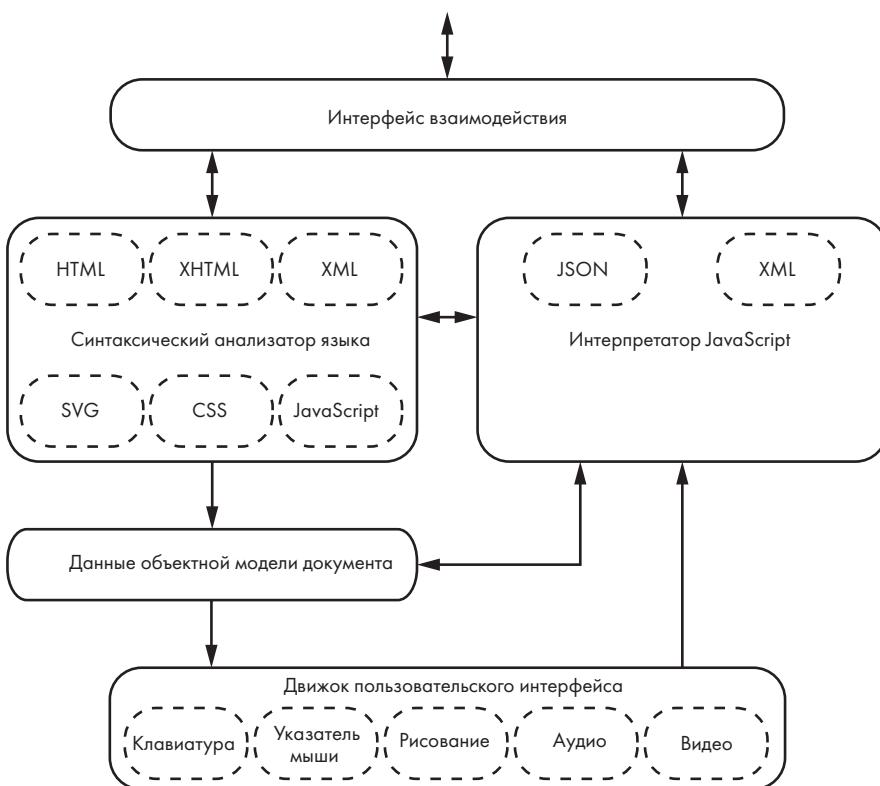


Рис. 9.9. Блок-схема браузера

На первый взгляд схема может показаться сложной, но она всего лишь объединяет все, что вы уже видели: синтаксические анализаторы, деревья синтаксического анализа, регулярные выражения, интерпретаторы, ввод и вывод данных.

Здесь также выражена разница между проектированием программного и аппаратного обеспечения. Проектировать аппаратное обеспечение значительно дороже, чем программное. Проектировщик аппаратного обеспечения вряд ли создаст систему с шестью несовместимыми методами для одной и той же задачи еще до завтрака. В то же время из-за отсутствия значительных начальных затрат разработчики ПО часто более расслаблены. В результате они создают исходно дешевые, большие и сложные системы, стоимость поддержки и обслуживания которых в дальнейшем оказывается довольно высокой.

Получив знания о сложности веб-браузера как интерпретатора, в следующей главе мы потренируемся в написании программ для него. Вы увидите примеры браузерных программ на языках JavaScript и С. Они продемонстрируют некоторые соображения, важные для разработки систем, но зачастую скрытые от глаз системных программистов.

10

Прикладное и системное программирование



В главе 9 мы рассмотрели работу веб-браузеров — сложных прикладных программ, которые предоставляют виртуальные машины для обработки высокоуровневых инструкций. В этой главе мы перейдем к написанию программ для браузеров, а за ними и программ, которые в браузере не используются. Структура двух программ представлена на рис. 10.1.

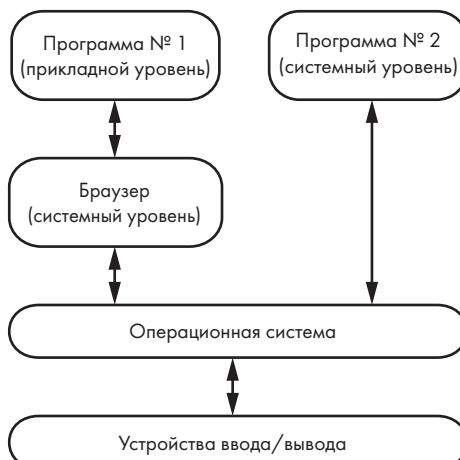


Рис. 10.1. Сценарии для двух программ

Операционная система скрывает всю сложность устройств ввода/вывода от пользовательских программ. Таким же образом сложные пользовательские программы наподобие браузеров скрывают сложность взаимодействия с операционной системой от прикладных программ, которые на них надстраиваются. Если вам достаточно писать высокоуровневое прикладное ПО, можете на этом остановиться. Однако, чтобы стать хорошим системным программистом, придется узнать больше.

В этой главе будут представлены более развернутые фрагменты кода на JavaScript и C. Не переживайте, если вы недостаточно знакомы с этими языками, — мы не будем вдаваться в несущественные детали.

Рассмотрим игру, в которой компьютер задает последовательные вопросы, пытаясь угадать название животного. По мере необходимости в программу добавляются названия новых животных и вопросы, позволяющие отличить их от остальных. Таким образом программа «обучается», строя двоичное дерево знаний.

Взаимодействие между *компьютером* (текст, выделенный монотонным шрифтом) и *пользователем* (монотонный шрифт с полужирным начертанием) выглядит примерно так:

Загадай животное.
Оно лает?
Да
Это собака?
Да
Я угадал!
Давай сыграем заново.
Загадай животное.
Оно лает?
Да
Это собака?
Нет
Сдаюсь. Что это за животное?
Гигантский фиолетовый сноркльвакер¹
Какой вопрос я могу задать, чтобы отличить гигантский фиолетовый сноркльвакер от собаки?
Оно живет в шкафу тревог?
Спасибо. Запомню это.
Давай сыграем заново.
Загадай животное.
Оно лает?
Да
Это собака?
Нет

¹ Гигантский фиолетовый сноркльвакер — The Giant Purple Snorklewacker — персонаж комиксов Bloom County. — *Примеч. пер.*

Оно живет в шкафу тревог?

Да

Это гигантский фиолетовый сноркльвакер?

Да

Я угадал!

Давай сыграем заново.

На рис. 10.2 представлен план реализации алгоритма.

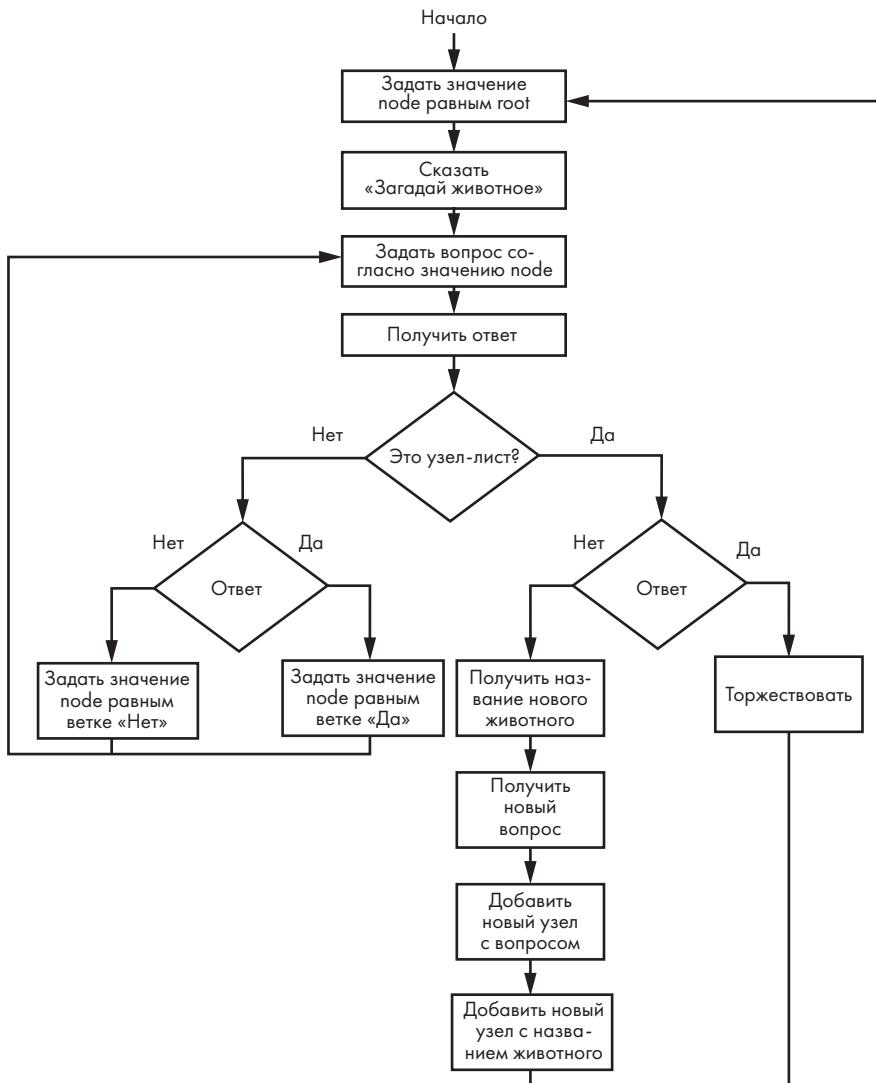


Рис. 10.2. Блок-схема алгоритма «Угадай животное»

Как видите, сначала мы задаем вопросы и продвигаемся вниз по дереву знаний. Если угадываем животное — радуемся, если нет — просим пользователя предоставить ответ и сопутствующий вопрос, добавляем их к дереву знаний и начинаем заново.

Программа продвигается вниз с левой стороны дерева знаний. Достигнув конца пути на правой стороне дерева, она либо радуется удаче, либо добавляет новые знания в базу.

«Угадай животное», версия 1: HTML и JavaScript

Приступим к написанию программы. Пойдем удобным путем (хотя он может расстроить кое-кого из моих коллег). Сделаем *ловкий трюк* — работающий, хотя и немного запутанный. Как вы узнали из предыдущей главы, модель DOM представляет собой дерево — подмножество НАГ, как и двоичное дерево. Так вот, мы построим двоичное дерево знаний в DOM в виде набора вложенных друг в друга невидимых элементов `<div>`. Подобную структуру данных можно создать и в JavaScript, но с помощью встроенных инструментов браузера это будет сделано проще. Как показано на рис. 10.3, программа начинается с первого вопроса и двух ответов из двоичного дерева.

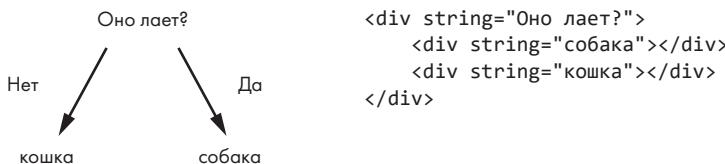


Рис. 10.3. Начало двоичного дерева

Сыграем в игру. Мы ответили «Да» на вопрос программы «Оно лает?» и «Нет» на вопрос «Это собака?». Затем программа спросила: «Что это за животное?», и мы ответили: «Гигантский фиолетовый сноркльвакер». После чего программа уточнила, какой вопрос она может задать в следующий раз, чтобы отличить гигантского фиолетового сноркльвакера от других животных, и мы ответили: «Оно живет в шкафу тревог?» Соответствующий вопрос был добавлен в дерево знаний, как показано на рис. 10.4.

Каркас прикладного уровня

В листинге 10.1 показан каркас веб-страницы, куда мы позже добавим код. Аdeptы чистоты кода очень сильно расстроятся, увидев этот фрагмент, поскольку в нем смешаны и HTML, и CSS, и JavaScript. Но мы пишем не весь сайт, а лишь учебный пример, поэтому для простоты объединим весь код в одном файле.

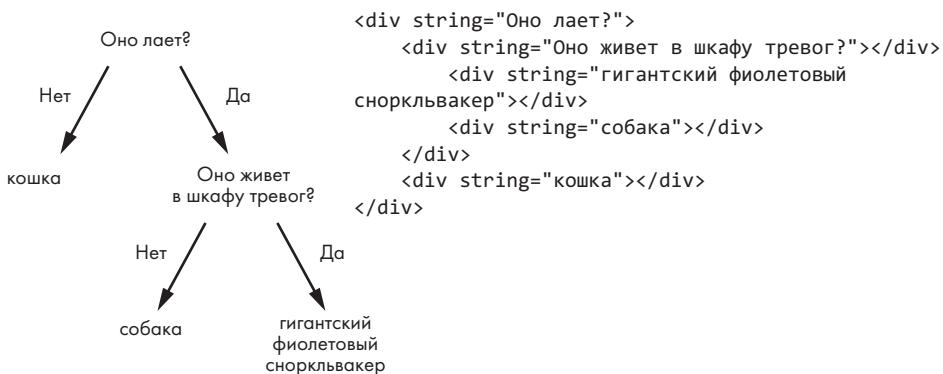


Рис. 10.4. Изменения в дереве знаний

Листинг 10.1. Каркас веб-страницы

```

1 <html>
2   <head>
3     <!-- Включение jQuery -->
4     <script type="text/javascript" src="https://code.jquery.com/
5       jquery-3.1.1.min.js"> </script>
6   <title>Каркас веб-страницы</title>
7
8   <style>
9     <!-- CSS-код -->
10  </style>
11
12  <script type="text/javascript">
13
14    <!-- JavaScript-код -->
15
16    $(function() {
17      <!-- JavaScript-код выполняется по готовности документа -->
18    });
19
20  </script>
21 </head>
22
23 <body>
24   <!-- HTML-код -->
25 </body>
26 </html>
  
```

Содержимое тега `title` можно изменить, например, на `Угадай животное`.

В предыдущей главе вы уже познакомились с компонентами веб-браузера (см. рис. 9.9). Теперь используем некоторые из них на практике.

Тело веб-страницы

Начнем с тега `<body>`, содержимое которого представлено в листинге 10.2. Этот фрагмент кода нужно вставить в программу из листинга 10.1 вместо строки 24 (`<!-- HTML-код -->`).

Листинг 10.2. HTML-код программы «Угадай животное»

```
1 <!-- Невидимое дерево знаний -->
2
3 <div id="root" class="invisible">
4   <div string="Оно лает">
5     <div string="собака"></div>
6     <div string="кошка"></div>
7   </div>
8 </div>
9
10 <div id="dialog">
11   <!-- Здесь выполняется преобразование -->
12 </div>
13
14 <!-- Название нового животного -->
15
16 <div id="what-is-it" class="start-hidden">
17   <input id="what" type="text"/>
18   <button id="done-what">Done</button>
19 </div>
20
21 <!-- Вопрос о новом животном -->
22
23 <div id="new-question" class="start-hidden">
24   Какой вопрос я могу задать, чтобы отличить
25   <span id="new"></span> от <span id="old"></span>?
26   <input id="question" type="text"/>
27   <button id="done-question">Готово</button>
28 </div>
29
30 <!-- Кнопки "Да" и "Нет" -->
31
32 <div id="yesno" class="start-hidden">
33   <button id="yes">Да</button>
34   <button id="no">Нет</button>
35 </div>
```

Видим, что в строках с 3-й по 8-ю задается начало дерева знаний — первый вопрос с ответами. Атрибут `string` хранит в себе вопрос, кроме узлов-листьев, где этот же атрибут означает название животного. Вопрос содержит два элемента `<div>`, первый из которых предназначен для ответа "да", а второй — для ответа "нет". Все дерево обернуто в элемент `<div>` со стилями, делающими его невидимым.

Элемент с идентификатором `dialog` в строках 10–12 содержит разговор компьютера и игрока. Внутри элемента с идентификатором `what-is-it` (строки 16–19)

хранятся текстовое поле для названия нового животного и кнопка, которую игрок должен нажать по завершении ввода. Элемент с идентификатором `new-question` (строки 23–28) содержит текстовое поле для нового вопроса и такую же кнопку, которую игрок должен нажать по завершении ввода. Кнопки «Да» и «Нет» находятся в элементе с идентификатором `yesno` (строки 32–35). Три тега `<div>` для пользовательского ввода (строки 16, 23 и 32) обозначены классом `start-hidden`, чтобы оставаться невидимыми в начале игры.

JavaScript

Теперь перейдем к коду на языке JavaScript. Первая его часть представлена в листинге 10.3.

Для начала объявим переменную `node` в строке 14 в коде каркаса веб-страницы (листинг 10.1) вместо строки `<!-- JavaScript-код -->`. Этот код можно перенести в функцию, которая запускается сразу после загрузки документа, но мы специально написали его здесь, чтобы к нему было легко обращаться из консоли разработчика в браузере. Еще две функции также объявлены не в коде загрузки документа как раз потому, что они не зависят от времени загрузки страницы.

Листинг 10.3. Переменная и функции JavaScript для игры «Угадай животное»

```
1 var node; // текущая позиция в дереве знаний
2
3 // Добавить указанный html-код к диалогу. Не добавлять вопрос, если
4 // новый узел не имеет дочерних узлов, то есть не имеет связанного вопроса.
5 // Иначе сделать новый узел текущим и задать вопрос, используя атрибут
6 // string узла. Если это узел-лист, использовать его как вопрос вместо
7 // названия животного. Функция вернет true, если новый узел – это узел-лист.
8
9 function
10 question(new_node, html)
11 {
12     $('#dialog').append(html); // добавить указанный html-код к диалогу
13
14     if ($(new_node).length == 0) { // не добавлять вопрос, если у узла нет
15         // дочерних узлов
16         return (true);
17     }
18     else {
19         node = new_node; // переход к новому узлу
20
21         if ($(node).children().length == 0)
22             $('#dialog').append('Is it a ' + $(node).attr('string') + '?');
23         else
24             $('#dialog').append($(node).attr('string') + '?');
25
26         return (false);
27     }
28 }
```

```
27 }
28
29 // Перезапуск игры. Все кнопки и текстовые поля скрываются, ввод
30 // сбрасывается, устанавливается начальный узел и приветствие, задается
31 // первый вопрос, отображаются кнопки да/нет.
32
33 function
34 restart()
35 {
36   $('.start-hidden').hide();
37   $('#question,#what').val('');
38   question($('#root>div'), '<div><b>Загадай животное.</b></div>');
39   $('#yesno').show();
40 }
```

Далее заменим строку 17 (`<!-- JavaScript-код выполняется по готовности документа -->`) в листинге 10.1 на код из листинга 10.4.

Листинг 10.4. Функция, которая запустится сразу после загрузки документа для игры «Угадай животное»

```
1 restart(); // Настроить все для первого запуска.
2
3 // Пользователь ввел новый вопрос. Создать узел с этим вопросом и поместить
4 // в него старый узел с ответом "нет". Затем создать узел с новым названием
5 // животного и поместить его в узел с новым вопросом перед старым узлом
6 // с ответом "нет", чтобы превратить его в ответ "да". Начать заново.
7
8 $('#done-question').click(function() {
9   $(node).wrap('<div string="' + $('#question').val() + '"></div>');
10  $(node).parent().prepend('<div string="' + $(what).val() + '"></div>');
11  $('#dialog').append("<div> Спасибо. Запомню это.</div><p>");
12  restart();
13 });
14
15 // Пользователь ввел новое название животного и нажал кнопку "Готово". Скрыть
16 // элементы диалога и сделать видимыми текстовое поле new-question для нового
17 // вопроса и кнопку "Готово". Передать в запрос новое и старое названия животного.
18
19 $('#done-what').click(function() {
20   $('#what-is-it').hide();
21   $('#new').text($('#what').val());
22   $('#old').text($(node).attr('string'));
23   $('#new-question').show();
24   $('#dialog div:last').append(' <i>' + $('#what').val() + '</i>');
25 });
26
27 // Пользователь нажал "да" при ответе на вопрос. Перейти вниз по дереву, если
28 // это был не последний узел – иначе восторжествовать и начать новую игру.
29
30 $('#yes').click(function() {
31   if (question($(node).children(':first-child'), ' <i>yes</i><br>')) {
```

```
32     $('#dialog').append("<div>Я угадал! Я такой умный!</div><p>");  
33     restart();  
34 }  
35});  
36  
37 // Пользователь нажал "нет" при ответе на вопрос. Перейти вниз по дереву, если  
38 // это был не последний узел – иначе скрыть кнопки "да" и "Нет" и сделать видимыми  
39 // текстовое поле what-is-it для нового названия животного и кнопку "Готово".  
40  
41 $('#no').click(function() {  
42     if (question($(node).children(':last-child'), ' <i>no</i><br>')) {  
43         $('#yesno').hide();  
44         $('#dialog').append('<div> Сдаюсь. Что это за животное?</div>');  
45         $('#what-is-it').show();  
46     }  
47 });
```

Для запуска игры вызывается функция `restart` (строка 1). Остальные функции – это *обработчики событий*, в JavaScript они заменяют обработчики прерываний, которые мы рассматривали в главе 5, – по одному обработчику событий на каждый элемент `button`. Каждый обработчик вызывает анонимную функцию (встраиваемую безымянную функцию) при нажатии на соответствующую кнопку.

Потренируйтесь в написании программы, вводя код из листингов вручную. Сохраните результат в отдельном файле под названием, например, `gta.html` и откройте его в браузере. Попробуйте сыграть в игру. Откройте инструменты разработчика в браузере и найдите HTML-инспектор для просмотра исходного кода текущей страницы. Проверьте, как меняется дерево знаний по мере добавления новых животных в игру.

CSS

В главе 9 мы затронули тему CSS-классов – они позволяют помечать элементы, чтобы к ним было легко обращаться при задании стилей. В основном CSS используется для статического определения свойств, а сделать его более динамичным можно с помощью программирования. В HTML-документе из листинга 10.2 мы видим два CSS-класса: динамический класс `start-hidden` и статический `invisible`.

Атрибут `class` используется для того, чтобы некоторые HTML-элементы из листинга 10.5 стали членами класса `start-hidden`. Это нужно не просто чтобы написать классный код – нам потребуется простой селектор для обращения к элементам. При запуске или перезапуске программы элементы с классом `start-hidden` становятся невидимыми, а их видимость задается в процессе игры, поэтому заданный класс позволяет легко управлять стилями для выбранных элементов.

Элемент с классом `invisible` невидим всегда, поскольку он хранит в себе дерево знаний. Теперь заменим строку 9 (`<!-- CSS-код -->`) в листинге 10.1 CSS-кодом из листинга 10.5.

Листинг 10.5. CSS-код для игры «Угадай животное»

```
1 invisible {  
2   display: none; /* элементы этого класса не отображаются */  
3 }
```

Помните, что в браузере всегда есть несколько способов делать одно и то же? Для простых CSS-правил можно использовать *встроенные стили*. Например, замените в листинге 10.2 строку 3 на код `<div id="root" style="display: none">`, и вы получите тот же результат.

«Угадай животное», версия 2: С

Как я уже говорил, браузеры представляют собой высокоуровневые виртуальные машины — вся их функциональность задается программно. Это позволяет писать программу быстро и легко, порой не обращая внимания на ее важные элементы. Перепишем программу с использованием языка С, чтобы увидеть простейшие действия, которые браузер от нас скрывает. Предполагается, что работа выполняется в операционной системе на основе UNIX.

Терминалы и командная строка

Создадим программу на С в виде ретрокода, без красочных графических элементов или кнопок. Подобно древней компьютерной игре *Adventure*, она будет работать лишь с командной строкой. Это поможет лучше изучить работу ввода и вывода, не полагаясь на изысканные интерфейсы, встроенные в браузер.

Что я имею в виду, говоря «ретро» и «командная строка»? В главе 1 мы обсуждали, что человеческий язык, вероятно, начался с отдельных звуков и жестов, но уж *точно* не с письменности. Компьютерные языки устроены иначе. Взаимодействие с пользователем началось с реакции на кнопки и переключатели на передней панели компьютеров, затем последовали письменные языки, а за ними — распознавание голосовых сигналов и жестов. Человек набирает что-то на компьютере, а компьютер «печатает в ответ» (см. раздел «Терминалы» на с. 226).

Скорее всего, для взаимодействия с компьютером вы используете *графический пользовательский интерфейс* (graphical user interface, *GUI*). Если задуматься, это сильно напоминает общение времен каменного века. «Ах ты! Смотри! Кнопка! Нажать! В друзья! Видео с котиками! Лайк! Твит! Твит! Твит!» Чаще всего графический интерфейс использует жестовый язык, что вполне удобно

обычным пользователям. В этом случае ему не приходится сильно полагаться на человеческую память — по крайней мере, так было до того, как пошла мода на интерфейсы, в которых абсолютно невозможно понять назначение иконок.

Многие компьютерные системы до сих пор поддерживают интерфейс командной строки, несмотря на изысканную современную графику. Сегодняшние терминалы перекочевали из внешних устройств в программное обеспечение. Открыв приложение терминала на компьютере, вы увидите окошко с *запросом на ввод команды* — введя что-нибудь, вы получите ответ.

Вместо нажатия на кнопки «да» и «нет» в версии программы на языке С необходимо вводить с клавиатуры буквы *у* или *н* соответственно. После ввода выбранной буквы пользователь должен нажать клавишу **Enter**, **Return** или ↵ в зависимости от клавиатуры. Похожим образом вводятся названия новых животных и связанные вопросы. Для выхода из программы будет использоваться ввод буквы *q*.

Создание программы

Язык С компилируемый, поэтому не получится просто взять и запустить исходный код на С, как мы это делали с интерпретированной версией на JavaScript. Первым делом нужно перевести его на машинный язык, а для этого пригодится простая командная строка. Если исходный код находится в файле с названием, скажем, *gta.c*, для получения файла на машинном языке с названием *gta* нужно ввести команду, показанную на рис. 10.5, в окно приложения терминала.

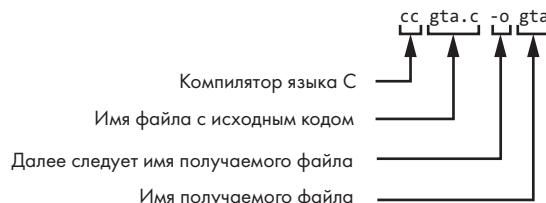


Рис. 10.5. Создание программы

Получив файл на машинном языке, вы запустите его, просто введя имя файла в командную строку.

Терминалы и драйверы устройств

Терминал — это устройство ввода/вывода, как уже упоминалось в разделе «Пространство системы и пользователя» на с. 178. Пользовательские программы не взаимодействуют с устройствами ввода/вывода напрямую — за них это делает операционная система (рис. 10.6).



Рис. 10.6. Операционная система — посредник между устройствами ввода/вывода и пользовательскими программами

Когда терминалы еще были отделены от компьютера, их приходилось подключать при помощи последовательного соединения RS-232 (см. «Последовательная связь» на с. 198). Таким образом, между терминалами и компьютерами тянулись самые обычные провода. Операционные системы копируют такие связи, воспроизводя их в виде ПО, поэтому устаревшие программы для терминалов до сих пор работают в современных системах в неизменном виде.

Переключение контекста

Драйвера устроены немного сложнее, чем кажется. Операционные системы были созданы именно потому, что потребовалось выполнять несколько задач одновременно. Компьютер имеет всего один набор регистров, и ОС нужно сохранять и восстанавливать их содержимое каждый раз, когда пользователь переключается между программами. Помимо регистров ЦП, регистров MMU и состояний всех устройств ввода/вывода компьютеру приходится сохранять и восстанавливать огромное количество дополнительной информации. Все это называется *контекстом процессов*, или просто *контекстом*. *Переключение контекста* — довольно затратная операция для компьютера, поэтому подходить к ней нужно с умом. Процесс системного вызова представлен на рис. 10.7.

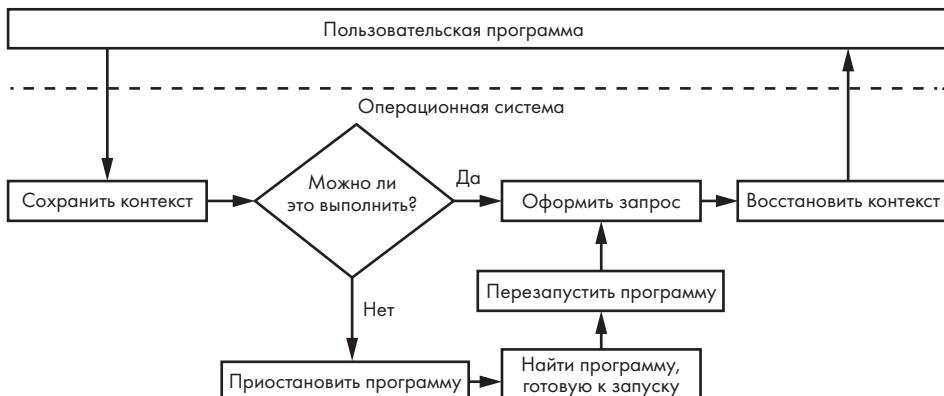


Рис. 10.7. Переключение контекста

Как видим, один системный вызов запускает множество скрытых процессов. Как я уже упоминал в разделе «Относительная адресация» на с. 173, ОС иногда *приостанавливает* пользовательскую программу, чтобы запустить другую программу, даже если запрос может быть выполнен.

Вовсе не нужно переключать контекст каждый раз, когда пользователь нажимает любую клавишу на клавиатуре. Обычно достаточно дождаться нажатия клавиши **ENTER** — она будет обозначать конец ввода. Пользовательская программа использует системный вызов, чтобы указать, что ей нужно *прочитать* команду из терминала. После этого программа приостанавливается или входит в режим ожидания, что позволяет ОС перейти к другой задаче. *Драйвер устройства*, обрабатывающий все запросы от физического устройства, сохраняет символы из терминала в *буфер* и *перезапускает* пользовательскую программу только после того, как пользователь нажмет **ENTER**, вместо того чтобы реагировать на нажатие любой клавиши.

Что представляет собой буфер? Пример буфера показан на рис. 6.25 — в программировании это структура данных, работающая по принципу «*первым пришел — первым обслужен*» (*first-in, first-out, FIFO*). На языке аппаратного обеспечения буфером называется электрическая схема, которая предотвращает перегрузку особо чувствительных компонентов. На рис. 10.8 изображена FIFO-структурата под названием *очередь*, похожая на очередь покупателей в супермаркете. Как и стек, очередь может быть переполнена, если в ней не хватает места, и исчерпана, если мы пытаемся обратиться к элементу в пустой очереди.



Рис. 10.8. Слово «кот» в очереди

Обычно терминалы работают в полнодуплексном режиме (см. «Последовательная связь» на с. 198), это означает, что клавиатура и экран не соединяются напрямую — вместо этого клавиатура посылает данные компьютеру, а экран получает от компьютера эти же данные. В далекие времена для соединения всех этих устройств использовались физические провода. Однако для драйвера устройства терминала недостаточно просто сохранить данные ввода в буфер — пользователь должен видеть, что его ввод с клавиатуры *повторяется* на экране. В дополнение к *буферу ввода* используется *буфер вывода*, потому что терминалы обычно работают медленнее программ, отправляющих им данные. Программа приостанавливается при попытке записать что-то в переполненный буфер вывода. Иногда для оповещения пользователя о том, что буфер ввода заполнен, драйвер использует звуковой сигнал или другой формат обратной связи. Часть драйвера устройства, которую мы здесь обсудили, изображена на рис. 10.9.

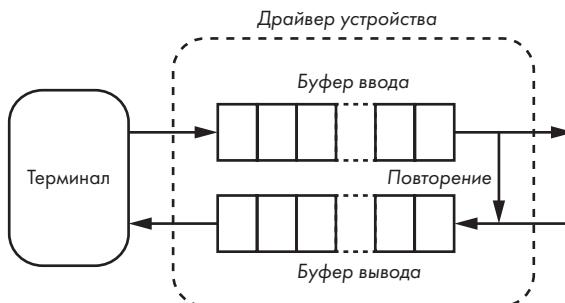


Рис. 10.9. Помещение данных в буфер и их повторение в драйвере устройства терминала

Реальные драйверы устройств не настолько просты. Для изменения настроек драйвера используются дополнительные системные вызовы, а повторение можно включить или выключить. Также можно выключить помещение в буфер — в этом случае драйвер будет работать в так называемом *необработанном* режиме (включенный буфер, соответственно, означает *обработанный* режим). Можно задать сочетания клавиш для запуска пользовательской программы, стирания введенных символов (обычно BACKSPACE или DELETE) и многих других задач.

Стандартный ввод/вывод

Буферизация данных в драйвере устройства лишь частично решает проблему. С этим же сталкиваются и пользовательские программы. Нет никакого смысла иметь драйвера устройства только для того, чтобы тот записывал входные данные и позволял пользовательским программам выполнять системные вызовы для каждого введенного символа. То же верно и для буфера вывода — его работу нельзя назвать эффективной, если пользовательская программа отправляет системные вызовы для записи каждого отдельного символа. Все это привело к созданию так называемой *стандартной библиотеки ввода/вывода* (`stdio`), которая содержит в себе буферизованные функции ввода/вывода для пользовательских программ.

Библиотека `stdio` поддерживает буферизованный ввод данных, при котором с драйвера устройства считывается как можно больше вводимых символов за один системный вызов, и все они помещаются в буфер. Пользовательская программа получает все данные из буфера, до тех пор пока он не станет пустым, а затем пробует получить оставшуюся часть данных. Символы помещаются в буфер вывода либо до тех пор, пока он не заполнится, либо до получения специального символа (например, символа новой строки). Весь процесс показан на рис. 10.10.

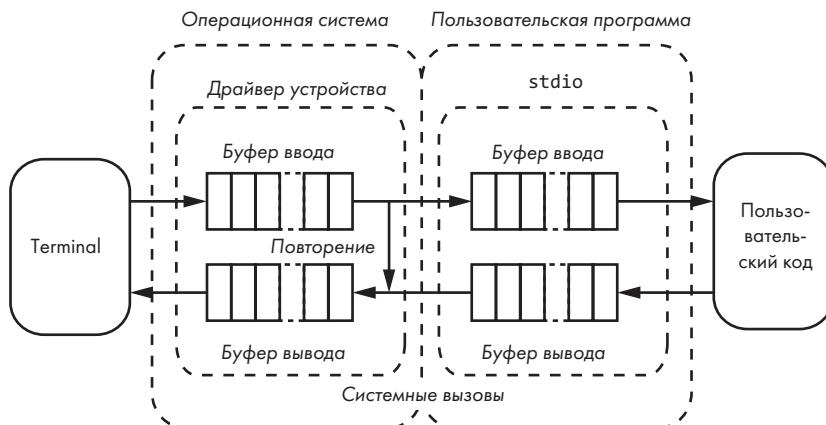


Рис. 10.10. Пользовательская программа с буферизацией через библиотеку stdio

Похоже, что для эффективной работы системы нужно хорошо потрудиться — и даже это еще не всё. Каким образом пользовательская программа подключается к драйверу устройства терминала?

Для указания на какой-либо объект достаточно называть его, вместо того чтобы давать его полное описание. Примерно такого же подхода придерживается операционная система, чтобы получить доступ к файлам. Системный вызов `open` превращает имя файла в *ссылку* или *дескриптор файла*, который может использоваться для обращения к файлу вплоть до его закрытия при помощи системного вызова `close`. Дескриптор файла похож на бирку, которую вы получаете при проверке ручной клади на стойке регистрации на рейс. В библиотеке `stdio` представлены соответствующие функции `fopen` и `fclose` — они используют системные вызовы и в дополнение к этому собирают и разбирают систему буферизации. Поскольку абстракции в UNIX работают с устройствами так же, как с файлами, для доступа к устройству терминала понадобится открыть специальный файл — `/dev/tty`.

Кольцевые буфера

Как я уже упоминал, очереди в программировании похожи на очереди в супермаркете. Но, несмотря на внешнее сходство, буфера вроде буфера вывода в `stdio` на рис. 10.10 реализованы несколько иначе.

Представьте себе очередь на кассу. Как только кассир заканчивает обслуживать первого покупателя, вся остальная очередь сдвигается вперед на одну позицию. Рассмотрим это на примере очереди букв из слова *рыба*, как показано на рис. 10.11. Согласно рисунку, нужно все время следить за концом очереди, чтобы добавлять в нее новые буквы.

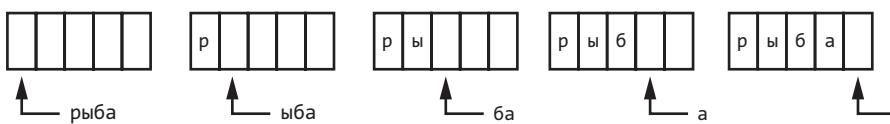


Рис. 10.11. Добавление букв в очередь

Теперь перейдем к последовательному удалению букв из очереди (рис. 10.12).

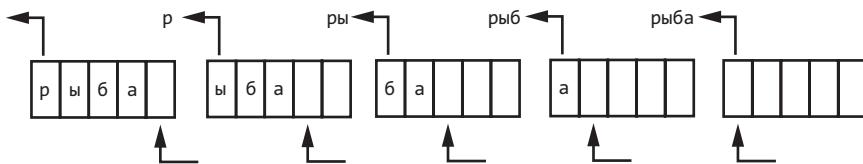


Рис. 10.12. Удаление букв из очереди

Как видите, это не так просто, как может показаться на первый взгляд. При удалении буквы *р* буква *ы* должна сдвинуться на место буквы *р*, буква *б* — на место буквы *ы* и т. д. Попробуем другой подход: вместо того чтобы сдвигать все буквы, будем перемещать позицию счетчика, как показано на рис. 10.13.

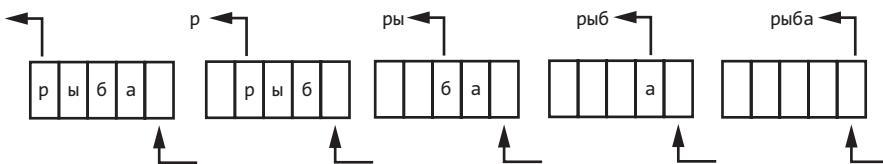


Рис. 10.13. Удаление букв из очереди и перемещение счетчика

Благодаря такой оптимизации никому не придется делать слишком много работы (кроме счетчика, конечно). Но теперь возникает новая проблема — рано или поздно счетчик упрется в дверь даже при наличии свободного места, и ни один человек не сможет попасть в очередь.

Все, что нам нужно, — это каким-то образом направлять новых людей в свободные места в начале очереди. Для этого согнем очередь в кольцо, как показано на рис. 10.14.

Как видите, данные можно добавлять в очередь, если стрелка *входа* направлена по часовой стрелке от стрелки *выхода*. Для удаления данных из очереди, соответственно, стрелка выхода должна быть направлена против часовой стрелки от

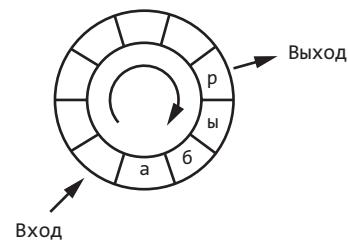


Рис. 10.14. Кольцевой буфер

стрелки входа. Чтобы перейти от конца буфера к началу, понадобится немного посчитать. Следующая позиция — это остаток от деления текущей позиции плюс 1 на размер буфера.

Такие структуры данных называются по-разному: *кольцевые буфера*, *кольцевые очереди* или *циклические буфера*. Они широко используются в программировании, не только в библиотеке `stdio` или драйверах устройств.

Больше абстракций — лучше код

Играя в игру «Угадай животное», мы всегда начинаем сначала — с момента, когда программа знает о существовании лишь кошек и собак. Было бы неплохо, если бы все сыгранные партии запоминались и не приходилось каждый раз обновлять базу знаний. Благодаря побочным эффектам абстракции файлов в языке С можно легко усовершенствовать программу.

На рис. 10.15 показано, почему это не так-то просто сделать в версии программы на языке JavaScript.

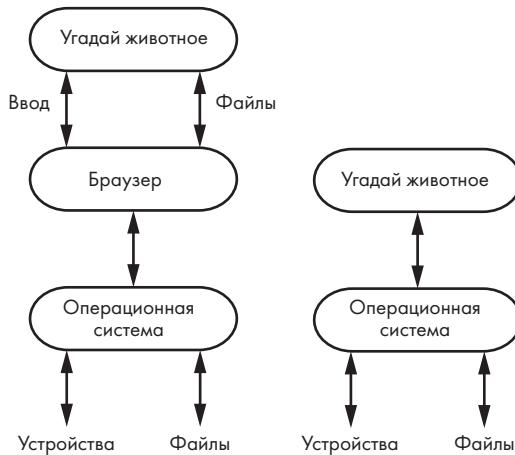


Рис. 10.15. Интерфейсы браузера и операционной системы

Как видите, в ОС для устройств и файлов используется один и тот же интерфейс. Он же применяется в браузере (слева на рисунке) и в версии программы на языке С (справа на рисунке). Это означает, что программа на С может читать как ввод из файла, так и пользовательский ввод с устройства. Браузер же не может похвастаться таким уровнем абстракции при передаче данных JavaScript-программисту — вместо этого ему потребуется абсолютно новый интерфейс, который мог бы решить данную задачу. Выбор интерфейса влияет как на простоту разработки, так и на понятность результата.

Важные мелочи

Вернемся к программе на языке С. Чтобы подготовить ее к запуску, необходимо скомпилировать ее и *связать* с кодом, с которым она работает — в нашем случае с библиотекой `stdio`. В разделе «Запуск программ» на с. 183 я упоминал, что нужно учитывать и библиотеку времени выполнения — в С она обычно называется `crt0`. Она отвечает за предварительную настройку стека и кучи, а также за открытие двух файлов, связанных с драйвером устройства терминала: одного — для ввода, второго — для вывода.

Библиотека `stdio` соединяет дескрипторы системных файлов с *указателями на файлы* — адресами структур данных, которые используются для буферизации и регистрации данных. Для начала понадобится три указателя: `stdin` (стандартный ввод), `stdout` (стандартный вывод) и `stderr` (стандартный вывод ошибок). Последний указатель нужен для сообщений, которые стоит выводить не через `stdout` — они оба указывают на одно и то же местоположение, но `stderr`, в отличие от `stdout`, не буферизуется. Отправленные в `stdout` сообщения об ошибках будут помещены в буфер, из-за чего вы можете даже не увидеть их, если в программе что-то пойдет не так. Если не задано иное, указатели на файлы `stdout` и `stderr` используют один и тот же дескриптор файла, как показано на рис. 10.16.

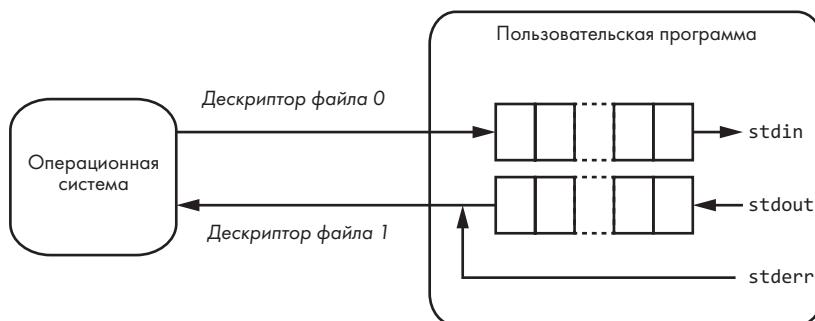


Рис. 10.16. Указатели на файлы `stdin`, `stdout` и `stderr`

Изобретениям часто предшествуют странные события. Стив Джонсон говорил, что указатель `stderr` изначально не был включен в библиотеку `stdio` — он был добавлен позже как побочный эффект разработки первого компьютерного типографского ПО (языка `troff`, созданного Джозефом Оссанной (Joseph Ossanna), 1928–1977) для фотонаборного автомата С/А/Т. Сегодня мы привыкли к лазерной и струйной печати, а в те времена приходилось проецировать изображения на серебряную фотобумагу, а затем еще и проявлять их. Технология значительно возросла в цене, когда братья Хант захватили рынок серебра, — поэтому от этого метода в итоге пришлось отказаться. Нередко вместо готового изображения из фотонаборного автомата получали прекрасно отформатированный лист

с надписью «невозможно открыть файл». Тогда для экономии денег и был создан указатель `stderr`, чтобы сообщение об ошибке выводилось в окне терминала, а не на бумаге.

Переполнение буфера

Раз уж мы затронули тему `stdio`, поговорим о классе весьма серьезных ошибок в программировании — *переполнении буфера*. Изначально при разработке библиотеки `stdio` в нее была добавлена функция `gets`, которая считывала строку до символа новой строки из `stdin` в предоставленный пользователем буфер. Можно использовать этот подход, как показано в листинге 10.6, чтобы прочитать ответ `u`, `n` или `q`. В переменной `buffer` достаточно места для одного символа и NUL-ограничителя.

Листинг 10.6. Применение функции `gets` для считывания ввода

```
1 char buffer[2];
2
3 gets(buffer);
```

Что не так с этим фрагментом кода? Функция `gets` не проверяет возможность того, что пользователь вводит слишком много символов — столько, сколько не помещается в буфер. Представьте, что мы написали более сложную программу, в которой есть еще и переменная `launch_missiles`, которую также нужно поместить в память (рис. 10.17).



Рис. 10.17. Переполнение буфера в памяти

Пользователь- злоумышленник может смекнуть, что ответ `ууу` помещает `у` в переменную `launch_missiles`, которая так или иначе представляет собой то же самое, что и несуществующая переменная `buffer[2]`. Может произойти непоправимое — и оно обязательно произойдет. Фактически именно из-за подобного недосмотра были обнаружены многие дыры в безопасности существующих систем. Для исправления ошибки в новую версию библиотеки `stdio` была добавлена функция `fgets`, проверяющая границы буфера. Однако и это не панацея — вызвать переполнение буфера можно многими другими способами, так что впредь будьте предельно внимательны. *Ни в коем случае не полагайтесь на то, что в буфере достаточно места для данных!* В главе 13 мы еще вернемся к переполнению буфера и рассмотрим некоторые детали.

Программа на языке С

В языке С существует огромное количество библиотек помимо `stdio`. К примеру, библиотека `string` включает в себя функции для сравнения и копирования строк, а стандартная библиотека-сборник `stdlib` предоставляет функции для управления памятью.

В листинге 10.7 представлена вводная часть игры — фрагмент программы на языке С. Вначале собираются все нужные библиотеки (строки 1–3). Затем объявляется структура узлов (строки 5–9), в которой присутствуют два указателя на узлы-листья и заглушку для строки с новым вопросом или названием животного. В версии программы на языке JavaScript ничего подобного не было благодаря преимуществам существующих элементов `<div>` в HTML. В противном случае код на С был бы аналогичен коду на JavaScript. Обратите внимание: структура узлов объявлена таким образом, что память и для узла, и для строки может быть выделена одновременно (см. раздел «Более эффективное выделение памяти» на с. 248).

Листинг 10.7. Введение в игру «Угадай животное» на языке С

```
1 #include <stdio.h> // стандартная библиотека ввода/вывода
2 #include <stdlib.h> // стандартная библиотека с функциями exit и malloc
3 #include <string.h> // библиотека для работы со строками
4
5 struct node {
6     struct node *no; // ссылается на узел с ответом "нет"
7     struct node *yes; // ссылается на узел с ответом "да"
8     char string[1]; // вопрос или животное
9 };
```

Затем нам понадобится функция-помощник для выделения памяти (листинг 10.8). В выделении памяти нет ничего сложного, но мы применим одинаковый подход в разных местах программы, поэтому заранее избавимся от копирования одних и тех же проверок на ошибки. Во многих новых языках добавлены конструкции для обработки исключений, что упрощает задачу.

Выделение памяти понадобится только при создании нового узла (`node`), поэтому новая функция будет принимать строку (`string`) и помещать ее в `node`. Помимо выделения памяти скопируем строку в переменную `node` и инициализируем указатели для ответов «да» (`yes`) и «нет» (`no`).

Листинг 10.8. «Угадай животное» на языке С: функция для выделения памяти

```
10 struct node      *
11 make_node(char *string)
12 {
13     struct node      *memory;      // текущая выделенная память
14
15     if ((memory = (struct node *)malloc(sizeof (struct node) +
16                                         strlen(string))) == (struct node *)0) {
17         (void)fprintf(stderr, "gta: out of memory.\n");
```

```

17         exit(-1);
18     }
19
20     (void)strcpy(memory->string, string);
21     memory->yes = memory->no = (struct node *)0;
22
23     return (memory);
24 }

```

Для вывода сообщения об ошибке используем функцию `fprintf` из библиотеки `stdio`. Ранее мы обсуждали, что данные в `stderr` не буферизуются, поэтому мы увидим сообщение об ошибке сразу, как только что-то пойдет не так.

Обратите внимание, что в строке 16 используется оператор приведения для приведения результата выполнения функции `fprintf` к типу `void`. Если `fprintf` вернет результат, который мы не рассматриваем, приведение укажет компилятору на то, что мы делаем это намеренно, а не просто пропускаем нечто важное. В этом случае мы избавимся от лишних предупреждающих сообщений. То же мы сообщаем и тем, кто будет читать этот код, — результаты функции `fprintf` игнорируются намеренно, а не по ошибке. В последние версии компиляторов были добавлены изменения, чтобы избавиться от лишних предупреждающих сообщений — но при необходимости их можно включить в настройках.

Вызов функции `exit` в строке 17 завершает выполнение программы. Это единственное, что стоит сделать, если программе для дальнейшей работы не хватает выделенной памяти.

Функция `printf` (сокращение от `print formatted` — *форматированный вывод*) является частью библиотеки `stdio` и имеет множество эквивалентов в других языках программирования. Первый ее аргумент — это *строка формата*, которая определяет, как будут интерпретироваться следующие аргументы. Символ «%», за которым следует код, говорит: «Замените меня на следующий аргумент в соответствии с кодом». В нашем примере «%s» означает «Обработать следующий аргумент как строку».

Оставшаяся часть программы представлена в листинге 10.9.

Листинг 10.9. «Угадай животное» на языке C: основная часть программы

```

25 int
26 main(int argc, char *argv[])
27 {
28     char           animal[50]; // буфер с названием нового животного
29     char           buffer[3]; // буфер с пользовательским вводом
30     int            c;        // текущий символ из буфера
31     struct node   **current; // текущий узел при обходе дерева
32     FILE           *in;      // входной файл учебных данных или ввода
33     struct node   *new;     // созданный узел
34     FILE           *out;     // выходной файл для сохранения учебных данных
35     char           *p;       // указатель для удаления символа новой строки

```

```
36     char          question[100]; // буфер с новым вопросом
37     struct node    *root;        // корневой узел дерева знаний
38
39 // Обработка аргументов
40
41     in = out = (FILE *)0;
42
43     for (argc--, argv++; argc > 1 && argc % 2 == 0; argc -= 2, argv += 2) {
44         if (strcmp(argv[0], "-i") == 0 && in == (FILE *)0) {
45             if ((in = fopen(argv[1], "r")) == (FILE *)0) {
46                 (void)fprintf(stderr, "gta: can't open input file `%s'.\n",
47                               argv[1]);
48                 exit(-1);
49             }
50         }
51         else if (strcmp(argv[0], "-o") == 0 && out == (FILE *)0) {
52             if ((out = fopen(argv[1], "w")) == (FILE *)0) {
53                 (void)fprintf(stderr, "gta: can't open output file `%s'.\n",
54                               argv[1]);
55                 exit(-1);
56             }
57         }
58         else
59             break;
60     }
61
62     if (argc > 0) {
63         (void)fprintf(stderr, "usage: gta [-i input-file-name]
64                                     [-o output-file-name]\n");
65         exit(-1);
66     }
67 // Чтение стандартного ввода, если не указан входной файл
68
69     if (in == (FILE *)0)
70         in = stdin;
71
72 // Создание дерева знаний
73
74     root = make_node("Оно лает");
75     root->yes = make_node("собака");
76     root->no = make_node("кошка");
77
78     for (;;) { // прохождение игры, пока пользователь не выйдет из нее
79
80         if (in == stdin)
81             (void)printf("Загадай животное.\n");
82
83         current = &root; // переход к началу
84
85         for (;;) { // прохождение игры
```

```
86
87     for (;;) {      // получение допустимого пользовательского ввода
88         if (in == stdin) {
89             if ((*current)->yes == (struct node *)0)
90                 (void)printf("Это ");
91
92             (void)printf("%s?[ynq] ", (*current)->string);
93         }
94
95         if (fgets(buffer, sizeof (buffer), in) == (char *)0 ||
96             strcmp(buffer, "q\n") == 0) {
97             if (in != stdin) {
98                 (void)fclose(in);
99                 in = stdin;
100            }
101            else {
102                if (in == stdin)
103                    (void)printf("\nСпасибо за игру. Пока.\n");
104                exit(0);
105            }
106        else if (strcmp(buffer, "y\n") == 0) {
107            if (out != (FILE *)0)
108                fputs("y\n", out);
109
110            current = &((*current)->yes);
111
112            if ((*current == (struct node *)0) {
113                (void)printf("Я угадал!\n");
114                break;
115            }
116        }
117        else if (strcmp(buffer, "n\n") == 0) {
118            if (out != (FILE *)0)
119                fputs("n\n", out);
120
121            if ((*current)->no == (struct node *)0) {
122                if (in == stdin)
123                    (void)printf("Сдаюсь. Что это за животное?");
124
125            fgets(animal, sizeof (animal), in);
126
127            if (out != (FILE *)0)
128                fputs(animal, out);
129
130            if ((p = strchr(animal, '\n')) != (char *)0)
131                *p = '\0';
132
133            if (in == stdin)
134                (void)printf(
135                    "Какой вопрос я могу задать, чтобы отличить
136                    %s от %s?",
137                    animal, (*current)->string);
138            fgets(question, sizeof (question), in);
```

```
138
139         if (out != (FILE *)0)
140             fputs(question, out);
141
142         if ((p = strchr(question, '\n')) != (char *)0)
143             *p = '\0';
144
145         new = make_node(question);
146         new->yes = make_node(animal);
147         new->no = *current;
148         *current = new;
149
150         if (in == stdin)
151             (void)printf("Спасибо. Запомню это.\n");
152
153         break;
154     }
155
156     else
157         current = &(*current)->no;
158 }
159 else {
160     if (in == stdin)
161         (void)printf("Нажмите у для ответа \"да\", н для
162                     ответа \"нет\" или q для выхода.\n");
163
164     while ((c = getc(in)) != '\n' && c != EOF)
165         ;
166     }
167
168     break;
169 }
170
171 if (in == stdin)
172     (void)printf("Давай сыграем заново.\n\n");
173 }
174 }
```

Помимо управления памятью, в этом коде нет ничего интересного. В основном эта программа на языке С делает то же самое, что и ее версия на JavaScript. В строках 28–37 объявляются переменные. В строках 74–76 создаются начальные узлы, как показано на рис. 10.18. Обратите внимание, что все строки завершаются NUL-ограничителем ('\0').

Сыграем в игру, как мы уже делали в разделе «“Угадай животное”, версия 1: HTML и JavaScript» на с. 323. Нас интересуют некоторые детали реализации в момент определения игроком нового вопроса и создания нового узла для него. Используйте функцию `strlen` (сокращение от `string length` — *длина строки*) с осторожностью. Она возвращает действительную длину строки, а не количество занятой этой строкой памяти, — на 1 байт больше, учитывая NUL-ограничитель. Заметьте, что при выделении памяти для строки мы не

добавляем еще 1 байт, потому что в нашем примере при выделении памяти для нового узла дополнительный байт памяти учитывается неявно.

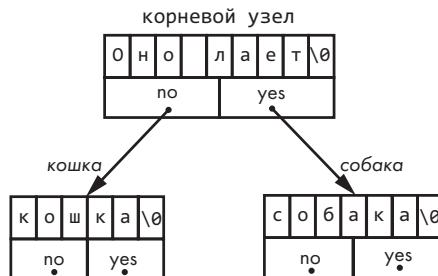


Рис. 10.18. «Угадай животное» на языке С: начальные узлы дерева

Пробираясь по дереву знаний в соответствии с ответами «да» или «нет», для упрощения добавления новых узлов мы используем указатель `current`. Он помогает программе различать ответы "да" и "нет", ссылаясь на указатель заменяемого узла. Поскольку `current` ссылается на указатель узла, он представляет собой указатель на указатель. Используя код `*current = new;`, мы переопределяем указатель и говорим программе: «Замени то, на что ссылается этот указатель». Согласно рис. 10.19, указатель `no` в узле `new` изначально имеет значение `current` (старый ответ), а `current` указывает на указатель `yes` в корневом узле, который заменяется на указатель на узел `new`.

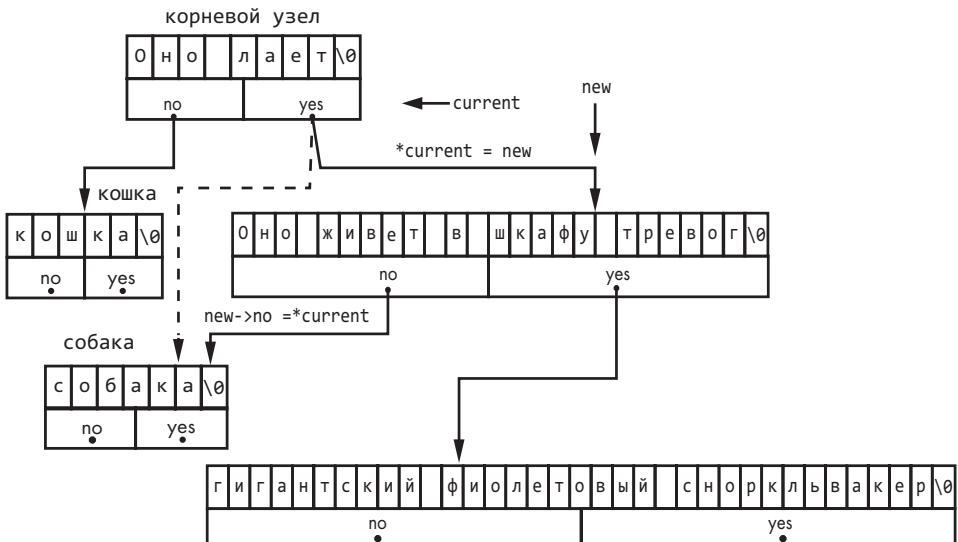


Рис. 10.19. «Угадай животное» на языке С: добавление новых узлов

Тренировка

Напомним, что наша программа на языке С может быть запущена с опциями командной строки для чтения и записи обучающих данных. Мы можем запустить программу следующим образом:

```
prompt> gta -o training
Загадай животное.
Оно лает?
n
Это собака?
n
Сдаюсь. Что это за животное?
гигантский фиолетовый сноркльвакер
Какой вопрос я могу задать, чтобы отличить гигантский фиолетовый сноркльвакер от
собаки?
Оно живет в шкафу тревог?
Спасибо. Запомню это.
Давай сыграем заново.
Загадай животное.
Оно лает?
q
Спасибо за игру. Пока.
```

Открыв файл `training`, вы увидите, что в нем сохранены все данные, которые вы только что вводили:

```
n
n
гигантский фиолетовый сноркльвакер
Оно живет в шкафу тревог?
```

Если перезапустить программу с помощью команды:

```
prompt> gta -i training
```

она прочитает данные из файла `training`, и нам не придется проходить обучение заново.

Давным-давно в разделе «Что такое программирование компьютеров?» на с. 29 я говорил, что нужно знать многое, чтобы стать хорошим программистом. Наша программа не лучший образец грамматически правильного программирования. Например, в вопросе «Какой вопрос я могу задать, чтобы отличить гигантский фиолетовый сноркльвакер от собаки?» отсутствует согласование в падежах. Как это исправить? Можно ли изменить код, чтобы программа задавала грамматически правильные вопросы?

Выводы

В этой главе мы написали две версии одной программы: одна представляет собой высокоуровневый прикладной код, а вторая — низкоуровневый системный. С одной стороны, высокоуровневые прикладные программы писать проще, потому что многие процессы в них скрыты от разработчика. С другой — некоторые программные возможности, такие как запись и воспроизведение, намного труднее реализовать в среде без унифицированных интерфейсов.

Более того, запуск простых приложений в сложной прикладной среде чреват ошибками. Вероятность ошибок представляет собой сумму кода приложения и кода среды, в которой оно запускается. Сколько раз вам приходилось перезапускать подвисающий браузер из-за внутренних ошибок управления памятью? А как часто браузер вообще отказывался работать?

Вы наверняка заметили, что системное программирование требует большего внимания к деталям — к управлению строками, памятью и буферами. Однако это важно, если вы действительно хотите создавать лаконичный и безопасный код. В следующей главе мы перейдем к новым подробностям: структурированию задач таким образом, чтобы их было проще решить.

11

Сокращения и приближения



Мы уже немало узнали о том, как проводить вычисления эффективно и не в ущерб памяти. Однако лучший способ вычислить что-либо — не вычислять вообще. В этой главе мы рассмотрим два способа избежать вычислений: сокращения и приближения.

Мы воспринимаем компьютеры как некие строгие и точные устройства, но, если вспомнить раздел «Представление действительных чисел» на с. 54, они таковыми не являются. Например, команда `bc` в UNIX представляет собой калькулятор с произвольной точностью — он отлично подходит для очень точных вычислений, но на самом деле не очень эффективен, потому что аппаратное обеспечение компьютеров произвольную точность не поддерживает. Возникает вопрос: насколько важна точность в том или ином случае? Эффективно использовать ресурсы компьютера — значит не обременять его лишней работой. Согласитесь, что вычислять все цифры после запятой в числе π каждый раз, когда оно используется, — не лучшая идея.

Поиск по таблице

Часто найти значение в таблице оказывается быстрее, чем вычислить его заново. Ниже мы рассмотрим несколько примеров. Поиск по таблице похож на инвариантную оптимизацию цикла, которую мы обсуждали в главе 8, — если какое-то значение понадобится программе несколько раз, его лучше вычислить заранее.

Преобразование

Предположим, что нам нужно получить значение температуры от теплового датчика и представить его в десятых долях градусов Цельсия ($^{\circ}\text{C}$). Умный проектировщик аппаратного обеспечения собрал схему, которая выдает значение напряжения в зависимости от измеренной температуры — результат можно прочитать с помощью аналого-цифрового преобразователя (АЦП) (см. «Аналого-цифровое преобразование» на с. 210). График зависимости представлен на рис. 11.1.



Рис. 11.1. График значений теплового датчика

Видим, что на рисунке кривая, а вовсе не удобная прямая линия. Можно вычислить значение температуры (t) с учетом напряжения (v) по следующей формуле, где A , B и C — это константы, которые зависят от конкретной модели датчика:

$$t = \frac{1}{A + B \times \log_e v + (C \times \log_e v)^2}.$$

Как видите, придется выполнить немало действий с вещественными числами, включая натуральные логарифмы, — для этого понадобятся определенные затраты. Так что давайте просто откажемся от вычислений. Вместо них составим таблицу, сопоставив значения температуры и напряжения. Предположим, что у нас есть 10-битный АЦП, а для хранения значения температуры нам хватит и 8 бит. Таким образом, чтобы избавиться от затратных вычислений, нам понадобится таблица размером 1024 байта, как показано на рис. 11.2.

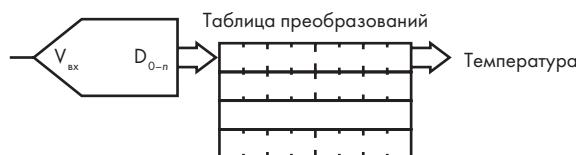


Рис. 11.2. Преобразование с помощью поиска по таблице

Отображение текстур

Поиск по таблице является основой *отображения текстур* — техники, которая позволяет создавать реалистичные изображения в видеоиграх и фильмах. Идея заключается в том, что вычислительной машине проще поместить изображение на объект вроде стены, чем генерировать все нужные детали с помощью алгоритмов. Представим, что у нас есть текстура кирпичной стены, как на рис. 11.3.



Рис. 11.3. Текстура кирпичной стены

Выглядит неплохо, но стены в видеоиграх не статичны. Представьте, что вы пытаетесь спастись от зомби и убегаете все дальше и дальше от стены. Текстура должна меняться в зависимости от того, как далеко вы от нее находитесь. На рис. 11.4 показано, как будет выглядеть одна и та же стена, если смотреть на нее, находясь вдалеке (слева) и вблизи (справа).

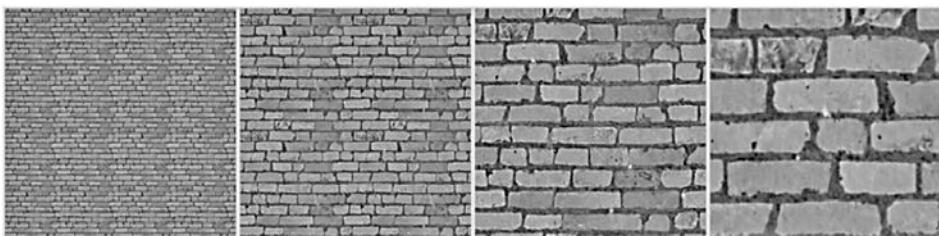


Рис. 11.4. Текстура кирпичной стены на разном удалении от нее

Наверняка вы уже догадались, что изменение текстуры в зависимости от расстояния — непростая задача. Чем дальше игрок отходит от текстуры, тем больше соседних пикселей сливаются друг с другом. Соответствующие вычисления нужно выполнять быстро, чтобы изображение не дергалось.

Ланс Уильямс (Lance Williams) (1949–2017), работавший в лаборатории графических языков Нью-Йоркского технологического института, придумал так

называемое *MIP-текстурирование* (MIP mapping, от лат. *multum in parvo* – «многое в малом»). Его работа на эту тему, «Pyramidal Parametrics», была опубликована в материалах конференции SIGGRAPH (июнь 1983 г.). Данный метод до сих пор используется не только в программном, но и в аппаратном обеспечении.

Как мы видели в разделе «Представление цветов» на с. 69, один пиксель можно разделить на три 8-битных компонента для красного, зеленого и синего цветов. Уильямс заметил, что при представлении этих компонентов в виде квадратов в 32-битных системах четверть пространства остается незанятой, как показано на рис. 11.5.

Оставлять пустое пространство слишком расточительно, поэтому он придумал хитроумный способ его занять – отличный от того, что предложили Том Дафф и Томас Портер (см. «Добавление прозрачности» на с. 72). Уильямс заметил, что пустое пространство также представляет собой квадрат, куда можно поместить копию исходного изображения, уменьшенную в 4 раза, а в *ее* пустое пространство – еще одну уменьшенную копию и т. д., как показано на рис. 11.6. Данный способ организации получил название «MIP-текстурирование».

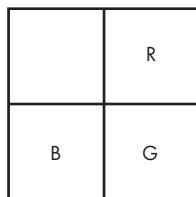


Рис. 11.5. Незанятое пространство при организации компонентов одного цвета

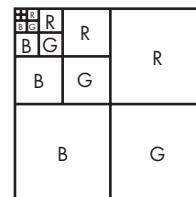


Рис. 11.6. Схема с множеством копий одного и того же изображения

Применив MIP-текстурирование к изображению кирпичной стены, получим рис. 11.7 (вместо оттенков серого представьте компоненты цвета).



Рис. 11.7. MIP-текстурирование для кирпичной стены

Изображения крупного плана позволяют видеть больше деталей — запомните эту мысль. Умный механизм хранения копий интересен сам по себе, но какая от него практическая польза? Взгляните на рис. 11.8, где MIP-изображение одного из цветов разворачивается в виде пирамиды.

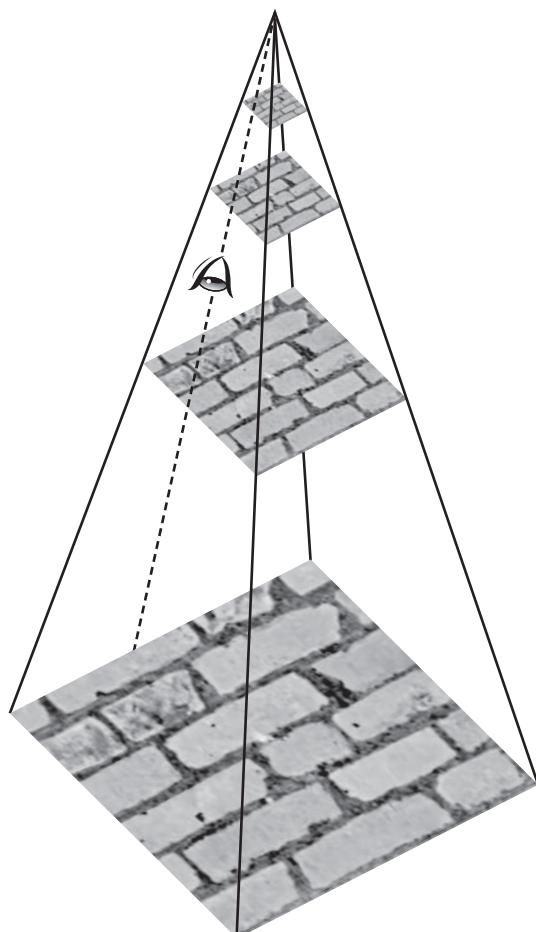


Рис. 11.8. Пирамида MIP-текстурирования

Изображение в верхушке пирамиды — это то, как мы будем видеть текстуру, находясь очень далеко от нее. Чем ниже мы спускаемся к основанию пирамиды, тем более детальным становится изображение. Чтобы получить нужную текстуру на заданной дистанции на рис. 11.8, нам не придется объединять все пиксели исходного изображения — достаточно использовать пиксели ближайшего слоя пирамиды. Так мы экономим много времени для вычислений, особенно если точка обзора находится далеко.

Предварительное вычисление информации, которая будет повторно использоваться, — в нашем случае текстуры с более низким разрешением — эквивалентно инвариантной оптимизации цикла.

Классификация символов

Методы поиска по таблице привели к добавлению новых библиотек в язык программирования С. Из главы 8 мы знаем, что *классификация символов* (определение, является ли символ буквой, цифрой и т. п.) играет важную роль в лексическом анализе. Вернувшись к таблице кодов ASCII (см. табл. 1.11), вы с легкостью напишете фрагмент кода для выполнения классификации — пример показан в листинге 11.1.

Листинг 11.1. Код для выполнения классификации символов

```
int
isdigit(int c)
{
    return (c >= '0' && c <= '9');
}

int
ishexdigit(int c)
{
    return (c >= '0' && c <= '9' || c >= 'A' && c <= 'F' || c >= 'a' && c <= 'f');
}

int
isalpha(int c)
{
    return (c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z');
}

int
isupper(int c)
{
    return (c >= 'A' && c <= 'Z');
}
```

Некоторые программисты из Bell Labs утверждают, что часто используемые функции, такие как код из листинга 11.1, нужно выносить в *библиотеки*. Деннис Ричи (Dennis Ritchie) (1941–2011) не был с этим согласен — он считал, что легко написать собственные версии библиотек. Однако Нильс-Питер Нельсон (Nils-Peter Nelson) из этого же компьютерного центра создал вариант программы, которая использовала таблицу вместо множества выражений *if*. Таблица индексировалась по значению символов, и каждый ее элемент хранил биты для символов в верхнем и нижнем регистрах, цифр и т. д., как показано на рис. 11.9.

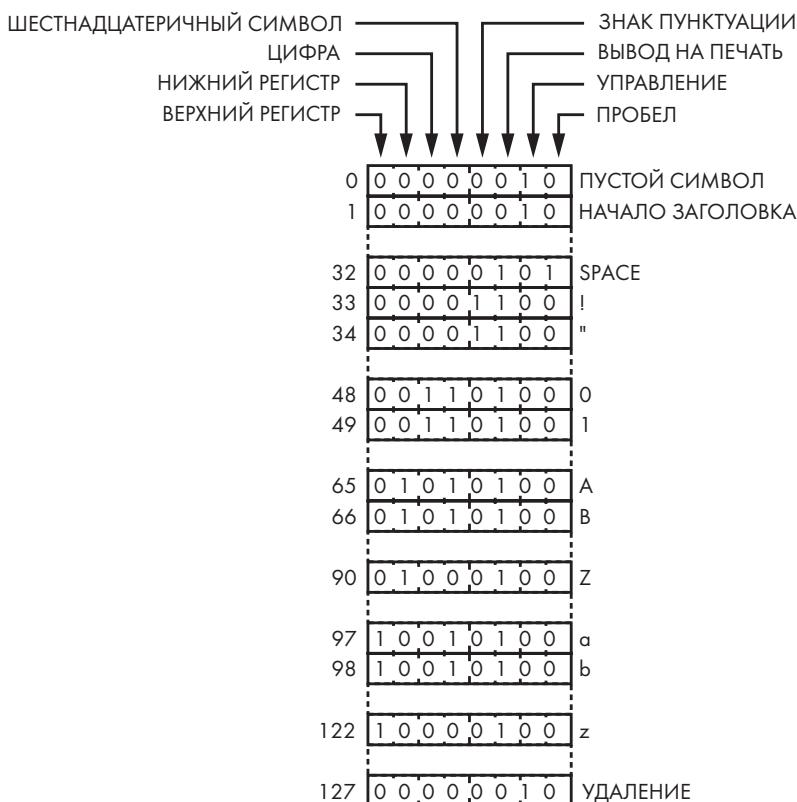


Рис. 11.9. Таблица для классификации символов

В данном случае классификация предусматривает поиск значения в таблице и проверку битов, как показано в листинге 11.2.

Листинг 11.2. Код для классификации символов с помощью таблицы

```
unsigned char table[128] = [ ... ];

#define isdigit(c)    (table[(c) & 0x7f] & DIGIT)
#define ishexdigit(c) (table[(c) & 0x7f] & HEXADECIMAL)
#define isalpha(c)    (table[(c) & 0x7f] & (UPPER | LOWER))
#define isupper(c)    (table[(c) & 0x7f] & UPPER)
```

Видим, что функции в листинге 11.2 выглядят проще, чем в листинге 11.1. Более того, результаты их выполнения одинаковы — различаются только значения констант, добавленных к содержимому таблицы. Такой подход оказался в 20 раз быстрее, чем остальные предложенные решения, поэтому Ричи создал библиотеку с этими функциями. Позже на ее основе были добавлены новые библиотеки.

ПРИМЕЧАНИЕ

Вы могли заметить, что в листинге 11.2 я использую макросы, а в листинге 11.1 — функции. На случай, если вы не знакомы с макросами — это конструкции языка программирования, которые заменяют код справа на код слева. Например, если в коде встречается выражение `isupper('a')`, препроцессор заменит его на `table[('a') & 0x7f] & UPPER`. Это отлично подходит для небольших фрагментов кода, потому что так мы избавляемся от оверхедов, связанных с вызовами функций. Однако код в листинге 11.1 заменить на макросы не получится — нужно учитывать вариант, когда пользователь может вызвать функцию `isupper(*p++)`. При использовании макросов в листинге 11.1 в функции `ishexdigit`, к примеру, значение `p` неожиданно для пользователя увеличилось бы на 1 шесть раз. Версия кода в листинге 11.2 ссылается на аргумент только один раз, поэтому такого не произойдет.

Целочисленные методы

Уже зная о том, как работает аппаратное обеспечение, вы могли предположить, что на одни вычисления требуется больше времени и энергии, чем на другие. К примеру, сложение и вычитание целых чисел компьютер выполнит без особого труда. Умножение и деление потребует больше ресурсов, но умножение и деление на 2 можно упростить, заменив их операциями сдвига. Операции с числами с плавающей точкой еще более затратны, особенно сложные вычисления — например, тригонометрических и логарифмических функций. Чтобы не отклоняться от темы главы, поговорим о способах оптимизации сложных вычислений.

Рассмотрим пару примеров. В листинге 11.3 представлен измененный вариант веб-страницы из листинга 10.1 — мы изменили содержимое тегов `style`, `script` и `body`.

Листинг 11.3. Простой пример холста

```
1 <style>
2   canvas {
3     border: 5px solid black;
4   }
5 </style>
6 ...
7 <script>
8   $(function() {
9     var canvas = $('canvas')[0].getContext('2d');
10
11    // Получаем ширину и высоту холста. Переводим их в целочисленные значения,
12    // поскольку метод attr возвращает строки. Использование строк в качестве
13    // целых чисел в JavaScript часто приводит к неожиданным результатам
14
15    var height = Number($('canvas').attr('height'));
16    var width = Number($('canvas').attr('width'));
```

```
17
18     canvas.translate(0, height);
19     canvas.scale(1, -1);
20   });
21 </script>
22 ...
23 <body>
24   <canvas width="500" height="500"></canvas>
25 </body>
```

Я кратко упоминал холст в разделе «HTML5» на с. 317. *Холст* — это элемент, на котором можно рисовать произвольные фигуры (наподобие листа миллиметровой бумаги).

Холст все же отличается от миллиметровки, потому что по умолчанию он не задействует стандартную декартову систему координат. Вместо нее используется нечто подобное растрю, как на телевидении (см. «Растровая графика» на с. 230) — растр начинается в левом верхнем углу. Координаты оси *X* представлены как обычно, тогда как координаты оси *Y* начинаются вверху, и их значения уменьшаются при движении вниз по оси. Подобная система координат не изменилась, даже когда телевизионные экраны стали использовать компьютерную графику.

Современные системы компьютерной графики работают и с произвольными системами координат — им часто требуется поддержка специального графического оборудования. К каждой указанной точке с координатами (x, y) применяется так называемая *трансформация* — соотношение заданных координат с координатами экрана (x', y') при помощи следующих формул:

$$\begin{aligned}x' &= Ax + By + C; \\y' &= Dx + Ey + F.\end{aligned}$$

Параметры *C* и *F* отвечают за *перевод* — перемещение точки на экране. Параметры *A* и *E* отвечают за *масштабирование* — увеличение или уменьшение размеров точки. Параметры *B* и *D* отвечают за *вращение* — изменение положения точки в пространстве. Зачастую они представляются в матричной форме.

Сейчас нас интересуют только перевод и масштабирование — они дают возможность выразить систему координат холста в более привычной форме. В строке 13 мы переводим координаты вниз на высоту холста, а затем в строке 14 меняем направление оси *Y*. Порядок имеет значение — если выполнить эти действия в обратном порядке, начало координат окажется выше холста.

Графика создается путем нанесения капель основных цветов на лист миллиметровой бумаги (см. «Представление цветов» на с. 69). Но какой толщины должен быть этот лист бумаги и насколько точно нужно распределять цветовые капли?

Атрибуты `width` и `height` в строке 19 задают размер параметра `canvas` в пикселях (см. «Цифровые изображения» на с. 222). *Разрешение* экрана представляет собой количество пикселей на дюйм (или сантиметр). Таким образом, размер холста на экране зависит от разрешения этого же экрана. Если только ваш монитор не слишком стар, вы не сможете разглядеть на нем отдельные пиксели. (Не забудьте, что разрешение человеческого глаза в контексте поля зрения не постоянно. См. публикацию Майкла Диринга (Michael Deering) «A Photon Accurate Model of the Human Eye, SIGGRAPH», 2005.) Даже современным высокоточным мониторам требуются вспомогательные технологии, такие как избыточная выборка, чтобы изображение выглядело на них хорошо.

Чтобы разобраться в деталях, начнем с рисования разных объектов в очень низком разрешении. Создадим некое подобие миллиметровой бумаги с помощью JavaScript-функции, которая очищает холст и рисует на нем сетку (листинг 11.4). Для получения целочисленных значений координат пересечения линий сетки нам понадобятся масштабирующие преобразования. Масштабирование применяется ко всему, что нарисовано на холсте, поэтому уменьшим ширину рисуемой линии.

Листинг 11.4. Создание сетки на холсте

```
1 var grid = 25;                                // задать шаг сетки равным 25 пикселей
2
3 canvas.scale(grid, grid);
4 width = width / grid;
5 height = height / grid;
6 canvas.lineWidth = canvas.lineWidth / grid;
7 canvas.strokeStyle = "rgb(0, 0, 0)";           // черный
8
9 function
10 clear_and_draw_grid()
11 {
12   canvas.clearRect(0, 0, width, height);        // стереть содержимое холста
13   canvas.save();                                // сохранить настройки холста
14   canvas.setLineDash([0.1, 0.1]);                // пунктирная линия
15   canvas.strokeStyle = "rgb(128, 128, 128)";    // серый
16   canvas.beginPath();
17
18   for (var i = 1; i < height; i++) {           // горизонтальные линии
19     canvas.moveTo(0, i);
20     canvas.lineTo(height, i);
21   }
22
23   for (var i = 1; i < width; i++) {            // вертикальные линии
24     canvas.moveTo(i, 0);
25     canvas.lineTo(i, width);
26   }
27
28   canvas.stroke();                            // восстановить настройки холста
29   canvas.restore();
```

```

30 }
31
32 clear_and_draw_grid();           // вызвать функцию при открытии
                                    // страницы

```

Прямые линии

Нарисуем пару линий, поместив закрашенные круги на сетку с помощью кода из листинга 11.5. Одна линия будет горизонтальной, а для второй зададим наклон в 45 градусов. Цветовые капли диагональной линии имеют больший размер, поэтому мы увидим обе линии в их точке пересечения.

Листинг 11.5. Горизонтальная и диагональная линии

```

1 for (var i = 0; i <= width; i++) {
2   canvas.beginPath();
3   canvas.fillStyle = "rgb(255, 255, 0)"; // желтый
4   canvas.arc(i, i, 0.25, 0, 2 * Math.PI, 0);
5   canvas.fill();
6
7   canvas.beginPath();
8   canvas.fillStyle = "rgb(255, 0, 0)"; // красный
9   canvas.arc(i, 10, 0.2, 0, 2 * Math.PI, 0);
10  canvas.fill();
11 }

```

При запуске программы у вас должно получиться изображение как на рис. 11.10. Пиксели диагональной линии находятся дальше друг от друга, чем пиксели горизонтальной (согласно теореме Пифагора, дальше на $\sqrt{2} - 1$). Почему это важно? Обе линии состоят из одинакового количества пикселей, излучающих свет, но пиксели диагональной линии находятся на большем расстоянии друг от друга. Соответственно, плотность света диагональной линии меньше, и она выглядит более тусклой, чем горизонтальная. Существует не так много способов справиться с этим — например, дизайнеры экранов придают пикселям другую форму, чтобы избавиться от подобного эффекта. Такое поведение пикселей чаще встречается на дешевых экранах — на современных мониторах и телефонах это не так заметно.

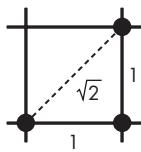


Рис. 11.10. Пространство между пикселями

Горизонтальные, вертикальные и диагональные линии не представляют особой сложности. Вот более интересная задача — как определить, какие именно пиксели должны подсвечиваться при добавлении новых линий? Создадим программу

для рисования линий, добавив пару элементов управления после тега `canvas` в теге `body`, как показано в листинге 11.6.

Листинг 11.6. Разметка для простой программы рисования линий

```
1 <div>
2   <label for="y">координата Y: </label>
3   <input type="text" size="3" id="y"/>
4   <button id="draw">Нарисовать</button>
5   <button id="erase">Стереть</button>
6 </div>
```

Следующий шаг — изменим код из листинга 11.5 в листинге 11.7, добавив обработчики событий для кнопок Нарисовать и Стереть. Функция `draw` использует устрашающее уравнение $y = mx + b$, где b всегда равно 0 (исключительно для примера). Наконец-то нам пригодилась математика!

Листинг 11.7. Функции для рисования и стирания линий, использующие числа с плавающей точкой

```
1 $('#draw').click(function() {
2   if ($('#y').val() < 0 || $('#y').val() > height) {
3     alert('y value must be between 0 and ' + height);
4   }
5   else if (parseInt($('#y').val()) != $('#y').val()) {
6     alert('y value must be an integer');
7   }
8   else {
9     canvas.beginPath();                      // нарисовать идеальную линию
10    canvas.moveTo(0, 0);
11    canvas.setLineDash([0.2, 0.2]);          // пунктирная линия
12    canvas.lineTo(width, $('#y').val());
13    canvas.stroke();
14
15    var m = $('#y').val() / width;          // наклон
16
17    canvas.fillStyle = "rgb(0, 0, 0)";
18
19    for (var x = 0; x <= width; x++) { // нарисовать точки на сетке
20      canvas.beginPath();
21      canvas.arc(x, Math.round(x * m), 0.15, 0, 2 * Math.PI, 0);
22      canvas.fill();
23    }
24
25    $('#y').val('');                      // очистить значение поля у
26  }
27 });
28
29 $('#erase').click(function() {
30   clear_and_draw_grid();
31 });
```

Зададим значение координаты Y равным 15. В результате должно получиться изображение, как на рис. 11.11.

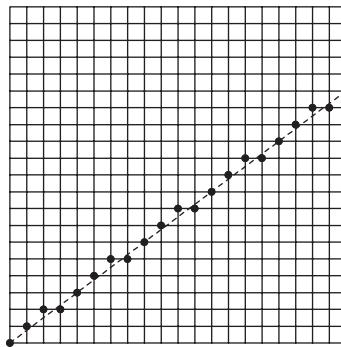


Рис. 11.11. Рисование линии при помощи операций над числами с плавающей точкой

Вблизи выглядит так себе, но если отойти подальше, то результат можно принять за прямую линию. Подобная проблема занимает не только специалистов в области компьютерной графики — все, кто вышивает крестиком, подтверждают.

Программа, которую мы только что написали, работает, но не очень эффективно. На каждом шаге она выполняет умножение и округление чисел с плавающей точкой. Даже на современных устройствах те же операции над целыми числами займут намного меньше времени. Мы все же попытались оптимизировать производительность, заранее проведя вычисления для наклона линии (строка 15). Это пример инвариантной оптимизации цикла, поэтому есть шанс, что оптимизатор (см. раздел «Оптимизация» на с. 293) автоматически сделает что-то подобное.

В 1962 году, когда операции над числами с плавающей точкой были непомерно дорогими, Джек Брезенхэм (Jack Bresenham) из IBM придумал удобный способ рисования линий без операций над числами с плавающей точкой. Он рассказал о своей идее в патентном бюро IBM, но там не оценили ее практической пользы, отказав в выдаче патента на изобретение. Оказалось, что это к лучшему — в дальнейшем идея Брезенхэма выросла в один из фундаментальных алгоритмов компьютерной графики, а отсутствие патента означало, что этим алгоритмом мог воспользоваться любой желающий. Брезенхэм осознал, что к проблеме рисования линий можно подойти инкрементно. Поскольку мы вычисляем значение y для всех последующих значений x , можно просто прибавлять к предыдущему y значение наклона (строка 9 в листинге 11.8), что избавляет от необходимости перемножать числа. Такое улучшение производительности вряд ли можно провести при помощи стандартного оптимизатора — мы действительно уменьшили конечную сложность алгоритма.

Листинг 11.8. Последовательное вычисление значения y

```

1 var y = 0;
2
3 canvas.fillStyle = "rgb(0, 0, 0)";
4
5 for (var x = 0; x <= width; x++) {           // нарисовать точки на сетке
6   canvas.beginPath();
7   canvas.arc(x, Math.round(y), 0.15, 0, 2 * Math.PI, 0);
8   canvas.fill();
9   y = y + m;
10 }

```

Нам приходится выполнять операции над числами с плавающей точкой, потому что наклон, вычисляемый по формуле $\Delta y / \Delta x$, — это дробное число. Однако деление можно заменить на сложение и вычитание, имея *условную переменную* d и добавляя к ней на каждой итерации. Значение y увеличивается, пока $d \geq \Delta x$, а затем мы вычитаем Δx из d .

Еще одна проблема — округление до целого. Нам нужны точки ровно посередине пикселей, а не по их нижним границам. Для этого начальное значение d нужно задать равным $1/2m$, а не 0. Чтобы избавиться от дробного числа, избавляемся от $1/2$, просто умножая все на 2. Так мы получаем $2\Delta y$ и $2\Delta x$. Замените фрагмент кода для рисования точек на сетке целочисленной версией из листинга 11.9 (в JavaScript не получится напрямую управлять типами данных, как это можно было сделать в языке наподобие С). Такой код будет работать корректно, если значение наклона находится в пределах от 0 до 1. Попробуйте улучшить этот код так, чтобы он работал с любыми значениями наклона.

Листинг 11.9. Рисование линий для целых чисел

```

1 var dx = width;
2 var dy = $('#y').val();
3 var d = 2 * dy - dx;
4 var y = 0;
5
6 dx *= 2;
7 dy *= 2;
8
9 canvas.fillStyle = "rgb(255, 255, 0)";
10 canvas.setLineDash([0,0]);
11
12 for (var x = 0; x <= width; x++) {
13   canvas.beginPath();
14   canvas.arc(x, y, 0.4, 0, 2 * Math.PI, 0);
15   canvas.stroke();
16
17   if (d >= 0) {
18     y++;
19     d -= dx;
20   }
21   d += dy;
22 }

```

У вас может возникнуть вопрос: почему условные вычисления в листинге 11.9 выглядят не так, как в листинге 11.10?

Листинг 11.10. Вариант условных вычислений

```
1 var dy_minus_dx = dy - dx;
2
3 if (d >= 0) {
4     y++;
5     d -= dy_minus_dx;
6 }
7 else {
8     d += dy;
9 }
```

На первый взгляд этот код лучше, потому что значение условной переменной изменяется только один раз на каждой итерации. В листинге 11.11 показано, что может произойти с этим кодом на гипотетическом ассемблере вроде тех, что мы рассматривали в главе 4.

Листинг 11.11. Вариант условных вычислений на ассемблере

load	d	load	d		
cmp	#0	cmp	#0		
blt	a	blt	a		
load	y	load	y		
add	#1	add	#1		
store	y	store	y		
load	d	load	d		
sub	dx_plus_dy	sub	dx		
bra	b				
a:	add	dy	a:	add	dy
	store	d		store	d
b:		

Обратите внимание, что альтернативный вариант кода на одну инструкцию длиннее, чем начальный. На многих устройствах сложение и ветвление выполняются за одно и то же время. Таким образом, код, который казался лучшим, на время выполнения одной инструкции медленнее, чем нужно для вычисления нового значения y .

Алгоритм Брезенхэма для рисования линий можно применять при решении огромного числа задач. Например, для создания *градиента* с плавным изменением цвета, как на рис. 11.12, значение y необходимо заменить на нужное значение цвета.



Рис. 11.12. Цветовой градиент

Градиент на рис. 11.12 был сгенерирован при помощи фрагмента кода из листинга 11.12, который нужно добавить в функцию, вызываемую после загрузки документа.

Листинг 11.12. Код для создания цветового градиента

```

1 var canvas = $('canvas')[0].getContext('2d');
2 var width = $('canvas').attr('width');
3 var height = $('canvas').attr('height');
4
5 canvas.translate(0, height);
6 canvas.scale(1, -1);
7
8 var m = $('#y').val() / width;
9
10 var dx = width;
11 var dc = 255;
12 var d = 2 * dc - dx;
13 var color = 0;
14
15 for (var x = 0; x <= width; x++) {
16   canvas.beginPath();
17   canvas.strokeStyle = "rgb(" + color + "," + color + "," + color + ")";
18   canvas.moveTo(x, 0)
19   canvas.lineTo(x, height);
20   canvas.stroke();
21
22   if (d >= 0) {
23     color++;
24     d -= 2 * dx;
25   }
26   d += 2 * dc;
27 }
```

Кривые линии¹

Целочисленные методы не ограничиваются прямыми линиями. Нарисуем простой эллипс, оси которого будут совпадать с осями координат, а центр — с началом отсчета. Такой эллипс можно описать следующим уравнением, где a — половина его ширины, а b — половина высоты:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$$

Представьте, что мы находимся в черной точке на рис. 11.13 — теперь нужно определить, какая из следующих возможных точек наиболее близка к идеальному эллипсу.

¹ В оригинале игра слов: curves ahead означает «впереди кривые», но это также официальное наименование дорожного знака, который в России называется «Извилистая дорога». — Примеч. ред.

Зададим $A = b^2$ и $B = a^2$. Теперь можно переписать уравнение как $Ax^2 + By^2 - AB = 0$. В случае с целочисленными значениями невозможно подобрать много точек для идеального эллипса, так как они не будут совпадать ни с контурами эллипса, ни с целочисленной сеткой. Находясь в точке с координатами (x, y) , мы должны подобрать следующую точку так, чтобы значение $Ax^2 + By^2 - AB$ было как можно ближе к 0. Кроме этого, решая уравнение, желательно избавиться от семикратного умножения.

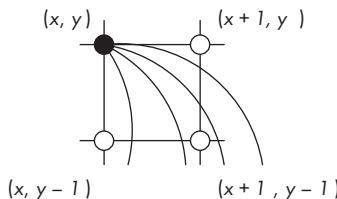


Рис. 11.13. Заданные координаты эллипса

Все, что можно сделать, — вычислить значения уравнения в каждой из трех возможных точек и выбрать точку, для которой результат будет наиболее близок к 0. Другими словами, рассчитать значение переменной отклонения d для каждой из трех точек, используя формулу $d = Ax^2 + By^2 - AB$.

Начнем с вычисления значения d в точке $(x + 1, y)$ без умножения. Подставим $(x + 1, y)$ в уравнение вместо значения x :

$$d_{x+1} = A(x+1)^2 + By^2 - AB.$$

Как известно, возвведение в квадрат можно заменить умножением:

$$d_{x+1} = A(x+1)(x+1) + By^2 - AB.$$

Перемножив, получаем:

$$d_{x+1} = x^2 + 2Ax + A + By^2 - AB.$$

Теперь, если вычесть конечное уравнение из начального, получим разность между уравнениями для x и $x + 1$:

$$dx = 2Ax + A.$$

dx можно прибавить к d , чтобы получить d_{x+1} . Это не совсем то, что нужно, потому что от умножения мы так и не избавились. Поэтому рассчитаем значение dx в точке $x + 1$:

$$d_{x+1} = 2A(x+1) + A;$$

$$d_{x+1} = Ax + 2A + A.$$

Выполнив вычитание, получим:

$$d2x = 2A.$$

В результате получили константу, благодаря которой легко вычислить d для координаты $(x + 1, y)$, не используя умножение. Нам понадобятся только промежуточные значения dx и $d2x$:

$$2A_{x+1} = 2Ax + d2x.$$

Таким образом мы рассчитаем направление по горизонтали. Направление по вертикали вычисляется почти так же, за исключением разницы в знаках, поскольку мы двигаемся в направлении $-y$:

$$\begin{aligned} dy &= -2By + B; \\ d2y &= 2B. \end{aligned}$$

Выполнив все вычисления, легко определить, какая из трех точек наиболее близка к идеальной кривой. Рассчитаем разности: по горизонтали dh для точки $(x + 1, y)$, по вертикали dv для точки $(x, y - 1)$ и по диагонали dd для точки $(x + 1, y - 1)$, а затем выберем меньшее из значений. Обратите внимание, что значение dx всегда должно быть положительным, а dv и dd могут быть отрицательными, поэтому для сравнения мы берем абсолютные значения, как показано на рис. 11.14.

Такой алгоритм рисования эллипса работает только в первой четверти системы координат. Оставшуюся часть эллипса нарисовать так же просто благодаря еще одному трюку — симметрии. Все четверти эллипса выглядят, как и первая, — разница в повороте кривой в направлении по горизонтали или вертикали. Начав с рисования точки с координатами (x, y) и добавив точки $(-x, y)$, $(-x, -y)$ и $(x, -y)$, мы получим целый эллипс. Если бы мы рисовали круг, то воспользовались бы симметрией в восьми направлениях.

В данном алгоритме используются сравнения, которые можно сократить, нарисовав только четверть эллипса. В точке, где наклон кривой равен 1, четверть эллипса можно разделить еще на две части. Таким образом, получим фрагмент кода, который отвечает за выбор между движением по горизонтали и по диагонали, и второй фрагмент кода, который отвечает за выбор между движением по вертикали и по диагонали. Значения a и b определяют, какой из фрагментов кода будет выполнен в первую очередь. На предварительные вычисления уйдет намного меньше времени, чем на выбор действий в цикле, так что это неплохая оптимизация кода.

У выбранного алгоритма есть одно слабое место — поскольку он начинается с половины ширины (a) и половины высоты (b), то он сработает только для эллипсов с нечетным числом пикселей в ширину и высоту. В итоге получится эллипс шириной $2a$ и высотой $2b$ плюс 1 для осей системы координат.

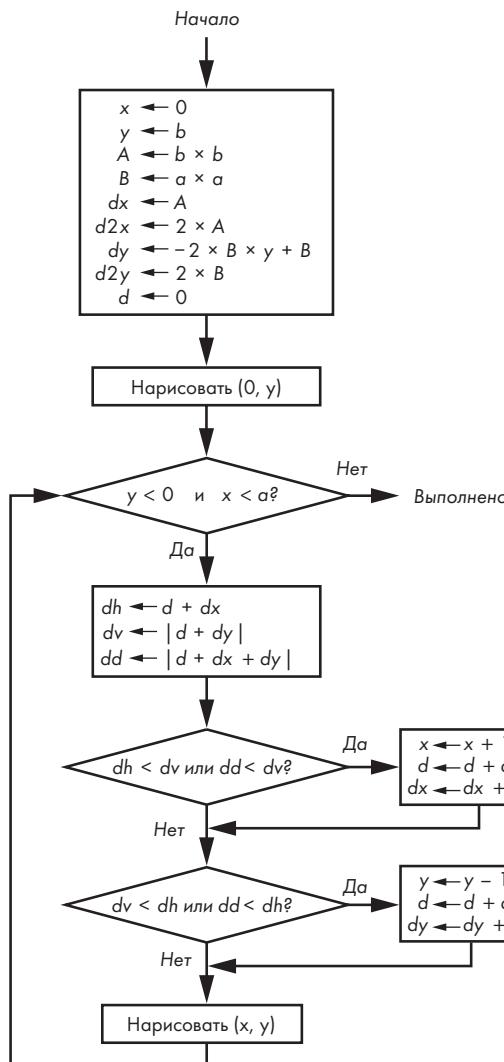


Рис. 11.14. Алгоритм рисования эллипса

Многочлены

Метод, который мы использовали для рисования эллипсов путем последовательного вычисления разностей, нельзя успешно масштабировать для значений, превышающих коническое сечение (введенных в квадрат). Это связано с тем, что уравнения более высокого порядка могут приводить к странным результатам — например, к нескольким сменам направления в пределах одного пикселя, что довольно сложно правильно протестировать.

Однако можно обобщить последовательное вычисление разностей, выразив его в форме многочлена $y = Ax^0 + Bx^1 + Cx^2 + \dots + Dx^n$. Все, что нужно, — сгенерировать n наборов разностей, чтобы начать поэтапно складывать значения с учетом заданных констант. Это сработает, поскольку в многочленах используется только одна независимая переменная. Вы наверняка помните разностную машину Чарльза Бэббиджа из раздела «Задача для цифровых компьютеров» на с. 75 — она как раз была предназначена для вычисления уравнений с использованием последовательных разностей.

Рекурсивное деление

Я мельком коснулся темы *рекурсивного деления* в разделе «Стеки» на с. 166. В этом разделе мы рассмотрим эту широко используемую технику более подробно и узнаем, как она помогает сократить объем выполняемой работы.

Сpirали

Код для рисования линий, который мы написали ранее, можно изменить так, чтобы рисовать более сложные линии путем нанесения на сетку нескольких точек и их соединения.

Наверняка из школьных уроков математики вы помните что-то об измерении углов в градусах и о том, что полный круг составляет 360 градусов. Помимо градусов, существуют и другие системы измерения углов, например *радианы*. Круг соответствует 2π радианам. Соответственно, 360 градусов — это 2π радиан, 180 градусов — π радиан, 90 градусов — $\pi/2$ радиан, 45 градусов — $\pi/4$ радиан и т. д. Я так подробно говорю об этом сейчас потому, что многие тригонометрические функции из математических библиотек, в том числе в JavaScript, подразумевают измерение углов в радианах, а не в градусах.

В наших примерах мы будем использовать кривые, нарисованные в полярной системе координат, — просто потому, что они красиво выглядят. На всякий случай упомяну, что в *полярной системе координат* используются радиус r и угол θ вместо x и y . Значения легко перевести в декартову систему координат: $x = r\cos\theta$ и $y = r\sin\theta$. В первом примере нарисуем спираль при помощи уравнения $r = \theta \times 10$. Рисуемая точка удаляется от центра по мере изменения углов. Большинству людей нелегко переключиться в уме с градусов на радианы, поэтому для примера используем входные данные в градусах. В листинге 11.13 представлена разметка для элементов управления.

Листинг 11.13. Разметка для элементов управления спиралью

```
<canvas width="500" height="500"></canvas>
<div>
```

```
<label for="degrees">Градусы: </label>
<input type="text" size="3" id="degrees"/>
<button id="draw">Нарисовать</button>
<button id="erase">Стереть</button>
</div>
```

Вместо кода для рисования сетки уделим внимание другим деталям. В листинге 11.14 точка с координатами $(0, 0)$ представляет собой центр спирали, поскольку мы используем полярную систему координат.

Листинг 11.14. Рисование спирали из точек в JavaScript

```
canvas.scale(1, -1);
canvas.translate(width / 2, -height / 2);

$('#erase').click(function() {
  canvas.clearRect(-width, -height, width * 2, height * 2);
});

$('#draw').click(function() {
  if (parseFloat($('#degrees').val()) == 0)
    alert('Градусы должны быть больше 0');
  else {
    for (var angle = 0; angle < 4 * 360; angle += parseFloat($('#degrees').val())) {
      var theta = 2 * Math.PI * angle / 360;
      var r = theta * 10;
      canvas.beginPath();
      canvas.arc(r * Math.cos(theta), r * Math.sin(theta), 3, 0, 2 * Math.PI, 0);
      canvas.fill();
    }
  }
});
```

Введите 10 (значение в градусах) и нажмите кнопку «Нарисовать». В результате должно получиться изображение, как на рис. 11.15.

Обратите внимание, что по мере удаления от центра спирали расстояние между точками становится все больше — в самом центре точки даже перекрывают друг друга. Можно ввести очень маленькое значение и получить практически идеальную кривую, но в этом случае точки перекроют друг друга еще больше. Рисование займет больше времени, и будет сложнее рассчитать значения произвольной функции.

Попробуем соединять точки линиями — замените фрагмент кода для рисования спирали на код из листинга 11.15.

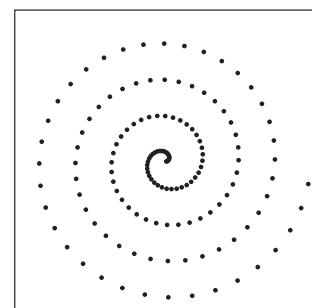


Рис. 11.15. Спираль из точек

Листинг 11.15. Рисуем спираль, соединяя точки линиями в JavaScript

```
canvas.beginPath();
canvas.moveTo(0, 0);

for (var angle = 0; angle < 4 * 360; angle += parseFloat($('#degrees').val())) {
  var theta = 2 * Math.PI * angle / 360;
  var r = theta * 10;
  canvas.lineTo(r * Math.cos(theta), r * Math.sin(theta));
}

canvas.stroke();
```

Введите 20 (значение в градусах) и нажмите кнопку «Нарисовать». Результат показан на рис. 11.16.

Сpirаль все еще не очень ровная. Опять же, она не-плохо выглядит в центре, но ближе к краям линия становятся все менее и менее плавной. Нужно найти способ вычислять координаты для дополнительных точек — и для этого пригодится уже знакомое рекурсивное деление. Сейчас мы рисуем линии, используя функцию для спирали между двух углов, θ_1 и θ_2 , — вместо этого будем опираться на некоторый критерий *достаточной близости*. Он означает, что, если две точки не находятся достаточно близко друг к другу, мы будем делить разницу между углами пополам и заново проверять критерий, пока не найдем достаточно близкую точку. Используем *формулу расстояния* $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, чтобы рассчитать расстояние между точками, как показано в листинге 11.16.

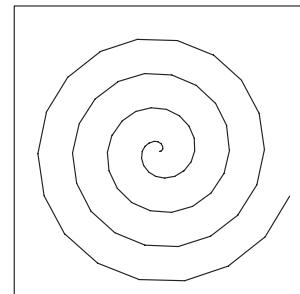


Рис. 11.16. Спираль

Листинг 11.16. Рекурсивное рисование спиральной линии в JavaScript

```
var close_enough = 10;

function
plot(theta_1, theta_2)
{
  var r;

  r = theta_1 * 10;
  var x1 = r * Math.cos(theta_1);
  var y1 = r * Math.sin(theta_1);

  r = theta_2 * 10;
  var x2 = r * Math.cos(theta_2);
  var y2 = r * Math.sin(theta_2);

  if (Math.sqrt(((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1))) < close_enough) {
    canvas.moveTo(x1, y1);
```

```
    canvas.lineTo(x2, y2);
}
else {
    plot(theta_1, theta_1 + (theta_2 - theta_1) / 2);
    plot(theta_1 + (theta_2 - theta_1) / 2, theta_2);
}
}

$('#draw').click(function() {
if (parseFloat($('#degrees').val()) == 0)
    alert('Градусы должны быть больше 0');
else {
    canvas.beginPath();

    for (var angle = 0; angle < 4 * 360; angle +=
        parseFloat($('#degrees').val())) {
        var old_theta;
        var theta = 2 * Math.PI * angle / 360;
        if (angle > 0)
            plot(old_theta, theta);
        old_theta = theta;
    }
}

canvas.stroke();
});
```

Пока значение переменной `close_enough` достаточно мало, увеличение значения в градусах несущественно, поскольку код автоматически генерирует нужное количество промежуточных углов. Попробуйте ради интереса подставить разные значения переменной `close_enough`. Для удобства добавьте в разметку соответствующее поле ввода.

В некоторых случаях определение достаточной близости весьма важно. Хотя это и выходит за рамки данной книги, вспомните, как в кино выглядят разные изогнутые объекты — при хорошей подсветке они кажутся более реалистичными. Теперь представьте зеркальную сферу, которую можно условно разбить на некоторое количество плоских граней. Наша спираль также разбивается на несколько отрезков. Если плоские грани слишком большие, вместо сферы мы получим что-то вроде диско-шара, который будет отражать свет совсем иначе.

Конструктивная геометрия

В главе 5 я кратко упомянул квадродеревья, или деревья квадрантов, показав, как с их помощью можно представлять различные формы. Они, очевидно, используют рекурсию вкупе с иерархическим механизмом разделения пространства.

Над квадродеревьями можно выполнять булевые операции. Например, нам нужно спроектировать что-то вроде прокладки ГБЦ (головки блока цилиндров) в двигателе автомобиля, как на рис. 11.17.

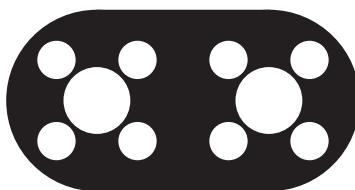


Рис. 11.17. Прокладка ГБЦ

Для узла квадродерева нам понадобятся структура данных и два специальных значения листьев: один — для 0 (белый цвет на рисунке), второй — для 1 (черный цвет). На рис. 11.18 показана подобная структура данных. Каждый узел ссылается на четыре других узла — хороший пример использования указателей в языках наподобие С.



Рис. 11.18. Узел квадродерева

В данном случае размер узла неважен. Все операции начинаются с корня дерева заданного размера, а каждый дочерний узел имеет размер, равный четверти родительского узла. На рис. 11.19 представлена схема для получения значения выбранного узла дерева.

На рис. 11.20 показано, как задаются значения (то есть обозначаются черным цветом) для точек квадродерева с координатами (x, y) . «Выполнено» означает «выход из функции», так как функция рекурсивная.

Подобным образом мы получали координаты точки на рис. 11.19. Алгоритм проходит по дереву в направлении сверху вниз, до тех пор пока не достигнет квадрата размером 1×1 с координатами (x, y) и не окрасит его в черный цвет. Если на пути встречается узел белого цвета, алгоритм заменяет его на новый узел с четырьмя дочерними узлами белого цвета. Таким образом создается дерево для продвижения сверху вниз. На обратном пути вверх все узлы с дочерними узлами черного цвета заменяются на отдельный черный узел. Это происходит каждый раз, когда в узле с тремя дочерними узлами черного цвета последний белый дочерний узел меняет цвет на черный, как показано на рис. 11.21.

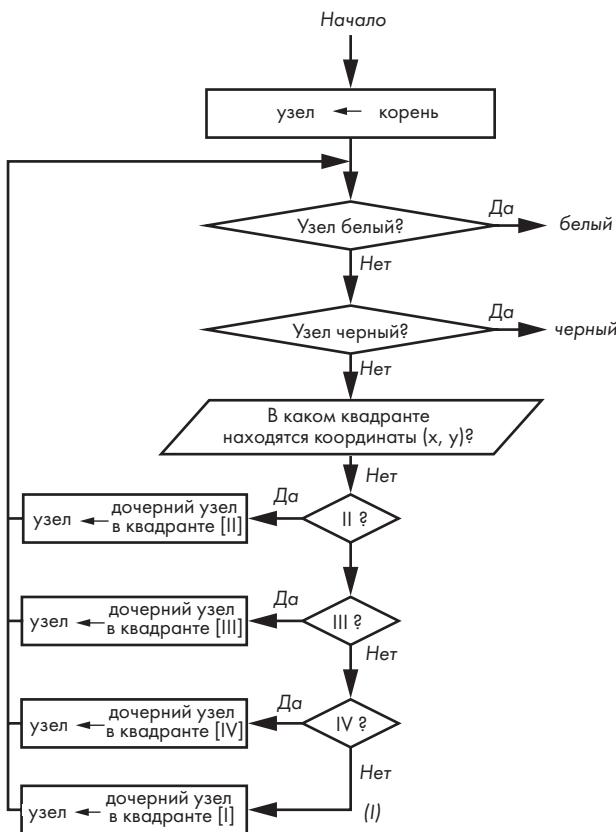


Рис. 11.19. Получение координат (x, y) в квадрограмме

Объединение узлов не только экономит память для хранения дерева, но и положительно сказывается на скорости операций, поскольку глубина дерева уменьшается.

Нужно найти способ очистить (то есть окрасить в белый цвет) значение координат (x, y) в квадрограмме. Решение этой задачи очень похоже на уже известный нам алгоритм задания цвета. Разница заключается в том, что теперь разделяются не белые, а черные узлы, а белые — наоборот, объединяются.

На основе функции задания значения можно создавать более сложные функции рисования. К примеру, задав необходимые координаты точек, можно нарисовать прямоугольник. Эта же функция пригодится и при рисовании эллипсов — мы лишь добавим принципы симметрии и алгоритм из раздела «Кривые линии» на с. 362.

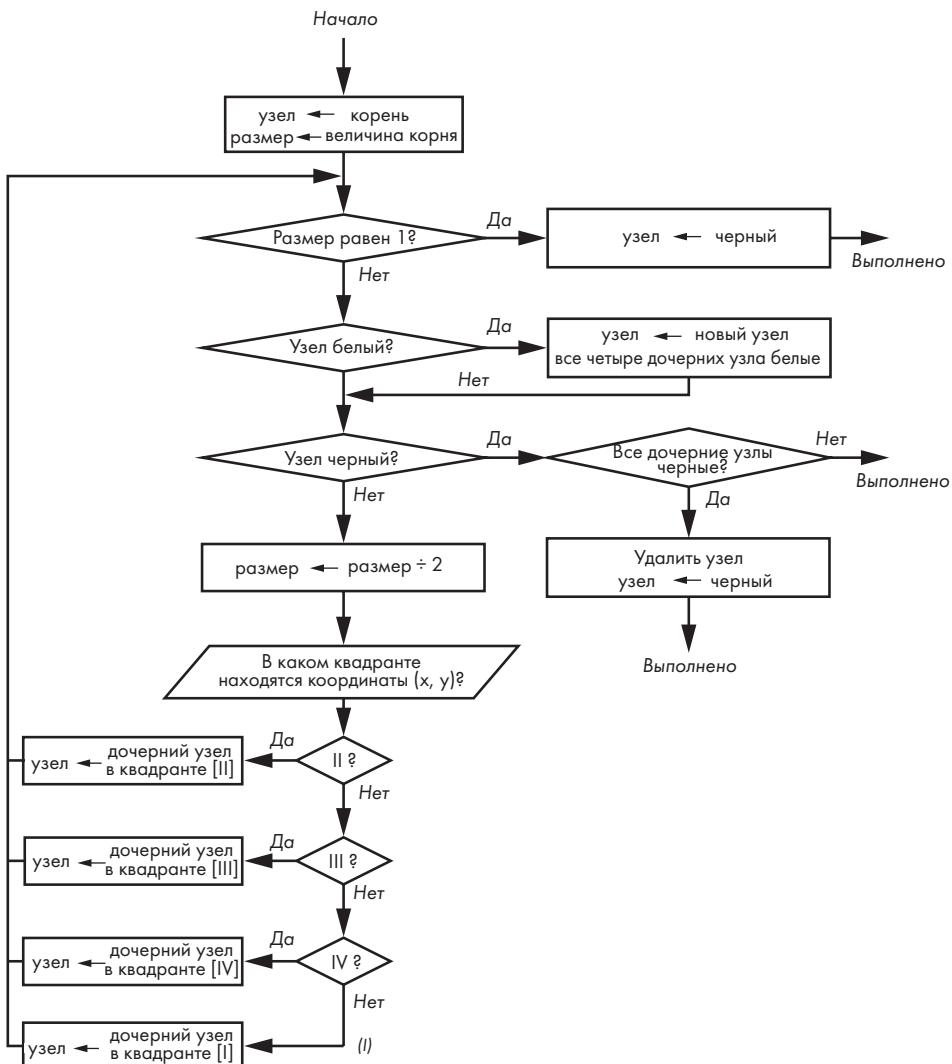
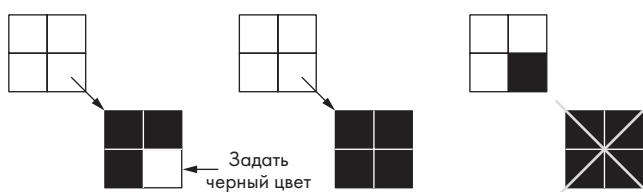
Рис. 11.20. Задание значения точки квадродерева с координатами (x, y) 

Рис. 11.21. Объединение узла

Забавы ради напишем версии функций булевой логики из главы 1 для квадродеревьев. Функция НЕ самая простая: спускаемся по дереву и заменяем все черные узлы белыми, а все белые — черными. Функции И и ИЛИ на рис. 11.22 требуют больше внимания. Эти алгоритмы не предназначены для вычисления эквивалентов выражений $C = a \text{ AND } b$ и $C = a \text{ OR } b$. Вместо этого вычисляются выражения $dst \& = src$ и $dst | = src$, как в случае с используемыми во многих языках операторами присваивания. При этом изменяется только операнд dst .

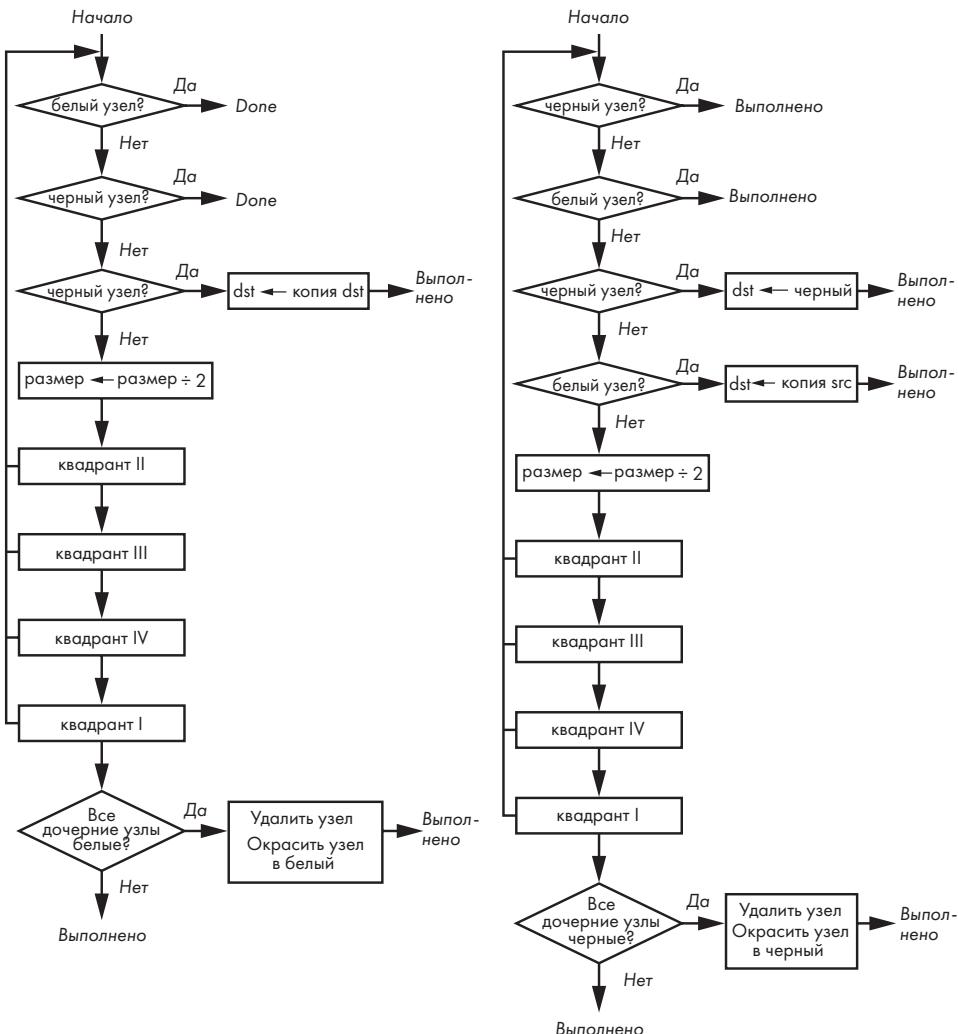


Рис. 11.22. Логические операции И и ИЛИ для квадродеревьев

Собрав все инструменты, перейдем к созданию прокладки ГБЦ. Разрешение возьмем низкое, чтобы были видны все детали. Начнем с квадродерева для пустой прокладки слева и начального квадродерева в центре, где нарисуем большой круг. Начальное квадродерево соединяется с прокладкой при помощи операции ИЛИ, в результате чего получим вариант справа на рис. 11.23. Обратите внимание: количество делений сведено к минимуму благодаря объединению узлов.

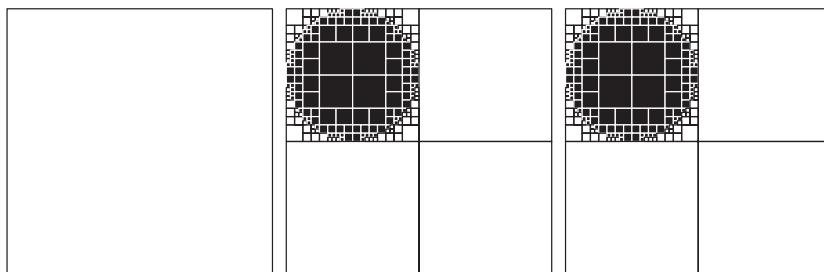


Рис. 11.23. Прокладка, круг, круг ИЛИ прокладка

Создадим новый круг в другом месте и объединим его с начальной версией прокладки, как показано на рис. 11.24.

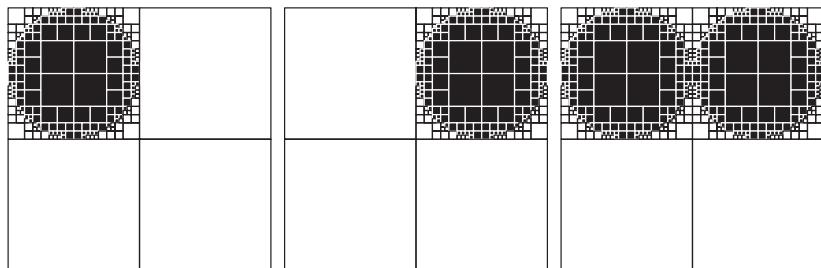


Рис. 11.24. Добавление к прокладке нового круга

Продолжим, создав черный прямоугольник и объединив его с прокладкой, как показано на рис. 11.25.

Следующий шаг — проделать в прокладке отверстие. Для этого нужно создать черный круг, а затем инвертировать его при помощи операции НЕ, получив круг белого цвета. Результат объединим с текущей версией прокладки, воспользовавшись операцией И. Так мы получим прокладку с отверстием, как показано на рис. 11.26.

К этому моменту уже может стать скучно. Тем же способом нужно добавить еще одно отверстие, как на рис. 11.26, и восемь новых мелких отверстий. Результат показан на рис. 11.27.

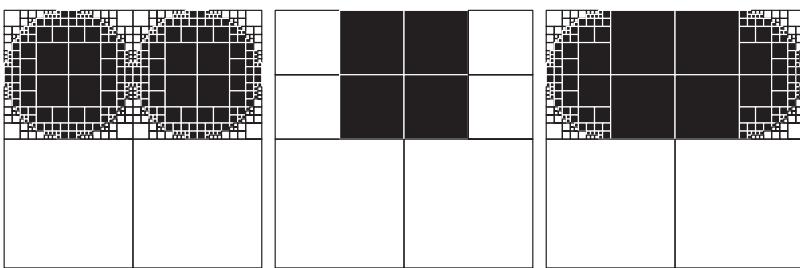


Рис. 11.25. Добавление прямоугольника

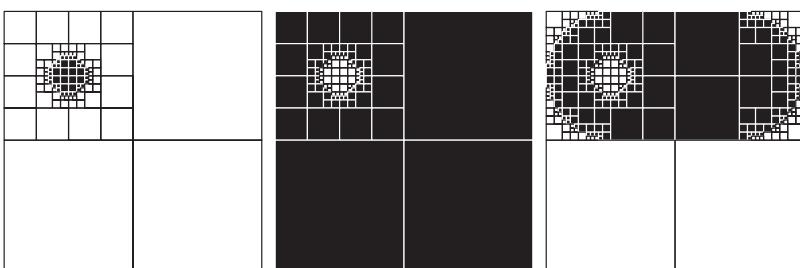


Рис. 11.26. Отверстие (операция НЕ) и добавление его к прокладке (операция И)

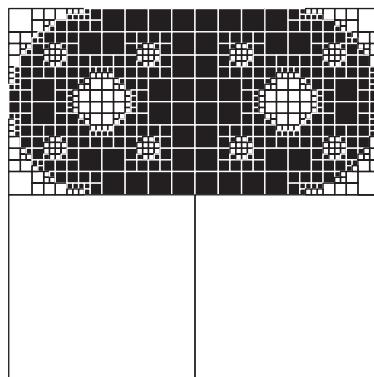


Рис. 11.27. Готовая прокладка

Как видим, в квадродеревьях мы использовали булевы функции, чтобы создать объекты сложной формы из геометрически простых частей. В примере мы ограничились двумерной прокладкой, но обычно такие вещи создаются в трех измерениях. Для трех измерений понадобится в два раза больше узлов, поэтому вместо квадродерева мы перейдем к *октодереву*, пример которого изображен на рис. 11.28.

Построение сложных трехмерных объектов на основе рассмотренных техник называется *конструктивной блочной геометрией*. Трехмерный аналог двумерного пикселя называется *вокселя* (voxel — сокращение от volume pixel («объемный пиксель»)).

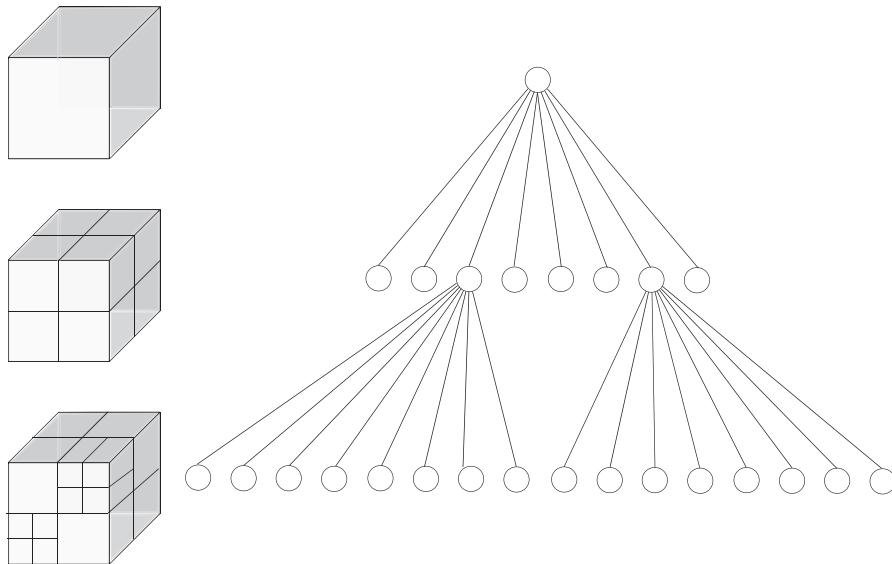


Рис. 11.28. Октордерево

Октордеревья часто используются для хранения данных, полученных от аппаратов компьютерной томографии и МРТ. Эти устройства генерируют стопки 2D-срезов, которые легко разделить, тем самым получив нужный вид в разрезе.

Сдвиг и наложение масок

Один из недостатков квадрдеревьев — расположение данных в памяти. Данные разбросаны по разным уголкам памяти, и отследить их местоположение очень сложно. Даже если два узла квадрдерева находятся рядом друг с другом, это не означает, что в памяти они расположены так же близко. Особенно остро эта проблема проявляется, когда нужно перевести данные из одной системы организации памяти в другую. Конечно, всегда можно перенести данные бит за битом, но для этого потребуется очень часто обращаться к памяти. Такой способ займет много времени, и потому он нам не подходит.

С подобной проблемой можно встретиться при выводении данных на экран. Организация памяти экрана зависит от аппаратного обеспечения. Как я уже упоминал в разделе «Растровая графика» на с. 230, каждый ряд растрового

изображения рисуется на экране последовательно, в заданном порядке. Растровый ряд называется *строкой развертки*, а весь набор строк развертки — *кадровым буфером*.

Предположим, что нам нужно вывести изображение готовой прокладки ГБЦ на экран компьютера. Простоты ради ограничимся монохромным дисплеем с памятью шириной в 16 бит из расчета 1 бит на 1 пиксель. Это означает, что верхние левые 16 пикселей будут относиться к первому слову, следующие 16 пикселей — ко второму и т. д.

Квадрат слева вверху на рис. 11.27 имеет размеры 4×4 пикселя и окрашен в белый цвет — то есть нужно очистить биты кадрового буфера. Будем использовать координаты и размер квадрата в квадродереве для создания маски, как показано на рис. 11.29.

0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Рис. 11.29. Маска операции И

Затем к полученной маске и всем связанным рядам можно применить операцию И, для чего потребуется всего по два обращения к памяти для каждого ряда: одно — для чтения, второе — для записи. Нечто подобное выполним и для битов кадрового буфера, только в этом случае маска будет содержать единицы, и мы применим операцию ИЛИ вместо И.

Еще один пример, где может пригодиться данный алгоритм, — рисование текстовых символов. Большинство текстовых символов хранятся в виде *битовых матриц* — двумерных массивов битов, как показано на рис. 11.30. Битовые матрицы нескольких символов объединяются и хранятся вместе в целях экономии памяти. Ранее все текстовые символы были организованы таким образом, сейчас же мы перешли к использованию геометрических описаний. Однако ради улучшения производительности описания символов часто переводятся в битовые матрицы перед использованием, и повторно используемые символы кэшируются.

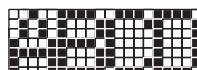


Рис. 11.30. Битовая матрица текстовых символов

Заменим изображенный выше символ *B* на *C*, как показано на рис. 11.31.

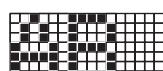


Рис. 11.31. Битовая матрица текстовых символов

Символ C хранится в битах 10–14, и теперь его нужно переместить в биты 6–10. Для этого потребуется взять символ C в каждом ряду и наложить маску на оставшиеся символы в слове. Затем необходимо сдвинуть символ C в заданную позицию. Место назначения должно быть прочитано, а на перезаписываемые позиции предварительно должна быть наложена маска. Затем сдвинутый символ C объединяется с остальными позициями и записывается, как показано на рис. 11.32.



Рис. 11.32. Рисование текстового символа

В данном примере на каждый ряд приходится три обращения к памяти: одно — для получения источника, второе — для получения места назначения, третье — для записи результата. Если выполнять все то же самое с каждым отдельным битом, это займет в пять раз больше времени.

В некоторых случаях могут возникнуть дополнительные сложности — например, если символ переносится за границы начального слова.

Еще меньше математики

Мы обсудили несколько способов избежать громоздких математических вычислений в разделе «Целочисленные методы» на с. 354. Основываясь на полученных знаниях, перейдем к более сложным примерам.

Приближения степенного ряда

Разберем еще одну возможность получить достаточно близкое значение. Допустим, нам нужно сгенерировать синусоиду, не имея аппаратного обеспечения, которое выполнило бы эту задачу. Для этого воспользуемся формулой ряда Тейлора:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}.$$

На рис. 11.33 представлены синусоидальная волна и приближения ряда Тейлора для разного количества значений. Как видим, чем больше значений мы используем, тем лучший результат получаем.

Проще всего добавлять новые значения, пока не будет достигнута желаемая степень точности. Также стоит отметить, что для углов острее 90 градусов требуется меньшее количество значений, поэтому для других углов можно ради эффективности использовать симметрию.

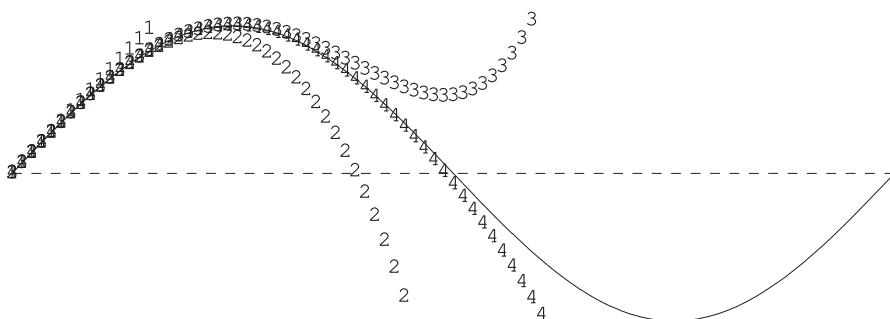


Рис. 11.33. Ряд Тейлора для синусоиды

Обратите внимание, что можно сократить количество операций умножения, задав *результат* как x , предварительно вычислив $-x^2$ и перемножив *результат* с $-x^2$ для получения каждого нового значения. В формуле все знаменатели являются константами — это означает, что их можно хранить в таблице, проиндексировав по показателям степени. Кроме того, все значения вычислять не обязательно. Если требуется точность в две цифры, то выполнение алгоритма стоит прекратить, как только добавление новых значений перестанет влиять на последние две цифры результата.

Алгоритм Волдера

Джек Волдер (Jack Volder) из компании Convair в 1956 году изобрел алгоритм под названием *CORDIC* (сокращение от *COordinate Rotation DIgital Computer* — цифровой вычислитель поворота системы координат). Алгоритм Волдера был создан как более точная замена аналоговой части системы навигации бомбардировщика B-58. Данный алгоритм может использоваться для генерации тригонометрических и логарифмических функций с применением целочисленных вычислений. Он же был добавлен в первый портативный научный калькулятор HP-35, выпущенный в 1972 году, а также в семейство сопроцессоров Intel 80x87 для вычислений с плавающей точкой.

Основная идея алгоритма Волдера изображена на рис. 11.34. Поскольку это единичная окружность (с радиусом, равным 1), координаты X и Y концов стрелки представляют собой косинус и синус угла. Нужно обернуть стрелку относительно ее начальной позиции вдоль оси X , используя все меньшие и меньшие шаги, пока не будет достигнуто желаемое значение угла, — и после этого определить координаты.

Допустим, нам нужен синус угла в 57.529 градуса. Как видите, мы начали с угла в 45 градусов, и этого оказалось недостаточно. Следующий шаг — 25.565 градуса. Так мы получили новый угол в 71.565 градуса — слишком большой. Затем мы вернулись назад на 14.036 градуса, получив желаемый угол в 57.529 градуса.

Очевидно, что это еще одно применение уже знакомого нам деления, только со странными значениями углов.

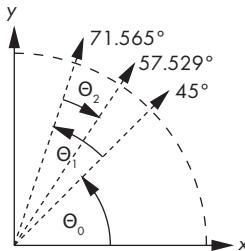


Рис. 11.34. Обзор алгоритма Волдера

Мы уже рассматривали уравнения трансформации в разделе «Целочисленные методы», ограничившись переводом и масштабированием. Алгоритм Волдера основан на вращении. Следующие уравнения, с общей формой которых вы уже знакомы, показывают, как координаты (x, y) вращаются относительно угла θ для получения новых значений координат (x', y') :

$$\begin{aligned} x' &= x \cos(\theta) - y \sin(\theta); \\ y' &= x \sin(\theta) - y \cos(\theta). \end{aligned}$$

Эти уравнения математически корректны, но кажутся бесполезными — мы не стали бы обсуждать алгоритм генерации синусов и косинусов, заранее зная их значения.

Перед тем как улучшить формулы, еще усложним их, переписав уравнения с использованием тангенсов (вспомните следующее тригонометрическое тождество):

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)}.$$

Чтобы добавить деление на $\cos(\theta)$, умножим на него же результат уравнения:

$$x' = \cos(\theta) \times \left[x \frac{\cos(\theta)}{\cos(\theta)} - y \frac{\sin(\theta)}{\cos(\theta)} \right] = \cos(\theta) \times [x - y \tan(\theta)];$$

$$y' = \cos(\theta) \times \left[x \frac{\sin(\theta)}{\cos(\theta)} - y \frac{\cos(\theta)}{\cos(\theta)} \right] = \cos(\theta) \times [x \tan(\theta) + y].$$

Выглядит, мягко говоря, ужасно. Кажется, что все стало еще хуже — но это потому, что мы еще не поговорили о приеме, который возвращает нас к странным значениям углов. Оказывается, $\tan(45^\circ) = 1$, $\tan(26.565^\circ) = \frac{1}{2}$, а $\tan(14.036^\circ) = \frac{1}{4}$. Определенно, это похоже на простое деление на 2, или, как сказал бы Максвелл

Смарт (Maxwell Smart)¹, «старый трюк со сдвигом вправо». Таким образом, мы пришли к двоичному поиску тангенсов углов.

Рассмотрим, как это соотносится с примером на рис. 11.34. Видим три вращения, которые приводят от начальных координат к конечным. Не забудьте, что, согласно рис. 11.34, $x_0 = 1$ и $y_0 = 0$:

$$x_1 = \cos(45^\circ) \times \left[x_0 - y_0 \times \tan(45^\circ) \right] = \cos(45^\circ) \times \left[x_0 - \frac{y_0}{1} \right];$$

$$y_1 = \cos(45^\circ) \times \left[x_0 \times \tan(45^\circ) + y_0 \right] = \cos(45^\circ) \times \left[\frac{x_0}{1} + y_0 \right];$$

$$x_2 = \cos(26.565^\circ) \times \left[x_1 - y_1 \times \tan(26.565^\circ) \right] = \cos(26.565^\circ) \times \left[x_1 - \frac{y_1}{2} \right];$$

$$y_2 = \cos(26.565^\circ) \times \left[x_1 \times \tan(26.565^\circ) + y_1 \right] = \cos(26.565^\circ) \times \left[\frac{x_1}{2} + y_1 \right];$$

$$x_3 = \cos(-14.036^\circ) \times \left[x_2 - y_2 \times \tan(-14.036^\circ) \right] = \cos(-14.036^\circ) \times \left[x_2 - \frac{y_2}{4} \right];$$

$$y_3 = \cos(-14.036^\circ) \times \left[x_2 \times \tan(-14.036^\circ) + y_2 \right] = \cos(-14.036^\circ) \times \left[\frac{x_2}{4} + y_2 \right].$$

Обратите внимание на изменение знака в последнем наборе уравнений — он появился после смены направления (по часовой стрелке). При вращении по часовой стрелке тангенс становится отрицательным. В результате подстановки уравнений для (x_1, y_1) в уравнения для (x_2, y_2) , подстановки всего этого выражения в уравнения для (x_3, y_3) , разложения косинусов на множители (и избавления от умножения на 1) получаем:

$$x_3 = \cos(45^\circ) \times \cos(26.565^\circ) \times \cos(-14.036^\circ) \times \left(x_0 - y_0 - \frac{x_0 - y_0}{2} + \frac{\frac{x_0 - y_0}{2} + (x_0 + y_0)}{4} \right);$$

$$y_3 = \cos(45^\circ) \times \cos(26.565^\circ) \times \cos(-14.036^\circ) \times$$

$$\times \left(\frac{-(x_0 - y_0) - \frac{(x_0 + y_0)}{2}}{4} + \frac{(x_0 - y_0)}{2} + (x_0 + y_0) \right).$$

¹ Главный герой сериала «Напряги извилины». — Примеч. ред.

И что же делать с косинусами? Опустив математические доказательства, заметим, что при достаточном количестве значений:

$$\cos(45^\circ) \times \cos(26.565^\circ) \times \cos(-14.036^\circ) \times \dots = 0.607252935008881.$$

Это константа, а константы мы любим. Назовем ее C . В итоге можно преобразовать уравнение, умножив его на C :

$$x_3 = C \times \left\{ \left[x_0 - y_0 \right] - \frac{x_0 - y_0}{2} + \frac{\frac{x_0 - y_0}{2} + [x_0 + y_0]}{4} \right\};$$

$$y_3 = C \times \left\{ -\frac{\left[x_0 - y_0 \right] - \frac{(x_0 + y_0)}{2}}{4} + \frac{(x_0 - y_0)}{2} + [x_0 + y_0] \right\}.$$

Либо оставить умножение, используя константу для x_0 , как показано ниже. Также избавимся от y_0 , поскольку он равен 0. В результате получим:

$$x_3 = C - \frac{C}{2} + \frac{\frac{C}{2} + C}{4} = 0.531;$$

$$y_3 = -\frac{C - \frac{C}{2}}{4} + \frac{C}{2} + C = 0.834.$$

Можно удостовериться, что значения x_3 и y_3 довольно близки к значениям косинуса и синуса угла в 57.529 градуса. И это только при трех заданных значениях — чем больше значений предоставить, тем точнее будет результат. А еще, заметьте, все это можно рассчитать при помощи лишь сложения, вычитания и деления на 2.

Переведем это на программный язык, заодно узнав несколько новых приемов. Во-первых, применим немного другую версию алгоритма Волдера под названием *режим наведения*. До этого мы использовали *режим вращения*, более простой для понимания. В режиме вращения мы начинали с вектора (стрелки) вдоль оси X и вращали его до тех пор, пока не получили нужный угол. Режим наведения представляет собой обратные действия — мы начинаем с нужного угла и вращаем вектор до тех пор, пока не достигнем параллели с осью X (угол при этом равен 0). Это означает, что мы только проверяем знак угла, чтобы определить направление вращения, избавляясь от сравнения двух чисел.

Во-вторых, нам понадобится поиск по таблице. Заранее рассчитаем таблицу углов с тангенсами 1, $\frac{1}{2}$, $\frac{1}{4}$ и т. д. Все это мы проделаем только один раз. Конечный вариант алгоритма представлен на рис. 11.35.

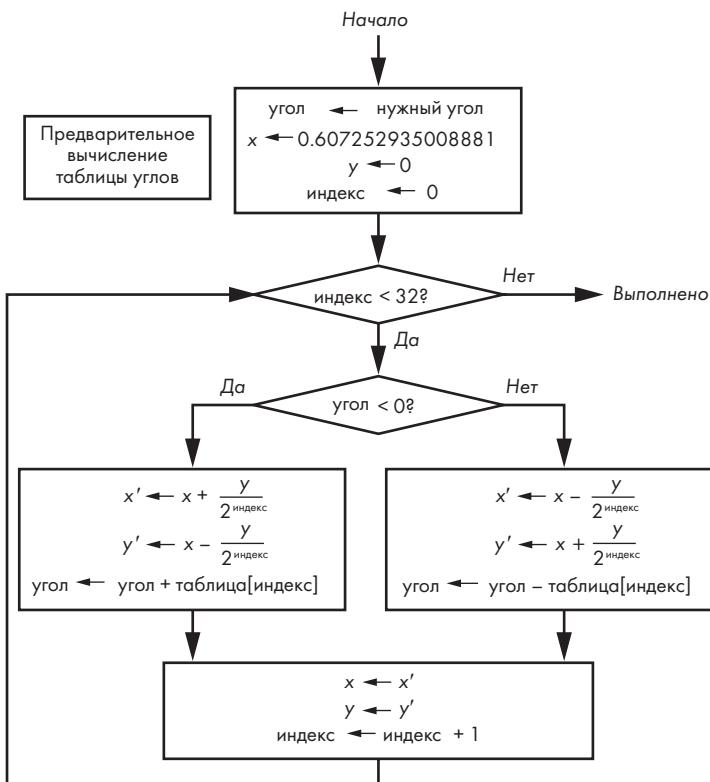


Рис. 11.35. Блок-схема алгоритма Волдера

Напишем программу на языке C, реализующую данный алгоритм, — заодно разберем некоторые приемы. Во-первых, перейдем к измерению углов в радианах вместо градусов.

Второй прием связан с первым. Вы могли заметить, что нам еще не встречались числа больше 1. Спроектируем программу так, чтобы она работала только для первого квадранта (между 0 и 90 градусами), а значения для других квадрантов получим с помощью симметрии. Угол в 90 градусов – это $\pi/2$ (≈ 1.57). Поскольку разброс значений небольшой, можно ограничиться целыми числами с фиксированной точкой вместо чисел с плавающей точкой.

В основе примера реализации будут лежать 32-битные целые числа. Нам нужен диапазон $\approx \pm 1.6$, поэтому бит 30 будет использоваться для единиц, бит 29 — для половин, бит 28 — для четвертей, бит 27 — для восьмых долей и т. д. Также будем использовать старший (бит 31) и знаковый биты. Чтобы перевести числа с плавающей точкой (если они находятся внутри заданного диапазона) в числа с фиксированной точкой, умножим их на нашу версию 1 (0x40000000) и приведем (преобразуем) их в целые числа. Подобным образом переведем результаты

обратно в систему чисел с плавающей точкой, использовав то же приведение и деление на 0x40000000.

В листинге 11.17 представлен довольно простой исходный код.

Листинг 11.17. Реализация алгоритма Волдера на С

```

1 const int angles[] = {
2     0x3243f6a8, 0x1dac6705, 0x0fadbafc, 0x07f56ea6, 0x03feab76, 0x01ffd55b,
3     0x00ffffaa, 0x007fff55,
4     0x003ffffa, 0x001ffffd, 0x000fffff, 0x0007ffff, 0x0003ffff, 0x0001ffff,
5     0x000003fff, 0x00001fff, 0x00000fff, 0x000000ff, 0x000003ff, 0x000001ff,
6     0x000000ff, 0x0000007f,
7     0x0000003f, 0x0000001f, 0x0000000f, 0x00000008, 0x00000004, 0x00000002,
8     0x00000001, 0x00000000
9 };
10 int x = (int)(0.6072529350088812561694 * 0x40000000);
11 int y = 0;
12
13 for (int index = 0; index < 32; index++) {
14     int x_prime;
15     int y_prime;
16
17     if (angle < 0) {
18         x_prime = x + (y >> index);
19         y_prime = y - (x >> index);
20         angle += angles[index];
21     }
22     else {
23         x_prime = x - (y >> index);
24         y_prime = y + (x >> index);
25         angle -= angles[index];
26     }
27
28     x = x_prime;
29     y = y_prime;
30 }
```

В реализации алгоритма Волдера используется много приемов из нашего пополняющегося арсенала: рекурсивное деление, предварительные вычисления, поиск по таблице, деление на 2, арифметика чисел с фиксированной точкой и симметрия.

Парочка случайностей

Очень сложно научить компьютеры выполнять случайные операции — им обязательно нужна заданная формула для генерации случайных чисел, что делает

эти числа по своей сути повторяемыми. Для большинства вычислительных задач этой мнимой «случайности» достаточно — кроме криптографии, которую мы обсудим в главе 13. В этом разделе мы рассмотрим некоторые приближения, основанные на *псевдослучайности*. Разберем несколько наглядных примеров — они интересные, и их легко изобразить.

Заполняющие пространство кривые

Итальянский математик Джузеппе Пеано (Giuseppe Peano) (1858–1932) впервые продемонстрировал миру *заполняющие пространство кривые* в 1890 году. На рис. 11.36 представлены три варианта таких кривых.

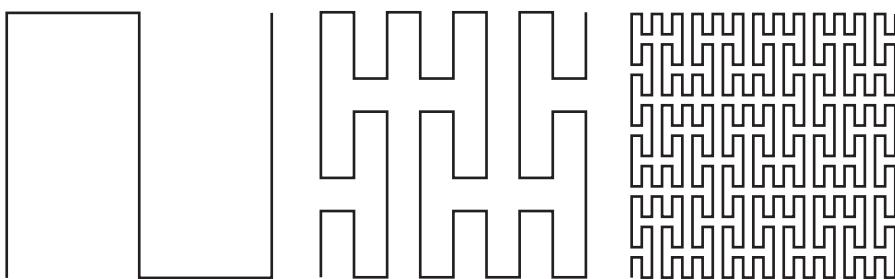


Рис. 11.36. Кривые Пеано

Кривая представляет собой простую форму — она разворачивается и повторяется в разных направлениях. Чем чаще повторяется узор, тем больше пространства заполняет кривая.

Заполняющие кривые — отличный пример *самоподобия*. Это означает, что вблизи и вдали они выглядят примерно одинаково. Еще один вариант заполняющих кривых представляют собой так называемые фракталы — они обрели популярность после того, как Бенуа Мандельброт (Benoit Mandelbrot) (1924–2010) выпустил книгу «Фрактальная геометрия природы» (Институт компьютерных исследований, 2002). Многие явления природы самоподобны — например, береговая линия имеет одинаковую зубчатую форму при наблюдении со спутника и через микроскоп.

Термин *фрактал* (fractal) происходит от английского слова fraction — «дробь». В геометрии можно встретить множество целочисленных отношений. Например, удвоение длин сторон квадрата приводит к увеличению его площади в 4 раза. Однако целочисленное увеличение длины стороны фрактала может привести к дробному увеличению его площади — отсюда и название.

Снежинка Коха — это простая фрактальная кривая, впервые описанная в 1904 году шведским математиком Хельге фон Кохом (Helge von Koch) (1870–1924). Она начинается с равностороннего треугольника. Затем каждая его

сторона делится на трети, и центральная третья заменяется треугольником, размер которого равен трети размера исходного треугольника. Края каждого нового треугольника, совпадающие с линиями исходного треугольника, стираются, как показано на рис. 11.37.

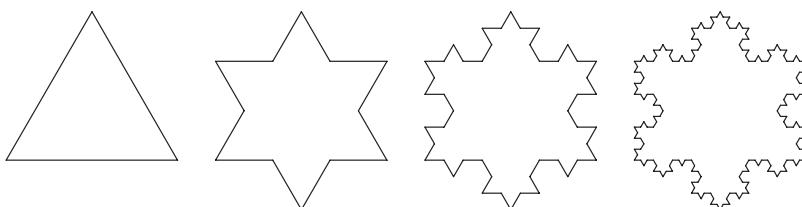


Рис. 11.37. Четыре итерации снежинки Коха

Сложные и интересные формы можно генерировать при помощи пары строк кода и рекурсии. Рассмотрим немного более сложный пример: кривую Гильберта, впервые описанную в 1891 году немецким математиком Давидом Гильбертом (David Hilbert) (1862–1943), изображенную на рис. 11.38.

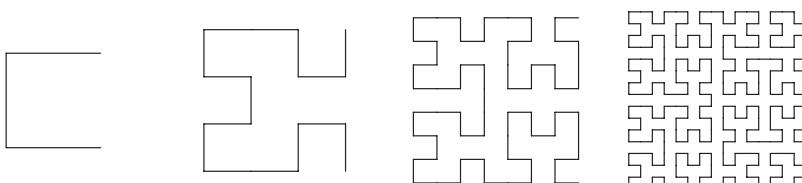


Рис. 11.38. Четыре итерации кривой Гильберта

Правила следующей итерации для кривой Гильберта чуть сложнее, чем для снежинки Коха, поскольку каждый раз повторяется не полностью идентичный узор. Всего используются четыре расположения формы «чашки», которые затем заменяются их же меньшими версиями, как показано на рис. 11.39. Помимо графической реализации, существует версия с буквами, указывающими направления: право (П), верх (В), лево (Л), низ (Н). На каждой итерации каждый угол формы слева заменяется четырьмя формами справа (в заданном порядке), в 4 раза меньшими по размеру, чем форма слева. Готовый узор соединяется прямыми линиями.

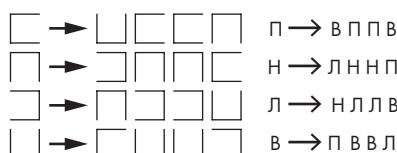


Рис. 11.39. Правила кривой Гильберта

L-системы

Правила на рис. 11.39 работают подобно регулярным выражениям из раздела «Регулярные выражения» на с. 281, с которыми вы уже знакомы, но в обратном порядке. Вместо задания шаблона для поиска данные правила сами определяют, какие шаблоны можно получить. Эти правила называются *L-системами*, или *системами Линденмайера*, по имени венгерского биолога Аристида Линденмайера (Aristid Lindenmayer) (1925–1989), разработавшего их в 1968 году. Поскольку правила определяют, что может быть получено, их также называют *порождающей грамматикой*.

На рис. 11.39 видно, что замена *P* на последовательность *B P P B* приводит к трансформации крайней левой кривой с рис. 11.38 в изображенную рядом с ней кривую.

Порождающие грамматики хороши тем, что они компактны и их легко определить и реализовать. Их можно использовать для моделирования самых разных явлений. Они снискали огромную популярность после того, как Алви Рей Смит (Alvy Ray Smith) из Lucasfilm опубликовал работу «Plants, Fractals, and Formal Languages» (SIGGRAPH, 1984). С тех пор стало невозможно выйти из дома, не наткнувшись на куст, воспроизведенный при помощи L-систем. Работа Линденмайера легла в основу многих приемов компьютерной графики, которые можно встретить в современных фильмах.

Посадим пару деревьев, чтобы уменьшить углеродный след от этой книги. В нашей грамматике используются четыре символа, как показано в листинге 11.18.

Листинг 11.18. Символы грамматики дерева

```
К нарисовать прямую, заканчивающуюся листом
В нарисовать прямую для ветви дерева
Л сохранить позицию и угол, повернуть влево на 45°
П восстановить позицию и угол, повернуть вправо на 45°
```

В листинге 11.19 создадим грамматику, включающую в себя два правила.

Листинг 11.19. Правила грамматики дерева

```
В → В В
К → В Л К П К
```

Символы и правила напоминают генетический код. На рис. 11.40 показаны несколько итераций грамматики, начиная с *K*. Обратите внимание: мы не рисуем листья на концах ветвей дерева. Кроме того, набор символов, которые определяют дерево, слишком длинный для каждой итерации, кроме первых трех.

В результате без особого труда получается неплохое дерево. L-системы прекрасно подходят для изображения естественных объектов.

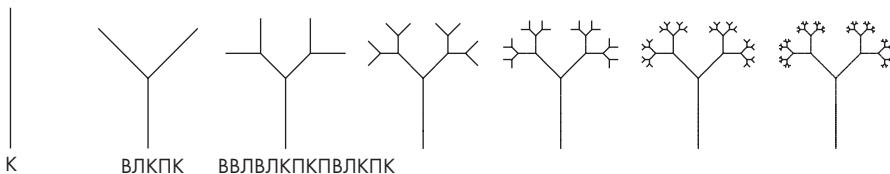


Рис. 11.40. Простое дерево L-системы

Порождающие грамматики использовались для генерации объектов задолго до появления первых компьютеров. Схемы вязания узоров представляют собой традиционные примеры порождающих грамматик, как показано в листинге 11.20.

Листинг 11.20. Порождающая грамматика для вязания шарфа на рис. 11.41

л = лицевая петля

и = изнаночная петля

п = провязать первую петлю как изнаночную

ряд₁ → п и л ли ил ли и л и ил ли ил ли ил и л
 ряд₂ → п л и ил ли ил ли л ли ил ли ил ли ил ли ил
 ряд₅ → п и ил ли ил ли ии ил ли ил ли ии ил ли ил ли ил
 ряд₆ → п л ли ил
 рапорт → ряд₁ ряд₂ ряд₁ ряд₂ ряд₅ ряд₅ ряд₆ ряд₂ ряд₂ ряд₅ ряд₅ ряд₆ ряд₅
 шарф → рапорт...

Если следовать грамматике в листинге 11.20, используя вязальную спицу в качестве устройства ввода/вывода, то, повторив узор несколько раз, получим шарф как на рис. 11.41.



Рис. 11.41. Шарф, созданный с помощью порождающей грамматики

Стохастические приемы

«Стохастический» — термин, которым можно блеснуть, если слово «случайный» звучит недостаточно солидно. Алан Форньер (Alan Fournier) и Дон Фуссель (Don Fussell) из Техасского университета в Далласе ввели понятие случайности в компьютерную графику в 1980 году. Добавление толики случайности ведет к разнообразию. Например, на рис. 11.42 показано стохастическое изменение деревьев L-системы из предыдущего раздела.

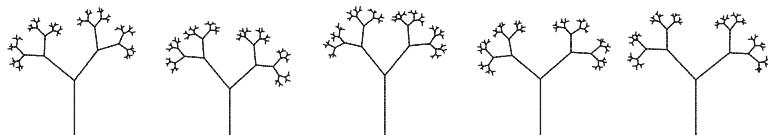


Рис. 11.42. Стохастические деревья L-систем

Как видите, в результате получился занятный набор похожих друг на друга деревьев. Лес выглядит более естественно, если не все деревья в нем одинаковы.

Лорен Карпентер (Loren Carpenter) из компании Boeing выпустил статью, которая осветила один из первых простых способов создания фракталов («Computer Rendering of Fractal Curves and Surfaces», SIGGRAPH, 1980). В SIGGRAPH 1983 вышла жаркая дискуссия между Карпентером и Мандельбротом на тему того, действительно ли результаты, полученные Лореном, представляют собой фракталы.

В итоге Карпентер ушел из Boeing и продолжил работу над фракталами в Lucasfilm. На основе созданных им фрактальных гор была спроектирована планета в фильме «Звездный путь 2: Гнев Хана». Говорят, что на генерацию планеты ушло шесть месяцев компьютерного времени. Из-за того что планета была создана при помощи графики, на некоторых кадрах в полете гроб Спока касался склона горы. В итоге художникам пришлось исправлять очертания горы вручную. Техника Карпентера была проста. Он случайным образом выбирал точку на прямой и несколько раз перемещал ее — также случайным образом. Эти действия рекурсивно повторялись для двух сегментов прямой, пока те не оказывались достаточно близко друг к другу. В целом алгоритм напоминал добавление случайности к генератору кривых Коха. На рис. 11.43 представлены несколько случайным образом созданных горных вершин.

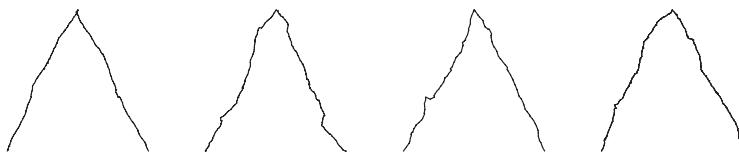


Рис. 11.43. Фрактальные горы

Опять же, неплохой результат, не требующий больших усилий.

Квантование

Иногда возможности выбрать определенную степень приближения нет, и тогда приходится просто делать все, что в наших силах. Например, нужно напечатать в черно-белой газете цветную фотографию. Рассмотрим, как выполняется трансформация в таких случаях. На рис. 11.44 представлена черно-белая фотография, потому что эта книга напечатана не в цвете. Поскольку на фото используются только оттенки серого, все три компонента каждого цвета идентичны друг другу и находятся в диапазоне от 0 до 255.



Рис. 11.44. Кот Тони

Здесь задействуется процесс под названием *квантование* — когда цвета оригинального изображения соотносятся с цветами его трансформированной версии. В этом заключается еще одна проблема дискретизации — взять аналоговый (даже более аналоговый в нашем случае) сигнал и разделить его на заданное число емкостей. Как уместить 256 значений в 2?

Начнем с простого подхода под названием *разделение по порогу*. Как понятно из названия, мы выбираем определенный порог и относим все, что светлее порога, к белому цвету, а все, что темнее, — к черному. В листинге 11.21 представлен пример присвоения значений больше 127 белому цвету, а всех остальных значений — черному.

Листинг 11.21. Разделение по порогу в псевдокоде

```
for (y = 0; y < height; y++)
    for (x = 0; x < width; x++)
        if (value_of_pixel_at(x, y) > 127)
            draw_white_pixel_at(x, y);
        else
            draw_black_pixel_at(x, y);
```

Если запустить данный псевдокод для изображения на рис. 11.44, получим результат, как на рис. 11.45.



Рис. 11.45. Результат выполнения алгоритма разделения по порогу

Выглядит так себе, но сейчас мало что можно исправить. Поэкспериментировав со значениями порога, получим всего лишь набор плохих результатов. Попробуем улучшить их при помощи оптических иллюзий.

Британский ученый Генри Тальбот (Henry Talbot) (1800–1877) изобрел *полутоновую печать* в 1850-х исключительно по этой причине — в те времена фотографии были только черно-белыми, как и печатные издания. При использовании полутоновой печати картинки разбивались на черные точки разных размеров, как показано на увеличенном изображении слева на рис. 11.46. Справа представлен результат, видимый человеческим глазом, — разные оттенки серого.

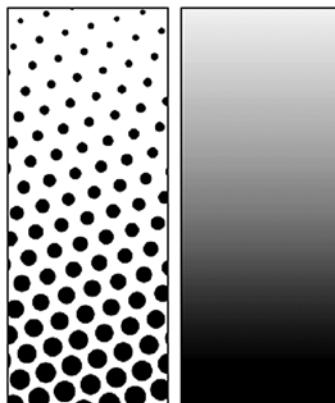


Рис. 11.46. Узор в полутоновой печати

Нам нужен примерно такой же эффект, но экран компьютера не позволяет регулировать размер точек. Придется поискать другой способ. Изменить характеристики отдельной точки (кроме цвета — черного или белого) мы не можем, поэтому переключимся на находящиеся рядом точки. Их нужно отрегулировать так, чтобы глаз воспринимал их как разные оттенки серого. Ради восприятия большего числа оттенков или цветов придется пожертвовать разрешением изображения.

Данный процесс называется *псевдосмещением*, и его увлекательная история начинается еще с аналоговых компьютеров Второй мировой войны. Было замечено, что компьютеры лучше работали на борту летящего самолета, чем на земле. Выяснилось, что случайная вибрация двигателей самолета не позволяла шестеренкам, колесикам, винтам и прочему «залипать» внутри механизма. Впоследствии вибрационные двигатели были добавлены и к наземным компьютерам, чтобы улучшить их работу за счет дрожания. Такая случайная вибрация называется *подмешиванием* (dither, от средневекового английского слова *differen* — «дрожать»). Существует множество алгоритмов псевдосмещения, или дизеринга, и некоторые мы рассмотрим в этой книге.

Основная идея заключается в использовании разных порогов для разных пикселей. В середине 1970-х американский ученый Брюс Байер (Bruce Bayer) (1929–2012) из компании Eastman Kodak изобрел ключевую для цифровых камер технологию — названный впоследствии в его честь *фильтр Байера*. *Матрица Байера* — это одна из версий фильтра, которая как раз подходит для наших задач. Несколько ее примеров представлено на рис. 11.47.

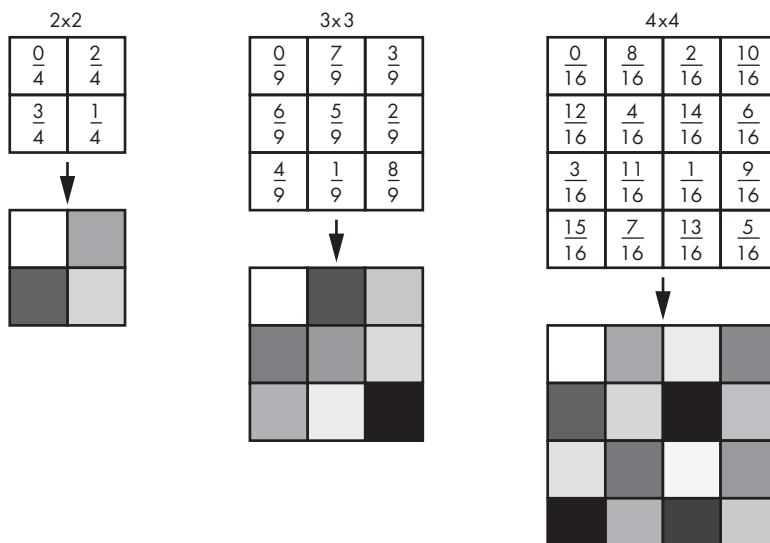


Рис. 11.47. Матрицы Байера

Эти матрицы *выложены* поверх изображения — это означает, что они повторяются в направлениях X и Y , как показано на рис. 11.48. Псевдосмешение с использованием выложенных узоров называется *упорядоченным псевдосмешением*, поскольку в зависимости от положения матрицы на изображении можно предсказать, каким будет узор.

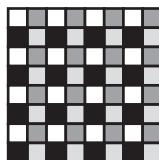


Рис. 11.48. Узор матрицы Байера размером 2×2

В листинге 11.22 представлен псевдокод для матрицы Байера с рис. 11.47.

Листинг 11.22. Псевдокод для упорядоченного псевдосмешения матрицы Байера

```
for (y = 0; y < height; y++)
    for (x = 0; x < width; x++)
        if (value_of_pixel_at(x, y) > bayer_matrix[y % matrix_size][x % matrix_size])
            draw_white_pixel_at(x, y);
        else
            draw_black_pixel_at(x, y);
```

А теперь важный вопрос — что об этом думает кот Тони? На рис. 11.49–11.51 мы разместили его фотографии с использованием псевдосмешения и трех разных матриц.



Рис. 11.49. Псевдосмешение на фотографии Тони с использованием матрицы Байера 2×2



Рис. 11.50. Псевдосмешение на фотографии Тони с использованием матрицы Байера 3×3



Рис. 11.51. Псевдосмешение на фотографии Тони с использованием матрицы Байера 4×4

Если прищуриться, картинка будет выглядеть даже вполне приемлемо — и ее можно улучшить, используя большие матрицы. Конечно, каждый волосок не разглядеть, но получилось все равно лучше, чем при разделении по порогу. Если взять матрицу покрупнее и потратить чуть больше времени, результаты будут еще лучше. Однако в любом случае заметен еще и узор в виде плиток поверх изображения. Кроме того, на некоторых изображениях этот алгоритм вызывает помехи, получившие специальное название — *муаровый* рисунок. Вы поймете, о чём я говорю, если попытаетесь рассмотреть что-то сквозь ряд оконных стекол.

Можно ли уменьшить количество помех на изображении? Сравним каждый пиксель со случайным числом вместо использования определенного узора при помощи псевдокода из листинга 11.23. Результат представлен на рис. 11.52.

Листинг 11.23. Псевдокод для псевдосмешения случайных чисел

```
for (y = 0; y < height; y++)
    for (x = 0; x < width; x++)
        if (value_of_pixel_at(x, y) > random_number_between_0_and_255())
            draw_white_pixel_at(x, y);
        else
            draw_black_pixel_at(x, y);
```



Рис. 11.52. Псевдосмешение на фотографии Тони с использованием случайных чисел

Помех с узором меньше, но изображение стало ворсистым — хотя для кошек это норма. Результат выглядит хуже, чем при упорядоченном псевдосмешении.

Основная проблема всех описанных выше подходов заключается в том, что все решения принимаются на уровне отдельных пикселей. Представим разницу между начальным пикселям и пикселям после обработки. В случае если начальный пиксель не был белым или черным, мы получим некоторую степень ошибки. Вместо того чтобы игнорировать ошибки, как мы делали раньше, распределим их по соседним пикселям.

Начнем с простого. Возьмем ошибку для заданного пикселя и применим ее же к следующему пиксели по горизонтали. Псевдокод представлен в листинге 11.24, а результат его выполнения — на рис. 11.53.

Листинг 11.24. Псевдокод одномерного распространения ошибки

```
for (y = 0; y < height; y++)
    for (error = x = 0; x < width; x++)
```

```

if (value_of_pixel_at(x, y) + error > 127)
    draw_white_pixel_at(x, y);
    error = -(value_of_pixel_at(x, y) + error);
else
    draw_black_pixel_at(x, y);
    error = value_of_pixel_at(x, y) + error;

```



Рис. 11.53. Псевдосмешение на фотографии Тони с использованием одномерного распространения ошибки

Не идеально, но лучше, чем было. Результат явно лучше, чем при разделении по порогу или использовании случайных чисел, и он чем-то напоминает применение матрицы размером 2×2 , только с другим типом помех. Поразмыслив, вы догадаетесь, что распространение ошибки — это еще один пример приема с условной переменной, который мы использовали при рисовании прямых и кривых.

Американские программисты Роберт Флойд (Robert Floyd) (1936–2001) и Луи Стейнберг (Louis Steinberg) нашли новый подход в середине 1970-х, который, по сути, является гибридом распространения ошибки и матрицы Байера. Идея заключается в распределении ошибки по соседним пикселям с использованием набора весов, как показано на рис. 11.54.

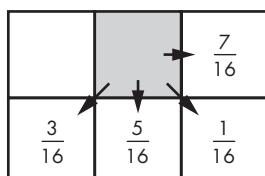


Рис. 11.54. Взвешенное распределение ошибки по методу Флойда — Стейнберга

В листинге 11.25 представлен псевдокод для метода Флойда — Стейнберга. Обратите внимание, что значения ошибок приходится хранить в двух рядах. Каждый из этих рядов намеренно на 2 элемента длиннее, чем нужно, и со сдвигом на 1 элемент. Так мы убедимся в том, что не выйдем за пределы ряда при обработке первой и последней колонок.

Листинг 11.25. Код распределения ошибки по методу Флойда — Стейнберга

```
for (y = 0; y < height; y++)
    errors_a = errors_b;
    errors_b = 0;
    this_error = 0;

    for (x = 0; x < width; x++)
        if (value_of_pixel_at(x, y) > bayer_matrix[y % matrix_size][x % matrix_size])
            draw_white_pixel_at(x, y);
            this_error = -(value_of_pixel_at(x, y) + this_error + errors_a[x + 1]);
        else
            draw_black_pixel_at(x, y);
            this_error = value_of_pixel_at(x, y) + this_error + errors_a[x + 1];

    this_error = this_error * 7 / 16;

    errors_b[x] += this_error * 3 / 16;
    errors_b[x + 1] += this_error * 5 / 16;
    errors_b[x + 2] += this_error * 1 / 16;
```

Приложив гораздо больше усилий, мы получили довольно неплохой результат, представленный на рис. 11.55. (Только не путайте его с алгоритмом «Пинк Флойд» — Стейнберга, который использовался для создания обложек альбомов в 1970-х.)



Рис. 11.55. Псевдосмешение на фотографии Тони с использованием алгоритма Флойда — Стейнберга

Алгоритм Флойда – Стейнберга лег в основу многих других схем распределения, как правило, выполняющих больший объем работы, распределяя ошибку среди большего количества соседних пикселей.

Рассмотрим еще один подход, предложенный голландским программистом Тиадмером Римерсма (Thiadmer Riemersma) в 1998 году. Его алгоритм интересен сразу по нескольким причинам. Прежде всего он возвращается к воздействию только на один смежный пиксель, но в то же время отслеживает еще 16 пикселей с возможной ошибкой. После этого вычисляется средневзвешенное значение, так чтобы последний просмотренный пиксель имел больший вес, чем первый просмотренный. На рис. 11.56 показана кривая весов.

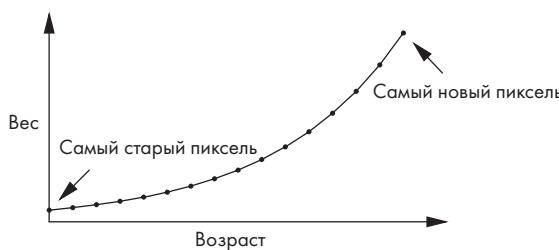


Рис. 11.56. Веса пикселей Римерсма

Алгоритм Римерсма не использует привычную сетку смежных пикселей (листинг 11.26). Вместо нее он следует тем же путем, что и кривая Гилберта на рис. 11.38.

Листинг 11.26. Псевдокод распределения ошибки по алгоритму Римерсма

```
for (каждый пиксель по кривой Гилберта)
    error = средневзвешенное значение последних 16 пикселей

    if (value_of_pixel_at(x, y) + error > 127)
        draw_white_pixel_at(x, y);
    else
        draw_black_pixel_at(x, y);
```

удалить старейшее взвешенное значение с ошибкой
добавить значение с ошибкой из текущего пикселя

Результат выполнения представлен на рис. 11.57. Все еще не за-мур-чательно, но на картинке хотя бы виден кот. Попробуйте применить код из листинга выше к градиенту наподобие представленного на рис. 11.12. Теперь вы узнали, что исходя из реальных условий применяется множество способов обработки приближений.



Рис. 11.57. Псевдосмешение на фотографии Тони с использованием алгоритма Римерсма

Выводы

В этой главе мы разобрали несколько приемов повышения производительности и эффективности, позволяющих избегать вычислений или сокращать их. Как сказал Джим Блайнн (Jim Blinn), один из монстров компьютерной графики: «Техника — это всего лишь уловка, которую вы используете раз за разом». Как и в случае с «кирпичиками», из которых строится аппаратное обеспечение, эти уловки можно объединять для решения сложных задач.

12

Взаимоблокировки и состояния гонки



Мы уже говорили о *многозадачности* — способности компьютеров выполнять несколько действий одновременно. Изначально компьютеры лишь имитировали многозадачность, поскольку могли только переключаться между задачами. Однако сейчас, в эпоху многоядерных процессоров, компьютеры *действительно* могут выполнять одновременно несколько действий. Многоядерные процессоры на самом деле не новость. Мы давно научились объединять несколько одноядерных процессоров для улучшения производительности, но лишь сейчас эта технология стала достаточно простой и популярной. Мультипроцессорные системы внедряются не только в дорогих специальных устройствах, но и в обычных смартфонах.

Иногда нам важен порядок действий. Представьте, что у вас есть объединенный банковский счет, к которому имеют доступ несколько человек. Текущий баланс счета — 100 долларов. Один из владельцев счета направляется к банкомату, чтобы снять 75 долларов, а в это же время вы идете в банк, чтобы снять 50 долларов. Такая ситуация называется *состоянием гонки*. Программное обеспечение банка должно *заблокировать* счет для одного из владельцев и провести только одну операцию снятия наличных за определенный момент времени, чтобы не допустить отрицательного остатка на счету. Таким образом, для определенных операций требуется временно отключать многозадачность. Однако, как вы узнаете из этой главы, это не так просто сделать, не потеряв ее преимуществ.

Что такое состояние гонки?

Состояние гонки возникает, когда две (или более) программы имеют доступ к одним и тем же ресурсам, а результат выполнения программы зависит от времени. Посмотрите на рис. 12.1, где две программы пытаются положить деньги на банковский счет.

Программа 1	Программа 2	Баланс
		\$100
прочитать \$100		\$100
добавить \$10		\$100
записать \$110		\$110
	прочитать \$110	\$110
	добавить \$50	\$110
	записать \$160	\$160

Верный результат

Программа 1	Программа 2	Баланс
		\$100
прочитать \$100		\$100
добавить \$10		\$100
	прочитать \$100	\$100
	добавить \$50	\$100
записать \$110		\$110
	записать \$150	\$150

Неверный результат

Рис. 12.1. Пример состояния гонки

В нашем примере *общий ресурс* — это баланс банковского счета. Как видим, результат просмотра баланса зависит от времени, когда одна из программ обращается к данному ресурсу.

Еще один прекрасный пример — футболка на рис. 12.2.



Рис. 12.2. Гоночная футболка

Общие ресурсы

Какими ресурсами можно поделиться? Да практически любыми. В предыдущем разделе мы рассматривали пример с совместно используемой памятью. Память

всегда участвует в коллективном использовании, даже если не является его конечным результатом. Так происходит потому, что необходим индикатор использования общего ресурса. Память не всегда рассматривается в привычном для нас виде — например, она может быть представлена битами в аппаратном обеспечении устройства ввода/вывода.

Разделение устройств ввода/вывода используется довольно часто (например, совместное использование принтера). Было бы странно, если бы принтер печатал на одном листе фрагменты разных документов. Я уже упоминал в разделе «Пространство системы и пользователя» на с. 178, что операционные системы работают с вводом/выводом для пользовательских программ. На самом деле это относится только к устройствам ввода/вывода, которые являются частью компьютера, например USB-контроллерам. Операционная система лишь проверяет, что устройства, подключенные через USB, работают корректно, но не-посредственный контроль над ними передается пользовательским программам.

Программируемые пользователем вентильные матрицы, или ППВМ (см. «Аппаратное и программное обеспечение» на с. 134) — настоящий Клондайк разделения ресурсов. Можно запрограммировать ППВМ, так чтобы она представляла специальную функцию аппаратного обеспечения для ускорения определенного ПО. Тогда можно быть уверенным, что используется аппаратное обеспечение, предназначеннное исключительно для выбранного ПО.

Это может быть не слишком очевидно, но программы, запущенные на разных компьютерах, также могут делить ресурсы при взаимодействии друг с другом.

Процессы и потоки

Как некоторым программам получать доступ к одним и тем же данным? Мы кратко затронули тему операционных систем в разделе «Относительная адресация» на с. 173. Как раз одна из функций операционной системы — управление многозадачностью.

Операционные системы управляют *процессами* — программами, запущенными в *пространстве пользователя* (см. «Пространство системы и пользователя» на с. 178). Несколько программ могут быть запущены одновременно в системах с многоядерными процессорами, но одного этого условия недостаточно для создания состояния гонки — программы еще и должны разделять между собой ресурсы.

Для разделения ресурсов процессы не используют никакой магии — по крайней мере с тех пор, как Тор вернул Тессеракт в Асгард, — для этого им требуются определенные соглашения. Это подразумевает, что процессы, разделяющие ресурсы, должны взаимодействовать между собой в той или иной форме. Взаимодействие процессов может быть заранее встроено в программу или настраиваться пользователем.

Иногда процесс должен обращать внимание сразу на несколько вещей. Хороший пример — *сервер печати*, программа, с которой могут взаимодействовать другие программы, отправляя документы на печать. До появления компьютерных сетей принтер должен был быть подключен к компьютеру, на котором располагался готовый к печати документ, при помощи порта ввода/вывода. Код компьютерной сети, разработанный в 1980-х в Калифорнийском университете в Беркли, упростил задачу взаимодействия компьютеров, создав несколько системных вызовов. Говоря простыми словами, программа ожидала, пока не активизируется какой-то из нескольких источников, и запускала соответствующий код обработчика. Такой подход сработал в основном потому, что код обработчика был довольно простым и запускался до ожидания новой активности. Код сервера печати мог напечатать целый документ, не ожидая появления следующего.

Все изменилось, когда пришли интерактивные программы с графическим пользовательским интерфейсом. Обработчики активности перестали быть простыми задачами, отрабатывавшими код от начала до конца, — иногда им приходилось останавливаться и ожидать завершения пользовательского ввода из нескольких источников. Программы можно было реализовать в виде кучи взаимодействующих процессов, но это было довольно сложно, потому что приходилось разделять между ними огромное количество данных.

Нужен был способ сделать обработчики прерываемыми — чтобы можно было их останавливать, сохранять в текущем состоянии и перезапускать. Впрочем, в этом не было ничего нового. Где хранится состояние? В стеке. Проблема заключалась в том, что на каждый процесс приходился только один стек, и, по-видимому, нужно было предоставить отдельный стек для каждого обработчика в процессе. Так и появились потоки выполнения. Мы уже изучали, как операционные системы организуют память процесса, в разделе «Организация данных в памяти» на с. 182. *Поток* — это часть программы, которая пользуется разделяемыми статическими данными и кучей, но имеет собственный стек, как показано на рис. 12.3. Каждый поток предполагает наличие отдельного доступа к журналам ЦП, поэтому их сохранением и восстановлением при переключении потоков занимается *планировщик потоков*. Похожим образом ОС переключается между процессами. Потоки иногда называют *облегченными процессами*, потому что они хранят гораздо меньше контекста, чем обычные процессы, и переключение между ними занимает меньше времени.

Ранние реализации потоков включали дополнительный код на ассемблере, который по определению был машинно-ориентированным. Потоки же оказались настолько полезными, что был стандартизован машинно-независимый API.



Рис. 12.3.
Схема памяти
для потоков

Потоки представляют интерес, потому что они обеспечивают состояние гонки внутри отдельного процесса, то есть эта проблема возникает не только в низкоуровневых программах на С — обработчики событий в JavaScript также являются потоками.

Однако потоки не панацея. Злоупотребление потоками может плохо повлиять на пользовательский интерфейс. Когда компания Microsoft впервые представила операционную систему Windows, та являлась, по сути, программой на основе MSDOS. А MSDOS как раз не из тех операционных систем, что следят за трендами и поддерживают многозадачность. В итоге Microsoft пришлось добавить части операционной системы во все приложения, чтобы, к примеру, пользователь мог открыть несколько документов одновременно. К сожалению, некоторые разработчики использовали такой же подход в программах, работающих в готовых операционных системах. Примеры мы видим в приложениях (таких, как LibreOffice и Firefox) и пользовательских интерфейсах (таких, как GNOME) со вкладками.

Чем плоха эта идея? Во-первых, потоки разделяют данные, поэтому возникает проблема с безопасностью. Во-вторых, и наверняка вам это знакомо, проблема или ошибка в одной вкладке часто приводит к завершению всего процесса — в результате можно потерять несохраненные данные. В-третьих (тоже, скорее всего, знакомо), поток, выполнение которого занимает много времени, не позволяет запускать другие потоки — например, долгая загрузка веб-страницы приводит к зависанию сразу нескольких вкладок браузера.

Мораль сей басни такова: пишите код с умом. Используйте операционную систему — она как раз предназначена для таких задач. Если же она не работает так, как нужно, или не предоставляет важную функцию — исправьте это. Просто не вмешивайтесь во все остальное.

Блокировки

В целом проблема не в общих ресурсах. Вопрос в том, как сделать операции *атомарными* (то есть неразделяемыми, непрерываемыми), в то время как они состоят из более мелких операций.

Мы бы не обсуждали все это в книге, если бы компьютеры умели выполнять инструкции наподобие «исправь баланс банковского счета». Но они, естественно, этого не умеют, иначе нам бы понадобились километры инструкций на все случаи жизни. Вместо этого приходится намеренно делать некоторые разделы кода атомарными, используя механизм *взаимного исключения*. Для этого применяются так называемые *рекомендательные блокировки*, чтобы избежать конфликта в программе (рис. 12.4).

Как видно на рис. 12.4, верхняя программа первой блокирует баланс счета, поэтому нижней программе приходится дожидаться снятия блокировки.

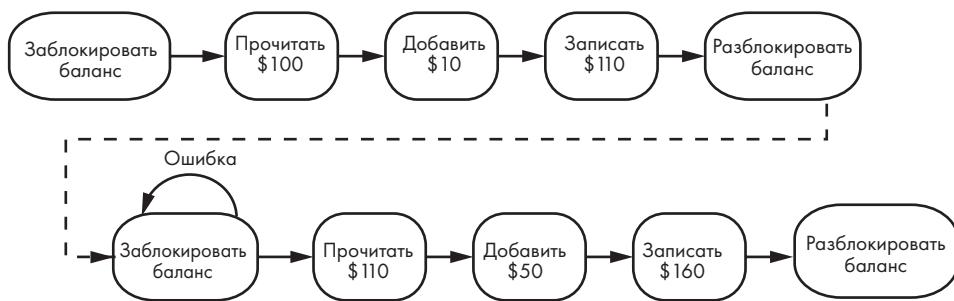


Рис. 12.4. Рекомендательная блокировка

Блокировка называется *рекомендательной*, потому что программы могут ее не использовать — не существует никакого механизма принуждения. На первый взгляд такая блокировка бесполезна, так как она не станет препятствием для грабителя. Однако важно понимать, где именно располагается блокировка — в банке (рис. 12.5), который самостоятельно контролирует ее, так что такая система сработает в нужный момент.

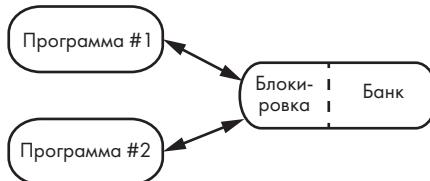


Рис. 12.5. Расположение блокировки

Так мы решили одну проблему, но создали новые. Что произойдет, если взаимодействие между программой № 1 и банком будет медленным? Очевидно, программе № 2 придется ждать, из-за чего потеряется преимущество много задачности. А что произойдет, если программа № 1 аварийно завершится или в ней возникнет ошибка, из-за чего блокировка никогда не будет снята? Что делает программа № 2 во время ожидания?

Рассмотрим все эти проблемы далее.

Транзакции и детализация

Для каждой операции, которую выполняет программа № 1 на рис. 12.5, требуется взаимодействие с банком. Оно должно быть двусторонним, поскольку перед выполнением следующей операции необходимо знать, успешно ли прошла предыдущая. Для улучшения производительности набор из нескольких операций объединяется в так называемую *транзакцию* — группу операций, каждая из

которых завершилась успешно или неуспешно (рис. 12.6). Термин *транзакция* позаимствован из мира баз данных. Вместо того чтобы отправлять результат выполнения каждой операции по отдельности, мы объединяем их.

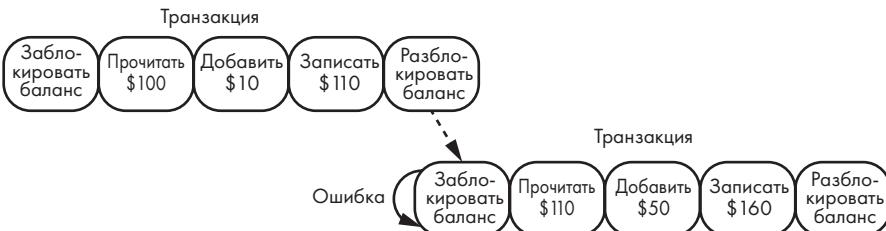


Рис. 12.6. Транзакции

Очевидно, что цель — сокращение времени блокировок, поскольку из-за нее на параллельное выполнение уходит больше времени. Еще одно важное преимущество — уменьшение *детализации* блокировок, то есть количества вещей, на которые влияет данная блокировка. В нашем примере блокируется баланс банковского счета. При этом подразумевается только один счет — нет никакой нужды блокировать всю банковскую систему ради обновления баланса одного из клиентов. X Window System — это один из примеров плохо спроектированной системы блокировок. Существует множество типов блокировок, но в некоторых случаях приходится намеренно блокировать всю систему — тогда время параллельного выполнения увеличивается.

Блокировки, которые затрагивают лишь малую часть системы, называются блокировками с *высокой степенью детализации*, а те, что затрагивают значительную ее часть, — блокировками с *низкой степенью детализации*.

ПРИМЕЧАНИЕ

Обработка прерывания процессора включает в себя механизм блокировки. Для каждого прерывания задается определенная маска, чтобы процессор не получал прерывания того же типа (если это не указано явно), до тех пор пока не завершится обработка.

Ожидание захвата ресурсов

Ни транзакции, ни блокировки с высокой степенью детализации не принесут особой пользы, если программа, ожидающая захвата нужных ей ресурсов, не выполняет полезной работы. Не зря же слово «многозадачность» начинается с «много».

Иногда в самом деле невозможно ничего сделать, пока блокировка не будет снята. Например, вам придется стоять под дождем, пока банкомат не выдаст купюры. Однако бездействовать можно двумя способами. Можно *раскручивать* блокировку в цикле, то есть снова и снова проверять, не снята ли она. Ждущий

цикл часто подразумевает использование таймера, чтобы между попытками проходило какое-то время. Автогонки на полной скорости требуют большего расхода горючего, чем нужно. Та же гонка в вычислительной сети напоминает тщетные попытки толпы людей попасть в магазин в «черную пятницу». В некоторых обстоятельствах — и это второй способ бездействовать — сущность, ожидающая захвата ресурсов, может *зарегистрировать* запрос на получение контроля над ресурсами и *получить уведомление*, когда запрос будет принят. Это позволяет программе выполнять другую работу во время ожидания. Такой подход не очень хорошо масштабируется и не поддерживается архитектурой интернета напрямую, но его можно наложить поверх существующей архитектуры.

Из главы 6 мы узнали, что Ethernet использует интересный подход к ожиданию. Захват ресурсов не применяется, но, если несколько устройств пытаются одновременно получить доступ к разделяемому ресурсу (проводу), им предлагается подождать случайное количество времени и затем попробовать снова.

Некоторые операционные системы предоставляют функциональность захвата ресурсов, которая обычно связана с обработчиком, похожим на дескриптор файла. Захват и освобождение ресурсов происходят при установке блокирующего или неблокирующего режима соответственно. *Блокирующий режим* означает, что система приостанавливает вызывающую программу (то есть прекращает ее выполнение), до тех пор пока не будет доступен нужный ресурс. *Неблокирующий режим* означает, что программа продолжает работать и лишь получает уведомление о том, что ресурс не был захвачен.

Взаимоблокировки

Как вы уже знаете, программам приходится ожидать в течение некоторого времени, пока нужные им ресурсы не будут доступны к захвату. В сложных программах часто используется сразу несколько разделяемых ресурсов. Что произойдет в случае, показанном на рис. 12.7?

Программа № 1 успешно захватывает ресурс А, а программа № 2 — ресурс Б. Затем программа № 1 пытается завладеть ресурсом Б, но не может, потому что в это время он находится у программы № 2. Подобным образом программа № 2 пытается захватить ресурс А, но безуспешно, потому что им владеет программа № 1. Ни одна из программ не может перейти к следующему шагу и освободить захваченные ресурсы. Такая ситуация называется *взаимоблокировкой*.

Проблему взаимоблокировок можно решить несколькими способами (если не учитывать совет «научиться писать хороший код»). В некоторых случаях занятые ресурсы можно освободить вручную, не нанеся при этом большого вреда всей системе. Наверняка вам уже встречались ситуации, когда программа отказывалась работать из-за отсутствия нужных ресурсов, и при желании вы могли

освободить их самостоятельно. Такое случается, если программа, занявшая ресурсы, аварийно завершает работу.

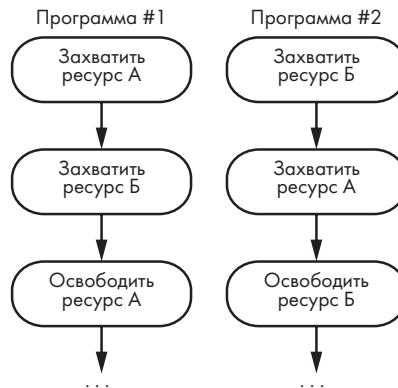


Рис. 12.7. Взаимоблокировка

Реализация кратковременного захвата ресурсов

На самом деле захват ресурсов реализуется только одним способом, а вот предоставить ресурсы программам можно по-разному. Для реализации захвата ресурсов нужно, чтобы аппаратное обеспечение включало в себя специальные команды и тем самым позволяло создавать блокировки. Программные решения, разработанные десятилетия назад, уже не сработают из-за шагнувших далеко вперед технологий процессоров — исполнения с изменением очередности и поддержки нескольких ядер.

Многие процессоры включают в себя команду «*проверить и установить*», созданную специально для реализации блокировок. Эта атомарная команда проверяет, равно ли расположение памяти 0, и, если нет, задает его равным 1. Если удалось успешно сменить значение, возвращается 1, в противном случае — 0. Таким образом напрямую реализуется захват ресурсов.

Еще один вариант реализации захвата в случае, если за ресурсы борются много программ, — «*сравнить и обменять*». Эта команда похожа на «*проверить и установить*», но вместо использования единственного значения вызывающая программа предоставляет как старое, так и новое значения. Если старое значение совпадает с тем, что хранится в расположении памяти, оно заменяется новым, а программа захватывает ресурс.

Использование этих команд обычно ограничивается системным режимом, поэтому пользовательским программам они недоступны. В некоторые из последних языковых стандартов, например C11, была добавлена поддержка атомарных операций на уровне пользователя. Кроме этого, были стандартизированы и добавлены в библиотеки многие операции, связанные с блокировками.

Функциональность захвата ресурсов можно улучшить, добавив дополнительный код. Например, ресурсы можно связать с очередями, чтобы регистрировать программы, находящиеся в ожидании.

Реализация долговременного захвата ресурсов

До этого мы все время подразумевали, что ресурсы захватываются на максимально короткий промежуток времени. Однако иногда необходимо долго удерживать ресурсы — обычно в ситуациях, когда доступ нескольких программ к одному и тому же ресурсу запрещен. Представьте, например, текстовый редактор, который не предназначен для одновременного редактирования текста несколькими пользователями.

Для долговременного захвата ресурсов требуется более постоянное хранение, чем память компьютера. Чаще всего это реализуется при помощи файлов. Существуют системные вызовы, позволяющие создавать подобные файлы: программа, добравшаяся до файла первой, получает исключительный доступ к нему. Можно сказать, что программа таким образом «приобретает» нужный ресурс. Обратите внимание, что системные вызовы представляют собой высоковывесную абстракцию — в них используются атомарные команды.

JavaScript в браузере

При написании программ для работы в браузере следует обращать особое внимание на параллельное выполнение. Если вы знакомы с документацией по JavaScript, вы наверняка удивитесь, потому что JavaScript определяется как однопоточный язык программирования. Так откуда проблемы с параллельным выполнением?

Причина в том, что сегодня язык JavaScript зачастую используется при решении задач, для которых он изначально не был спроектирован. Одна из первых целей создания JavaScript заключалась в том, чтобы предоставить быструю обратную связь для пользователя и сократить интернет-трафик, поскольку интернет был еще слишком медленным. Представьте веб-страницу с полем для ввода номера кредитной карты. До появления JavaScript этот номер приходилось отправлять на веб-сервер, который проверял, состоит ли номер из одних только цифр, и возвращал сообщение об ошибке или продолжал обработку. Благодаря JavaScript веб-браузер может самостоятельно проверить, правильно ли введен номер карты. Это значит, что пользователю не приходится долго ждать ответа и расходовать ценный трафик в случае опечатки. При этом вы наверняка встречали плохой код на JavaScript, который распознает пробелы в номере как опечатки.

JavaScript был создан для запуска небольших программ в ответ на действия пользователя, поэтому он использует модель *цикла событий*, принцип работы которой показан на рис. 12.8.

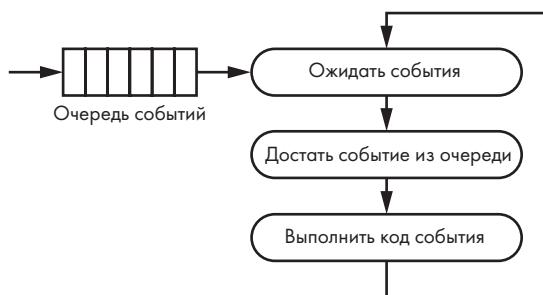


Рис. 12.8. Цикл событий в JavaScript

В этом случае задачи, которые нужно выполнить, помещаются в *очередь событий*. JavaScript по одной достает задачи из очереди и выполняет их. Задачи считаются непрерываемыми, поскольку JavaScript — однопоточный язык. Однако это означает, что программист не может контролировать порядок добавления событий в очередь. Например, представьте, что вы создали обработчики событий для всех кнопок мыши. Порядок нажатия кнопок вы контролировать не можете, как не можете повлиять и на порядок событий. Поэтому программе нужно написать таким образом, чтобы она правильно обрабатывала события в любом порядке.

JavaScript появился в 1995 году, и первые его версии не предусматривали возможности асинхронного взаимодействия. Вплоть до этого момента браузеры отправляли заполненные веб-формы на серверы, а те возвращали соответствующие веб-страницы. Два фактора повлияли на изменение старого способа взаимодействия. Первый — публикация объектной модели документа (Document Object Model, DOM) в 1997 году (несмотря на то, что более-менее стабильная ее версия была выпущена в 2004 году). Благодаря DOM появилась возможность изменять содержимое веб-страниц, а не заменять их полностью. Второй фактор — появление в 2000 году XMLHttpRequest (XHR), который лег в основу AJAX. Интерфейс XHR и фоновое взаимодействие браузера и сервера пришли на смену существующей модели загрузки страницы.

Из-за этих изменений сложность веб-страниц значительно возросла. С тех пор написано столько кода на JavaScript, что он стал действительно популярным языком программирования. Веб-страницы стали все чаще использовать фоновое асинхронное взаимодействие с серверами. Однако JavaScript не мог в полной мере соответствовать ожиданиям, так как не был предназначен для решения конкретно этих задач, в первую очередь потому, что однопоточная модель противоречила асинхронному взаимодействию.

Набросаем простое веб-приложение для сортировки картин в альбоме по художникам. Будем использовать это приложение на воображаемом веб-сайте. Для этого нужно превратить название альбома и имя художника в идентификатор

альбома, чтобы использовать этот идентификатор для получения выбранной картины. Здесь может пригодиться код программы из листинга 12.1 (фрагменты, выделенные курсивом, означают пользовательский ввод).

Листинг 12.1. Программа для альбома с картинами — первый вариант

```
var album_id;
var album_art_url;

// Отправить имя художника с названием альбома на сервер и получить ID альбома

$.post("some_web_server", { artist: имя_художника, album: название_альбома },
       function(data) {
    var decoded = JSON.parse(data);
    album_id = decoded.album_id;
});

// Отправить идентификатор альбома на сервер и получить URL изображения
// для выбранной картины из альбома.
// Добавить тег с изображением в документ для отображения картины из альбома

$.post("some_web_server", { id: album_id }, function(data) {
    var decoded = JSON.parse(data);
    album_art_url = decoded.url;
});
$(body).append('');
```

Функция `post` в языке jQuery отправляет данные (второй параметр функции) на URL (первый параметр) и, получив ответ, вызывает функцию (третий параметр). Обратите внимание: функция при этом не вызывается напрямую, а помещается в очередь событий и вызывается только в определенном порядке.

На первый взгляд программа кажется симпатичной, простой и упорядоченной. Однако она не очень надежна. Почему? Разберем подробно, что она делает. Взгляните на рис. 12.9.

Как видим, программа не выполняется в ожидаемом порядке. Операции `post` внутренним образом запускают потоки, которые ожидают ответа с сервера. При получении ответа функции обратного вызова помещаются в очередь событий. Показано, что ответ для второй функции `post` получен первым, но точно так же можно получить первым и ответ для первой функции `post`.

Существует вероятность, что программа запросит картину из альбома, до того как будет получен `album_id` из первой функции `post`. И почти наверняка она попытается добавить изображение на веб-страницу до получения `album_art_url`. Так происходит потому, что, несмотря на однопоточность JavaScript, обработка данных на сервере выполняется параллельно. Иначе говоря, интерпретатор JavaScript предоставляет программисту однопоточную модель, но сама ее суть многопоточна.

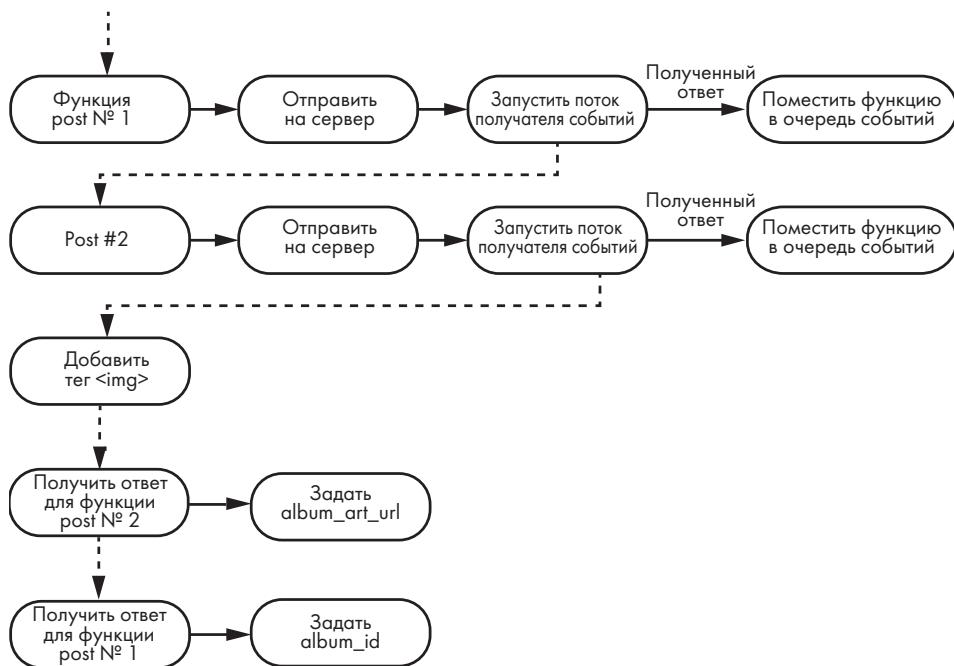


Рис. 12.9. Поток выполнения программы для альбома с картинами

В листинге 12.2 представлен рабочий вариант программы.

Листинг 12.2. Программа для альбома с картинами — второй вариант

```

$.post("some_web_server", { artist: имя_художника, album: название_альбома },
function(data) {
  var decoded = JSON.parse(data);

  $.post("some_web_server", { id: decoded.id }, function(data) {
    var decoded = JSON.parse(data);
    $(body).append('');
  });
});
  
```

В этот раз функцию `append` для изображения переместили в функцию обратного вызова для второй операции `post`, а ту, в свою очередь, — внутрь функции обратного вызова первой операции `post`. Это означает, что вторая операция `post` будет запущена только после выполнения первой.

Такая вложенность необходима, чтобы убедиться в наличии всех зависимостей. Если добавить обработку ошибок, код станет еще хуже. В следующем разделе мы рассмотрим способ исправить это.

Асинхронные процессы и промисы

С программой из листинга 12.2 все в полном порядке — она работает как надо, потому что функция `post` в jQuery реализована корректно. Однако нельзя сказать то же самое про остальные библиотеки, особенно относящиеся к Node.js, — число плохих библиотек на его основе растет со стремительной скоростью. Очень сложно отладить программу, если используемые в ней библиотеки неправильно реализуют функции обратного вызова. Это вылилось в настоящую проблему, потому что, как я уже упомянул во введении к этой книге, к программированию сейчас подходят как к склеиванию функций из разных библиотек.

Недавно JavaScript попытался решить эту проблему, введя новую конструкцию под названием *промис*. Впервые о концепции промисов мы услышали в середине 1970-х, а добавление в JavaScript ознаменовало ее возрождение. Благодаря промисам механизм асинхронных функций обратного вызова переместился напрямую в язык JavaScript, и потому сломать его при помощи библиотек стало несколько сложнее. Конечно, не стоит ожидать изменений в языке в ответ на любую ошибку программиста, но проблема с функциями обратных вызовов стала настолько частой, что потребовалось глобальное решение.

Объяснение промисов в JavaScript может показаться запутанным, потому что оно связывает два несопоставимых компонента. Проще разобраться в промисах, рассмотрев эти компоненты по отдельности. Важнее всего то, что использование промисов благотворно влияет на работу библиотек для асинхронных операций. Вторая, менее важная задача промисов (в последнее время говорить о ней стали чаще) заключается в изменении самой парадигмы программирования. Мы еще затронем эти парадигмы и связанные с ними «культы» в последней главе. В некотором смысле конструкция промиса представляет собой *синтаксический сахар* — подсластитель, который упрощает определенные виды программирования, но утяжеляет сам язык.

Бесконтрольное нагромождение асинхронных запросов в JavaScript приводит к тому, что код начинает выглядеть как так называемая *пирамида судьбы*, как показано в листинге 12.3. Лично я не вижу ничего плохого в такой записи. Если вас смущают отступы, держитесь подальше от языка Python — он повергнет вас в шок.

Листинг 12.3. Пирамида судьбы

```
$.post("server", { parameters }, function() {  
  $.post("server", { parameters }, function() {  
    $.post("server", { parameters }, function() {  
      $.post("server", { parameters }, function() {  
        ...  
      });  
    });  
  });  
});
```

Конечно, такой вид отчасти обусловлен способом написания программы. Анонимные функции подразумевают, что тело функции следует сразу за ее объявлением. Код можно переписать, избавившись от пирамиды судьбы, как показано в листинге 12.4, но читать его будет не слишком удобно.

Листинг 12.4. Перепишем код, избавившись от анонимных функций

```
$.post("some_web_server", { artist: имя_художника, album: название_альбома }, got_id);

function
got_id(data)
{
  var decoded = JSON.parse(data);
  $.post("some_web_server", { id: decoded.id }, got_album_art);
}

function
got_album_art(data)
{
  var decoded = JSON.parse(data);
  $(body).append('');
}
```

На самом деле нужен более простой способ писать код. В других языках это сделать проще — все усложняет однопоточная модель выполнения в JavaScript. Если бы JavaScript был многопоточным языком, можно было просто создать новый поток для запуска кода из листинга 12.5. Подразумевается, что функция `post` блокирует выполнение остального кода до своего завершения, то есть является синхронной. Такой код просто выглядит, и его легко прочитать.

Листинг 12.5. Пример возможной блокировки в JavaScript

```
var data = $.post("some_web_server", { artist: имя_художника, album: название_альбома } );
var decoded = JSON.parse(data);

var data = $.post("some_web_server", { id: decoded.id }, got_album_art);
var decoded = JSON.parse(data);

$(body).append('');
```

Подобный код в JavaScript просто не сможет работать хорошо. Однопоточная природа JavaScript не позволит выполниться всему остальному коду, пока не завершатся функции `post`, а значит, обработчики событий для нажатия кнопок мыши и других пользовательских взаимодействий очень долго будут ждать запуска.

Промисы в JavaScript реализуются как в листинге 12.4 — определение промиса подобно определению функции, то есть отделено от его выполнения.

Промис создается, как показано в листинге 12.6. Этот код почти не отличается от фрагментов на JavaScript, рассмотренных выше, и промис похож на функцию `post` в jQuery, принимающую функцию как параметр. Отличие заключается в том, что функция промиса не выполняется, поскольку мы рассматриваем здесь только фазу создания промиса.

Листинг 12.6. Создание промиса

```
var promise = new Promise(function(resolve, reject) {
  if (все действия завершились успехом)
    resolve(возвращаемое_значение);
  else
    reject(возвращаемое_значение);
});
```

Рассмотрим этот пример подробнее. В промисе определяется функция, выполняющая некоторую асинхронную операцию. Данная функция принимает два параметра, которые также являются функциями: первый (`resolve` в листинге 12.6) помещается в очередь событий в JavaScript при успешном выполнении асинхронной операции, а второй (`reject` в листинге 12.6) помещается в очередь в случае ошибки.

Программа запускает промис при помощи его метода `then`, как показано в листинге 12.7. Данный метод принимает пару функций в качестве параметров, которые соотносятся с функциями `resolve` и `reject`, определенными при создании промиса.

Листинг 12.7. Выполнение промиса

```
promise.then(
  function(value) {
    операция с возвращаемым_значением из resolve
  },
  function(value) {
    операция с возвращаемым_значением из reject
  }
);
```

Не впечатляет. То же самое можно сделать и без новшеств — мы и раньше писали подобный код. Стоит ли заморачиваться? Одно из преимуществ промисов заключается в использовании синтаксического сахара под названием *цепочка промисов*. Благодаря этому можно написать код в виде `что-то().then().then().then()...`. Это сработает, потому что метод `then` возвращает новый промис. Обратите внимание: как и в случае с исключениями, второй параметр в функции `then` можно опустить, обработав ошибки при помощи блока `catch`. В листинге 12.8 представлен вариант программы для альбома с картинами с использованием цепочек промисов.

Листинг 12.8. Программа для альбома с картинами с использованием цепочек промисов

```
function
post(host, args)
{
  return (new Promise(function(resolve, reject) {
    $.post(host, args, function(data) {
      if (success)
        resolve(JSON.parse(data));
      else
        reject('failed');
    });
  }));
}

post("некий-веб-сервер", { artist: имя_художника, album: название_альбома })
.then(function(data) {
  if (data.id)
    return (post("некий-веб-сервер", { id: data.id }));
  else
    throw ("nothing found for " + artist_name + " and " + album_name);
}).then(function(data) {
  if (data.url)
    $(body).append('');
  else
    throw (`nothing found for ${data.id}`);
}).catch(alert);
```

Не думаю, что этот код легче читать, чем его вариант с пирамидой судьбы, но вы можете считать иначе. Варианты кода из листингов 12.2 и 12.8 иллюстрируют другой аспект искусства программирования: компромиссы между простотой разработки и обслуживания. В общей схеме жизненного цикла продукта простота обслуживания важнее, чем написание кода в привычном разработчику стиле. Я еще вернусь к этой мысли в главе 15. Объединение промисов в цепочки позволяет писать код в виде `function().function().function()`... в отличие от пирамиды судьбы. В первом случае проще проверить, не потерялись ли фигурные скобки. Однако язык JavaScript — в отличие, например, от Ruby — был разработан с использованием второго способа. Два разных стиля написания кода в одном языке приводят к путанице и снижению производительности разработчиков. Благодаря промисам количество ошибок программирования определенного класса сокращается, но не стоит воспринимать промисы как панацею от плохо написанного кода.

Промисы — это синтаксический сахар, позволяющий уменьшить количество вложенных блоков кода. Однако, если вы действительно хотите сделать код удобочитаемым, вы наверняка предпочтете что-то похожее на листинг 12.5. В JavaScript существует еще один способ написания «асинхронных» программ на основе промисов, отражающий стиль синхронного программирования: `async` и `await`.

В листинге 12.9 представлен вариант программы для альбома с картинами с использованием `async` и `await`.

Листинг 12.9. Программа для альбома с картинами с использованием `async` и `await`

```
function
post(host, args)
{
  return (new Promise(function(resolve, reject) {
    $.post(host, args, function(data) {
      if (success)
        resolve(JSON.parse(data));
      else
        reject('failed');
    });
  }));
}

async function
get_album_art()
{
  var data = await post("некий-веб-сервер, { artist: имя_художника,
  album: название_альбома } ");

  if (data.id) {
    var data = await post("некий-веб-сервер, { id: data.id });

    if (data.url)
      $(body).append('');
  }
}
```

Мне кажется, что этот код выглядит проще, чем фрагмент в листинге 12.8.

Конечно, необходимо понимать, что все описанное выше заметно искажает, если и вовсе не ломает однопоточную модель JavaScript. Асинхронные функции, по сути, представляют собой непрерывные потоки.

Выводы

В этой главе вы узнали о некоторых проблемах, связанных с использованием разделяемых ресурсов. Вы познакомились с состоянием гонки и взаимоблокировками, с процессами и потоками. Также вы немного погрузились в тему параллельного выполнения в JavaScript и способов его реализации. Таким образом, мы заложили основу для понимания более сложных концепций — концепций безопасности, — о которых поговорим в следующей главе.

13

Безопасность



Тема безопасности сложна для понимания. В частности, аспекты, касающиеся криптографии, предполагают заумную математику. Однако важно рассмотреть хотя бы основы безопасности. В данной главе мы не будем изучать подробности, а лишь проведем краткий обзор — этих знаний не хватит, чтобы стать экспертом в области безопасности, но будет достаточно, чтобы задавать вопросы о жизнеспособности тех или иных решений в этой сфере.

Не нужно быть экспертом, чтобы писать достаточно безопасный код.

В целом компьютерная безопасность не слишком отличается от того же понятия в других сферах, скажем безопасности жилища. Во многих отношениях появление компьютерных сетей позволило перейти в вопросах безопасности от маленькой квартиры к дворцу. Как вы понимаете, в огромном дворце больше входов, которым нужна защита, и обитателей, которые могут ее скомпрометировать. Чем больше дворец, тем больше мусора в нем накапливается, тем сложнее поддерживать чистоту и тем больше в нем мест, где могут спрятаться тараканы.

Основная задача безопасности — обеспечивать достаточную защиту имущества (понятие *защиты* определяете *вы сами*). Так, безопасность перестала быть технологической проблемой — теперь это проблема социальная. Вы, ваше имущество и ваше определение защиты существуют с другими пользователями, их имуществом и определениями защиты.

Безопасность и конфиденциальность тесно связаны между собой, отчасти потому, что безопасность начинается с обеспечения конфиденциальности. Например, ваш банковский счет не был бы безопасным, если бы пароль от него мог получить любой человек. Поддерживать конфиденциальность на должном уровне сложно, учитывая количество странных практик в организациях, с которыми

нам приходилось работать. Каждый раз, когда я прихожу к новому врачу, его клиника запрашивает все мои личные данные. Я всегда спрашиваю: «Зачем вам эта информация?» Они отвечают: «Для защиты вашей конфиденциальности». На что я всегда говорю: «Каким образом предоставление всех моих персональных данных по любому вопросу обеспечит защиту моей конфиденциальности?» Они раздраженно вздыхают и произносят: «Нам просто нужны ваши данные». И даже если это не так, они не обязаны говорить вам правду. Сегодня на конфиденциальность влияет также простота, с которой можно собрать воедино разрозненные фрагменты информации (подробнее об этом мы поговорим в следующей главе) благодаря повсеместному распространению технологий для сбора данных: камер наблюдения, автоматических систем считывания номерных знаков, отслеживания мобильных телефонов, в том числе ловцов IMSI (перехватчиков сигнала), перехвата интернет-коммуникаций (комната 641А¹), распознавания лиц и т. д. Обеспечить конфиденциальность и, соответственно, безопасность, невероятно сложно.

Безопасность — непростая тема. Ее как нельзя лучше описывает старая поговорка: «Прочность цепи определяется прочностью ее самого слабого звена». Представьте себе систему онлайн-банкинга. Она состоит из множества компонентов, включающих в том числе аппаратное и программное обеспечение, сети передачи данных и персонал. Однако даже лучшие технологии не смогут вас защитить, если вы оставите пароль от аккаунта на листочке у компьютера!

Обзор безопасности и конфиденциальности

В этой главе я кратко познакомлю вас с проблемами безопасности и конфиденциальности. Многие из введенных здесь терминов мы рассмотрим более детально в следующих разделах.

Модель угроз

Если бы угроз не существовало, нам бы не пришлось говорить о безопасности. Когда все ведут себя хорошо, беспокоиться не о чем, — но в жизни так не бывает.

Безопасность нельзя изучать в вакууме — она всегда связана с *моделью угроз*, которая определяет сущности, нуждающиеся в защите, и перечисляет возможные способы взлома этих сущностей. Таким образом, можно заранее разработать соответствующие средства защиты. Вопреки кажущейся простоте «умных устройств» — телевизоров с подключением к интернету, камер наблюдения, ламп накаливания и др., — для создания рабочей модели угроз недостаточно одного вопроса «Что может пойти не так?».

¹ Комната 641А — помещение в здании магистрального провайдера AT&T, использовавшееся для перехвата интернет-телефонии в интересах Агентства национальной безопасности. — Примеч. науч. ред.

Например, незадолго до того как я написал эту книгу, компания Fender выпустила гитарный усилитель с возможностью подключения по Bluetooth. Однако они не удосужились реализовать протокол связывания Bluetooth, который обеспечил бы безопасность беспроводного соединения между гитарой музыканта и усилителем. Это привело к тому, что некоторые умельцы подключались к усилителю на сцене с помощью обычного мобильного телефона, включая все, что им заблагорассудится. Позже это может превратиться в новое направление искусства, но вряд ли компания Fender изначально ставила перед собой подобную цель.

Понимание модели угроз важно, потому что обеспечить стопроцентную безопасность невозможно. Для каждой модели угроз разрабатываются отдельные методы защиты. Например, чтобы уберечь рюкзак с учебниками от кражи, можно нанять личного вооруженного охранника, но, скорее всего, это ударит по вашему кошельку, да и администрация учебного заведения посмотрит косо. Для устранения этой конкретной угрозы вполне подойдет обычный запирающийся шкафчик.

Вот еще пример: я живу на ферме в богом забытом месте. Я могу навесить на входную дверь десяток дорогих замков, но, если злоумышленник решит прорваться прямо сквозь стену с цепной пилой или динамитом, они меня не спасут. Никто из соседей не обратит внимания на подозрительные звуки, потому что, живя за городом, слышишь такое каждый день. Конечно, у меня есть дверные замки, но в систему безопасности входит и хорошая страховка, потому что защитить имущество одними лишь физическими средствами мне не по карману.

Многие мои соседи этого не понимают и своими действиями лишь ухудшают безопасность собственного жилища. К сожалению, зачастую люди, переехавшие за город, тут же устанавливают фонари на участках. Я спрашивал у многих, зачем они так делают, ведь один из плюсов жизни за городом — это как раз возможность видеть звездное небо, не скрытое за светом уличных фонарей. Ответ всегда один и тот же: «В целях безопасности». Я пытался объяснить, что все эти фонари лишь сигнализируют взломщикам: тут есть чем поживиться и хозяев нет дома.

Губительные средства защиты нередко встречаются и в мире компьютеров. Например, многие организации намеренно вводят правила создания паролей и частоты их обновления. В результате сотрудники выбирают легко угадываемые пароли или просто записывают их, чтобы не забыть.

Суть в том, что обеспечить эффективную защиту невозможно без определения модели угроз. Стоит соблюдать баланс между угрозами и средствами защиты. Наша цель — недорогие средства защиты, взлом которых дорого обойдется злоумышленникам. Появление интернета привело к тому, что стоимость взлома снизилась, а стоимость средств защиты — нет.

Доверие

Одна из самых сложных задач определения модели угроз — понять, чему можно доверять. В далекие времена доверие проверялось во время разговоров с глазу

на глаз, хотя иногда люди все же попадались на уловки харизматичных проходимцев. В современном мире намного сложнее выяснить, кому именно можно доверять. Можете ли вы определить надежность точки доступа Wi-Fi, посмотрев её в глаза? Вряд ли, даже если вы эти самые глаза найдете.

Если вы когда-нибудь просили друзей хранить что-то в секрете, вы знаете, насколько доверие важно. Шанс того, что друг вас не выдаст, — 50 на 50. Согласно теории вероятности, риск раскрытия секрета составит 75 %, если вы поделитесь им с двумя друзьями. Риск того, что секрет перестанет быть секретом, возрастает с каждым новым другом: 87 %, если поделиться с тремя друзьями, 94 % — с четырьмя, 97 % — с пятью и т. д. Очевидно, что чем больше вы полагаетесь на то, что не можете контролировать, тем хуже обстоит дело с безопасностью — все плохо начинается, а заканчивается и того хуже.

В случае с друзьями вы сами выбираете, кому можно доверять. Возможность такого же выбора в сетевом мире компьютеров сильно ограничена. Например, если вы одна из тех редких птиц, что читает условия договора до того, как их принять, вы наверняка заметили, что почти все такие документы содержат примерно одну и ту же фразу: «Ваша конфиденциальность очень важна для нас. Поэтому вы освобождаете нас от ответственности за ее нарушение». Звучит не очень надежно. Однако у вас нет выбора, если вы собираетесь пользоваться предложенными услугами.

В мире компьютерной безопасности термин *доверие* относится к компонентам, на которые приходится полагаться за неимением выбора. Ваша безопасность зависит от безопасности этих компонентов. Мы уже обсудили, что число таких компонентов лучше свести к минимуму ради все той же безопасности.

Пользуясь компьютером, вы полагаетесь на огромное количество сторонних устройств и программ. У вас нет доступа к каждой из них, и потому вам приходится слепо им доверять, даже если они не сделали ничего, чтобы ваше доверие заслужить. И даже если бы у вас был такой доступ, хватило бы вам знаний и времени, чтобы проверить абсолютно всё?

Понятие доверия всплывает снова и снова. А пока рассмотрим три класса его нарушений:

- **Преднамеренное.** Примеры такого нарушения — появившийся в 2005 году *руткит* (ПО для обхода защиты), который был установлен компанией Sony BMG на пользовательских компьютерах, и всплывающее сообщение, распространявшее *вредоносное ПО* на ноутбуках Lenovo пару лет назад. Эти программы не из тех, что были случайно установлены пользователями, — их установили сами поставщики компьютеров.
- **Некомпетентное.** Примеры некомпетентности включают в себя незашфрованные беспроводные датчики давления в шинах, которые позволяют отслеживать автомобиль, незашфрованные RFID-метки в новых

паспортах США, которые упрощают слежение за всеми, кто имеет такой паспорт, или предлагаемые стандарты связи между транспортными средствами, введенные ради «безопасности», — они лишь упростят загрузку вредоносной информации в связанные устройства. Даже не имея пароля администратора, хакеры нашли способ получить доступ к огромному числу роутеров Wi-Fi и изменили их настройки. Один из самых опасных примеров — случай с компанией Siemens, когда в их промышленных системах управления был обнаружен запрограммированный пароль. Это означает, что любой, кто его знал, мог получить доступ к оборудованию, которое считалось защищенным. Запрограммированный пароль нашли и в некоторых продуктах компании Cisco. Одна из крупнейших DDoS-атак (мы обсудим ее далее) выявила использование стандартных паролей в IoT-устройствах, выпущенных компанией Hangzhou XiongMai. К сожалению, все это возвращает нас к модели угроз по вопросу «Что может пойти не так?» и мышлению типа «безопасность через неясность» (поговорим об этом немного позже).

- **Расчетливое.** Возникает, когда кто-то откровенно лжет. Более подробно мы поговорим об этом в разделе «Социальный контекст» на с. 428. Хороший пример — ситуация, с которой столкнулись сотрудники американского Национального института стандартов и технологий (National Institute of Standards and Technology, NIST). Они разрабатывали новые стандарты шифрования при поддержке «экспертов» из Агентства национальной безопасности США (АНБ). Выяснилось, что эксперты из АНБ умышленно занижали уровень безопасности стандарта. Так им было проще шпионить, а другим злоумышленникам — взламывать банковские счета. Нарушения доверия встречаются настолько часто, что для описания класса нарушений, когда злоумышленник тайно и безопасно крадет нужные ему данные, был придуман специальный термин — *клептография*.

Выражение *безопасность через неясность* используется для классификации утверждений наподобие «это безопасно, потому что секретный ингредиент — внезапно — хранится в секрете». Не раз доказывалось, что это не так. На самом деле лучшая безопасность обеспечивается путем *прозрачности и открытости*. Чем больше людей знают о применяемых методах безопасности, тем чаще эти методы обсуждаются и тем больше их недостатков можно обнаружить. История подтверждает, что никто не совершенен и что невозможно продумать все заранее. В компьютерном программировании мы иногда называем это *принципом тысячи глаз*. Статистика отрасли показывает, что в ОС Windows в сто раз больше критических уязвимостей, чем в Linux.

Все это непросто. Иногда на обнаружение проблем безопасности уходят годы или даже десятилетия, даже если этим занимаются лучшие умы. Например, недавно найденные уязвимости Spectre и Meltdown исходят из решений по проектированию архитектуры ЦП, принятых еще в 1960-х годах.

Физическая безопасность

Представьте себе школьный шкафчик. Он нужен, чтобы хранить там вещи и не беспокоиться, что кто-то другой имеет к ним доступ. Он создан из довольно прочной стали и спроектирован так, чтобы его было сложно взломать. Те, кто работает с безопасностью, назвали бы дверь шкафчика *поверхностью атаки* — именно ею займется злоумышленник, пытающийся вскрыть шкафчик. Шкафчик достаточно защищен от угрозы воровства, потому что взломать его бесшумно просто невозможно. В течение школьного дня рядом обычно находится множество учеников — скорее всего, они заметят, если что-то пойдет не так. Конечно, взломать шкафчик можно и после уроков, но вряд ли в это время в нем будут храниться ценные вещи.

Кодовый замок на двери открывается, только если ввести правильный код, обычно известный только владельцу шкафчика. Вместе с кодом от двери школа наделяет владельца *авторизацией* для открытия конкретно этого шкафчика. Замок — это еще одна поверхность атаки. Замок спроектирован таким образом, что он не откроется, даже если отломать циферблат, поэтому после закрытия шкафчика добраться до внутренностей замка сложно. Здесь возникает очередная проблема — необходимость запоминать секретную комбинацию цифр. Если записать код на листочке, его может обнаружить кто-то еще. Открывая шкафчик, нужно убедиться, что никто не подсматривает, как вы вводите код. И, как вы уже знаете из фильмов, взломщики умеют вскрывать кодовые замки, а школы обычно экономят на замках. А еще существуют *устройства автонабора* — их можно прикрепить к кодовому замку для перебора всех возможных комбинаций цифр. Это могут быть специальные устройства или собранные народными умельцами маленькие недорогие компьютеры на основе Arduino и дешевых шаговых двигателей. Но, как и в случае с дверью шкафчика, в школьном коридоре постоянно шатается кто-то из учеников, так что попытка взломать замок не пройдет незамеченной. Чтобы открыть шкафчик, нужен либо талант взломщика, либо плохой замок (многие замки надежны только с виду). Кодовый замок одного из популярных брендов можно взломать меньше чем за минуту, если потратить совсем немного времени, чтобы изучить его устройство.

Есть и третья поверхность атаки, и ее можно не заметить. Это отверстие для ключа в замке. Специалисты в области безопасности назвали бы его *лазейкой* или «задней дверью» (хотя в нашем случае это как раз передняя дверь). Замочная скважина — еще один способ добраться до шкафчика без ведома его владельца. Для чего это нужно? В администрации школы, очевидно, знают секретную комбинацию цифр, открывающую шкафчик, но отверстие для ключа гарантирует, что при необходимости можно быстро открыть шкафчик любого ученика. Это серьезный удар по безопасности — замки со скважинами можно взломать за считанные секунды. И поскольку ко всем замкам подходит один ключ, то стоит злоумышленнику получить его копию (это не так сложно, как может показаться), как под угрозой окажутся сразу все шкафчики в школе.

Передав вам код от шкафчика, администрация школы тем самым предоставила вам *полномочие* — возможность открыть шкафчик. Сотрудник с ключом имеет более высокий *уровень полномочий*, так как ему разрешено открывать все шкафчики. Наличие доступа к копии ключа повысит ваш уровень полномочий. Многие подающие надежды инженеры, в том числе автор этой книги, в юности нашли способ стать «уполномоченными», открыв для себя слесарное дело.

Безопасность связей

Мы узнали, как можно защитить имущество, — пришло время более сложных задач. Как передать другим что-то, что принадлежит вам? Начнем с простого примера. Скажем, вы получили домашнее задание — подготовить доклад о созвездии Орион, но вам придется пропустить урок, потому что вы собираетесь к врачу. В коридоре вы встречаете своего друга Эдгара и просите его сдать домашнее задание за вас. Выглядит все достаточно просто.

Первый шаг данного процесса называется *аутентификацией*. Вы должны убедиться, что человек, которому вы передаете домашнее задание, действительно ваш друг Эдгар. Например, в спешке вы можете забыть, что у Эдгара есть злой брат-близнец. Или «Эдгар» может оказаться кем-то, кто лишь надел такой же костюм, как у Эдгара. Вряд ли вы захотите случайно аутентифицировать Эдгара-жука вместо Эдгара-человека (см. фильм «Люди в черном» (1997))!

Эдгари-самозванцы не единственная поверхность атаки. Как только домашнее задание оказывается не в ваших руках, все резко меняется. Теоретически теперь Эдгар должен действовать в ваших интересах. Но он может прозевать урок и забыть сдать доклад. Злой близнец Эдгара мог изменить домашнее задание, заменив некоторые ответы на неправильные, или, что еще хуже, сдать копию чужого доклада вместо вашего. Невозможно доказать *подлинность* работы — что Эдгар сдал именно то, что вы ему передали. Если бы вы все спланировали заранее, то могли бы положить домашнее задание в конверт, защищенный восковой печатью. Но, конечно, такие конверты можно несколько раз открывать и запечатывать заново, не оставляя следов.

Все становится еще сложнее, если у вас нет аутентифицированного, *доверенного* курьера для передачи домашнего задания. Возможно, возникла непредвиденная ситуация, и учитель предложил вам отправить домашнее задание по почте. Ваша посылка может пройти через руки большого количества людей, а потому она становится уязвимой для *атаки через посредника* — когда злоумышленник находится где-то между заинтересованными сторонами и может вмешиваться в их взаимодействие или изменять его результат. Вы не знаете, кто именно будет передавать посылку, и потому даже не можете провести аутентификацию.

Все эти проблемы решаются с помощью *криптографии*. Можно *зашифровать* сообщение, применив секретный код, известный только вам и получателю. Получатель сможет *расшифровать* сообщение, используя секретный код. Конечно, как

и в случае с комбинацией цифр от школьного шкафчика, секретный код должен оставаться секретным. Коды также можно взломать, и вы останетесь в неведении, если кто-то узнает или взломает ваш код. Правильно спроектированная *крипtosистема* позволяет отказаться от обязательных доверенных компонентов между сторонами — даже если сообщение будет перехвачено, расшифровать его будет непросто, что уменьшает возможные риски.

В случае взлома кода стороны просто меняют код на новый. Интересно, что во время Второй мировой войны использовались различные приемы для маскировки действий, возникающих в результате взлома кодов. Например, отправка самолета для «случайного» обнаружения перемещений флота и последующей атаки на него скрывала, что взлом кода фактически заключался в способе определения местоположения флота. Если вас заинтересовал этот тип информационной безопасности, рекомендую прочитать роман Нила Стивенсона (Neal Stephenson) «Криптономикон».

Наше время

Эпоха «подключенных компьютеров» объединяет проблемы физической безопасности с проблемами безопасности обмена данными. Психоделический ковбой, поэт, автор текстов и футурист Джон Перри Барлоу (John Perry Barlow) (1947–2018) на конференции SIGGRAPH в 1990 году заметил, что «киберпространство — это место, где находятся ваши деньги». И речь не только о купюрах. Раньше люди покупали музыку на пластинках или компакт-дисках, а фильмы — на видеокассетах или DVD. Теперь эти развлечения превратились в биты на компьютере. И конечно же, банковские операции также перешли в онлайн.

Одно дело, если бы эти биты просто хранились на компьютерах. Но компьютеры, как и телефон, подключены к интернету. Поверхность атаки здесь настолько большая, что стоит смирииться с тем, что доверие будет нарушено как минимум в одном месте. Злоумышленники же практически невидимы.

В позабытом прошлом, если кто-то хотел досадить вам, он мог позвонить в дверь и убежать. Если же вы оказывались в нужное время в нужном месте, то вы даже могли поймать хулигана на месте преступления. И, разумеется, провернуть подобное можно было лишь пару раз за день. В интернете же вы мало что можете сделать, даже если личность злоумышленника вам известна. Чаще всего злоумышленники даже не люди — это компьютерные программы, которые могут пытаться взломать ваши устройства тысячи раз в секунду. Это уже совсем другой уровень.

Чтобы создавать проблемы, злоумышленникам даже не нужно взламывать устройство. Если бы хулиган звонил в дверь достаточно настойчиво, он бы помешал другим людям добраться до нее. Такой тип атаки называется «отказ в обслуживании» (denial of service, DoS), потому что из-за нее порядочные люди не могут получить доступ к сервису. Если вы управляете интернет-магазином, подобные атаки могут просто похоронить ваш бизнес. Большинство атак

такого рода сегодня представляют собой *распределенный отказ в обслуживании* (distributed denial of service, DDoS) — когда сразу несколько хулиганов согласованно звонят в дверь.

Отслеживать злоумышленников чаще всего бесполезно из-за использования ими *серверов-посредников*. Задача не составила бы труда, если бы они запускали миллионы атак прямо с собственных компьютеров, но вместо этого злоумышленники взламывают несколько устройств, устанавливают туда нужное им программное обеспечение (часто называемое *вредоносным ПО*) и позволяют этим устройствам делать за них всю грязную работу. Часто это выглядит как многоуровневое дерево, которое состоит из миллионов скомпрометированных устройств. Гораздо труднее перехватить относительно небольшое количество *командных и управляющих сообщений*, контролирующих другие скомпрометированные устройства. Да и результаты атаки можно не отправлять злоумышленнику напрямую, достаточно разместить их на каком-нибудь общедоступном веб-сайте в зашифрованном виде, чтобы взломщик мог получить их, когда ему удобно.

Как все это возможно? Прежде всего это заслуга программных продуктов Microsoft, установленных на множестве устройств по всему миру. Само ПО Microsoft задает стандарты для небезопасного и подверженного ошибкам программного обеспечения, и так вышло не случайно. В интервью журналу Focus в октябре 1995 года Билл Гейтс сказал: «Я говорю, что мы не создаем новые версии для исправления ошибок. Нет. Их просто бы не покупали». Недавно Microsoft все же выпустила некоторые улучшения, а лавры самого небезопасного программного обеспечения на рынке перешли к устройствам интернета вещей, многие из которых уже имеют большую вычислительную мощность, чем современные компьютеры.

Существуют два основных класса атак. Первый, взлом криптографической системы, встречается относительно редко, и его сложно реализовать в хорошо спроектированной системе. Гораздо чаще встречаются «социальные» атаки, когда пользователя обманом заставляют установить программу. Никакое шифрование вас не защитит, если вредоносный фрагмент кода, который вы установили, отслеживает, как вы вводите один из своих паролей. Некоторые популярные механизмы социальных атак со стороны выглядят рассчитанными на идиотов, но на их удачу попадаются, казалось бы, умные люди — например, запуск программ, отправленных по электронной почте или записанных на флешку, которую вы подобрали на дороге, или подключение телефона к случайному USB-порту. Что вообще могло пойти не так? Сейчас на смену этим механизмам приходят атаки через веб-браузеры, а они не так-то просты, если вы помните главу 9.

Одним из примеров чрезвычайно умной и опасной атаки стал взлом системы онлайн-банкинга в 2009 году. При входе пользователя в систему злоумышленники снимали часть денег с его банковского счета. Затем они переписывали возвращающуюся с сервера банка веб-страницу таким образом, чтобы снятие средств прошло

бесследно для владельца. Такой подход делает любую кражу незаметной, если только вы все еще не получаете бумажные выписки и тщательно их не проверяете.

Еще одна проблема современности заключается в том, что манипуляции с битами могут иметь физические последствия. Во имя прогресса или удобства все виды критически важной инфраструктуры теперь подключены к интернету. Это означает, что злоумышленник может вывести из строя электростанцию или просто отключить отопление в доме зимой, чтобы замерзли трубы. А с появлением робототехники и интернета вещей злоумышленник потенциально может запрограммировать пылесос так, чтобы он терроризировал кошек или включал охранную сигнализацию, когда никого нет дома.

Наконец, современные технологии значительно усложнили возможность определения подлинности чего-либо. Можно легко создавать *дипфейки* — реалистичные фальшивые фотографии, аудио и видео. Существует теория, что большая часть современных автоматических звонков просто собирает образцы голосов, чтобы использовать их где-то еще. Не за горами ли то время, когда собранные данные будут преобразованы в звонки роботов, похожих на ваших друзей?

Метаданные и наблюдение

Современные технологии привели еще к одному большому изменению. Даже если криптография сохраняет в тайне содержание сообщений, можно многому научиться, просто наблюдая за моделями общения. Как сказал покойный Йоги Берра (Yogi Berra)¹: «Можно понять многое, просто наблюдая». Например, даже если никто никогда не открывает ваши письма, многое можно узнать, просто изучив список адресатов, не говоря уже о размере и весе конвертов и частоте их отправки. В Америке это обычное дело, поскольку сотрудники почты фотографируют каждое письмо.

Информация на внешней стороне конверта называется *метаданными*. Это данные о данных, а не сами данные. Любой человек может использовать эту информацию, чтобы составить список ваших друзей. Вы можете считать, что в этом нет ничего плохого, если только живете в современном западном обществе. Но задумайтесь, к чему это могло бы привести, если бы вы и ваши друзья жили в более репрессивном обществе, где знание этой информации угрожало бы вашим друзьям. Пример — система «социального рейтинга» в Китае.

Конечно, вряд ли кто-то станет заниматься такими вещами, отслеживая почту. Сейчас достаточно просто открыть список друзей в социальных сетях, что значительно упрощает задачу. Кроме того, наблюдение за вами и вашими близкими больше не зависит от наличия достаточного количества агентов. Никто не будет закреплять за вами сыщика, когда вы выходите из дома, потому что все действия

¹ Американский бейсболист и тренер. — Примеч. ред.

в интернете можно отслеживать удаленно, а ваши перемещения в реальном мире видны по множеству камер. Если вы везде носите с собой мобильный телефон, то постоянно находитесь под наблюдением, потому что информация, которая используется для работы системы телефона, — это всё те же метаданные.

Социальный контекст

Трудно говорить о безопасности и не затрагивать при этом политику, потому что на самом деле существуют два аспекта безопасности. Первый — надежные методы обеспечения безопасности. Второй — компромисс между личной безопасностью и безопасностью общества. С этого момента все усложняется, потому что трудно обсуждать технические аспекты, не учитывая социальные цели.

Безопасность — это не только социальная проблема. В разных странах разные стандарты безопасности, потому что законы и нормы в них также разные. И ситуация тем более усложняется сейчас, когда коммуникации легко выходят за национальные границы и подчиняются различным правилам. Я ни в коем случае не хочу начинать здесь политические споры, важно лишь то, что нельзя обсуждать безопасность исключительно с позиции технологий. Политическая часть этой главы написана в основном исходя из американских реалий.

Многие думают, что «национальная безопасность» закреплена в Конституции США, но это не так. Суды обычно отклоняют дела о конституционных правах, когда чиновники начинают жонглировать аргументами вроде «национальная безопасность» и «государственная тайна». Четвертая поправка к Конституции США содержит самое четкое определение национальной безопасности, которое гласит: «Право народа на гарантии неприкосновенности личности, жилища, бумаг и имущества от необоснованных обысков и арестов не должно нарушаться, и никакие ордера не должны выдаваться иначе как при достаточных к тому основаниях». К сожалению, понятие «достаточный» не уточняется, вероятно потому, что на момент написания оно казалось очевидным. Обратите внимание, что эта поправка обеспечивает безопасность народа, а не государства — как раз то, что «создано народом для народа».

Вопрос заключается в том, что приоритетнее — обязанность государства защищать людей или права этих самых людей.

Большинство предпочло бы расслабиться, зная, что кто-то другой обеспечивает их безопасность, — это можно назвать общественным договором. К сожалению, ценность такого общественного договора была скомпрометирована из-за нарушений доверия.

Бытует совершенно ничем не подкрепленное убеждение, что люди в правительстве «лучше» или «честнее», чем все остальные. Но в лучшем случае они такие же, как и везде, — кто-то лучше, кто-то хуже. Документальных свидетельств о преступлениях, совершенных сотрудниками правоохранительных органов,

более чем достаточно. Отягчающим обстоятельством является секретность; на должностях, где отсутствует система надзора и подотчетности, плохие люди встречаются чаще, и именно поэтому понятия прозрачности и открытости лежат в основе надлежащей модели доверия. Например, в рамках экспериментов по контролю сознания в 1960-х годах ЦРУ незаконно вводило подопытным ЛСД и наблюдало за их реакцией. Организованного контроля за этой программой, известной как MKUltra, не существовало, что повлекло за собой как минимум одну известную смерть невольного подопытного. После закрытия MKUltra агент Джордж Уайт (George White) сказал: «Когда еще настоящий американский парень мог лгать, убивать, обманывать, воровать, насиливать и грабить с позволения и благословения Всевышнего?» А в ФБР под руководством Дж. Эдгара Гувера накопилась богатая история политических злоупотреблений — не только шпионаж в политических целях, но и активный саботаж предполагаемых врагов.

Если вы не в курсе, в последнее время выявляется все больше и больше злоупотреблений доверием, и, скорее всего, это лишь небольшая часть того, что происходило в реальности, учитывая секретность и отсутствие надзора. Наиболее важными для этой главы являются разоблачения Эдварда Сноудена (Edward Snowden) о незаконном наблюдении со стороны правительства.

Без надзора трудно сказать, прикрывает ли государственная тайна незаконную деятельность или только некомпетентность. Еще в 1998 году правительство США поощряло использование схемы шифрования под названием Data Encryption Standard (DES) (стандарт шифрования данных). Фонд Electronic Frontier Foundation (EFF) создал устройство под названием Deep Crack примерно за 250 000 долларов (что *намного* меньше бюджета АНБ), которое смогло взломать код DES. Одна из причин, по которой это было сделано, — желание показать, что эксперты АНБ или некомпетентны, или сильно преувеличивают безопасность алгоритма. Фонд EFF пытался разоблачить лицемерное нарушение доверия, совершенное АНБ ради удобства американских шпионов. И это отчасти сработало — эксперты агентства не изменили своего поведения, но хотя бы озабочились разработкой схемы Advanced Encryption Standard (улучшенный стандарт шифрования), которая вскоре заменила DES.

Легко утверждать, что «мир небезопасен». Но если бы эти секретные программы пригодились для поимки многих злоумышленников, мы бы об этом услышали. Вместо этого мы слышим о том, что в ловушку попадают «невежественные и разорившиеся» люди, не представляющие реальной угрозы. Многие говорят: «Меня не волнует, что правительство следит за мной, — мне нечего скрывать». Может быть, это и так, но даже те, кто это утверждает, наверняка *хотели бы* скрыть пароль от своего банковского счета. Часто кажется, что у тех, кто слишком яро протестует против слежки, действительно не все чисто.

Нарушение доверия имеет международные последствия. Люди не желают покупать товары, безопасность которых может быть скомпрометирована. Аутсорсинг также представляет угрозу. В вашей стране могут существовать законы,

захищающие ваши данные, но в других странах кто-то может иметь доступ к этим данным. Мы уже знакомы со случаями продажи аутсорсинговых данных. Недавно были выявлены доказательства того, что персональные данные, полученные внешними субъектами, использовались для вмешательства в политические процессы, что, возможно, ознаменует конец Вестфальского мира.

Нарушения доверия также влияют на свободу. Возникает так называемый «сковывающий эффект», когда люди прибегают к самоцензуре или боятся слежки при встрече или общении с другими людьми в интернете. Существует множество исторических свидетельств, подтверждающих влияние сковывающих эффектов на политические движения.

Существует множество способов разблокировать современный мобильный телефон: пароль или графический ключ, сканер отпечатков пальцев, система распознавания лица. Что лучше? По крайней мере, в Америке я рекомендую использовать пароль или графический ключ, хотя они не так удобны, как их альтернативы. На это есть три причины. Во-первых, некоторые суды истолковали ту часть пятой поправки к Конституции США («Никто... не может быть принужден в уголовном деле свидетельствовать против самого себя») таким образом, что вы не можете дать «свидетельские» показания, которые находятся лишь у вас в голове. Другими словами, нельзя заставить вас разглашать пароли, секретные коды, шаблоны и т. д. Но некоторые суды постановили, что вас *могут* обязать предоставить отпечатки пальцев или собственное лицо. Во-вторых, существует проблема доверия. Даже если вы не возражаете против разблокировки телефона по запросу, можете ли вы быть уверены в том, как телефон использует ваш отпечаток пальца или данные о лице? Нужна ли эта информация исключительно для разблокировки или же она будет загружена в базы данных для будущих тайных целей? Может, вы начнете получать таргетированную рекламу, когда пройдете мимо магазинов, узнавших ваше лицо? В-третьих, пластиковые отпечатки пальцев и фальшивые сетчатки, которыми изобилуют дурацкие боевики, действительно вошли в реальную жизнь. Биометрические данные подделать легче, чем пароль.

Аутентификация и авторизация

Я уже упоминал аутентификацию и авторизацию. *Аутентификация* доказывает, что кто-то или что-то является тем, за кого себя выдает. *Авторизация* ограничивает доступ к чему-либо, если не представлены соответствующие «учетные данные».

Авторизация считается более простой, чем аутентификация, — для нее требуется надлежащим образом спроектированное и реализованное аппаратное и программное обеспечение. Аутентификация намного сложнее. Как программе определить, что именно вы ввели пароль?

Двухфакторная аутентификация (two-factor authentication, 2FA) встроена сегодня во многие системы. *Фактор* — это независимое средство проверки.

Факторы включают вещи, которые должны быть для вас уникальными (например, отпечатки пальцев), личное имущество (например, мобильный телефон) и что-то, что знаете только вы (например, пароли или PIN-коды). Двухфакторная аутентификация, как следует из названия, задействует два из этих факторов — например, при использовании банковской карты с PIN-кодом или вводе пароля на телефон отправляется сообщение для ввода одноразового кода. Одни такие системы работают лучше, другие — хуже. Очевидно, что другие люди могут иметь доступ к вашему телефону, поэтому отправка сообщения не так уж и безопасна. Сама инфраструктура сотовой связи влияет на безопасность использования 2FA. Злоумышленники могут использовать ваш адрес электронной почты и другую легкодоступную информацию, чтобы перенести ваш номер телефона на SIM-карту в подконтрольном им телефоне. Таким образом они получают доступ к вашим данным, а вы ничего не можете с этим сделать.

Криптография

Как я уже упоминал, криптография позволяет отправителю зашифровать сообщение так, чтобы расшифровать его могли только назначенные получатели. Это очень важно — например, снимая деньги со своего банковского счета, вы вряд ли хотите, чтобы это мог сделать кто-то еще.

Однако криптография важна не только для конфиденциальности и безопасности. Криптографические подписи позволяют подтвердить достоверность данных. Раньше считалось, что к физическим оригиналам можно было обращаться, если возникали сомнения в надежности источника информации. В случае с документами, аудио, видео и т. д. оригиналов часто не существует, так как они создаются на компьютерах и никогда не преобразовываются в физическую форму. Криптографические методы могут использоваться для предотвращения и обнаружения подделок.

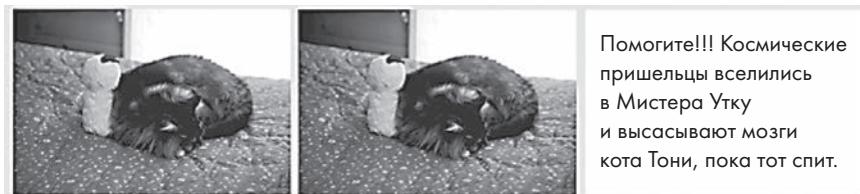
Однако криптография сама по себе не превратит замок в могучую крепость — она лишь часть цельной системы безопасности, где все части имеют значение.

Стеганография

Сокрытие одной вещи внутри другой называется *стеганографией*. Это отличный способ передать секреты, потому что практически невозможно отследить связь между отправителем и получателем. Раньше для этого публиковались объявления в газетах, но теперь стеганография намного чаще используется в интернете, поскольку количество мест для публикации там практически бесконечно.

Стеганография технически не является криптографией, но она вполне подходит для наших целей. Взгляните на рис. 13.1. Слева вы видите фото Мистера Утки и кота Тони. В центре — та же фотография, на которой скрыто секретное

сообщение. Отличите ли вы две фотографии друг от друга? Секретное сообщение находится справа.



Помогите!!! Космические
пришельцы вселились
в Мистера Утку
и высасывают мозги
кота Тони, пока тот спит.

Рис. 13.1. Секретное сообщение, спрятанное в фотографии

Как мы это сделали? Слева — 8-битная черно-белая фотография. Центральное изображение было создано путем замены младшего значащего бита в каждом пикселе соответствующим младшим битом из секретного сообщения справа. Таким образом, чтобы восстановить секретное сообщение, нужно просто удалить семь наибольших значащих битов из центрального изображения.

Это не лучший способ зашифровать послание. Гораздо менее очевидно передать секретное сообщение в кодах символов ASCII, а не в самих символах. Биты секретного сообщения почти невозможно обнаружить, если распределить их по всему изображению или зашифровать. Согласно другому подходу, недавно опубликованному исследователями Чаном Сяо (Chang Xiao), Чэном Чжаном (Cheng Zhang) и Чанси Чжэном (Changxi Zheng) из Колумбийского университета, сообщения кодируются путем небольшого изменения формы текстовых символов. Эта идея не нова — «первая женщина-криптоаналитик Америки» Элизабет Смит Фридман (Elizabeth Smith Friedman) (1892–1980) использовала подобную технику, чтобы нанести секретное сообщение на надгробие своего мужа.

Стеганография используется рекламодателями для отслеживания посещаемых веб-страниц. Они не понимают, что «нет — значит нет», если вы заблокировали рекламу. Многие веб-сайты содержат изображение размером в один пиксель, скрытое на веб-страницах, связанных с определенным URL-адресом. Это не так уж и безобидно: в 2016 году в Турции из-за подобного отслеживающего ПО тысячи людей были обвинены в государственной измене.

Данный метод не ограничивается изображениями. Секретные сообщения могут быть даже закодированы в виде количества пустых строк в публикации в блоге или комментарии на веб-странице. Сообщения могут быть разбросаны по кадрам в видео или скрыты в цифровом аудио, подобно предыдущему примеру с одним пикселеем. Один из необычных примеров последнего — *маркетинг бесшумных свистков*: веб-страницы и рекламные объявления воспроизводят ультразвук, не воспринимаемый человеческим ухом. Эти звуки улавливает микрофон вашего мобильного телефона, что позволяет рекламодателям устанавливать связь между вашими устройствами и определять, какую рекламу вы видели.

Стенография применяется и в других целях. Например, студия может вставлять уникальные идентификационные метки в фильмы, еще не вышедшие в прокат и предназначенные для отправки рецензентам. Это позволит им отследить источник в случае незаконного распространения фильма — тот же принцип, что и при нанесении водяных знаков на изображения.

Стеганография используется почти в каждом компьютерном принтере. Фонд EFF получил документ в ответ на запрос, касающийся Закона о свободе информации, который предполагает существование секретного соглашения между правительством и производителями в целях отслеживания всех печатных документов. Цветные принтеры, например, добавляют маленькие желтые точки на все страницы — с их помощью шифруется серийный номер принтера. EFF выпустил специальные светодиодные фонарики, с помощью которых эти точки можно было найти, поскольку такой вид шифрования может быть расценен как вторжение в частную жизнь.

Шифры подстановки

Если у вас когда-нибудь было кольцо-декодер, то вы, скорее всего, уже знакомы с простым *шифром подстановки*. Идея довольно проста: создается таблица, где каждый символ заменяется другим, как показано на рис. 13.2. Сообщение *зашифровывается* путем замены всех исходных символов аналогами из таблицы, а *расшифровывается* в обратном порядке. Исходное сообщение называется *открытым текстом*, а зашифрованная версия — *зашифрованным текстом*.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
q	s	a	o	z	w	e	n	y	d	p	f	c	x	k	g	u	t	m	v	l	b	r	h	j	i

Рис. 13.2. Шифр подстановки

Этот шифр соотносит *c* с *a*, *r* с *t*, *y* с *j* и т. д., так что слово *cryptography* (криптография) будет зашифровано как *atjgvketqgnj*. Расшифровать зашифрованный текст можно с помощью обратного связывания (*a* с *c*, *t* с *r* и т. д.). Такой код называется *симметричным*, поскольку для кодирования и декодирования сообщения используется один и тот же шифр.

Почему это не лучшая идея? Шифры подстановки легко взломать, используя статистику. Люди проанализировали, как часто буквы используются в разных языках. Например, в английском языке наиболее распространены следующие пять букв: *e*, *t*, *a*, *o*, *n* — именно в таком порядке. По крайней мере, так было, когда Герберт Зим (Herbert Zim) (1909–1994) опубликовал книгу «*Codes & Secret Writing*» в 1948 году. Взлом подстановочного шифра включает в себя сначала поиск наиболее часто встречающейся буквы в зашифрованном тексте — можно предположить, что это *e*, и т. д. Правильно разгадав несколько

букв, можно с легкостью разобрать некоторые слова — так будет проще определить оставшиеся буквы. Возьмем абзац открытого текста из листинга 13.1 в качестве примера. Чтобы упростить задачу, сделаем все буквы строчными и удалим знаки препинания.

Листинг 13.1. Пример открытого текста

```
theyre going to open the gate at azone at any moment
amazing deep untracked powder meet me at the top of the lift
```

Этот же абзац в виде зашифрованного текста (мы применили код с рис. 13.2):

```
vnzjtz ekyxe vk kgzx vnz eqvz qv qikxz qv qxj ckczxv
qcqiyxe ozzg lxtqapzo gkrozt czzv cz qv vnz vkg kw vnz fywv
```

В листинге 13.2 показано распределение букв в зашифрованной версии. Текст отсортирован по частоте появления букв — наиболее часто встречающаяся буква находится вверху.

Листинг 13.2. Анализ частоты появления букв

```
zzzzzzzzzzzzzzzz
vvvvvvvvvvvvvvv
qqqqqqqqq
kkkkkkkk
xxxxxxxx
cccccc
eeee
gggg
nnnn
ooo
ttt
yyy
ii
jj
ww
a
f
l
p
r
```

Взломщик может использовать этот анализ, чтобы догадаться, что буква *z* в зашифрованном тексте соответствует букве *e* в открытом тексте, поскольку в зашифрованном тексте их больше, чем любых других букв. Продолжая в том же духе, мы также можем предположить, что *v* означает *t*, *q* означает *a*, *k* означает *o*, а *x* означает *n*. Заменим буквы соответствующим образом. Расшифрованные буквы сделаем заглавными, чтобы отличать их от нерасшифрованных.

```
TnEjtE e0yNe TO 0gEN TnE eATE AT A1ONE AT ANj c0cENT
AcAiNe oEEg 1NTtAapEo g0roEt cEET cE AT TnE T0g Ow TnE fywT
```

На этой стадии можно сделать несколько простых догадок, основанных на общих знаниях английского языка. Существует очень мало трехбуквенных слов, начинающихся с *t* и заканчивающихся на *e*, и *the* встречается чаще всего, так что предположим, что *n* означает *h*. Это легко проверить в моей системе Linux, так как в ней есть словарь слов и программа сопоставления с образцом; команда `grep '^t.e$' /usr/share/dict/words` находит все трехбуквенные слова, начинающиеся с *t* и заканчивающиеся на *e*. Кроме того, в сочетании *cEET* с *E* грамматически правильным будет выбор только одной буквы на замену *c* — *m*.

THEjtE e0yNe TO OgEN THE eATE AT AiONE AT ANj MOMENT
AMAiyNe oEEg 1NTtAapEo gOroEt MEET ME AT THE T0g Ow THE fywt

В английском только четыре слова соответствуют сочетанию *o□en*: *omen*, *open*, *oven* и *oxen*; только *open* имеет смысл, поэтому *g* заменяем на *p*. Точно так же единственное слово, имеющее смысл в сочетании *to open the □ate*, — это *gate*, поэтому *e* заменяется на *g*. В английском только одно двухбуквенное слово начинается на *o*, поэтому *ow* мы заменим на *of*, то есть *w* соответствует *f*. Буква *j* должна быть заменена на *y*, потому что слова *and* и *ant* не подходят по смыслу.

THEYtE GOyNG TO OPEN THE GATE AT AiONE AT ANY MOMENT
AMAiYNG oEEP 1NTtAapEo POrOEt MEET ME AT THE TOP OF THE fyFT

Можно не расшифровывать сообщение полностью — вместо этого мы воспользовались статистикой и знанием языка, причем нам пригодилась лишь простая статистика по наиболее часто встречающимся буквам. Также можно использовать знание распространенных пар букв в английском языке, таких как *th*, *er*, *on* и *an*, — они называются *диграфами*. Существует статистика по наиболее часто удаивающимся буквам, таким как *ss*, и многим другим приемам.

Как видите, простые шифры подстановки интересны, но не очень безопасны.

Шифры перестановки

Еще один способ закодировать сообщение — зашифровать позиции символов. Древняя система шифра перестановки, предположительно использовавшаяся греками, называется *скитала*. Звучит впечатляюще, но на самом деле так называется закругленная палка, обмотанная лентой пергамента. Сообщение записывалось в строку вдоль палки, где-то между дополнительными ложными сообщениями. В результате на ленте пергамента получался случайный набор символов. Для расшифровки сообщения получатель должен был обернуть ленту вокруг палки того же диаметра, что и палка, которая использовалась для кодирования.

Можно легко сгенерировать шифр перестановки, записав сообщение на сетке определенного размера. В нашем случае именно размер является ключом. Для примера запишем открытый текст из листинга 13.1 на сетке из 11 столбцов, удалив пробелы, как показано на рис. 13.3. В нижней строке пробелы в тексте

заменим несколькими случайными буквами, выделенными курсивом. Чтобы сгенерировать показанный внизу зашифрованный текст, нужно прочитать сетку по столбцам, а не по строкам.

т	h	е	у	г	е	г	о	и	н	г
т	о	о	р	е	н	т	h	е	г	а
т	е	а	т	а	з	о	н	е	а	т
а	н	у	м	о	м	е	н	т	а	м
а	з	и	н	г	д	е	е	п	у	н
т	г	а	с	к	е	д	р	о	в	д
е	г	м	е	е	т	м	е	а	т	т
h	е	т	о	р	о	f	т	h	е	l
i	f	т	а	с	д	f	g	h	i	j
tttaatehioenzrrefeoayiamtlyptmnceoareaogkepsenzmdetodgtoeedmffohnnperetgieetpoahhngaaauwteigatmndtlj										

Рис. 13.3. Сетка шифра перестановки

Частота букв в шифре перестановки такая же, как и в открытом тексте, но это не особо поможет взломщику, поскольку порядок букв в словах также скрывается. Тем не менее подобные шифры по-прежнему довольно легко разгадать, особенно сейчас, когда компьютеры могут сами генерировать сетки разных размеров.

Более сложные шифры

Существует бесконечное множество более сложных шифров, которые представляют собой варианты шифров замены и перестановки или их комбинации. Обычно буквы преобразуются в числовые значения, затем числа преобразуются в буквы снова после выполнения некоторых математических операций над ними. Некоторые коды включают дополнительные таблицы чисел, добавляемые, чтобы взломщик не мог проанализировать частоту, с которой встречаются буквы.

История взлома кодов во время Второй мировой войны очень увлекательна. Одним из способов взлома было прослушивание сообщений, передаваемых по радио. Эти *перехваты* затем подвергались исчерпывающему статистическому анализу, что помогало в итоге их взломать. Важным фактором также была способность человеческого разума распознавать закономерности и некоторые хитрые уловки.

Из сообщений об известных событиях также можно было получить подсказки. Американцы одержали крупную победу в битве за Мидуэй, потому что знали,

что японцы собираются атаковать, но не знали, где именно. Они взломали код, но японцы называли цели кодовыми именами, в данном случае AF. Американцы организовали отправку сообщения с Мидуэя — специально, чтобы японцы его перехватили. В сообщении говорилось, что на острове не хватает пресной воды. Вскоре японцы повторно отправили это сообщение в закодированном виде, подтвердив, что слово AF означало Мидуэй.

Сложность шифров ограничивалась скоростью человека. Американцы уже имели в своем распоряжении несколько машин с перфокартами, которые умели составлять таблицы и тем самым помогали взламывать коды, но это было до компьютерной эры. Коды должны были быть простыми, чтобы сообщения можно было кодировать и декодировать достаточно быстро и без потерь.

Одноразовые блокноты

Самый безопасный метод шифрования носит название *одноразовый блокнот* — он был разработан американским криптографом Фрэнком Миллером (Frank Miller) (1842–1925) в 1882 году. Одноразовый блокнот представляет собой набор уникальных шифров подстановки, каждый из которых применяется только один раз. Раньше шифры печатались в бумажных блокнотах, и лист с использованным шрифтом нужно было оторвать, чтобы применить следующий, — это и объясняет название метода.

Представьте, что мы кодируем предыдущее сообщение. Мы вырываем страницу из блокнота, которая выглядит примерно так, как показано в листинге 13.3.

Листинг 13.3. Одноразовый блокнот

```
FGDDXEFZEZOUZGBQJTKVAZGNYYSMWGRBKRAKDSMKMKAHBFGRYHUPNAFJQDOJ  
IPTVWQWZKHJLDUWITRQGJYGMZNVIFDHOLAFERE0ZKBYAMCXCVNOUROWPBFNA
```

Это работает следующим образом: каждая буква в исходном сообщении преобразуется в число от 1 до 26, как и каждая соответствующая ей буква в одноразовом блокноте. Значения складываются с использованием арифметики с основанием 26. Например, первая буква в сообщении — T со значением 20. Она соотносится с первой буквой одноразового блокнота, F, со значением 6. Цифры складываются, и получается 26, поэтому закодированная буква — Z. Аналогично вторая буква, H, имеет значение 8 и находится в паре с G, значение которой равно 7, поэтому закодированная буква — это 0. Четвертая буква в сообщении — Y со значением 24, что в сочетании с D со значением 4 дает 28. Затем вычитается 26, остается 2, и так мы получаем закодированную букву B. Для расшифровки используется вычитание вместо сложения.

Одноразовые блокноты абсолютно безопасны при правильном использовании, но и тут не обошлось без проблем. Во-первых, обе стороны коммуникации должны использовать один и тот же блокнот. Во-вторых, они должны быть

синхронизированы, то есть шифр для обеих сторон должен быть одинаковым. Коммуникация становится невозможной, если кто-то забывает оторвать страницу или случайно отрывает больше одной. В-третьих, размер блокнота должен быть не меньше длины сообщения, чтобы шаблоны не повторялись.

Интересным применением одноразовых блокнотов была система шифрования голоса SIGSALY времен Второй мировой войны. Она была введена в эксплуатацию в 1943 году и использовалась для шифровки и расшифровки аудио с помощью одноразовых блокнотов, хранившихся на грампластинках. Это нельзя было назвать портативным устройством — каждая такая система весила более 50 тонн!

Проблема обмена ключами

Одна из проблем симметричных систем шифрования — обе стороны коммуникации должны использовать один и тот же ключ. Вы можете отправить кому-нибудь одноразовый блокнот по почте или с надежным курьером, но убедиться, что он не был перехвачен в пути или скопирован, не получится. Кроме того, блокнот можно потерять или повредить. Это так же небезопасно, как отправить другу почтой ключ от дома — вы не узнаете, не сделал ли кто-то копию по пути. Другими словами, ключ уязвим для атаки через посредника.

Криптография с открытым ключом

Криптография с открытым ключом решает многие из рассмотренных проблем. В этом случае используется пара *связанных* ключей. Представьте себе дом с почтовым ящиком на входной двери. Первый ключ, называемый *открытым ключом*, можно дать любому — он позволяет класть почту в ящик. Но только вы можете открыть входную дверь с помощью второго — *закрытого* — ключа и *прочитать* письма.

Криптография с открытым ключом является *асимметричной* системой, в которой ключи кодирования и декодирования неодинаковы. Это решает проблему обмена ключами: открытый ключ нельзя использовать для декодирования сообщений, поэтому не столь важно, попала ли его копия в руки злоумышленникам.

Криптография с открытым ключом основана на *односторонних функциях с потайным входом* — математических функциях, которые легко вычислить в одном направлении, но невозможно в другом, если у вас нет части секретной информации. Функция называется так, потому что выбраться из потайного входа легко, но найти его без подсказки практически невозможно. Рассмотрим очень простой пример с функцией $y = x^2$. Довольно легко вычислить y , имея x . Но вычислить x из y с помощью уравнения уже сложнее. Ненамного, потому что это простой пример, но вы наверняка согласитесь, что проще выполнить умножение, чем найти квадратный корень. Для этой функции математического

секрета не существует, будем считать, что секрет — это наличие калькулятора, потому что он делает решение для x таким же простым, как и для y .

Идея состоит в том, что открытый и закрытый ключи связаны сложной математической функцией, где открытый ключ выступает в роли потайного входа, а закрытый — в роли подсказки, что упрощает шифрование сообщений, но затрудняет их расшифровку. На высоком уровне принято определять ключи в виде факторов действительно большого случайного числа.

Асимметричное шифрование требует больших вычислительных ресурсов. В результате оно зачастую используется только для того, чтобы тайно сгенерировать симметричный *сеансовый ключ*, который затем будет использован для фактического содержимого сообщения. Частный пример подобного шифрования — *обмен ключами по методу Диффи — Хеллмана*, названный в честь американских криптографов Уитфилда Диффи (Whitfield Diffie) и Мартина Хеллмана (Martin Hellman).

Диффи и Хеллман опубликовали статью о криптографии с открытым ключом в 1976 году, но первая ее реализация стала доступной только в 1977 году, поскольку, несмотря на простоту концепции односторонней функции с потайным входом, создать ее очень сложно. Эта задача оказалась под силу криптографу Рональду Ривесту (Ronald Rivest) в 1977 году. Легенда гласит, что он сделал это после того, как выпил несколько бутылок «Манишвица», тем самым доказав, что математические способности не связаны с вкусовыми рецепторами. Вместе с израильским криптографом Ади Шамиром (Adi Shamir) и американским ученым Леонардом Адлеманом (Leonard Adleman) Ривест разработал алгоритм *RSA*, название которого составлено из первых букв фамилий каждого участника. К сожалению, разоблачитель правительственный заговоров Эдвард Сноуден выяснил, что их компания RSA Security взяла деньги у АНБ в обмен на установку криптографической лазейки в их генераторе случайных чисел. Из-за этого АНБ и всем связанным организациям стало проще расшифровать сообщения, закодированные RSA.

Прямая секретность

Одна из практических проблем, связанных с использованием сеансового ключа симметричного шифрования, заключается в том, что при обнаружении ключа могут быть прочитаны все сообщения. Мы знаем, что у многих правительств есть технические возможности для записи и хранения сообщений. Если, например, вы активист-правозащитник, чья безопасность зависит от безопасности переписки, вы не захотите, чтобы ваш ключ обнаружили и расшифровали все ваши сообщения.

Чтобы избежать этого, вы можете использовать *прямую секретность* — подход, при котором для каждого сообщения создается новый сеансовый ключ. Таким образом, обнаружив ключ, злоумышленник сможет декодировать только одно сообщение.

Криптографические хеш-функции

Мы кратко упомянули хеш-функции как метод быстрого поиска еще в главе 7. Они также используются в криптографии, но для этого подходят только некоторые из них, обладающие определенными свойствами. Как и в случае с обычными хеш-функциями, криптографические функции преобразуют произвольные входные данные в числа фиксированного размера. Хеш-функции для поиска преобразуют свои входные данные в гораздо меньший диапазон выходных данных, чем их криптографические собратья, поскольку первые используются как ячейки памяти, а вторые — только как числа.

Ключевым свойством криптографических хеш-функций является то, что они *односторонние*. Это означает, что сгенерировать хеш из входных данных несложно, но нецелесообразно генерировать входные данные из хеша.

Еще одно важное свойство — небольшие изменения входных данных генерируют некоррелированные хеши. Еще в главе 7 мы использовали хеш-функцию, которая суммировала значения символов по модулю некоторого простого числа. В этом случае строка *b* будет иметь хеш-значение на 1 больше, чем строка *a*. Для криптографии это слишком предсказуемо. В табл. 13.1 показаны хеши SHA-1 (Secure Hash Algorithm № 1, защищенный алгоритм хеширования № 1) для трех строк, отличающихся только одной буквой. Как видите, между входными данными и хеш-значением нет заметной связи.

Таблица 13.1. Хеши для фразы *Corned Beef*

Входные данные	Хеш-значение SHA-1
Corned Beef	005f5a5954e7eadabbbf3189ccc65af6b8035320
Corned Beeg	527a7b63eb7b92f0ecf91a770aa12b1a88557ab8
Corned Beeh	34bc20e4c7b9ca8c3069b4e23e5086fba9118e6c

Криптографические хеш-функции должно быть трудно подделать — сложно придумать входные данные, которые генерировали бы известное хеш-значение. Другими словами, необходимо, чтобы было сложно добиться коллизии данных. Используя хеш-алгоритм из главы 7 с простым числом 13, получим хеш-значение 4 для входных данных *Corned Beef*. Но это же значение хеш-функции мы получили бы и для фразы *Tofu Jerky Tastes Weird*.

Долгое время чаще всего использовалась хеш-функция MD5. Но в конце 1990-х был найден способ создания *коллизий*, который вы рассмотрели еще в разделе «Создание хешей». На момент написания книги MD5 был заменен вариациями алгоритма SHA. К сожалению, вариации SHA-0 и SHA-1 были разработаны АНБ, что делает их ненадежными.

Цифровые подписи

С помощью криптографии можно проверить подлинность данных, используя *цифровые подписи*, которые обеспечивают целостность, неподдельность и аутентификацию.

Проверка *целостности* означает, что можно определить, было ли сообщение изменено. Например, в былые времена табели успеваемости представляли собой настоящие печатные карточки со списком класса и оценок, которые ученики приносили домой и показывали родителям. Я помню одноклассника-двоечника, который в четвертом классе вручную исправил двойки на пятерки. Его родители даже не догадались об этом.

Проверка целостности осуществляется путем прикрепления криптографического хеша данных. Но, конечно, прикрепить хеш к сообщению может кто угодно. Чтобы предотвратить это, отправитель шифрует хеш с помощью закрытого ключа, который получатель может расшифровать, используя соответствующий открытый ключ. Обратите внимание, что в случае с подписями роли открытого и закрытого ключей меняются местами.

Использование закрытого ключа обеспечивает как неподдельность, так и аутентификацию. *Неподдельность* означает, что отправитель вряд ли сможет заявить, что он не подписывал сообщение, если на нем его закрытый ключ. *Аутентификация* означает, что получатель знает, кто подписал сообщение, поскольку его открытый ключ связан с закрытым ключом подписавшего.

Инфраструктура открытых ключей

В шифровании с открытым ключом зияет одна большая дыра. Предположим, вы используете веб-браузер для подключения к банку с использованием безопасного (HTTPS) соединения. Банк отправляет открытый ключ в ваш браузер, чтобы тот мог зашифровать ваши данные, а банк — расшифровать их, используя закрытый ключ. Но как убедиться, что этот открытый ключ поступил от вашего банка, а не от какой-то третьей стороны, которая смогла подключиться к передаче сообщений? Как браузер аутентифицирует этот ключ? Кому, если не ключу, он может доверять?

Сегодня для решения подобных задач используется *инфраструктура открытых ключей* (*public key infrastructure, PKI*), хотя это не лучший выбор. Частью такой инфраструктуры является доверенная третья сторона, называемая *центром сертификации* (ЦС), которая гарантирует подлинность ключей. Теоретически ЦС проверяет, что сторона является той, за кого она себя выдает, и отправляет криптографически подписанный документ (*сертификат*), который можно использовать для проверки ключа. Эти сертификаты имеют формат X.509 — стандарт, определенный Международным союзом электросвязи (МСЭ).

Хотя PKI в целом работает, все возвращается к проблеме доверия. ЦС иногда взламывают, а небрежности в них привели к случайной публикации закрытых ключей, что позволило любому подписывать поддельные сертификаты (к счастью, для сертификатов существует механизм *отзыва*). Некоторые ЦС оказались небезопасными, поскольку они не аутентифицировали стороны, которые запрашивали сертификаты. И вполне допустимо, что правительство считает, что оно имеет право принуждать центры сертификации генерировать поддельные сертификаты.

Блокчейн

Технология *распределенных реестров*, или *блокчейн*, — еще одно применение криптографии. В ее основе лежит простая идея, которая реализуется в большом объеме сложных математических выкладок. Большая часть дискуссий в СМИ о распределенных реестрах в связи с биткоином и другими криптовалютами посвящена их применению, а не принципу работы.

Можно представить распределенный реестр как механизм управления бухгалтерской книгой или выпиской по банковскому счету. Проблема с бухгалтерскими книгами в том, что их легко изменить на бумаге и еще легче — в электронном виде, поскольку компьютеры не оставляют следов ластика.

Бухгалтерская книга обычно состоит из набора записей в последовательности строк. Эквивалент строки бухгалтерской книги в распределенном реестре — это *блок*. Реестр добавляет криптографический хеш предыдущего блока (строки) и метку времени его создания к следующему блоку. В результате получается цепочка блоков (отсюда и английское название — *blockchain*, «цепочка блоков»), связанных хешами и метками времени, как показано на рис. 13.4.



Рис. 13.4. Упрощенная схема распределенного реестра

Как видите, если бы содержимое блока *n* было изменено, его хеш изменился бы и перестал совпадать с хешем, хранящимся в блоке *n + 1*. Благодаря свойствам криптографических хешей маловероятно, что можно изменить содержимое блока без изменения хеша. Фактически каждый блок включает в себя цифровую подпись предыдущего блока.

Единственный действенный способ атаковать блокчейн — скомпрометировать управляющее им программное обеспечение. Такой подход можно несколько смягчить, сделав данные реестра общедоступными и продублировав их в нескольких системах. Для атаки на подобную распределенную систему необходим сговор нескольких человек.

Управление паролями

Криптография также применяется для *управления паролями*. В старые добрые времена компьютеры хранили файл паролей в *открытом* виде. Когда пользователь входил в систему, введенный им пароль сравнивался с паролем, хранящимся в файле.

Это плохой подход прежде всего потому, что каждый, у кого есть доступ к файлу, может узнать все пароли. Имейте в виду, что для этого даже не обязательно взламывать компьютер. Многие организации отправляют резервные копии данных на хранение третьим лицам (хорошо иметь не менее трех резервных копий, географически удаленных друг от друга, желательно на разных тектонических плитах). Здесь опять встает вопрос доверия, потому что любой человек может получить доступ к файлу паролей или другим данным в этих резервных копиях. Резервные копии можно зашифровать, но это ненадежно, так как дефекты носителя небольшого размера (например, неисправный блок дисковода) могут сделать всю резервную копию невосстановимой. Придется учитывать компромисс между защитой данных и возможностью их восстановления.

Простое решение этой проблемы — хранить пароли в зашифрованном формате, например в виде криптографического хеша. Когда пользователь пытается войти в систему, его пароль преобразуется в криптографический хеш, который затем сравнивается с тем, что содержится в файле. Свойства криптографических хешей делают вероятность угадывания пароля весьма маловероятной. В качестве дополнительной меры предосторожности большинство систем запрещают обычным пользователям доступ к файлу паролей.

Однако сами пароли могут создавать проблемы даже при таких мерах предосторожности. На заре общих вычислений достаточно было парочки паролей для нескольких систем. Однако сейчас приходится держать в голове бесчисленное количество паролей от банковских счетов, школьных сайтов, множества интернет-магазинов и т. д. Многие люди выходят из этой ситуации, используя везде один и тот же пароль. Оказывается, что самый распространенный пароль — это *password*, а второе место занимает *password123* для сайтов, требующих, чтобы в пароле была хотя бы одна цифра. Повторное использование пароля эквивалентно отказу от прямой секретности — если один сайт скомпрометирован, пароль может быть использован на любом другом сайте. Для каждого сайта можно задать отдельный пароль, но тогда придется запомнить их все. Можно использовать менеджер паролей, который хранит все пароли в одном месте,

защищенным единственным паролем, но, если этот пароль или менеджер паролей скомпрометирован, раскрыты будут и все оставшиеся пароли. Вероятно, наиболее эффективной, хотя и проблематичной, является двухфакторная аутентификация, о которой мы говорили выше. Но она часто требует наличия мобильного телефона и не позволяет получить доступ к учетным записям, если вы находитесь вне зоны действия сети. Кроме того, она отнимает много времени, из-за чего люди просто не выходят из аккаунтов.

Гигиена ПО

Теперь вы узнали кое-что о безопасности и криптографии, но как эти знания пригодятся вам как программисту? Не нужно быть экспертом в криптографии или безопасности, чтобы избежать многих распространенных ошибок. Подавляющее большинство уязвимостей безопасности возникают в результате ситуаций, которых легко можно было избежать, — многие из них описаны в книге Генри Спенсера (Henry Spencer) «The Ten Commandments for C Programmers». Некоторые ситуации мы рассмотрим в этом разделе.

Защищайте только необходимое

При разработке системы безопасности возникает искушение сделать так, чтобы абсолютно все было безопасно. Но это не всегда хорошая идея. Например, пользователям приходится входить в систему для доступа к чему-то, что не обязательно защищать. Поскольку вошедшие в систему пользователи могут просматривать «защищенный» контент, это увеличивает вероятность того, что кто угодно может получить к нему доступ, когда, например, пользователь недолго отходит от компьютера.

В этом состоит принцип работы многих мобильных телефонов. По большей части в них блокируется все, кроме, возможно, камеры. Есть вещи, которые *нужно* блокировать, — вряд ли вы хотите, чтобы кто-то отправлял сообщения от вашего имени, если вы потеряете телефон. Но предположим, что вы слушаете музыку с друзьями. Вы передаете разблокированный телефон кому-то, кто выбирает мелодии, предоставляя доступ ко всему содержимому телефона. Отправка кода на телефон часто используется в качестве второго фактора в двухфакторной аутентификации, и передача этого фактора третьей стороне противоречит безопасности.

Проверьте логику трижды

Довольно легко написать программу, которая, *как вы думаете*, что-то делает, хотя на самом деле это не так. Незамеченные ошибки в логике можно использовать против вас, особенно когда злоумышленник имеет доступ к исходному коду. Чтобы этого избежать, вы можете, например, пройтись по коду, обсудив

его с кем-нибудь еще. Читая вслух, вы просматриваете код медленнее, чем при чтении про себя, — и всегда можете заметить что-то необычное.

Поиските ошибки

Код, который вы пишете, будет использовать системные вызовы и вызывать библиотечные функции. Большинство таких вызовов возвращают коды ошибок, если что-то пойдет не так. Не игнорируйте их! Например, если вы безуспешно пытаетесь выделить память, не используйте ее. Если не удается прочитать пользовательский ввод, не предполагайте, что было введено допустимое значение. Таких примеров много, и обработка каждой ошибки может быть утомительна, но все же займитесь этим.

Избегайте библиотечных функций, которые могут незаметно перестать работать или выйти за допустимые пределы. Убедитесь, что в инструментах языка включены отчеты об ошибках и предупреждениях. Считайте ошибки выделения памяти критическими, потому что многие библиотечные функции полагаются на выделенную память и могут внезапно дать сбой, после того как память не будет выделена.

Сведите к минимуму поверхности атаки

В этом разделе я перефразирую некоторые показания исследователя криптографии Мэтта Блейза (Matt Blaze) от 19 апреля 2016 года перед подкомитетом Палаты представителей США после стрельбы в Сан-Бернардино. Советую вам прочитать источник целиком.

Из-за сложности программ мы должны исходить из того, что все они содержат ошибки. Исследователи пытались создать «формальные методы», похожие на математические доказательства, которые можно было бы использовать для демонстрации «правильности» компьютерных программ. К сожалению, на сегодняшний день эта проблема все еще не решена.

Из этого следует, что каждая функция, добавленная в часть программного обеспечения, представляет собой новую поверхность атаки. Мы даже не можем доказать, что какая-то из поверхностей атаки защищена полностью. Но мы знаем, что каждая новая поверхность атаки добавляет новые уязвимости и что все они суммируются. Это основная причина того, что настоящие специалисты по безопасности, в отличие от политиков, выступают против идеи установки лазеек для правоохранительных органов. Это усложняет программу, добавляя новую поверхность атаки, и вдобавок, как в примере со школьным шкафчиком из этой главы, весьма вероятно, что неавторизованные стороны поймут, как получить доступ к выделенной лазейке.

Оказывается, Мэтт Блейз действительно знает, о чем говорит. АНБ объявило о разработке системы конфиденциальной связи на чипе Clipper в 1993 году, и в пла-нах АНБ было закрепление обязательств по использованию данной системы для

шифрования сообщений на государственном уровне. Система включала в себя лазейку для правительенного доступа, поэтому продать ее другим странам было сложно: иностранцы не хотели использовать американские продукты, которые могли бы шпионить за ними. Успех Clipper не состоялся как из-за политического сопротивления, так и из-за опубликованной Блейзом в 1994 году статьи под названием «Protocol Failure in the Escrowed Encryption Standard», в которой он показал, как легко использовать эту лазейку. Между прочим, в то время Блейз обнаружил, что он совершенно не был готов к даче показаний в Конгрессе, но теперь он выступает очень даже уверенно. Тогда же для него это была совершенно неизведанная вселенная. Постарайтесь получить навык публичных выступлений — когда-нибудь он может пригодиться.

Хорошая практика обеспечения безопасности заключается в том, чтобы сделать код как можно более простым, чтобы свести к минимуму количество поверхностей атаки.

Не выходите за пределы

В главе 10 была представлена концепция переполнения буфера. Это один из примеров класса ошибок, которые долго остаются незамеченными, — и их также могут использовать злоумышленники.

Напомним, что буфер может переполниться, если программное обеспечение не проверяет доступные пределы и может в итоге перезаписать другие данные. Например, если переменная «вы авторизованы» не проверяет выход за пределы буфера паролей, пользователь сможет авторизоваться, даже используя слишком длинный и неверный пароль. Переполнение буфера в стеке может быть особенно проблематичным, потому что в результате злоумышленник может изменить адрес возврата из вызова функции, что приведет к непредсказуемому поведению других частей программы.

Переполнение буфера не ограничивается только строками. Необходимо также убедиться, что индексы массива учитывают размер самого массива.

Еще одна проблема с пределами — размер переменных. Не исходите, например, из того, что целое число занимает ровно 32 бита. Оно может состоять из 16 бит, и при задании 17-го бита может произойти что-то неожиданное. Проверяйте пользовательский ввод и убедитесь, что любые введенные пользователем числа соответствуют используемым вами переменным. Большинство систем включают в себя файлы определений, которые ваш код может использовать, чтобы убедиться в задании правильных размеров сущностей. В худшем случае определения используются, чтобы предотвратить сборку кода при выявлении неправильных размеров. В лучшем случае можно применять определения для автоматического выбора правильных размеров. Определения подходят и для размеров чисел, и даже для количества битов в байте. Просто не делайте никаких предположений!

Также важно не выходить за пределы памяти. Если вы используете динамически выделяемую память и выделяете n байт, убедитесь, что обращения к памяти находятся в диапазоне от 0 до $n - 1$. Мне пришлось делать отладку кода, в котором после выделения памяти адрес памяти увеличивался, потому что алгоритму было удобно ссылаться на `memory[-1]`. Затем код освобождал `memory` вместо `memory[-1]`, что приводило к ошибкам.

Многие микрокомпьютеры, предназначенные для встроенного использования, имеют гораздо больше памяти, чем требуется программе. В таких случаях избегайте динамического распределения и используйте только статические данные – это позволяет избежать многих потенциальных проблем. Конечно, стоит убедиться, что код учитывает границы хранилища данных.

Еще одна ограниченная область — время. Убедитесь, что ваша программа выполняет задачу, когда ввод поступает быстрее, чем способны реагировать обработчики прерываний. Избегайте прерывания обработчиков, чтобы не выйти за пределы стека.

Существует так называемый *фаззинг* – вид тестирования с использованием инструментов, которые выявляют подобные ошибки. Но это лишь статистический метод, который не заменит написания хорошего кода. В ходе фаззинга код проверяется на большом количестве вариаций допустимых входных данных.

Генерировать хорошие случайные числа — сложно

В криптографии высоко ценятся хорошие случайные числа. Как их получить?

Наиболее распространенные генераторы случайных чисел на самом деле генерируют *псевдослучайные* числа, потому что логические схемы не умеют генерировать настоящие случайные числа. Если использовать одни и те же входные данные, всегда будет получаться одна и та же последовательность чисел. В качестве генератора псевдослучайных чисел (pseudorandom-number generator, PRNG) можно использовать простую схему под названием *регистр сдвига с линейной обратной связью* (linear feedback shift register, LFSR). Пример такого генератора изображен на рис. 13.5.

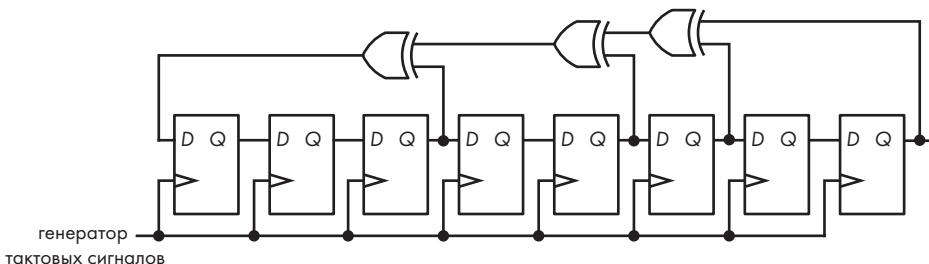


Рис. 13.5. Регистр сдвига с линейной обратной связью

Видим, что по мере сдвига числа вправо слева появляется новый бит, который генерируется из некоторых других битов. Вersия на рисунке генерирует только 8-битные числа, но можно создавать и более крупные варианты. Подобный генератор имеет два недостатка. Во-первых, числа повторяются циклически. Во-вторых, зная самое последнее случайное число, всегда можно определить следующее — если только что было сгенерировано число 0xa4, следующим всегда будет 0x52. Заметьте, что это проблема криптографии, но знание этого факта весьма пригодится вам при отладке программ.

Первое заданное значение в регистре называется *начальным числом*. Многие программные решения позволяют устанавливать начальное число. В LFSR было внесено множество улучшений — например, вихрь Мерсенна, — но, в конце концов, все они столкнулись с двумя проблемами, о которых я говорил. Настоящей случайности не существует.

Современное программное обеспечение решает эту проблему, собирая энтропию из различных источников. Термин *энтропия* был заимствован из термодинамики, где он относится к повсеместной склонности к случайности.

Один из первых источников энтропии под названием LavaRand был изобретен в SGI в 1997 году. Принцип его работы заключался в том, что на пару лавовых ламп направлялась веб-камера, в результате чего генерировалось почти 200 Кбайт случайных данных в секунду. Производительность источников энтропии важна — например, если веб-сайт генерирует множество идентификаторов сеансов для множества клиентов, ему нужен способ быстро получать много хороших случайных чисел.

Было бы здорово поставлять пару лавовых ламп с каждым компьютером, но в реальности это нецелесообразно. Некоторые производители чипов встроили в создаваемое ими оборудование генераторы случайных чисел. В 2012 году компания Intel выпустила встроенный в чип генератор случайных чисел на основе генератора теплового шума, который производил 500 Мбайт случайных чисел в секунду. Но люди отказывались его использовать, потому что он был выпущен сразу после разоблачений Сноудена и не вызывал доверия.

Существует еще один фактор доверия к встроенным генераторам случайных чисел. Производитель может выложить собственный проект в широкий доступ. Можно даже *снять крышу* с чипа и изучить его с помощью электронного микроскопа, чтобы убедиться, что чип соответствует проекту. Однако все же можно незаметно изменить чип в процессе производства — этакий аппаратный эквивалент допингового скандала. В главе 2 я упомянул добавление примесей, когда мы обсуждали транзисторы. *Легирующие добавки* — это неприятные химические вещества, которые используются для создания областей *p* и *n*. Поведение схемы можно изменить, слегка отрегулировав уровни добавок, и эти изменения нельзя будет обнаружить даже под микроскопом.

Специалисты по безопасности поняли, что аппаратные генераторы случайных чисел не заслуживают доверия. Энтропия собирается из случайных событий, которые не зависят от компьютерных программ, — из движений мыши, времени между нажатиями на клавиши, скорости доступа к диску и т. д. Этот подход работает очень хорошо, но мы все еще не можем эффективно и быстро создавать большое количество случайных чисел.

Сбор энтропии привел к возникновению некоторых серьезных глупых ошибок, особенно в операционной системе Android на базе Linux. Оказывается, телефоны с ОС Android не генерируют энтропию быстро, поэтому случайные числа, которые использовались вскоре после генерации, не были такими уж случайными. А еще оказалось, что некоторые из ранних реализаций копировали код, который собирал энтропию из времени доступа к диску. Конечно, в мобильных телефонах вместо дисков используется флеш-память с предсказуемым временем доступа, из-за чего энтропия также становится предсказуемой.

Если ваша безопасность зависит от хороших случайных чисел, убедитесь, что вы разобрались в системе, которая их генерирует.

Знайте свой код

Крупные проекты часто используют *сторонний код* — то есть код, написанный не членами проектной группы. Во многих случаях у команды даже нет доступа к исходному коду, и приходится верить поставщику на слово, что его код работает и безопасен. Что же может пойти не так?

Прежде всего откуда вы знаете, что этот код действительно работает и безопасен? Можете ли вы быть уверены, что тот, кто с ним работал, не установил секретную лазейку? Я спрашиваю не просто так — в 2015 году в продуктах крупного сетевого поставщика была обнаружена секретная лазейка. В 2016 году в продуктах другого поставщика нашли дополнительную учетную запись с жестко запрограммированным паролем. Весьма вероятно, что в будущем на поверхность всплывут еще более внушительные нарушения.

Лекция Кена Томпсона «Reflections on Trusting Trust» 1984 года, посвященная премии Тьюринга, дает представление о том, какой ущерб может нанести злумышленник.

Использование стороннего кода приводит к новой, более сложной проблеме, которая с ужасающей частотой проявляется в программном обеспечении физической инфраструктуры — обеспечивающем работу электростанций и т. п. Вы наверняка думали, что такое важное ПО должны разрабатывать инженеры, но это на самом деле редкость. Инженеры могут руководить созданием подобного ПО, но обычно его пишут «системные интеграторы». Это люди, которых «учили писать код», — системная интеграция в значительной степени сводится

к импорту кода, написанного другими, и склеиванию вызовов функций. В результате код продукта выглядит как на рис. 13.6.



Рис. 13.6. Неиспользуемый код поставщика и код продукта

Это означает, что в готовом продукте будет много неиспользуемого кода — на рисунке больше кода поставщика, чем итогового кода продукта. Однажды я провел серию докладов об этом феномене, назвав его «цифровым герпесом» — весь лишний код обвивается вокруг центральной нервной системы продукта в ожидании внешнего стимула, чтобы вырваться наружу, по аналогии с тем, как действует настоящий вирус в организме.

Это ставит программиста в затруднительное положение. Как решить, какой сторонний код безопасен? Не все работники электростанции являются экспертами в области криптографии или сетевых протоколов.

Прежде всего в этой области открытый исходный код имеет преимущество. Открытый исходный код можно просмотреть, и высока вероятность, что кроме вас на него обратит внимание кто-нибудь еще. Этот принцип «тысячи глаз» означает, что вероятность того, что ошибки в таком будут замечены, выше, чем для закрытого исходного кода, который могут просматривать лишь немногие. Конечно, это не панацея. В 2014 году в популярной криптографической библиотеке OpenSSL была обнаружена серьезная ошибка. В этом был и плюс — обнаружение ошибки заставило многих обратить внимание на этот код и другие критически важные для безопасности пакеты.

Еще одна хорошая практика — следить за соотношением фактически используемого кода из стороннего пакета к общему размеру пакета. Однажды я работал над проектом медицинского инструмента, где руководство сказали: «Давайте использовать эту классную операционную систему, ведь она обойдется недорого». Но она включала в себя всевозможные виды функций, которые мы не собирались использовать. Я отказался, и мы просто написали собственный код для действительно

нужной функциональности. Это было пару десятилетий назад, и только недавно были обнаружены ошибки в некоторых развертываниях этой ОС.

Еще одна область, на которую стоит обратить внимание, — отладка кода. Очень часто во время разработки продукта в систему добавляется код, предназначенный для отладки. Убедитесь, что перед поставкой он был удален, в том числе все жестко закодированные пароли. Если в целях отладки кода вы добавили пароли по умолчанию или другие сокращения, убедитесь, что ничего этого не осталось.

Чрезвычайная изобретательность — ваш враг

Если вы используете сторонний код, избегайте использования малоизвестных умных решений. Это связано с тем, что поставщики часто прекращают поддержку функций, редко используемых их клиентами. Если это так, скорее всего, эта часть приложения перестанет обновляться. Поставщики часто предоставляют исправления только для последних версий продуктов. Соответственно, если ваш код зависит от больше не поддерживаемой функции, вы не сможете установить важные исправления безопасности.

Разберитесь с видимостью

Подумайте, как другие программы могут получить доступ к конфиденциальным данным, в том числе к метаданным. Кто еще может видеть данные вашей программы? Это важная часть определения модели угроз. Что может быть скомпрометировано, если кто-то незаметно проникнет в вашу совершенно безопасную систему? Может ли злоумышленник обойти защиту, вытащив микросхемы памяти из вашего устройства и получив к ним прямой доступ?

Помимо обеспечения безопасности кода, необходимо следить за *атаками по сторонним каналам* — попытками взлома, основанными на метаданных или побочных эффектах реализации. Представьте, что у вас есть код, проверяющий пароль. Если на проверку почти правильного пароля уходит больше времени, чем на проверку заведомо неверного пароля, это дает злоумышленнику ключ к разгадке. Такой подход называется *атакой по времени*.

Камера, направленная на клавиатуру банкомата, — это пример атаки по стороннему каналу.

В истории встречаются атаки, основанные на электромагнитном излучении. Одна из наиболее интересных — *перехват Ван Эйка*, при котором используется антенна для улавливания излучения монитора и создания удаленной копии отображаемого изображения. Было доказано, что таким образом можно нарушить тайну волеизъявления в некоторых электронных системах голосования.

Атаки по сторонним каналам действительно коварны, и, чтобы их избежать, требуется серьезный системный подход. Недостаточно просто хорошо писать

код. Примеров таких атак было немало, особенно в эпоху Второй мировой войны, когда было положено начало современной криптографии. Немцы смогли определить местоположение Лос-Аламосской национальной лаборатории из-за того, что несколько сотен каталогов Sears были отправлены по почте на один и тот же адрес. Британские химические заводы были рассекречены по результатам матчей заводских футбольных команд, опубликованным в местных газетах.

В общем, убедитесь, что внешне видимое поведение важного для безопасности кода не зависит от того, какие задачи он выполняет на самом деле. Избегайте раскрытия информации по сторонним каналам.

Не переусердствуйте

Сейчас я скажу чрезвычайно очевидную вещь — и я бы даже не стал о ней говорить, но мой опыт свидетельствует, что мало кто это понимает. Лучший способ сохранить что-то в безопасности — вообще ничего не хранить. Не собираите конфиденциальную информацию, которая на самом деле вам не нужна.

Классический пример — многие медицинские анкеты. У меня всегда вызывает недоумение тот факт, что в анкете запрашивают и дату моего рождения, и мой возраст. Неужели мне стоит идти к врачу, который не может вычислить возраст по дате рождения? Собирая и то и другое, вы обязываетесь защитить все, поэтому стоит запрашивать только действительно необходимые данные.

Не копите

Собранную конфиденциальную информацию вовсе не обязательно хранить вечно. Избавьтесь от нее как можно скорее. Например, вам может понадобиться чей-то пароль для входа в некую систему. Пароль перестает быть нужным сразу после того, как вы им воспользуетесь. Приберитесь. Чем дольше пароль находится без присмотра, тем больше вероятность, что кто-то его обнаружит.

Уничтожение информации становится все более важным с юридической точки зрения, поскольку мир изо всех сил пытается понять Общий регламент Европейского союза по защите данных (General Data Protection Regulation, GDPR), который описывает последствия утечки персональных данных.

Не полагайтесь на динамическое выделение памяти

В главе 7 мы рассматривали динамическое выделение памяти с использованием кучи. В этом разделе мы поговорим о стандартных функциях библиотеки C: `malloc`, `realloc` и `free`, которые могут вызвать ряд проблем. Разберем, что происходит при освобождении динамически выделенной памяти (рис. 13.7).

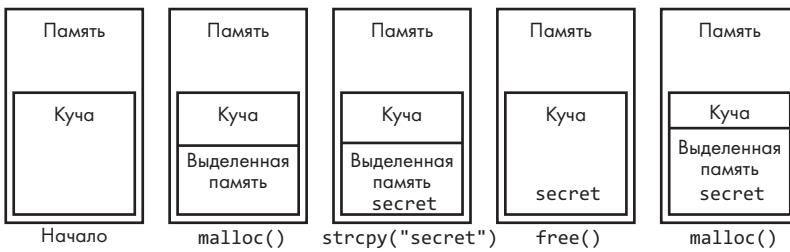


Рис. 13.7. Освобождение памяти

В левой части рис. 13.8 видим кучу, хранящуюся в памяти. По мере продвижения вправо часть памяти из кучи выделяется для использования программой. Затем часть секретной информации (*secret*) копируется в выделенную память. В какой-то момент эта память больше не нужна программе, поэтому она освобождается и возвращается в кучу. Наконец, справа память выделяется из кучи для другой цели. Но секретная информация все еще хранится в отрезке памяти, и ее можно прочитать. Первое правило использования динамической памяти заключается в том, чтобы убедиться, что вся конфиденциальная информация удаляется из памяти перед освобождением.

Функция `realloc` позволяет увеличить или уменьшить размер выделенной памяти. Рассмотрим пример сокращения на рис. 13.8.

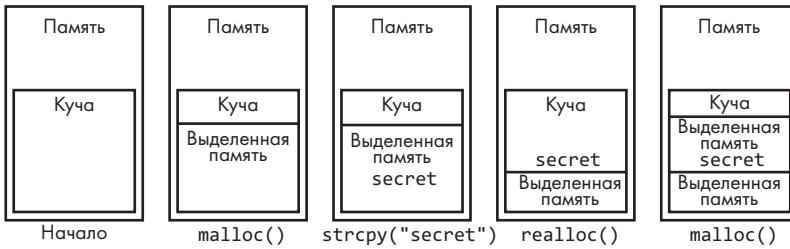


Рис. 13.8. Сокращение памяти

Первые шаги совпадают с предыдущим примером, но затем объем выделенной памяти уменьшается. Секретная информация хранилась в избыточной памяти, поэтому она возвращается в кучу. Затем при последующем выделении памяти мы получаем блок, содержащий секрет. Второе правило использования динамической памяти: убедитесь, что любая память, которая возвращается в кучу при сокращении, очищается. Это правило похоже на первое.

При использовании функции `realloc` для увеличения размера выделенного блока памяти необходимо учитывать два случая. Первый считается простым и не относится к проблемам безопасности. Как показано на рис. 13.9, если над уже выделенным блоком в куче есть место, размер блока увеличивается.



Рис. 13.9. Хороший пример выделения памяти

На рис. 13.10 показан случай, который может привести к проблемам с безопасностью.

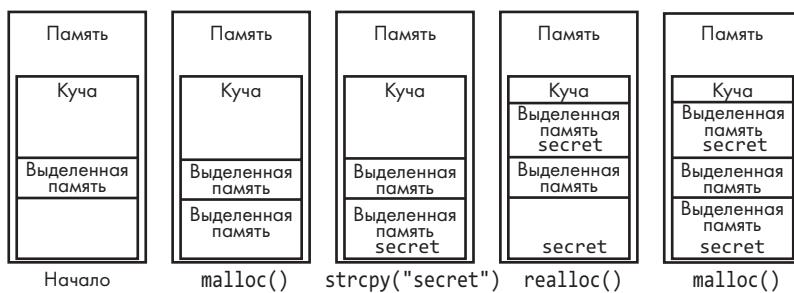


Рис. 13.10. Плохой пример выделения памяти

В этом примере видим еще одну часть памяти, которая уже выделена для другой цели. При попытке увеличить размер выделенного блока памяти места не хватает из-за другого блока, поэтому приходится искать в куче достаточно большой непрерывный блок памяти. Память выделяется, данные из старого блока копируются, и старый блок освобождается. Теперь в памяти содержатся две копии секретной информации, и одна из них находится в нераспределенной памяти, которую можно выделить. Большая проблема здесь в том, что код, вызывающий функцию `realloc`, ничего об этом не знает. Единственный способ проверить, был ли перемещен блок памяти, — сравнить его новый адрес с первоначальным. Но даже если он сдвинулся, мы узнали об этом слишком поздно, потому что уже не можем контролировать старый блок. Отсюда следует третье правило: не используйте функцию `realloc`, если вопрос безопасности очень важен. Используйте `malloc` для выделения новой памяти, скопируйте старые данные в новое место, сотрите старые данные и вызовите функцию `free`. Не так эффективно, зато безопасно.

Не полагайтесь и на сборку мусора

Я уже говорил об удалении важных данных, которые больше не нужны. В предыдущем разделе было показано, что при явном управлении памятью сделать

это не так просто. Системы со сборкой мусора сталкиваются с уникальными проблемами. Предположим, мы написали программу на C, которая содержит что-то «конфиденциальное», как показано на рис. 13.11.

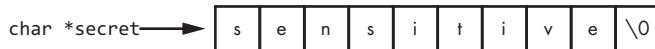


Рис. 13.11. Стока конфиденциальных данных в С

Как очистить ее по завершении работы? Поскольку строки в С заканчиваются NUL-ограничителем, можно сделать строку пустой, задав первый символ как NUL, что показано на рис. 13.12.



Рис. 13.12. Плохо очищенная строка конфиденциальных данных в С

Злоумышленник без труда восстановит содержимое строки, поэтому все-таки придется перезаписать ее полностью.

Эта строка может находиться в массиве памяти или в памяти, полученной динамически с помощью функции `malloc`. Убедитесь, что для строки установлено значение NUL или какое-либо распознаваемое значение, которое легко заметить при отладке, чтобы стереть каждый символ перед вызовом функции `free`. В противном случае конфиденциальные данные просто возвращаются в кучу и могут быть получены позже при вызове функции `malloc`.

Что делать, если выбранный язык программирования использует сборку мусора вместо явного управления памятью? Код наподобие `secret = "xxxxxxxxxxxxxx"` не делает то, что вы думаете, в таких языках, как JavaScript, PHP и Java. Вместо того чтобы перезаписывать конфиденциальные данные, эти языки могут просто создать новую строку. Стока с конфиденциальными данными при этом попадет в список сущностей, подлежащих сборке мусора. Конфиденциальные данные не удаляются сами, и принудительно удалить их вы тоже не можете.

Было бы неплохо, если бы проблемы такого рода ограничивались языками программирования и средами, но это не так. Флеш-память используется повсеместно, в том числе в твердотельных накопителях (solid-state disk drive, SSD). Поскольку флеш-память изнашивается, эти накопители используют *сглаживание нагрузки*, чтобы уравнять использование различных микросхем флеш-памяти. Это означает, что запись новых данных не гарантирует стирание старых, как при освобождении выделенной памяти и ее размещении в куче.

Как видите, даже процесс безопасного стирания чего-либо более сложен, чем все знания, которыми вас пичкают, когда вы «учитесь писать код». Чтобы писать

хороший код, необходимо глубокое понимание всех аспектов среды, и, конечно же, именно поэтому вы читаете эту книгу.

Данные как код

Вы уже должны знать, что «код» — это просто данные в определенном формате, понятном компьютеру. Первоначально имелось в виду аппаратное обеспечение, но теперь многие программы, такие как, например, веб-браузеры, запускают данные, в том числе в виде программ JavaScript. И JavaScript также может запускать данные — оператор `eval` обрабатывает любую строку как программу и выполняет ее.

В главе 5 мы познакомились с некоторыми аппаратными средствами, не позволяющими компьютерам обрабатывать произвольные данные как код. Блоки управления памятью или машины с гарвардской архитектурой хранят код отдельно от данных, предотвращая запуск данных как кода. Программы, которые могут запускать данные, не имеют такой защиты, поэтому вы должны обеспечить ее самостоятельно.

Классический пример — *внедрение SQL* (Structured Query Language, язык структурированных запросов). Он представляет собой интерфейс для многих систем баз данных. Слово «*структурный*» отвечает за организацию данных, например в виде картотеки. «*Запрос*» позволяет получить доступ к этим данным. И конечно же, «*язык*» — это способ получения данных.

Базы данных SQL организуют данные в виде набора *таблиц* — прямоугольных массивов строк и столбцов. Программисты могут создавать таблицы и задавать столбцы. С помощью запросов можно добавить, удалить, изменить или получить строки в таблицах.

Не нужно быть специалистом в SQL, чтобы разобраться в следующем примере. Имейте в виду, что операторы SQL заканчиваются точкой с запятой (;), а комментарии начинаются с цифры или знака решетки (#).

У вашего учебного заведения наверняка есть сайт, где можно проверить свои оценки. Для примера мы взяли базу данных SQL, которая включает в себя таблицу с именами учеников, как показано в табл. 13.2.

Таблица 13.2. Таблица учеников в базе данных

Ученик	Предмет	Оценка
Дэвид Лайтман	Биология 2	F
Дэвид Лайтман	Английский язык 11Б	D
Дэвид Лайтман	Всемирная история 11Б	C

Ученик	Предмет	Оценка
Дэвид Лайтман	Тригонометрия 2	B
Дженнифер Мак	Биология 2	F
Дженнифер Мак	Английский язык 11Б	A
Дженнифер Мак	Всемирная история 11Б	B
Дженнифер Мак	Геометрия 2	D

Сайт предлагает текстовое поле из HTML, где учащийся может ввести название предмета. Пара строк кода на JavaScript и jQuery отправляет название предмета на веб-сервер и отображает полученную оценку. На веб-сервере хранится код на PHP, который ищет оценку в базе данных и отправляет ее на веб-страницу. Когда ученик входит в систему, переменной `$student` присваивается его имя, поэтому он может получить доступ только к собственным оценкам. Код представлен в листинге 13.4.

Листинг 13.4. Фрагменты кода со школьного веб-сайта

HTML

```
<input type="text" id="class"></input>
```

JavaScript

```
$('#class').change(function() {
  $.post('school.php', { class: $('#class').val() }, function(data) {
    // показать оценки
  });
});
```

PHP

```
$grade = $db->queryAll("SELECT * FROM students
  WHERE class='$_REQUEST['class']]' && student='$student']");
header('Content-Type: application/json');
echo json_encode($grade);
```

Запрос к базе данных довольно простой. Мы выбираем все (*) столбцы из всех строк в таблице `students`, где столбец `class` соответствует полю `class` на веб-странице, а столбец `student` соответствует переменной, содержащей имя вошедшего в систему ученика. Что может пойти не так?

Что ж, Дэвид, возможно, не уделяет особого внимания биологии, зато он хорошо разбирается в компьютерах. Он входит в свою учетную запись, и вместо того, чтобы ввести `Biology 2` (Биология 2) в качестве названия предмета, он вводит `Biology 2' || 1=1 || ';`. В результате оператор `select` превращается в то, что показано в листинге 13.5.

Листинг 13.5. Внедрение SQL

```
SELECT * FROM students WHERE class='Biology 2' || 1=1 || '';  
# && student='David Lightman'
```

Этот запрос действует по следующему алгоритму: «выбрать все столбцы из таблицы students, где class — это Biology 2, или 1 равно 1, или пустая строка». Точка с запятой завершает запрос раньше времени, а оставшаяся часть строки превращается в комментарий. Поскольку 1 всегда равно 1, Дэвид только что получил доступ ко всем оценкам. Я мог бы развить эту тему и показать, как Дэвид произвел впечатление на Дженифер, изменив ее оценку по биологии, но вы можете просто посмотреть фильм «Военные игры» 1983 года. Чтобы вас заинтересовать — он мог ввести кое-что в поле с точкой с запятой, а затем выполнить команду обновления базы данных.

Это не просто теоретические сценарии научно-фантастических фильмов — полистайте комикс xkcd № 327. Совсем недавно, в 2017 году, в популярном ПО для веб-сайтов WordPress была обнаружена серьезная ошибка внедрения SQL, которая затронула очень многие веб-сайты. Это еще один пример проблем, с которыми вы можете столкнуться, если будете полагаться на сторонний код.

А вот еще пример — многие веб-сайты дают пользователям возможность оставлять комментарии, и перед публикацией стоит тщательно проверить каждый из них на наличие JavaScript-кода. В противном случае пользователь, просматривающий эти комментарии, непреднамеренно запустит код. И если на вашем сайте пользователи могут отправлять комментарии в формате HTML, то вы, скорее всего, обнаружите, что куча комментариев представляет собой рекламу и ссылки на другие сайты. Убедитесь, что данные, которые вводят пользователи, никогда не интерпретируются как код.

Выводы

В этой главе вы получили общее представление о принципах безопасности. Вы немного узнали о криптографии — ключевой технологии компьютерной безопасности. Вы также узнали о том, как сделать код более безопасным.

Кроме того, надеюсь, вы поняли, что тема безопасности очень сложна и не предназначена для любителей. Консультируйтесь с экспертами, пока не станете одним из них. Не занимайтесь безопасностью в одиночку.

Подобно безопасности, машинный интеллект — еще одна сложная тема, в которой стоит хотя бы немного разобраться. Мы перейдем к ней в следующей главе.

14

Машинный интеллект



Сколько раз, пытаясь найти в интернете что-то типа «коты и котлеты», вы видели фразу «Найдены результаты по запросу *коты и котлеты*»? Наверняка вы принимали это как должное и даже не задумывались о том, как поисковый движок обнаруживает ошибку и, мало того, предлагает ее исправить. Бряд ли кому-то оказалось под силу создать программу, которая перебирает все возможные ошибки в словах и соотносит их с исправленными вариантами. Вместо этого вся работа ложится на плечи *машинного интеллекта*.

Машинный интеллект — сложная тема, которая затрагивает связанные области *машинного обучения, искусственного интеллекта и больших данных*. В этой главе мы кратко рассмотрим все перечисленные концепции, поскольку вы, как программист, наверняка еще встретитесь с ними.

Термин «искусственный интеллект» впервые прозвучал на семинаре Дартмутского колледжа в 1956 году. В 1957 году, познакомившись с *перцептроном*, о котором мы поговорим позже, мир впервые услышал о *машинном обучении*. Сегодня машинное обучение играет ведущую роль отчасти благодаря двум тенденциям. Во-первых, технический прогресс резко увеличил размеры хранилищ данных, стоимость которых существенно снизилась, а также повлиял на появление более быстрых процессоров и сетей. Во-вторых, интернет упростил сбор больших объемов данных, и люди не преминули этим воспользоваться. Например, данные проекта одной крупной компании по сканированию и переводу книг были использованы для значительного улучшения отдельного переводческого проекта. Еще один пример — картографический проект, который способствовал

разработке беспилотных автомобилей. Эти и многие другие результаты были настолько убедительны, что теперь машинное обучение применяется повсеместно. Совсем недавно люди также осознали, что те же самые тенденции — более дешевое хранилище и большая вычислительная мощность вкупе со сбором больших объемов данных — способствуют возрождению искусственного интеллекта, что также приводит к большому количеству новых рабочих мест в этой области.

Однако спешка в использовании машинного интеллекта подобна философии «Что может пойти не так?», которая уже пронизывает мир компьютерной безопасности. Мы знаем еще недостаточно, чтобы избежать создания психотических систем, таких как HAL из фильма «2001: Космическая одиссея» (1968).

Обзор

Вы уже наверняка поняли, что программирование — это утомительная работа по поиску решений всевозможных задач. Постановка задач и поиск решений — это интересно и сложно. Многие предпочут посвятить всю жизнь тому, чтобы научить компьютер выполнять работу за них, вместо того чтобы делать ее самим (еще один пример «своеобразной инженерной лени» описан в главе 5).

Как сказано выше, все началось с искусственного интеллекта, затем пришли машинное обучение и большие данные. Понятие искусственного интеллекта (хотя это название он и получил лишь в 1956 году на семинаре в Дартмуте) восходит к греческой мифологии. С тех пор многие философи и математики пытались разработать формальные системы для кодирования человеческого разума — ни одну из них нельзя назвать настоящим искусственным интеллектом, но эти работы заложили фундамент для дальнейших исследований. К примеру, Джордж Буль, с которым мы познакомились, изучая алгебру в главе 1, в 1854 году опубликовал трактат «Исследование законов мышления, на которых основываются математические теории логики и вероятностей».

Мы накопили много информации о том, как люди принимают решения, но до сих пор не знаем, как они думают. Например, вы и ваши друзья наверняка сможете отличить кота от котлеты, но придет к этому выводу разными путями. Тот факт, что одни и те же данные дают один и тот же результат, не означает, что все мы обрабатываем их одинаково. Мы знаем входные и выходные данные, понимаем части «железа», но все же мало знаем о «программировании», которое преобразует одно в другое. Отсюда следует, что способы распознавания котов и котлет, которые используют компьютер и человек, также будут различаться.

И все-таки мы кое-что знаем о том, как человеческое мышление обрабатывает информацию, — мы действительно хорошо разбираемся в статистике на подсознательном уровне (это не то же самое, что специально изучать статистику как предмет). Например, лингвисты изучали, как люди усваивают язык. Младенцы проводят внушительный статистический анализ, учатся выделять важные звуки

из окружающей среды, разделять их на фонемы, а затем группировать в слова и предложения. Продолжаются споры о том, есть ли у людей специализированные механизмы для такого рода вещей или мы просто обрабатываем информацию универсальным образом.

Младенцы учатся с помощью статистического анализа именно потому, что располагают большим количеством данных для анализа. Если не брать в расчет исключительные случаи, младенцы постоянно подвергаются воздействию звука. Точно так же они обрабатывают постоянный поток визуальной информации и других сенсорных данных. Дети учатся, обрабатывая огромное количество данных, или просто *большие данные* (big data).

Стремительный рост вычислительной мощности, емкости хранилища и различных типов подключенных к сети датчиков (включая мобильные телефоны) привел к тому, что нам (а не только плохим парням из предыдущей главы) приходится собирать данные в больших объемах. Некоторые из этих данных организованы, а некоторые нет. Пример организованных данных — показатели высоты волн, собранные при помощи морских буев. Неорганизованные данные — это, к примеру, смесь окружающих звуков. Что нам делать со всем этим?

Что ж, статистика — это одна из четко определенных областей математики, что означает, что мы можем писать программы, выполняющие статистический анализ. Мы также можем писать программы, а затем обучать их, используя данные. Например, один из способов реализации спам-фильтров (который мы вскоре рассмотрим более подробно) состоит в том, чтобы отправить большое количество собранного спама и неспама в статистический анализатор, одновременно сообщая анализатору, что является спамом, а что — нет. Другими словами, мы загружаем организованный набор *обучающих данных* в программу и сообщаем ей, что означают эти данные (то есть что является спамом, а что — нет). Это называется *машинным обучением* (machine learning, ML). Как и в случае с Гекльберри Финном, мы «изучаем эту машину», а не «обучаем» ее.

Во многих отношениях системы машинного обучения аналогичны функциям вегетативной нервной системы человека. Человеческий мозг не участвует активно во многих низкоуровневых процессах, таких как дыхание, — он зарезервирован для функций более высокого уровня, например для выяснения того, что сегодня на ужин. Функции низкого уровня обеспечиваются вегетативной нервной системой, которая беспокоит мозг только тогда, когда что-то требует внимания. Машинное обучение сейчас хорошо подходит для распознавания чего-либо, но не для активных действий.

Неорганизованные данные — это нечто иное. Мы говорим о *больших данных* — их люди не могут охватить без посторонней помощи. В этом случае для поиска закономерностей и взаимосвязей используются различные статистические методы. Например, подход под названием *дата майнинг* (data mining) можно использовать для извлечения музыки из окружающих звуков (в конце концов,

как сказал французский композитор Эдгар Варез (Edgard Varèse), музыка — это организованный звук).

По сути, все вышеперечисленное представляет собой поиск способов преобразования сложных данных во что-то более простое. Например, можно научить систему машинного обучения отличать котов от котлет. Для этого придется преобразовывать очень сложные данные из изображения в простые части котов и котлет. Это процесс называется *классификацией*.

Еще в главе 5 мы обсуждали разделение инструкций и данных. В главе 13 я предостерег вас от того, чтобы рассматривать данные как инструкции, изображения безопасности. Но бывают случаи, когда это имеет смысл, потому что так работает классификация на основе данных.

Например, нельзя создать беспилотный автомобиль только на основе классификации. Чтобы реализовать такой вариант поведения, как «не сбивайте кошек, но проехаться по котлете можно — в обществе это не наказуемо», нужно написать целый набор сложных программ, которые воздействуют на выходные данные классификатора. Существует множество способов реализовать нужное поведение, включая комбинации отклонений и изменений скорости. Необходимо учитывать большое количество переменных, таких как встречное движение, относительное положение препятствий и т. д.

Люди не учатся водить машину по сложным и подробным инструкциям наподобие «поворните руль на один градус влево и слегка нажмите на педаль тормоза в течение трех секунд, чтобы избежать столкновения с котом» или «поворните руль на три градуса вправо и выжмите педаль газа, чтобы переехать котлету». Вместо этого они работают с такими задачами, как «не сбивать кошек». Как я уже говорил, люди «программируют» себя, и у нас пока нет возможности изучить механизмы этого программирования, чтобы определить, какие пути мы выбираем для достижения целей. Если вы наблюдаете за машинами на улице, вы увидите, насколько по-разному люди выполняют одни и те же базовые задачи.

В подобных случаях люди не просто совершенствуют классификаторы, а пишут новые программы. Если действия совершают компьютеры, мы называем это *искусственным интеллектом (ИИ)*. ИИ-системы пишут собственные программы для достижения целей. Чтобы в процессе ни один кот не пострадал, мы создаем системы искусственного интеллекта с генерацией симулированных входных данных. Конечно, философские термозвездные устройства наподобие Бомбы № 20 из фильма «Темная звезда» (1974) показывают, что не всегда все идет, как задумано.

Большая разница между машинным обучением и искусственным интеллектом заключается в способности исследовать систему и «понимать» «мыслительные процессы». Пока системы машинного обучения для этого не предназначены, но уже можно сделать нечто подобное с ИИ. Неясно, продолжит ли это работать,

когда системы ИИ сильно разрастутся. Маловероятно, что процессы внутри систем искусственного интеллекта будут похожи на человеческое мышление.

Машинное обучение

Посмотрим, найдем ли мы способ отличить фотографию кота от фотографии котлеты. Фотографии не передают весь объем информации, который мы получаем, видя настоящих котов и котлеты. Например, люди обнаружили, что коты обычно пытаются убежать, если за ними кто-то гонится, а вот котлеты обычно так себя не ведут. Мы попытаемся создать процесс, который будет сообщать, «видит» ли он на фотографии кота, котлету или же ничего из перечисленного. На рис. 14.1 представлены оригинальные изображения: слева — кот Тони, похожий на котлету, а справа — настоящая котлета.

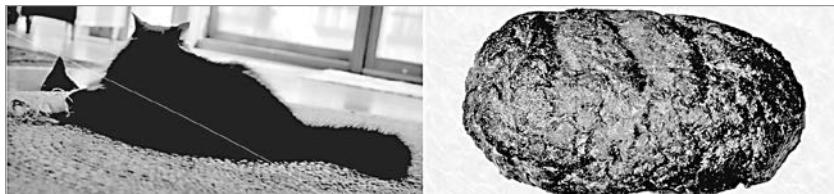


Рис. 14.1. Оригинальные изображения кота Тони и котлеты

Скорее всего, вы не раз столкнетесь со статистикой при работе с машинным интеллектом, поэтому пройдемся по теории.

Байес

Английского священника Томаса Байеса (Thomas Bayes) (1701–1761) наверняка очень интересовало, может ли его паства вознестись на небеса, поскольку он провел немало часов, размышляя о вероятности. В частности, Байес изучал вероятность возникновения сразу нескольких событий. Например, если вы играете в нарды, то уже разбираетесь в вероятности распределения чисел при подбрасывании пары шестигранных костей. Байес также стал автором одноименной теоремы.

В контексте данной книги нас больше всего интересует часть его трудов под названием *наивный байесовский классификатор*. Ненадолго оставим пример с котами и котлетами и попробуем отделить сообщения, содержащие спам, от других сообщений, вполне безобидных. Сообщения представляют собой наборы слов, и некоторые из этих слов чаще всего встречаются в спаме, благодаря чему можно предположить, что сообщения, в которых отсутствуют определенные слова, не содержат спама.

Начнем со сбора простых статистических данных. Допустим, у нас есть репрезентативная выборка сообщений, 100 из которых содержат спам, а 100 других — не содержат. Разделим все сообщения на отдельные слова и подсчитаем, как часто встречается каждое слово. Поскольку сообщений ровно 100, мы сразу получаем значения в процентах. Результат представлен в табл. 14.1.

Таблица 14.1. Статистика слов в сообщениях

Слово	Процентное соотношение в спаме	Процентное соотношение не в спаме
котлета	80	0
гамбургер	75	5
кошачья мята	0	70
лук	68	0
мышки	1	67
это	99	98
и	97	99

Как видим, некоторые слова встречаются и в спаме, и не в спаме. Сравним данные из таблицы с неизвестным сообщением, в котором есть слова «гамбургер» и «лук»: процент спама, соответственно, равен 74, 97 и 68, а процент неспама — 5, 97 и 0. Какова вероятность, что это сообщение содержит спам, и какова вероятность, что не содержит?

Теорема Байеса утверждает, что нужно сравнить вероятности (p), где p_0 — это вероятность того, что сообщение со словом «котлета» содержит спам, p_1 — вероятность того, что сообщение со словом «гамбургер» содержит спам, и т. д.:

$$p_{общая} = \frac{p_0 p_1 p_2 \dots p_n}{p_0 p_1 p_2 \dots p_n + (1-p_0)(1-p_1)(1-p_2)\dots(1-p_n)}.$$

Этот процесс наглядно представлен на рис. 14.2. События и их вероятности, такие как представленные в табл. 14.1, помещаются в классификатор, и в результате получается объединенная вероятность событий.

Можно создать пару классификаторов: один — для спама, а второй — для неспама. Подставив числа из таблицы выше, получим вероятность того, что сообщение содержит спам, равную 99.64 %. Вероятность того, что сообщение не содержит спама, — 0 %.

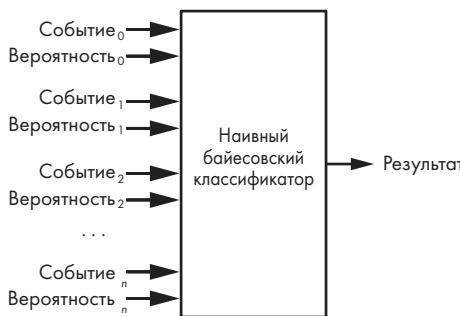


Рис. 14.2. Наивный байесовский классификатор

Очевидно, это довольно неплохой метод. Статистика рулит! Конечно, стоит изучить еще некоторые приемы, чтобы создать достойный спам-фильтр. Например, выясним, что имеется в виду под словом «наивный». Это вовсе не означает, что Байес понятия не имел, что он делает. Это лишь знак того, что рассматриваемые события не связаны между собой, как и в случае с подбрасыванием костей. Можно улучшить спам-фильтр, обратив внимание на связи между словами, приняв, например, тот факт, что словосочетание «и и» можно встретить, разве что когда речь идет о булевой алгебре. Многие спамеры пытаются обойти фильтры, добавив большое количество «словесного винегрета» в сообщения — в результате редко получается что-то грамматически корректное.

Гаусс

Немецкий математик Иоганн Карл Фридрих Гаусс (1777–1855) — не последний человек в мире статистики. Именно благодаря ему мы теперь знакомы с *колоколообразной кривой* (рис. 14.3), также называемой *нормальным распределением*, или *гауссовым распределением*.

Колоколообразная кривая интересна тем, что выборки наблюдаемых явлений соответствуют кривой. Например, если измерить рост баскетболистов в парке и определить средний рост μ , то некоторые игроки будут выше него, а некоторые — ниже. Среди игроков 68 % будут находиться в пределах одного *стандартного отклонения*, или σ , 95 % — в пределах двух стандартных отклонений и т. д. Точнее сказать, что распределение роста сходится к кривой нормального распределения по мере увеличения выборки, потому что рост одного игрока

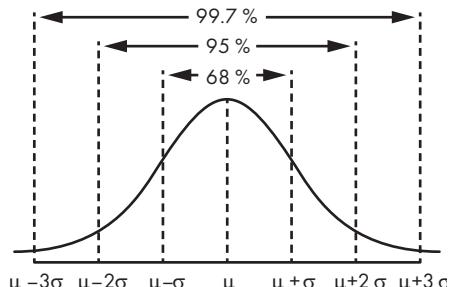


Рис. 14.3. Колоколообразная кривая

мало что нам скажет. Тщательно отобранные данные из четко определенной популяции можно использовать для предположений о более крупных популяциях.

Это очень интересно, однако существует множество других применений кривой нормального распределения, и некоторые из них можно использовать в задаче о котах и котлетах. Американский карикатурист Бернард Клибан (Bernard Kliban) (1935–1990) учит нас, что кот — это, по сути, котлета с ушами и хвостом. Из этого следует, что полезно было бы извлекать из фотографий такие признаки, как уши, хвосты и котлеты. Тогда их можно передавать классификатору для дальнейшей идентификации.

Можно сделать признаки объектов более узнаваемыми, выделив их очертания. Это невозможно сделать, не определив сначала контуры объектов, а как известно, и коты, и изрядное количество котлет не имеют четких контуров. У кошек, в частности, много отдельных волосков шерсти, которые сами по себе являются контурами, но нам это не поможет. Хотя это может показаться нелогичным, предварительно понадобится слегка размыть изображения, избавившись от некоторых из этих нежелательных аспектов. Размытие изображения означает применение фильтра нижних частот, что мы делали для звука в главе 6. Мелкие детали изображения — это «высокие частоты». Это проще понять, если представить, что мелкие детали меняются быстрее при сканировании изображения.

Посмотрим, как можно использовать труды Гаусса. Возьмем кривую с рис. 14.3 и обернем ее вокруг μ , чтобы превратить ее в трехмерную версию, как показано на рис. 14.4.

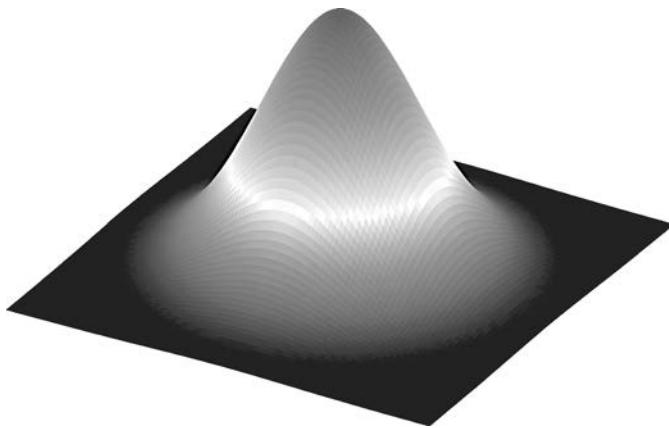


Рис. 14.4. Трехмерная версия гауссова распределения

Будем перемещать кривую по изображению, поочередно выставляя μ в центр каждого пикселя. Представьте, как части кривой покрывают другие пиксели, окружающие центральный. Затем рассчитаем новое значение каждого пикселя,

умножив значения пикселей под кривой на значения на кривой и складывая результаты. Этот процесс, показанный на рис. 14.5, называется гауссовым размытием. Изображение в середине представляет собой увеличенную копию выделенной квадратиком части фотографии слева. На изображении справа показано, как гауссово размытие взвешивает наборы пикселей центрального изображения.

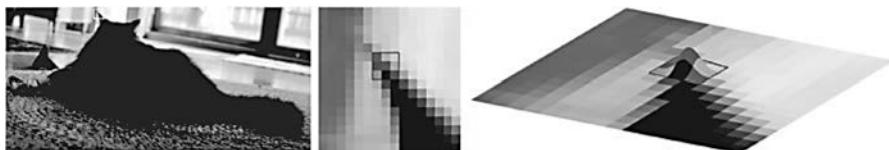


Рис. 14.5. Гауссово размытие

Процесс объединения значения пикселя и значений соседних пикселей в математике называется *сверткой*. Массив весов — это *ядро свертки*, или просто *ядро*. Разберем несколько примеров.

На рис. 14.6 представлены ядра размером 3×3 и 5×5 . Сумма всех весов составляет 1, что соответствует максимальной яркости изображения.

3x3			5x5				
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$	$\frac{1}{256}$	$\frac{4}{256}$	$\frac{6}{256}$	$\frac{4}{256}$	$\frac{1}{256}$
$\frac{2}{16}$	$\frac{4}{16}$	$\frac{2}{16}$	$\frac{4}{256}$	$\frac{16}{256}$	$\frac{24}{256}$	$\frac{16}{256}$	$\frac{4}{256}$
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$	$\frac{6}{256}$	$\frac{24}{256}$	$\frac{36}{256}$	$\frac{24}{256}$	$\frac{6}{256}$
			$\frac{4}{256}$	$\frac{16}{256}$	$\frac{24}{256}$	$\frac{16}{256}$	$\frac{4}{256}$
			$\frac{1}{256}$	$\frac{4}{256}$	$\frac{6}{256}$	$\frac{4}{256}$	$\frac{1}{256}$

Рис. 14.6. Ядра гауссовой свертки

На рис. 14.7 оригинал фотографии расположен слева. Очертания стволов деревьев можно определить, даже несмотря на то, что они не очень четкие. Центральное изображение показывает результаты наложения ядра размером 3×3 — оно более размытое, но контуры дерева легче различить. На правом изображении показаны результаты наложения ядра 5×5 .

Изображение можно представить как математическую функцию в формате $яркость = f(x, y)$. Значение данной функции — яркость пикселя в каждой точке координат. Обратите внимание, что эта функция *дискретна*, то есть значения x и y должны быть целыми числами. И, конечно же, они должны находиться внутри границ изображения. Аналогичным образом ядро свертки можно представить как маленькое изображение со значением $вес = g(x, y)$. Таким образом,

процесс наложения свертки состоит из прохождения по соседним пикселям, входящим в ядро, умножения значений пикселей на веса и их сложения.



Рис. 14.7. Пример гауссова размытия

На рис. 14.8 представлены результаты наложения гауссова ядра размером 5×5 на оригиналы фотографий.

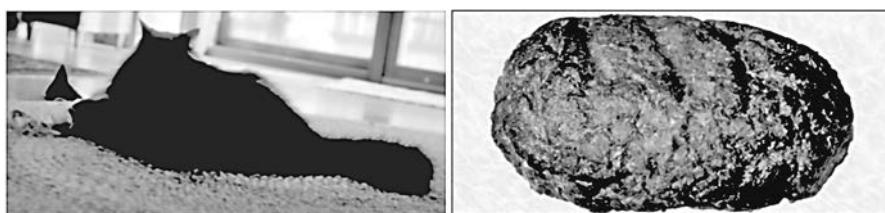


Рис. 14.8. Размытые кот и котлета

Обратите внимание: размер ядер свертки больше чем 1 пиксель, поэтому они выходят за рамки изображения. Эту проблему можно решить несколькими способами — например, можно не доходить до границ изображения (в этом случае результат будет меньшего размера) или нарисовать рамку, добавив недостающие пиксели.

Собель

На рис. 14.1 представлено много информации, и не вся она пригодится для определения предмета по фотографии — например, знать цвет для этого не обязательно. В книге «Understanding Comics: The Invisible Art» («Понимание комикса: невидимое искусство») Скотт Макклайд (Scott McCloud) показывает,

что достаточно нарисовать круг, две точки и линию, чтобы определить, что это лицо. Остальные детали избыточны, и их можно игнорировать. Таким же образом нужно упростить наши изображения.

После размытия будет легче найти контуры объектов на фотографиях. Вообще, *контур* имеет множество определений. Человеческий глаз легче воспринимает изменения яркости, поэтому будем опираться именно на них. Изменение яркости — это разница между яркостью пикселя и яркостью соседних с ним пикселей.

Почти половина вычислений связана с изменениями, поэтому их можно применить и для нашего примера. *Производная функции* — это наклон кривой графика функции. Если нужно вычислить изменение яркости от одного пикселя к другому, подойдет формула *яркость* = $f(x + 1, y) - f(x, y)$.

Обратите внимание на горизонтальный ряд пикселей на рис. 14.9. Под ним располагается график уровня яркости, а еще ниже — график изменений яркости. Нижний график по виду напоминает колючки, потому что изменения яркости никогда не равны нулю.

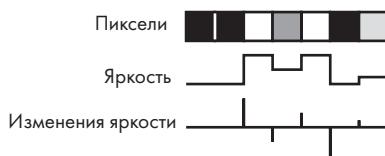


Рис. 14.9. Контуры — это изменения яркости

Проблема измерений изменения яркости заключается в том, что изменения происходят в щелях между пикселями, а нам нужно, чтобы они были в самих пикселях. Посмотрим, поможет ли нам в этом Гаусс. При наложении размытия мы располагали μ по центру пикселя. Будем использовать тот же подход, но вместо кривой нормального распределения возьмем ее первую производную (рис. 14.10), которая изображает наклон кривой с рис. 14.3.

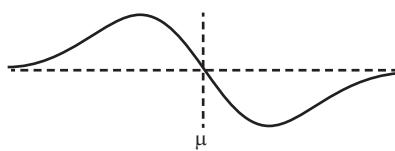


Рис. 14.10. Наклон гауссовой кривой с рис. 14.3

Отметим положительные и отрицательные пики кривой как +1 и -1 соответственно и расположим их над центрами соседних пикселей. Тогда изменение яркости пикселя вычисляется как $\Delta\text{яркость}_n = 1 \times \text{пиксель}_{n-1} - 1 \times \text{пиксель}_{n+1}$. Результат показан на рис. 14.11.

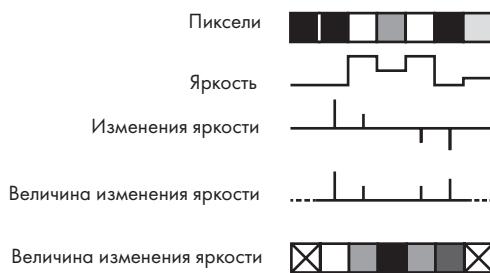


Рис. 14.11. Изменения яркости, размещенные над центрами пикселей

Конечно, проблема с границами изображения здесь та же, что и в предыдущем разделе, поэтому значения крайних пикселей не вычисляются. Сейчас нас интересует не направление изменения, а лишь количество, поэтому для расчета величины берем абсолютное значение.

Уже на данном этапе выделение контуров работает неплохо, но многие исследователи пытались улучшить этот алгоритм. Один из лучших подходов под названием *оператор Собеля* был представлен в 1968 году в статье американских ученых Ирвина Собеля (Irwin Sobel) и Гэри Фельдмана (Gary Feldman).

Как и в случае с гауссовым ядром размытия, мы генерируем ядро выделения контуров Собеля при помощи значений, полученных из наклона гауссовой кривой. Двумерная версия изображена на рис. 14.10, а трехмерная — на рис. 14.12.

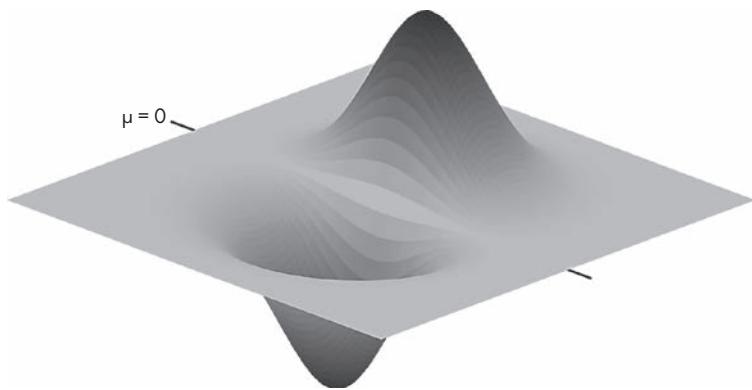


Рис. 14.12. Трехмерная версия наклона гауссовой кривой с рис. 14.10

Кривые не симметричны относительно осей координат, поэтому Собель использовал две версии ядра: одну для горизонтального направления, а вторую для вертикального. Оба ядра представлены на рис. 14.13.

Сobelъ _x			Сobelъ _y		
+1	0	-1	+1	+2	+1
+2	0	-2	0	0	0
+1	0	-1	-1	-2	-1

Рис. 14.13. Ядра Собеля

В результате использования данных ядер получим пару *градиентов*, G_x и G_y , для каждого пикселя. По своей сути градиент подобен наклону кривой. Поскольку градиент затрагивает все направления декартовой системы координат, можно перевести его в полярные координаты при помощи тригонометрии. В результате получаем *величину G* и *направление θ*, как показано на рис. 14.14.

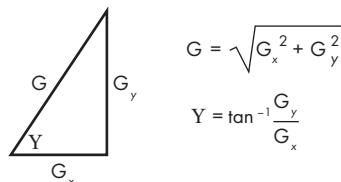


Рис. 14.14. Величина и направление градиента

Величина градиента говорит о том, насколько «четкий» контур мы получили, а направление, соответственно, определяет положение контура. Учтите, что направление перпендикулярно объекту, то есть у горизонтального контура вертикальный градиент.

Вы могли заметить, что вычисление величины и направления представляет собой не что иное, как перевод из декартовой системы координат в полярную — мы уже говорили об этом в главе 11. Изменять систему координат очень удобно — в случае с полярными координатами можно не думать о делении на ноль или больших числах при уменьшении знаменателей. Во многих математических библиотеках есть функция наподобие `atan2(y, x)`, которая вычисляет арктангенс без деления.

На рис. 14.15 представлены величины градиентов для обоих изображений.

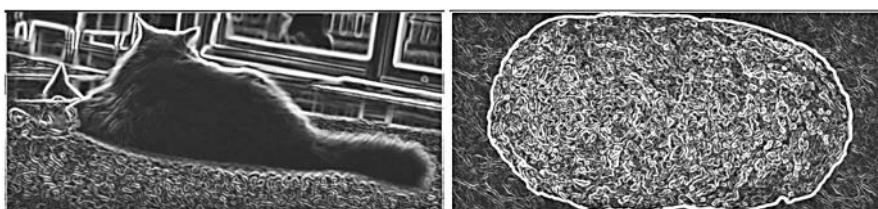


Рис. 14.15. Величины Собеля для размытых изображений кота и котлеты

Работая с направлением, мы сталкиваемся с очередной проблемой — направление предоставляет больше информации, чем нужно. Взгляните на рис. 14.16.



Рис. 14.16. Соседние пиксели

Поскольку каждый пиксель окружают ровно восемь соседних пикселей, можно говорить о четырех возможных направлениях. На рис. 14.17 показано, как направление разбивается на четыре группы.

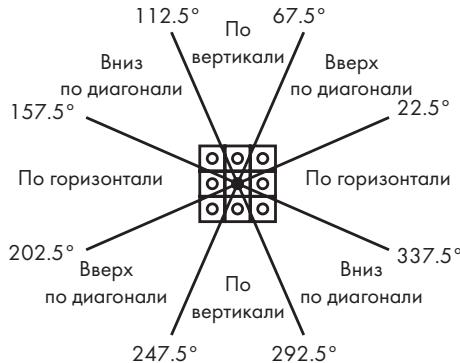


Рис. 14.17. Группы направлений градиента

На рис. 14.18 представлены сгруппированные направления Собеля для размытых изображений. Обратите внимание на соответствие между направлениями и величинами. Верхний ряд — это горизонтальная группа, за которой следуют группы «вверх по диагонали», вертикальная и «вниз по диагонали».

Видим, что оператор Собеля может найти контуры объектов, но они получаются слишком толстыми, и иногда их можно спутать с признаками объектов. Этого можно избежать, используя тонкие контуры, определяемые при помощи направлений Собеля.

Кэнни

Австралийский программист Джон Кэнни (John Canny) улучшил алгоритм выделения контуров в 1986 году, добавив к результату Собеля несколько шагов. Первый шаг называется *подавлением немаксимумов*. Вернувшись к рис. 14.15,

видим, что некоторые контуры толстые и нечеткие. Признаки объектов проще разглядеть, если их контуры тоньше. Подавление немаксимумов — это как раз один из способов *утончения контуров*.

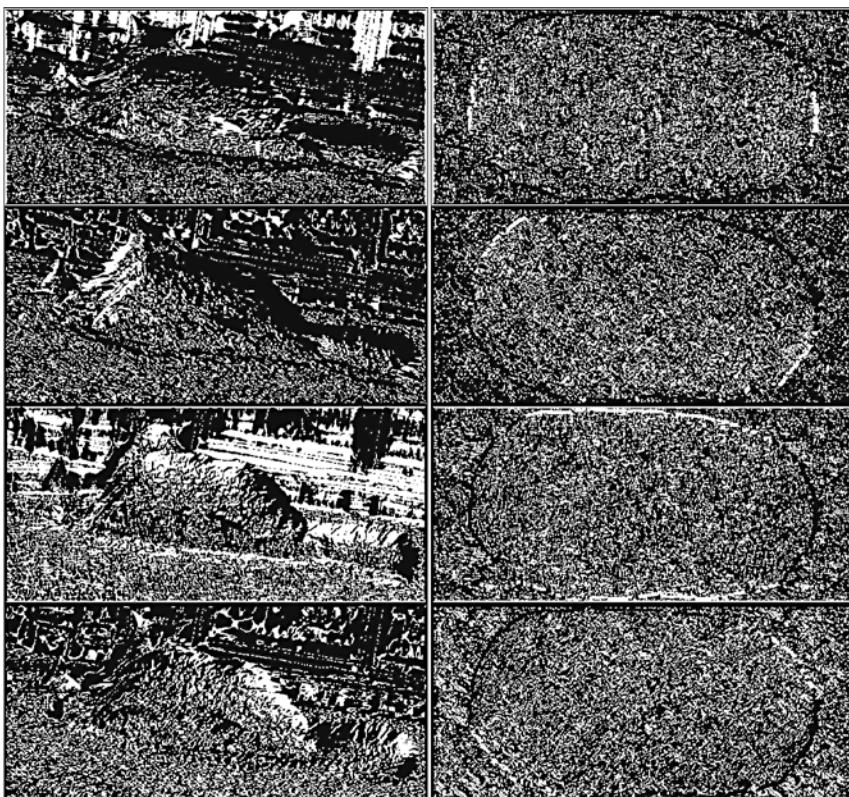


Рис. 14.18. Направления Собеля для размытых изображений кота и котлеты

У меня есть план. Сравним величины градиентов каждого пикселя с величинами градиентов соседних с ним пикселей в направлении градиента. Если величина выбранного пикселя больше, сохраним значение, в противном случае — применим подавление, то есть приравняем его значение к 0 (рис. 14.19).



Рис. 14.19. Подавление немаксимумов

По рис. 14.19 видно, что центральный пиксель слева сохраняется, поскольку он имеет большую величину (то есть большую яркость), чем соседние с ним

пиксели. Центральный пиксель справа подавляется, так как его величина меньше величин соседних пикселей.

На рис. 14.20 показано, как подавление немаксимумов позволяет сделать тоньше контуры, полученные при помощи оператора Собеля.

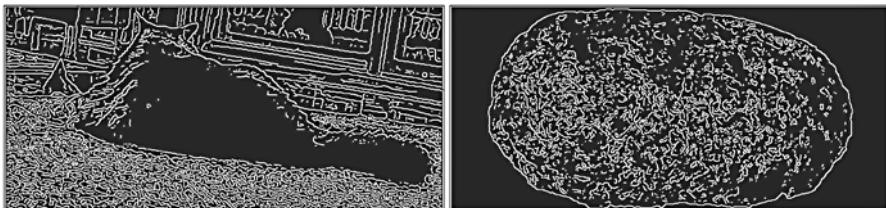


Рис. 14.20. Результат подавления немаксимумов на изображениях кота и котлеты

Выглядит неплохо, хотя можно разлюбить котлеты, взглянув на изображение справа. Подавление немаксимумов обнаружило множество контуров объектов. Если снова посмотреть на рис. 14.15, то увидим, что многие из этих контуров имеют низкие величины градиента. Последним этапом обработки по методу Кэнни является *отслеживание контуров с гистерезисом*, при котором удаляются «слабые контуры», а «сильные» остаются.

Еще в главе 2 вы узнали, что метод гистерезиса включает в себя сравнение с парой порогов. Просканируем результаты подавления немаксимумов, чтобы найти пиксели контура (на рис. 14.20 они белого цвета), у которых величина градиента превышает верхний порог. Найдя такие пиксели, пометим их как пиксели контура. Затем проверим все соседние пиксели — если величина градиента больше нижнего порога, этот пиксель также нужно пометить как пиксель контура. Повторим этот же алгоритм, используя рекурсию, пока не достигнем величины градиента, которая меньше нижнего порога. То есть мы находим начало четкого контура и отслеживаем его связи, пока это возможно. Результаты представлены на рис. 14.21. Сильные контуры окрашены в белый цвет, а отклоненные слабые — в серый.

Отметим, что мы избавились от множества контуров, полученных путем подавления немаксимумов, а края объектов стали хорошо видны.

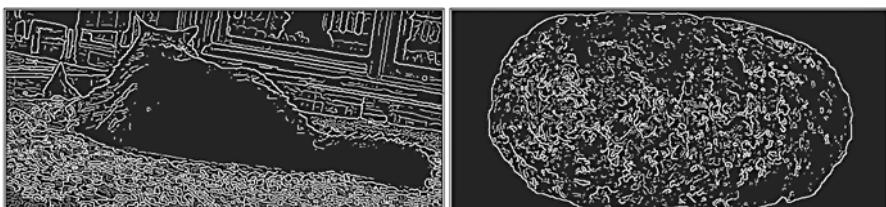


Рис. 14.21. Отслеживание контуров с гистерезисом

Существует отличная библиотека *OpenCV* с открытым исходным кодом для компьютерного зрения, которая пригодится для всех видов обработки изображений, включая примеры из этой главы.

Выделение признаков

Следующий шаг прост для людей, но сложен для компьютеров. Нам нужно выделить признаки изображений на рис. 14.21. Я не буду описывать выделение признаков подробно, потому что оно включает в себя много математических операций, с которыми вы, вероятно, еще не сталкивались, но основы мы все же рассмотрим.

Существует множество алгоритмов выделения признаков. Некоторые из них, такие как *преобразование Хафа*, хорошо подходят для выделения геометрических фигур, таких как линии и круги. Для нашего примера это не особо полезно, потому что мы ищем не геометрические фигуры. Начнем с простого — просканируем изображение на наличие контуров и пройдем по ним, чтобы выделить объекты. Если контуры пересекаются, будем выбирать кратчайший путь.

В результате получаем капли, уши, кошачьи игрушки и волнистые линии, как показано на рис. 14.22 (только репрезентативная выборка).

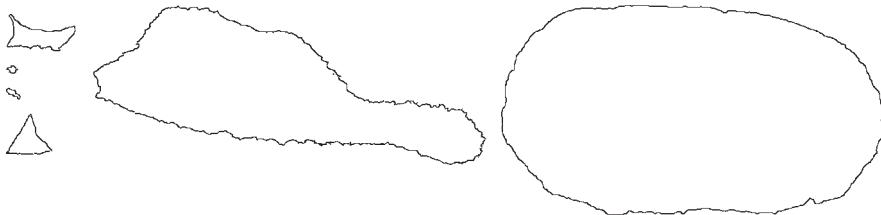


Рис. 14.22. Выделенные признаки

Получив признаки, мы отправляем их в классификатор (рис. 14.23), как и в примере с обнаружением спама среди сообщений. Входы классификатора, помеченные знаком «+», означают, что признак можно отнести к выбранному объекту. Знак «-» говорит о том, что признак не относится к выбранному объекту, а 0 означает, что признак нельзя отнести ни к одному из объектов.

Примечательно, что на изображениях мы выделили информацию, которую можно использовать для улучшения наивного классификатора — например, контуры кошачьих игрушек. Игрушки часто встречаются рядом с котами и редко — рядом с котлетами.

Этого примера достаточно для определения алгоритма классификации признаков, изображенного на рис. 14.24.

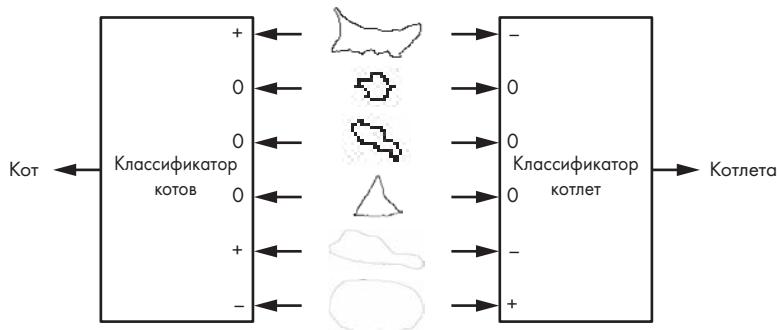


Рис. 14.23. Классификация признаков



Рис. 14.24. Алгоритм распознавания изображений

Известно, что котлеты в основном ведут сидячий образ жизни, в то время как кошки много двигаются и принимают множество милых поз. Наш алгоритм сработает только для объектов на фотографиях из примера, он не опознает кошку с рис. 11.44 на с. 390. И, не зная контекста, он вряд ли определит, что кот на рис. 14.25 — это Мит Лоуф (Meat Loaf)¹.



Рис. 14.25. Кот или Мит Лоуф?

¹ Игра слов. Мит Лоуф (певец), по-английски Meat Loaf пишется так же, как meatloaf (котлета). — Примеч. ред.

Нейронные сети

На определенном уровне тип данных, которые мы используем для представления объектов, перестает иметь значение. Мир постоянно меняется, и нужно уметь справляться с этим. Как и люди, компьютеры не могут изменять входные данные, поэтому придется улучшать классификаторы, чтобы работать со всем разнообразием.

Один из подходов, используемых в искусственном интеллекте, заключается в имитации человеческого поведения. Мы почти уверены, что оно сильно зависит от *нейронов*. У человека около 86 миллиардов нейронов, хотя не все они находятся в «мозге»: нервные клетки — это тоже нейроны. Возможно, именно поэтому некоторые люди думают нутром.

Нейрон можно представить как нечто среднее между логическими вентилями из главы 2 и аналоговыми компараторами из главы 6. Упрощенная схема нейрона показана на рис. 14.26.

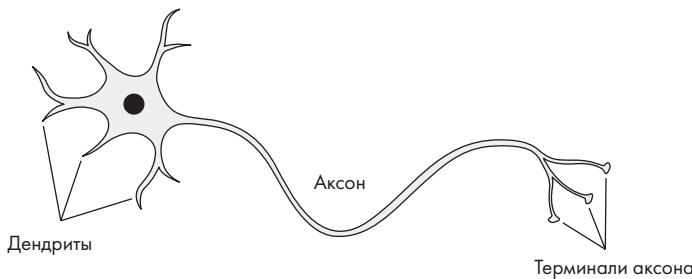


Рис. 14.26. Нейрон

Дендриты — это входы нейрона, а *аксон* — это его выход. *Терминали аксона* представляют собой связи аксона с другими нейронами, так как нейроны имеют всего один выход. Нейроны отличаются от вентиляй И в том плане, что они не обрабатывают все входы одинаковым образом. Взгляните на рис. 14.27.

Значение каждого входа дендрита умножается на заданный *вес*, после чего все взвешенные значения складываются — процесс напоминает байесовский классификатор. Если значения меньше *потенциала действия*, выход компаратора будет равен `false`. В противном случае — `true`, что приведет к *срабатыванию* нейрона и заданию выхода триггера равным `true`. Выход аксона представляет собой *импульс*. Как только импульс становится равным `true`, триггер сбрасывается и возвращается к значению `false`. Нейробиологи могут поспорить с тем, что компаратор в нашей схеме по умолчанию имеет гистерезис, — у настоящих нейронов он есть только в определенное время, чего наша модель не учитывает.

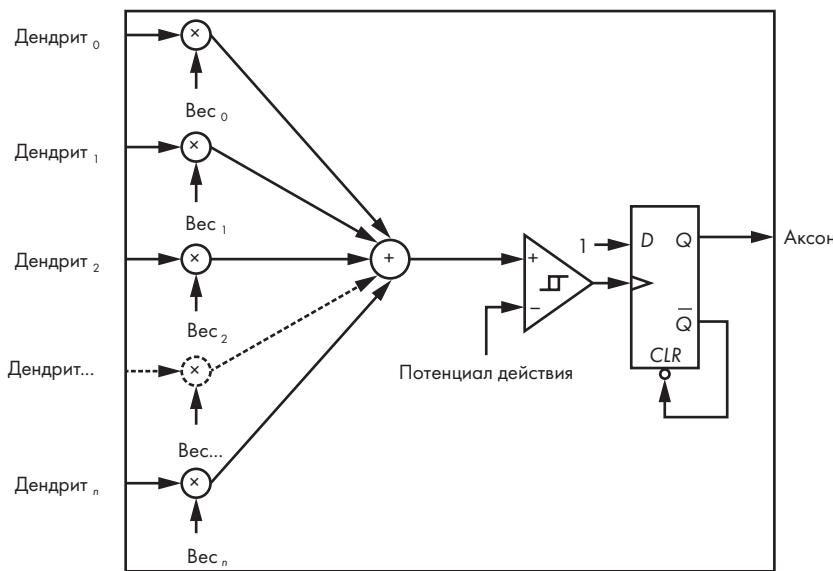


Рис. 14.27. Упрощенная модель логических вентилей нейрона

Нейроны похожи на вентили в том смысле, что они «просты», но при соединении друг с другом могут образовывать сложные «схемы», или *нейронные сети*. Самое важное в нейронах то, что они срабатывают на основе взвешенных входных данных. Определенные комбинации входных данных могут вызвать срабатывание нейрона.

Американский психолог Фрэнк Розенблatt (Frank Rosenblatt) (1928–1971) впервые попытался создать искусственный нейрон — *перцептрон*, схема которого показана на рис. 14.28. Важное свойство перцептронов заключается в том, что их входы и выходы двоичны, то есть их значения могут быть равны только 0 или 1. Веса и порог представляют собой действительные числа.

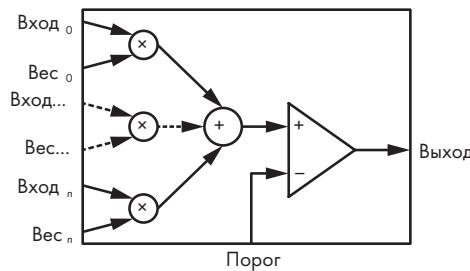


Рис. 14.28. Перцептрон

Перцептроны вызвали большой ажиотаж в мире ИИ, но позже выяснилось, что они не подходят для определенных классов задач. Это стало одной из причин наступления так называемой «зимы ИИ», когда финансирование новых разработок прекратилось.

Оказывается, проблема заключалась в способе использования перцепtronов. Они были организованы на одном «уровне», как показано на рис. 14.29, где каждый круг — это отдельный перцептрон.

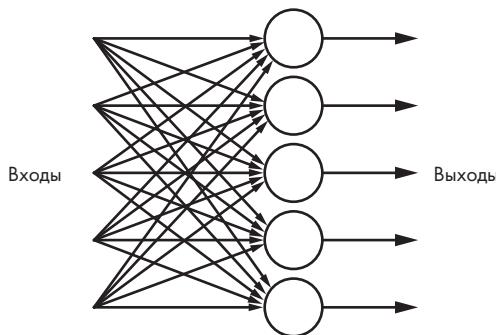


Рис. 14.29. Одноуровневая нейронная сеть

Входы могут соотноситься сразу с несколькими перцептрами, каждый из которых принимает единственное решение и получает выходные данные. Множество проблем с перцептрами было решено после изобретения многослойной нейронной сети, изображенной на рис. 14.30.

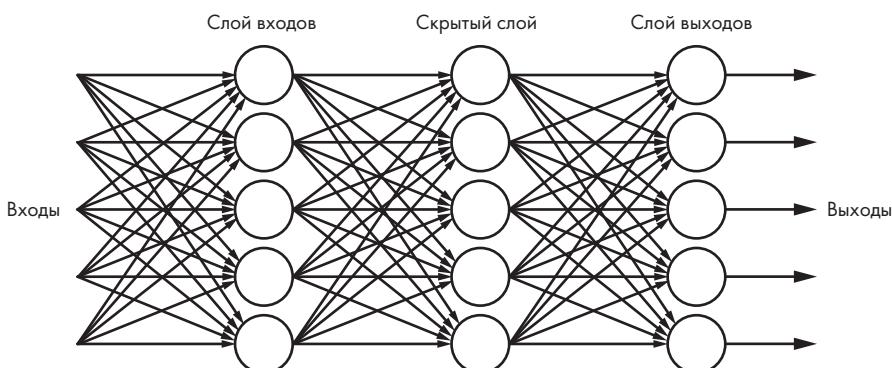


Рис. 14.30. Многослойная нейронная сеть

Такая схема еще называется *нейронной сетью прямого распространения*, поскольку выходные данные производятся на каждом слое и переносятся на

следующий слой. *Скрытых* слоев может быть несколько, и они называются так, потому что не подключаются ни ко входам, ни к выходам. На рис. 14.30 число нейронов на каждом слое одинаково, но так может быть не всегда. Определение количества слоев и количества нейронов в каждом слое для конкретной задачи — это черная магия, изучение которой выходит за рамки данной книги. У нейронных сетей наподобие той, что мы рассмотрели, гораздо больше возможностей, чем у простых классификаторов.

Нейробиологи пока еще не уверены, как определяются веса дендритов, — но программистам нужны хотя бы примерные значения, иначе искусственные нейроны окажутся бесполезными. Цифровая природа перцептронов усложняет задачу, поскольку небольшие изменения весов не ведут к пропорциональным изменениям выходных данных — перцептроны действуют по принципу «все или ничего». Для решения этой проблемы было предложено использовать *сигмовидные нейроны* — новую модель искусственных нейронов. В этом случае компаратор перцептронов заменяется на сигмоидную функцию, график которой представляет собой S-образную кривую. На рис. 14.31 изображены графики передаточных функций перцептронов и сигмоидной функции. Все это очень напоминает примеры из главы 2, где мы обсуждали аналоговый и цифровой подходы, не правда ли?

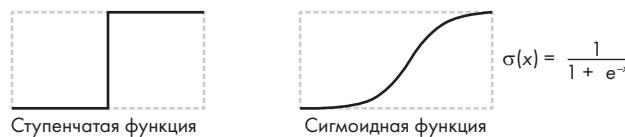


Рис. 14.31. Передаточные функции искусственных нейронов

На рис. 14.32 представлена структура сигмовидного нейрона.

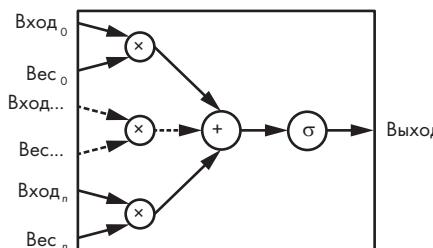


Рис. 14.32. Сигмовидный нейрон

Неочевидный факт заключается в том, что входы и выходы сигмовидного нейрона являются «аналоговыми» числами с плавающей точкой. Хочу уточнить — в сигмовидном нейроне также используется смещение, но для нас это несущественно.

Веса для нейронных сетей, построенных из сигмовидных нейронов, можно определить с помощью метода *обратного распространения*, который регулярно забывают и открывают вновь. Последний раз это произошло совсем недавно — в 1986 году была опубликована статья под авторством Дэвида Румельхарта (David Rumelhart) (1942–2011), Джейфри Хинтона (Geoffrey Hinton) и Роланда Уильямса (Roland Williams). Обратное распространение использует линейную алгебру, поэтому мы не будем вдаваться в детали. Вы, вероятно, уже умеете решать системы алгебраических уравнений, а линейная алгебра помогает справляться с огромным количеством уравнений с большим количеством переменных.

Общая идея обратного распространения заключается в том, что мы предоставляем входные данные для чего-то известного, например для признаков кота, а затем проверяем выходные данные. При условии, что на вход нейронной сети был передан кот, мы ожидаем, что на выходе получим 1 или очень близкое к нему число. Можно вычислить *функцию ошибок*, которая представляет собой фактический результат, выченный из желаемого. Затем веса корректируются так, чтобы значение функции ошибок было как можно ближе к 0.

Обычно это делается с помощью алгоритма *градиентного спуска*, который очень похож на спуск Данте в ад, если вы не любите математику. Разберем его на простом примере. Помните, что «градиент» означает то же, что и «наклон». Возьмем разные значения весов и построим график значений функции ошибок. Он может выглядеть примерно так, как показано на рис. 14.33, — наподобие физической карты, на которую нанесены горы, впадины и т. д.

Градиентный спуск представляет собой перекатывание мяча по карте, до тех пор пока он не приземлится в самой глубокой впадине, где значение функции ошибок минимально. Мы задаем веса для значений, которые представляют положение мяча. Этот алгоритм получил свое причудливое название именно потому, что он выполняется для очень большого числа весов, а не для двух, как в нашем примере. Все усложняется еще больше, если веса располагаются в нескольких слоях, как показано на рис. 14.30.

Заметим, что выходной импульсный механизм, который мы видели на рис. 14.27, таинственным образом исчез. Нейронные сети, которые мы рассматривали, по сути, подчиняются комбинаторной логике, а не последовательной и фактически являются НАГ. Существует вариант с последовательной логикой под названием *рекуррентная нейронная сеть*. Она не является НАГ, это означает, что выходные данные нейронов в слое могут соединяться с входами нейронов в одном из предыдущих слоев. Не развалиться этой конструкции помогают хранение выходов и синхронизация всего беспорядка. Такие типы сетей хорошо подходят для обработки последовательностей входных данных, например при распознавании почерка и речи.

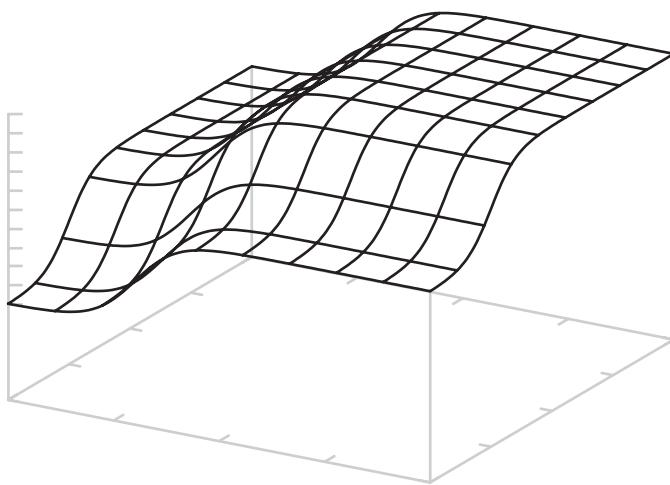


Рис. 14.33. Топология градиента

Существует еще одна разновидность нейронных сетей, которая особенно хорошо подходит для обработки изображений, — *сверточная нейронная сеть*. Входные данные в сверточных нейронных сетях представляют собой массив значений пикселей, что похоже на ядра свертки, о которых мы уже говорили.

Большая проблема нейронных сетей заключается в том, что их можно «отравить» плохими обучающими данными. Невозможно предугадать, как станут вести себя взрослые, которые слишком много смотрели телевизор в детстве, — то же верно и для систем машинного обучения. Возможно, в будущем нам понадобятся специальные психотерапевты для машин, хотя трудно представить, что они будут садиться рядом с машиной и говорить: «Расскажи, как ты на самом деле относишься к изображениям кошек».

По сути, нейронные сети представляют собой очень способные классификаторы. Их можно научить преобразовывать большой объем входных данных в меньший объем выходных данных, которые нужным образом описывают входные данные. Эстеты назовут это *уменьшением размерности*. Теперь мы должны выяснить, что делать со всей этой информацией.

Использование данных машинного обучения

Как создать что-то похожее на самоуправляемую бутылку с кетчупом при помощи выходных данных классификатора? Возьмем для примера сценарий на рис. 14.34. Будем перемещать бутылку с кетчупом как короля на шахматной доске так, чтобы она достигла котлеты, не столкнувшись с котом.

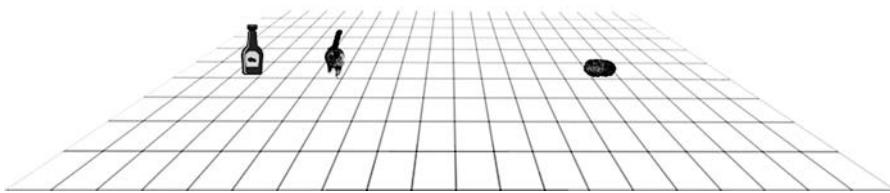


Рис. 14.34. Тестовый сценарий

В этом «примере из учебника» классификаторы предоставляют местоположения кота и котлеты. Кратчайшее расстояние между двумя точками — это прямая линия, а все объекты расположены на целочисленной сетке, поэтому самый эффективный способ добраться до котлеты — это использовать алгоритм Брезенхэма для рисования линий из раздела «Прямые линии» на с. 357. Конечно, его нужно изменить, как показано на рис. 14.35, потому что соус не очень подходит к котам.

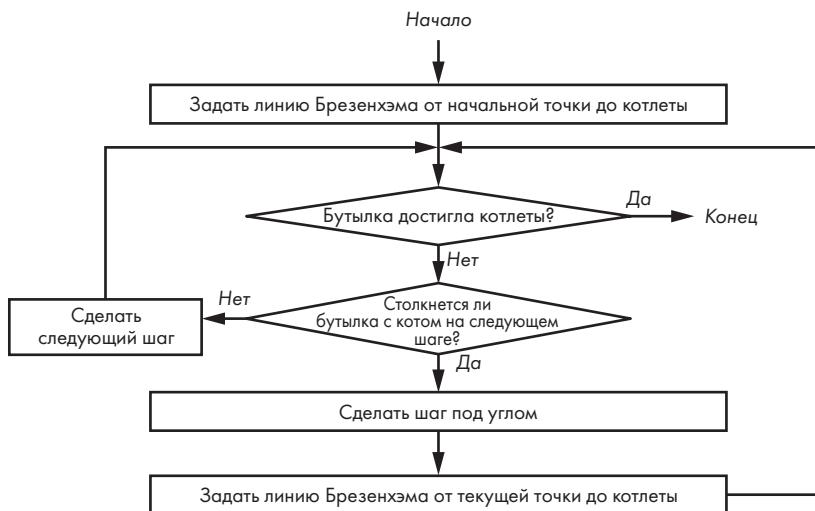


Рис. 14.35. Алгоритм движения самоуправляемой бутылки с кетчупом

Как видите, это несложно. Мы рисуем прямую линию до котлеты, и, если по пути встречается кот, мы отклоняемся и снова рисуем прямую линию до котлеты. Конечно, в реальной жизни все намного сложнее — кот сам может передвигаться, а на пути встречаются и другие препятствия.

Поговорим об одной из подобластей машинного обучения, которая подскажет другие способы решения подобных задач.

Искусственный интеллект (ИИ)

Ранние результаты применения искусственного интеллекта, например для обучения игре в шашки и решения различных логических задач, были впечатляющими и привлекли множество спонсоров. К сожалению, эти результаты нельзя было использовать для решения более сложных проблем, и финансирование прекратилось.

Одним из участников семинара в Дартмуте в 1956 году, где и появился термин «искусственный интеллект», был американский ученый Джон Маккарти (John McCarthy) (1927–2011), который создал язык программирования LISP, работая в Массачусетском технологическом институте. Этот язык использовался для большей части ранних работ в области ИИ, а LISP официально расшифровывается как *List Processor* (процессор для обработки списков), но любой, кто знаком с его синтаксисом, расшифрует эту аббревиатуру как *Lots of Insipid Parentheses* (множество бессодержательных скобок).

Благодаря LISP в языках программирования высокого уровня возникло несколько новых концепций. Конечно, в 1958 году это было несложно, поскольку в то время существовал только один язык высокого уровня (FORTRAN). В частности, в LISP односвязные списки (см. главу 7) рассматривались как тип данных, а программы — как списки инструкций. Это означало, что программы могли изменять сами себя, что существенно отличает их от систем машинного обучения. Нейронная сеть может корректировать веса, но не может изменить собственный алгоритм. Язык LISP может генерировать код и, соответственно, изменять или создавать новые алгоритмы. В некотором роде JavaScript также поддерживает *самомодифицирующийся код*, хотя в практически неограниченной среде интернета это небезопасно.

Ранние системы искусственного интеллекта быстро стали ограничиваться доступными аппаратными технологиями. Одним из наиболее распространенных компьютеров, использовавшихся в то время для исследований, был DEC PDP-10. Его адресное пространство изначально было ограничено 256 000 36-битных слов, а со временем расширилось до 4 000 000. Этого не хватит даже для запуска примеров из этой главы. Американский программист Ричард Гринблатт (Richard Greenblatt) и специалист по вычислительной технике Том Найт (Tom Knight) в начале 1970-х годов объединили свои усилия в Массачусетском технологическом институте с целью создания *LISP-машин* — компьютеров, оптимизированных для запуска кода на LISP. Однако даже в период их расцвета было выпущено всего несколько тысяч таких машин, возможно, потому что компьютеры общего назначения развивались более быстрыми темпами.

Искусственный интеллект начал возвращаться в 1980-х годах, когда мир впервые услышал об *экспертных системах*. Эти системы помогают пользователям —

например, медицинским работникам, — задавая им вопросы и ведя их по базе данных знаний. Наверняка вы уже узнали в них более серьезную версию нашей игры «Угадай животное» из главы 10. К сожалению, экспертные системы превратились в раздражающие телефонные меню.

Проблему самоуправляемой бутылки с кетчупом можно решить с помощью генетического алгоритма — метода, имитирующего эволюцию (рис. 14.36).

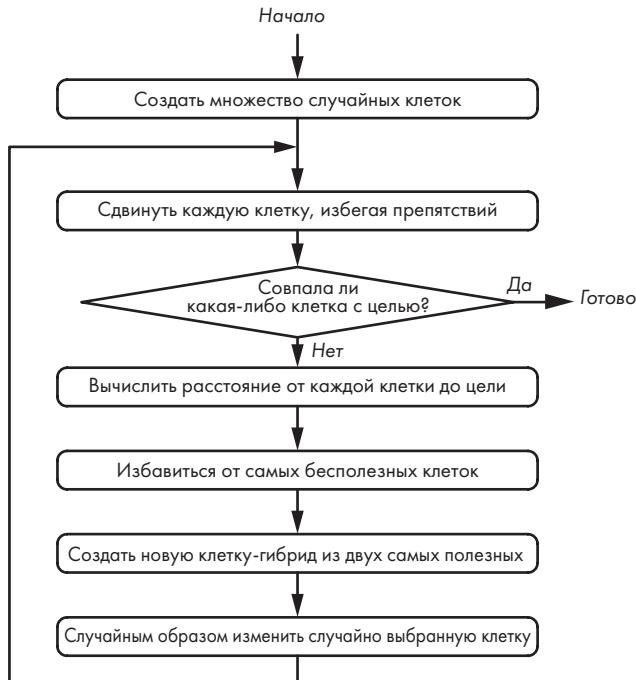


Рис. 14.36. Генетический алгоритм

Создадим случайный набор клеток, каждая из которых имеет определенное местоположение и направление движения. Каждая клетка сдвигается на один шаг, после чего заново рассчитывается ее рейтинг «полезности» — в нашем случае используется формула расстояния. После чего мы создаем новую клетку-гибрид из двух наиболее полезных и избавляемся от самых бесполезных клеток. Вместе с этим мы случайным образом изменяем одну из оставшихся клеток — в этом и заключается суть эволюции. Будем повторять эти шаги, до тех пор пока одна из клеток не достигнет цели. Набор шагов этой клетки вплоть до достижения цели — это и есть сгенерированная программа.

Проверим, как работает алгоритм, на сетке из 20 клеток. Нам повезло, и мы нашли решение, представленное на рис. 14.37, всего за 36 итераций.

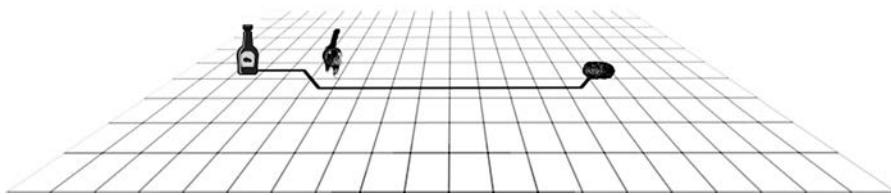


Рис. 14.37. Хорошие результаты генетического алгоритма

Алгоритм создает простую программу из листинга 14.1 для достижения цели. Важно понимать, что данная программа написана не программистом, а искусственным интеллектом. Переменные x и y задают местоположение бутылки с кетчупом.

Листинг 14.1. Сгенерированный код

```
x++;  
x++;  
x++; y++;  
x++;  
x++; y--;
```

Конечно, генетический алгоритм на то и генетический, что он слукаен и не всегда выдает такие же хорошие результаты. При новом запуске программы понадобилось 82 итерации, чтобы найти решение на рис. 14.38. Может, это пригодится для обоснования «разумного замысла»¹?



Рис. 14.38. Странные результаты генетического алгоритма

¹ Разумный замысел (Intelligent Design, ID) — одно из направлений креационизма, в рамках которого утверждается, что Вселенная и жизнь были созданы неким «разумным творцом». — Примеч. науч. ред.

Видим, что ИИ-программы могут генерировать удивительные результаты. Но они не так сильно отличаются от того, что придумывают дети, когда исследуют окружающий мир, — просто мы начали обращать на это больше внимания. На самом деле, многие удивительные результаты ИИ были предсказаны давно. Например, случай, когда системы ИИ в одной компании создали свой язык для общения друг с другом, вызвал серьезный интерес. Это не новость для тех, кто видел научно-фантастический фильм 1970 года «Колосс: Проект Форбина».

Большие данные

До сих пор в наших примерах обрабатывалось большое количество данных. Видеокамера с высоким разрешением (1920×1080) ежесекундно производит около трети гигабайта данных. Большой адронный коллайдер генерирует около 25 ГиБ в секунду. По некоторым оценкам, устройства с подключением к интернету генерируют около 50 ГиБ в секунду, по сравнению с 1 МиБ в секунду четверть века назад. Большая часть этой информации не представляет никакой ценности, и каждый раз приходится выискивать в ней что-то полезное.

Большие данные — очень интересная вещь; это данные, которые невозможно обработать методом «грубой силы» из-за их большого объема и сложности. Двадцать пять лет назад такое количество данных было бы легко обработать при помощи современных технологий, но тогда таких технологий не было. Объем собираемых данных, похоже, всегда будет превышать возможности технологий их анализа, поэтому приходится что-то изобретать.

Термин *большие данные* относится не только к анализу, но и к сбору, хранению и организации данных. Для целей этой книги достаточно будет рассмотреть только анализ.

Многие собранные данные носят личный характер. Не стоит делиться информацией о своем банковском счете или историей болезни с незнакомыми людьми. И данные, собранные для одной цели, зачастую используются совсем с другими намерениями. Например, нацисты использовали данные переписи для выявления евреев и их дальнейшего преследования. Данные американской переписи использовались для обнаружения и задержания американцев японского происхождения в целях интернирования, несмотря на то что по закону части этих данных, позволяющие установить личность, должны были оставаться конфиденциальными в течение 75 лет.

Многие данные публикуются в исследовательских целях в «анонимизированной» форме, что означает удаление любой информации, позволяющей установить личность. Однако это не так просто, как кажется, — методы больших данных часто могут повторно идентифицировать людей по анонимным данным. Многие политики, разработанные, чтобы затруднить повторную идентификацию, на самом деле упростили этот процесс.

В Америке номер социального страхования (Social Security number, SSN) регулярно используется в качестве личного идентификатора, хотя он никогда не был предназначен для этого. Фактически на карточке социального обеспечения есть фраза «не для идентификации», которая была частью первоначального закона, редко применявшегося и обросшего огромным количеством исключений.

SSN состоит из трех полей: трехзначного номера региона (area number, AN), двухзначного номера группы (group number, GN) и четырехзначного серийного номера (serial number, SN). Номер региона присваивается на основе почтового индекса адреса в анкете. Номера групп присваиваются в определенном, но непоследовательном порядке. Серийные номера присваиваются последовательно.

В 2009 году группа исследователей из Университета Карнеги — Меллона опубликовала статью, описывающую метод успешного угадывания SSN. Определить SSN оказалось легко по двум причинам. Прежде всего из-за наличия основного архива смертей — списка умерших, предоставленного службой социального обеспечения якобы для предотвращения мошенничества. Он включает имена, даты рождения, даты смерти, SSN и почтовые индексы в удобном формате. Как список умерших помогает угадывать SSN живых?

Нужно понимать, что это не единственный источник данных. Списки регистрации избирателей включают данные о рождении, как и многие онлайн-профили.

Статистический анализ номеров регионов и почтовых индексов в основном архиве смертей можно использовать для привязки AN к географическим регионам. Правила назначения GN и SN просты. В результате информация из основного архива смертей может использоваться для сопоставления AN с почтовыми индексами. Отдельно полученные данные о рождении также можно привязать к почтовым индексам. Эти два источника информации можно чередовать, выполнив сортировку по дате рождения. Любой пробел в последовательности SSN в основном архиве смертей — это живой человек, чей SSN находится между предыдущей и последующей записями архива. Пример приведен в табл. 14.2.

Конечно, на самом деле это не так просто, но и ненамного сложнее. Например, диапазон серийных номеров SSN можно узнать только в том случае, если в данных архива смертей есть разрыв, как между Джоном Чифом Краером и Джоном Смоллом Берризом в примере. Многие организации часто запрашивают последние четыре цифры SSN для идентификации. Как видно из примера, их труднее всего угадать, поэтому не упрощайте мошенникам задачу, выдавая их по первой просьбе. Настаивайте на других средствах идентификации.

Вот еще пример. Массачусетская комиссия коллективного страхования (Group Insurance Commission, GIC) опубликовала анонимные данные о больницах

с целью улучшения медицинского обслуживания и контроля над расходами. Губернатор Массачусетса Уильям Уэлд (William Weld) заверил общественность в сохранении конфиденциальности пациентов. Вы наверняка уже понимаете, к чему это привело. Мораль: он должен был держать рот на замке.

Таблица 14.2. Объединение данных о почтовом индексе 89044

Основной архив смертей			Предположительный SSN	Данные о рождении	
Имя	Дата рождения	SSN		Дата рождения	Имя
Джон Мэни Джарс	10.01.1984	051-51-1234			
Джон Фиш	01.02.1984	051-51-1235			
Джон Ту Хорнс	12.02.1984	051-51-1236			
			051-51-1237	14.02.1984	Джон Стейнхарт
Джон Уорфин	20.02.1984	051-51-1238			
Джон Бигбут	15.03.1984	051-51-1239			
Джон Я Я	19.04.1984	051-51-1240			
			051-51-1241	20.04.1984	Джон Гилмор
Джон Фледглинг	21.05.1984	051-51-1242			
			051-51-1243	22.05.1984	Джон Перри Барлоу
Джон Грим	02.06.1984	051-51-1244			
Джон Литлджон	03.06.1984	051-51-1245			
Джон Чиф Краер	12.06.1984	051-51-1246			
			051-51-1247	05.07.1984	Джон Джейкоб Джинглхаймер Шмидт
Джон Смолл Берриз	03.08.1984	051-51-1250			

Губернатор Уэлд потерял сознание во время церемонии 18 мая 1996 года и был госпитализирован. Аспирантка Массачусетского технологического института Латанья Суини (Latanya Sweeney) знала, что губернатор проживает в Кембридже,

и потратила 20 долларов на покупку полного списка избирателей этого города. Она объединила данные GIC с данными об избирателях, как это сделали мы в табл. 14.2, и легко раскрыла данные губернатора, отправив его медицинские записи, включая рецепты и диагнозы, в его же офис.

Это оказалось довольно просто, поскольку губернатор был публичной личностью, но и ваш телефон, вероятно, обладает большими вычислительными ресурсами, чем были в распоряжении у Суини в 1996 году. Сегодняшние вычислительные ресурсы позволяют решать гораздо более сложные задачи.

Выводы

В этой главе мы рассмотрели множество действительно сложных тем. Вы узнали о взаимосвязи машинного обучения, больших данных и искусственного интеллекта. Вы также узнали, что для более подробного изучения этих концепций понадобится подтянуть математику. При написании этой главы ни один кот не пострадал.

15

Влияние реальных условий



Поскольку эта книга предназначена для тех, кто учится писать код, вы наверняка уже знаете что-то о программах и аппаратных средствах, на которых те запускаются. Возможно, вы думаете, что уже готовы стать программистом. Однако программирование – это больше, чем знание аппаратной части и кодинг. Как узнать, какой именно код писать и как непосредственно его писать? Уверены ли вы, что ваш код будет работать?

Это еще не все важные вопросы, с которыми вам придется столкнуться. Смогут ли другие люди разобраться в вашем коде? Насколько легко добавить в него что-то новое или исправить баги? Можно ли перенести ваш код на другое устройство, отличающееся от компьютера, для которого он был изначально написан?

В этой главе рассматриваются темы, связанные с созданием ПО. Конечно, вы можете писать код для небольших задач, запершись в темной комнате с запасом фастфуда, но большинство проектов предполагают работу в команде и умение договариваться с другими людьми. Это не так просто, как может показаться. Люди – это программные/аппаратные системы, содержащие больше ошибок, чем самое плохое устройство интернета вещей. Забудьте о документации – она обязательно окажется устаревшей, если вообще будет в наличии.

Именно поэтому в этой главе вы встретите разбор некоторых философских и практических аспектов работы программиста. Да, старый ворчун все-таки попытается передать вам крупицы мудрости, добытой тяжким трудом.

Повышение ценности

Главный вопрос, который вы должны постоянно задавать себе, работая над проектом: «Повышаю ли я ценность проекта?» Я говорю не о внутренней ценности выполнения какой-то задачи, а о повышении производительности.

Если вы зарабатываете на жизнь программированием, от вас требуется достигать любых целей, поставленных работодателем. Однако делать это можно по-разному: просто выполнять работу, лишь бы хоть как-то решить задачу, или же подумать о том, что не пришло в голову руководству. Например, структурировать собственный код так, чтобы его было легко использовать повторно в другом проекте. Или исходить из того, что вам поручено реализовать частный случай общей проблемы: решив такую проблему, вы проложите путь для будущих улучшений. Конечно, предварительно стоит поговорить с руководством, чтобы это не стало сюрпризом.

Вы можете повысить собственную ценность, убедившись, что владеете различными технологиями. Параллельные проекты — популярный способ получить новый опыт. Это похоже на выполнение домашнего задания, только более увлекательного.

Один из классических способов, с помощью которого люди пытаются повысить ценность проекта, — создание инструментов. Это сложнее, чем кажется, потому что иногда повышение ценности для вас снижает ценность для других. Люди часто создают новые инструменты, потому что те, что есть, не имеют нужных им функций. Хороший пример этого — утилита `make`, изобретенная Стюартом Фельдманом (Stuart Feldman) в Bell Labs в 1976 году, которая используется для создания больших программных пакетов. Время шло, и понадобились новые функции. Некоторые из них были добавлены в `make`, но чаще люди без злого умысла создавали новые утилиты, несовместимые с `make` и выполняющие аналогичные функции. (Например, однажды я консультировал компанию, которая написала собственную утилиту только потому, что не удосужилась полностью прочитать документацию по `make` и не знала, что она умеет делать то, что им нужно.) В результате мы имеем `make`, `cmake`, `dmake`, `imake`, выберите-букву-`make` и другие программы, которые делают почти одно и то же несовместимыми способами. В результате специалистам вроде вас приходится изучать несколько инструментов в каждой категории. Это лишь усложняет всем жизнь и ничуть не добавляет ценности. Рисунок 15.1 хорошо иллюстрирует ситуацию.

Создание бремени для других не повышает ценность проекта. Опытные программисты знают, что переделка чего-то уже готового в соответствии с их личными предпочтениями редко приносит пользу. Это лишь демонстрирует их незрелость как программистов. Улучшайте существующие инструменты везде, где только можно, потому что это пригодится большему количеству людей. Сохраните создание новых инструментов для чего-то принципиально нового. Убедитесь,

что вы полностью разобрались в существующих инструментах, потому что они могут быть более функциональными, чем кажется на первый взгляд.



Рис. 15.1. Снижение ценности
(любезно предоставлено Рэнделем Манро (Randall Munroe), xkcd.com)

Порча экосистемы, в которую вы выпускаете код, не повышает ценность проекта. Многие разработчики ведут себя как отдыхающие в другой стране стереотипные американцы или как, если на то пошло, мой тест в гостях: «Я только что пришел, так что теперь все должно быть по-моему».

Например, в системах UNIX есть команда, которая отображает справочные страницы для программ. Введите `man foo` — и откроется справочная страница для команды `foo`. Существует также соглашение, согласно которому для действительно сложных команд, таких как `yacc`, есть как справочная страница, так и документ с подробным описанием программы. Когда проект GNU (о котором я вскоре расскажу) добавлял команды в UNIX, он использовал собственную систему для руководств — `texinfo`, несовместимую с `man`. В результате пользователям приходилось применять обе команды, `man` и `info`, чтобы найти документацию. Даже если, как некоторые считают, подход GNU был лучше, любые возможные преимущества были нивелированы огромной потерей эффективности UNIX-сообщества из-за раскола экосистемы.

Есть много других примеров — например, замена системы `init` на `systemd`. Большая часть философии UNIX, как будет показано далее в этой главе, модульная, но `systemd` заменила модульную систему `init` огромным монолитным монстром. Не было даже попыток внедрить новые функции в существующую систему. Продуктивность всей пользовательской базы снизилась, потому что людям пришлось осваивать новую систему, которая в основном делала то же, что и старая. Добавление многопоточности и других новых функций в существующую систему принесло бы больше пользы.

Еще один пример — утилита `jar`, которая является частью среды программирования Java. В 1970-х годах была создана утилита `tar` для упаковки нескольких файлов в один. Это решило проблему с магнитными лентами, использовавшимися для хранения информации. Магнитная лента представляет собой блочное устройство, а объединение файлов позволило использовать полные блоки, в результате чего повысилась эффективность. ZIP-файлы, которые впервые появились в Windows, действуют по тому же принципу. Однако вместо того чтобы использовать любой из этих существующих форматов, в Java создали свой собственный. В результате пользователям пришлось выучить еще одну команду без веской на то причины.

Так что не будьте похожи на «безобразного американца»¹. Работайте вместе с экосистемой, а не против нее. Руководствуйтесь правилом «наименьшего удивления». Вы повышаете ценность проекта, если ваша работа выглядит естественным продолжением существующей среды.

Как мы до этого дошли

Перед тем как перейти к практическим задачам, разберемся, как мы до этого дошли. Невозможно охватить абсолютно всю историю программирования в одной книге, поэтому коснемся лишь некоторых важных вех и последних разработок.

Краткая история

Давным-давно люди зарабатывали деньги, продавая компьютеры, которые стоили очень дорого. Специалисты писали и распространяли программы, чтобы компьютеры было легче продать. Существовала культура совместного использования и сотрудничества при улучшении программ. По мере того как компьютеры становились доступнее, все больше и больше людей смогли создавать и распространять программные продукты.

Операционная система *Multics*, запущенная на огромном суперкомпьютере GE645, была разработана в 1960-х годах компаниями Bell Telephone Laboratories, General Electric и Массачусетским технологическим институтом (MIT). Bell вышла из проекта, и некоторые из работавших над ним специалистов, в первую очередь Кен Томпсон и Деннис Ритчи, стали экспериментировать с идеями реализации файловой системы, пришедшими им на ум во время работы над *Multics*, используя произведенные корпорацией Digital Equipment (DEC) компьютеры меньшего размера. Результатом их работы стала инновационная операционная система под названием *UNIX*, которая воплощала в себе новую

¹ Отсылка к политическому роману Юджина Бердика (Eugene Burdick) и Уильяма Ледерера (William Lederer) «Безобразный американец» («The Ugly American»), в котором авторы критиковали работу американских дипломатов. — Примеч. ред.

минималистскую и модульную философию программного обеспечения. Она стала первой *переносимой*, или *порттируемой*, операционной системой, способной работать на нескольких типах компьютеров, хотя разработчики изначально не планировали делать ее таковой. Термин *UNIX* в этой книге относится ко всем подобным системам, включая Linux, FreeBSD, NetBSD, OpenBSD и современную macOS. Операционная система Microsoft Windows — это единственное крупное исключение, но даже она включает в себя все больше и больше функций UNIX, например модель сокетов для организации сети.

Компания Bell не единственная из участников Multics, кто пошел своим путем. Incompatible Timesharing System (ITS, несовместимая система разделения времени) была разработана в MIT. ITS сама по себе включает ряд новаторских функций, но ее наиболее узнаваемый продукт — это текстовый редактор Emacs (Editor MACroS, редактор макросов), который изначально создавался как набор макросов для текстового редактора DEC TECO (Text Editor and Corrector, текстовый редактор и корректор). Пользовательский интерфейс для ITS и Emacs повлиял на проект GNU, который также был запущен в Массачусетском технологическом институте.

В 1975 году Кен Томпсон, взяв академический отпуск для преподавания в Калифорнийском университете в Беркли, привез с собой копию UNIX. Это возымело огромный эффект, отголоски которого слышны до сих пор. Студенты получили доступ к настоящей рабочей системе, смогли изучать код, чтобы видеть, как все работает, и вносить изменения. Более того, они повлияли на саму философию операционной системы. В Беркли выпустили собственную версию UNIX под названием *BSD* (сокр. от Berkeley Software Distribution).

Благодаря студентам в систему было добавлено много новых важных функций. Сетевой стек BBN, лежащий в основе интернета, был встроен в UNIX в Беркли, где появился ныне повсеместно используемый интерфейс сокетов. Выпускники университетов стали использовать BSD-версию UNIX, что привело к появлению компаний вроде Sun Microsystems, которые занимались производством коммерческих систем на базе UNIX.

Все изменилось с приходом персональных компьютеров. Внезапно оказалось, что программисты, создающие программное обеспечение, и продавцы компьютеров — это не одни и те же люди, поэтому за соответствующую помочь им пришлось платить. Тогда все еще было мнение: «Здорово, что мы зарабатываем себе на жизнь, создавая эти классные штуки». Все пошло не так, когда Билл Гейтс (один из основателей Microsoft) вдруг заявил о себе. Как видно из многочисленных показаний в суде, он был сосредоточен на зарабатывании денег. Если бы ему нужно было сделать что-то крутое и тем самым заработать деньги, он бы это сделал, но его приоритеты расходились с приоритетами других специалистов в отрасли. К чему это привело?

Разработка программ стала больше зависеть от политики, юристов и иногда даже интриг, чем от превосходной инженерной мысли. Такой подход часто был

направлен на подавление новых разработок, которые конкурировали с существующими продуктами. Например, компания Microsoft начала с MS-DOS, программы, купленной у разработчика — американского программиста Тима Патерсона (Tim Paterson). Microsoft не развивала программу, поскольку именно в первозданном виде она приносила много денег. Компания Digital Research выпустила улучшенную версию программы под названием DR-DOS. Когда в Microsoft вышла ОС Windows, первая версия которой работала поверх DOS, они добавили скрытый зашифрованный фрагмент кода, который проверял, работает ли система под управлением MS-DOS или DR-DOS. Если обнаруживалась DR-DOS, код генерировал фальшивые ошибки. Это привело к краху DR-DOS на рынке, хотя, возможно, за свои деньги это был лучший продукт.

Однако дело было не только в Microsoft. Компания Apple также подала в суд на Digital Research за «копирование» их пользовательского интерфейса в продукте под названием GEM. Digital Research, вероятно, в итоге одержала бы победу, попутно обанкротившись, потому что у Apple кошельки явно были толще. Это довольно иронично, учитывая, что пользовательский интерфейс Apple был в большинстве своем скопирован с Xerox Alto.

К сожалению, такой подход сохраняется и сегодня. Крупные игроки обращаются в суды, если над ними нависает угроза конкуренции, вместо того чтобы пытаться улучшить собственные продукты. Примеров множество: SCO против IBM, Oracle против Google, Apple против Samsung, Samsung против Apple, офшорные компании Intellectual Ventures против всего мира и т. д.

Персональные компьютеры обрели популярность в середине 1980-х годов. Они не слишком подходили для запуска UNIX из-за отсутствия модуля управления памятью (см. главу 5), хотя существовал вариант под названием Xenix, который работал на оригинальных компьютерах IBM PC.

Колледжи начали использовать персональные компьютеры под управлением Microsoft Windows для обучения computer science, потому что эта ОС была дешевле. Однако, в отличие от выпускников UNIX-эры Калифорнийского университета в Беркли и других учебных заведений, у этих студентов не было возможности посмотреть исходный код используемой системы. Сама же система была значительно менее продвинутой, чем UNIX. В результате выпускники этой эпохи часто не настолько квалифицированы, как их более ранние коллеги.

Отчасти именно закрытость исходного кода подтолкнула Ричарда Столлмана (Richard Stallman) к запуску проекта GNU (Gnu's Not Unix) в 1983 году. Среди прочего, его целью было создание свободно доступной и юридически не обремененной версии UNIX. Сегодня мы называем это бесплатным программным обеспечением с открытым исходным кодом, или *FOSS* (free and open source software). *Открытый исходный код* означает, что он доступен всем для просмотра и, что более важно, изменения и улучшения. Столлман, поработав с адвокатом, создал *авторское лево*, или *копильтф* (copyleft), — вариант авторского права,

используемый специалистами для защиты программного обеспечения. По сути, копирайт гласит, что другие люди могут свободно использовать и изменять код, если их изменения доступны на тех же условиях. Другими словами, «мы поделимся с вами нашим кодом, если вы поделитесь своим со всеми остальными». Проект GNU внес большой вклад в воссоздание утилит UNIX, таких как `cp` и компилятора `gcc` для C, — работа над последним, возможно, была важнее всего. Но команда проекта не спешила создавать саму операционную систему.

Линус Торвальдс (Linus Torvalds) начал работу над продуктом, который сейчас известен как операционная система *Linux*, в 1991 году отчасти потому, что не существовало операционной системы GNU. В значительной степени его работа стала возможной благодаря существованию инструментов GNU (компилиатора C и др.) и зарождающемуся интернету, который стал площадкой для совместной работы. ОС Linux обрела необычайную популярность: она активно используется в центрах обработки данных (в том числе облачных), легла в основу программного обеспечения для Android-устройств и может применяться практически повсеместно. Даже эта книга была написана в системе Linux.

Крупные компании изначально скептически относились к использованию программ с открытым исходным кодом. Кому придется исправлять баги? Это довольно нелепый вопрос. Если вы когда-нибудь сообщали о баге в Microsoft, Apple или любую другую крупную компанию, вы знаете, сколько внимания вам уделят. В 1989 году Джон Гилмор, Д. В. Хенкель-Уоллес (он же Гамби) и Майкл Тименн (Michael Tiemann) запустили проект *Cygnus Support* для коммерческой поддержки ПО с открытым исходным кодом. Его существование значительно увеличило готовность компаний использовать такие программы.

Во многих отношениях Linux и GNU открыли для нас новую золотую эру, похожую на дни UNIX в Беркли. Однако она сияет *не так* ярко, как могла бы, потому что некоторые люди из эпохи ПК добавляют свои изменения, не понимая философии. В частности, некоторые программисты, не выросшие на UNIX, снижают ценность экосистемы, заменяя небольшие модульные компоненты огромными монолитными программами.

ПО с открытым исходным кодом

ПО с открытым исходным кодом обрело огромный успех, несмотря на кричащую пропаганду некоторых авторитетных компаний с закрытым кодом. К примеру, кто-то из управляющих Microsoft заявил: «Открытый исходный код разрушает интеллектуальную собственность. Я не могу представить что-то хуже этого для бизнеса в сфере ПО и интеллектуальной собственности». И это несмотря на то, что они тайно использовали инструменты с открытым исходным кодом внутри компаний. Основное преимущество программ с открытым исходным кодом заключается в неустанном соблюдении принципа тысячи глаз, благодаря чему можно говорить о повышенной безопасности и надежности кода. Прежде всего

они позволяют программистам брать за основу работу, проделанную другими, вместо того чтобы заново изобретать велосипед. Даже если вы работаете в компьютерной системе с закрытым исходным кодом, вполне возможно, что у некоторых ее компонентов он открыт. Даже Microsoft недавно, очевидно, достигла просветления и сделала многие инструменты UNIX доступными в своих системах.

Интернет и облачные сервисы очень помогли развитию ПО с открытым исходным кодом. Легко найти существующие проекты такого типа или запустить свой собственный. Но — и это большое «но» — большинство открытых проектов не обладают никакой ценностью, как и их аналоги с закрытым исходным кодом.

Многие программы с открытым исходным кодом выросли из студенческих проектов. Часто это первые проекты, авторы которых еще не овладели искусством написания хорошего кода. И большая часть этих программ остается незавершенной, поскольку студенты-программисты окончили курс и университет или просто забросили работу. Иногда проще переписать все заново, чем расшифровывать чужой плохо написанный и задокументированный код. Так начинается порочный круг, потому что переписать код часто не удается и в результате появляются несколько версий, которые не работают по разным причинам. Например, недавно мне нужно было извлечь теги из MP3-файлов, и я перепробовал шесть разных программ с открытым исходным кодом, каждая из которых по-своему не справилась с задачей. Часто из-за обилия мусора бывает сложно понять, есть ли у инструмента хорошая рабочая версия.

Ричард Столлман, запуская проект GNU, предполагал, что мир полон таких же классных программистов, как он сам и его коллеги. Это не подтвердилось. До сих пор существует убеждение, что одно из преимуществ ПО с открытым исходным кодом в том, что в него можно добавлять функции и исправлять в нем найденные ошибки. К сожалению, большая часть таких программ плохо написана и вообще не имеет документации, из-за чего обычному пользователю или даже опытному программисту придется приложить слишком много усилий.

Открытый исходный код сам по себе не делает проект лучше, но он позволяет узнать, что следует и чего точно не следует делать при создании собственного продукта.

Ниже я назову два признака, положительный и отрицательный, которые можно использовать для определения качества фрагмента кода.

Положительный признак в том, что проект находится в активной разработке и в нем задействованы как минимум два участника. Это не относится к проектам, которые существуют уже давно и фактически «закончены». Хорошо, если проект поддерживается какой-то организацией. Многие из крупных проектов с открытым исходным кодом возникли внутри компаний, которые до сих пор поддерживают их разработку. Однако стоит с осторожностью относиться к открытым проектам, созданным в компаниях, приобретенных организациями

с другой философией. Например, Sun Microsystems была выдающимся разработчиком ПО с открытым исходным кодом, включая OpenOffice, Java и VirtualBox. Однако затем Sun вошла в состав компании Oracle. Та прекратила поддержку некоторых из этих проектов и попыталась найти способы контроля и монетизации других — см. подробности в судебном процессе Oracle против Google. Проекты также могут передаваться компаниями в дар фондам, поддерживающим их развитие, что позволяет проектам держаться в нужном русле. Этот показатель не совсем надежен, так что не слишком ему доверяйте. Например, кодовая база веб-браузера Firefox представляет собой плохо документированный беспорядок.

Отрицательный признак — характер и количество обсуждений, которые можно встретить на разных сайтах «самопомощи» программистов. Если вы видите множество вопросов вроде «Я не понимаю, как это сделать» и «Как внести это изменение?», то это, вероятно, не очень хороший код. Кроме того, если ответы в основном бесполезны, язвительны или вообще отсутствуют, то проекту, вероятно, не хватает хороших разработчиков. Разработчики, которые обвиняют спрашивающего в том, что он плохой специалист, — не очень хороший пример для подражания. Конечно, если комментариев или вопросов нет вообще, это тоже плохой знак, ведь это означает, что код, скорее всего, не используется.

Если учесть все предостережения, открытый исходный код — отличная штука. Делайте код открытым, когда это имеет смысл. Но сначала научитесь выполнять работу хорошо, чтобы ваш код служил примером для других людей.

Creative Commons

Авторское право прекрасно сработало для программного обеспечения, но опираться на прошлое с выгодой для себя можно не только в области ПО. Во времена создания авторского права большинство компьютерных приложений существовали в текстовом формате, так как графика, изображения, аудио и видео были слишком дорогими для среднего потребителя. Сегодня звуки и визуальные эффекты, дополняющие программы, возможно, настолько же важны, насколько и сами программы.

Американский юрист и ученый Лоуренс Лессиг (Lawrence Lessig) осознал важность художественных произведений и создал для них набор лицензий, аналогичный авторскому праву, под названием *Creative Commons*. Существует множество вариантов этих лицензий — столько же, сколько лицензий с открытым исходным кодом для программного обеспечения. Они варьируются от «вы можете делать все, что хотите», «вы должны выразить признательность создателю» и «вы должны обнародовать все свои изменения» до «только для некоммерческого использования» и «запрещены производные работы».

Правовая база *Creative Commons* значительно расширила наши возможности использования чужих наработок в коде.

Расцвет переносимости

Термин *переносимость* имеет особое значение для программ. Переносимый код может работать в среде (и/или на оборудовании), отличной от той, для которой он был разработан. На заре вычислительной техники, когда было всего несколько поставщиков компьютеров, проблемы с переносимостью не существовало, хотя стандартные языки вроде COBOL и FORTRAN позволяли запускать программы на разных устройствах. Переносимость стала более важной в 1980-х годах, когда индустрия EDA (см. «Аппаратное и программное обеспечение» на с. 134) и доступность UNIX привели к появлению гораздо большего количества компьютерных компаний.

Новые поставщики компьютеров перенесли системы UNIX на выпускаемые ими устройства, чтобы избавить клиентов от беспокойства по этому поводу. Но примерно в то же время произошло еще одно изменение: системы UNIX, стоявшие дешевле конкурентов, проникли из академических кругов на коммерческий рынок. Не все системы имели открытый исходный код, поскольку конечные пользователи не стали бы самостоятельно создавать программы. Чтобы увеличить прибыль, некоторые компании начали взимать дополнительную плату за определенные инструменты UNIX, такие как компилятор С. Люди, которым они были нужны, стали обращаться к инструментам GNU, поскольку те были бесплатны и работали зачастую не хуже, а во многих случаях даже лучше, чем исходные инструменты UNIX.

Однако теперь пользователям приходилось самим переносить эти инструменты на другие системы, что быстро переросло в большую проблему. В разных системах файлы заголовков и библиотеки находились в разных местах, а многие библиотечные функции работали не совсем так, как ожидалось. Проблема решалась двумя способами. Во-первых, для обеспечения некоторой согласованности API и пользовательских сред были созданы такие стандарты, как POSIX (portable operating system interface, интерфейс переносимых операционных систем). Во-вторых, проект GNU создал набор *инструментов сборки* — `automake`, `autoconf` и `libtool` — для автоматизации некоторых системных проверок зависимостей. К сожалению, эти инструменты очень непонятны и сложны в использовании. Кроме того, они имеют собственные зависимости, поэтому код, созданный с помощью одной версии, часто нельзя собрать с помощью другой.

Таков мир сегодня. Современные системы больше похожи друг на друга, чем раньше, потому что в значительной степени они основаны на UNIX. И инструменты сборки GNU, несмотря на свою громоздкость, в большинстве случаев справляются с поставленными задачами.

Управление пакетами

Программное обеспечение с открытым исходным кодом, особенно Linux, сделало проблему распространения ПО еще более острой. Хотя люди относятся к Linux

как к единой системе, существует множество различных ее конфигураций для разных целей: от центров обработки данных и настольных компьютеров до основы системы Android для телефонов и планшетов. Даже если все системы имеют одинаковую конфигурацию, существует много разных версий каждой системы. Несмотря на доступность исходного кода, большая часть его теперь распространяется в предварительно скомпилированном и готовом к запуску виде.

Мы уже обсуждали общие библиотеки в разделе «Запуск программ» на с. 183. Предварительно скомпилированная программа не будет работать, если в систему не добавлены правильные версии нужных ей библиотек. Некоторые большие программы используют огромное количество библиотек, их все нужно включать в пакет с учетом правильно выбранных версий.

Управление пакетами стало по-настоящему популярным в Linux, хотя и до этого предпринимались разнообразные попытки внедрить его в проекты. Инструменты управления пакетами позволяют объединять программы в *пакеты*, содержащие список зависимостей. Инструменты управления пакетами, такие как `apt`, `yum` и `dnf`, не только загружают и устанавливают программы, но и проверяют целевую систему на наличие зависимостей, загружая и устанавливая их по мере необходимости.

Эти инструменты обычно работают хорошо, но, как правило, проблемы начинаются, если разным программам требуются разные версии одних и тех же зависимостей. Из-за несовместимости менеджеров пакетов подготовка программ для установки в разных системах занимает много времени и сил.

Контейнеры

Контейнеры — это недавно придуманный подход к решению проблем управления пакетами. Идея заключается в том, что приложение и все его зависимости объединены в контейнер, а все части контейнера, такие как файлы данных, запускаются в среде изолированно от остальной системы.

Контейнеры упрощают развертывание программ, поскольку они объединяют все зависимости (библиотеки и другие программы), необходимые приложению, в один пакет. Это означает, что, если ваш тип контейнера поддерживается, вы можете просто установить контейнерное приложение, не беспокоясь о нужных ему зависимостях. Недостаток этого подхода в том, что в результате мы избавляемся от разделяемых библиотек (см. «Запуск программ» на с. 183) и, как следствие, память используется менее эффективно. Сами контейнеры также занимают больше места, чем отдельные приложения.

Безопасность преподносится как преимущество контейнеров, поскольку запуск нескольких приложений в одной операционной системе приводит к тому, что приложения мешают друг другу, используя ошибки ОС. Может быть, это и так, но для решения проблемы необходимо просто использовать другой класс ошибок.

Во многих системах Linux доступны контейнерные приложения под названием *snap*-пакеты. Так называемая *CoreOS*, теперь *Container Linux*, — один из основных проектов Linux, связанных с контейнерами. Один из ее разработчиков приложил руку к заметкам, которые легли в основу этой книги, так что вы в хорошей компании.

Java

Работа по созданию языка программирования Java началась в 1991 году в компании Sun Microsystems под руководством Джеймса Гослинга (James Gosling). Гослинг был достаточно опытен, чтобы понять, что технологии изменились и придется использовать другой подход к разработке. В данном случае он осознал, что компьютеры работают настолько быстро, что вместо скомпилированного кода практичнее применять интерпретаторы. Язык Java во многом похож на С и С++.

Одна из идей Java заключалась в том, что вместо того, чтобы перекомпилировать код под каждый целевой компьютер, нужно обновить интерпретатор Java, и тогда однажды написанный код можно будет запустить где угодно. Эта концепция не нова, поскольку Java не был первым интерпретируемым языком.

Первоначально Java разрабатывался для телевизионных приставок (тогда он еще назывался Oak), а затем был перепрофилирован для запуска кода в браузерах. Цель заключалась в том, чтобы код не зависел от устройства, на котором работает браузер. В этой среде его несколько затмил язык JavaScript, хотя и Java все еще используется. JavaScript не связан с Java, и его нельзя назвать приятным языком, но создавать на нем код проще, так как он не требует применения специальных инструментов.

Java также важен, поскольку он часто используется для обучения программированию. Отчасти это потому, что в Java есть автоматическая сборка мусора, что освобождает новичков от сложностей с явным управлением памятью. С Java хорошо начинать, если только вы не собираетесь на нем останавливаться.

Java вырос в нечто большее, чем просто язык программирования, — его окружает целая экосистема ПО. Она включает в себя множество пользовательских инструментов и форматов файлов, что усложняет жизнь программистам. Экосистема настолько сложна и фрагментирована, что нередко можно услышать жалобы на то, что код пишется только один раз, но сложно каждый раз устанавливать и настраивать экосистему, чтобы запускать этот код.

Еще одним недостатком Java считается выросшая вокруг него культура программирования. Java-программисты склонны писать сотни строк кода там, где достаточно одной. Глядя на чужой Java-код, задаешься вопросом, где найти строку, которая хоть что-то делает. Отчасти это связано с тем, что Java — хороший объектно-ориентированный язык, а фанаты ООП одержимы красивой иерархией классов и часто ставят ее выше решения задачи.

Отличный пример — инструмент Java для работы с базами данных под названием *Hibernate*, который, насколько я могу судить, пытается решить две «проблемы». Во-первых, классы и подклассы Java отлично справляются с *сокрытием данных* или ограничением видимости внутренних переменных. Но, несмотря на скрытие данных, код в самом низу иерархии классов обращается к глобальной базе данных, что не дает некоторым покоя. *Hibernate* использует специальные комментарии в Java для управления базой данных, скрывая от программиста действительное положение дел. Все это хорошо до тех пор, пока что-то не сломается, — и тогда придется разбираться во всем самому.

Во-вторых, *Hibernate* — предоставление абстракции под названием *HQL* (*Hibernate Query Language*, язык запросов *Hibernate*) поверх базового API базы данных, в качестве которого обычно используется *SQL* (*Structured Query Language*, язык структурированных запросов). Теоретически это позволяет программистам работать с базами данных, не беспокоясь о различиях между их системами.

Еще до того, как язык программирования С был официально стандартизирован, между компиляторами существовал ряд несовместимостей. Вместо того чтобы изобретать «мета-С», люди придумали правила программирования, такие как «не используйте эту функцию». Если следовать этим рекомендациям, код будет работать на любом компиляторе.

Различия между реализациями *SQL* можно обработать похожим образом без введения еще одного механизма. Также стоит отметить, что большинство серьезных проектов *SQL* включают так называемые *хранимые процедуры*, не имеющие совместимости между реализациями. А *HQL* не обеспечивает их поддержку — его разработчики упустили место, где *HQL* мог быть действительно полезен.

В обмен на скрытие базовой системы баз данных приходится изучать новый язык, который даже не делает все необходимое.

Node.js

Как вы уже знаете из этой книги, *JavaScript* зародился как язык сценариев для браузеров. *Node.js* — это новейшая среда, которая позволяет запускать *JavaScript* вне браузера. Одно из основных преимуществ *Node.js* заключается в том, что с его помощью можно писать клиентскую и серверную части приложения на одном языке.

Идея хорошая, но результаты разные. Я избегаю *Node.js* по нескольким причинам. Во-первых, в *Node.js* включен собственный менеджер пакетов. То, что нужно, — еще один несовместимый метод, усложняющий обслуживание систем. К примеру, у *Perl* тоже есть собственный менеджер пакетов, но он не понижает ценность проекта, поскольку его пакеты доступны через системные менеджеры вроде *apt* и *dnf*.

Во-вторых, существуют сотни тысяч пакетов Node.js со сложными взаимозависимостями. Подавляющее большинство из них для серьезной работы не подходят. Node.js почему-то привлекает к себе плохой код.

Облачные вычисления

Облачные вычисления означают использование чужих компьютеров в сети. На самом деле эта концепция представляет собой обновленное изобретение 1960-х годов — разделение времени. Облако интересно по двум причинам.

- Сети распространились повсеместно, а их скорость резко увеличилась. В результате стали доступны потоковые аудио и видео, не говоря уже о снижении нагрузки на хранилища для таких вещей, как электронная почта.
- Цены на оборудование упали настолько, что за сравнительно небольшие деньги можно приобрести невероятное количество вычислительной мощности и хранилища. Это привело к появлению новых алгоритмов и ранее нецелесообразных способов решения задач. Конечно, то же самое можно сказать и о настольных компьютерах. Сейчас у моего компьютера восемь процессорных ядер, 64 Гбайт ОЗУ и 28 Тбайт на диске. Когда я начал программировать, это не было ни практично, ни экономично. Вместе с тем на устройстве, с которого я пишу эту книгу, больше оперативной памяти, чем было всего на диске компьютера, за которым я работал 20 лет назад.

В облачных вычислениях нет волшебства — это всего лишь аппаратная и программная части. Новый подход породил новые бизнес-модели для аренды вычислительных ресурсов.

Облачные вычисления легли в основу множества инноваций в области аппаратного обеспечения. Центры обработки данных (ЦОД) за счет своего масштаба работают совершенно иначе, чем обычные ПК, и для них очень важна надежность. Чтобы втиснуть огромное количество компьютеров в ограниченное пространство, необходимо тщательно продумать системы питания и охлаждения. Одна из творческих схем, предложенная Sun Microsystems, предполагает строительство ЦОД не в зданиях, а в транспортных контейнерах.

Виртуальные машины

Раньше считалось, что одна программа может запускаться одновременно только на одном компьютере. Операционные системы позволяли запускать несколько программ, используя разделение времени. Но не все нужные пользователям прикладные программы были доступны во всех ОС, особенно когда в моду вошли системы с закрытым исходным кодом. Многим приходилось использовать

несколько компьютеров с разными операционными системами или перезагружать компьютеры, запуская разные операционные системы.

Аппаратное обеспечение теперь достаточно быстрое, чтобы целые операционные системы можно было рассматривать как отдельные приложения. Это приводит к мысли, что разделение времени можно применить и между несколькими операционными системами. Имейте в виду, что для этого может понадобиться интерпретация набора инструкций, отличного от набора инструкций выбранного физического устройства. Кроме того, недостаточно просто иметь возможность запускать набор инструкций, поскольку необходима еще и ожидаемая аппаратная среда.

Поскольку операционные системы не обязательно работают прямо на аппаратном обеспечении физического устройства, они называются *виртуальными машинами*. Виртуальные машины предоставляют множество преимуществ, помимо того что они не блокируют базовую ОС. Они действительно полезны для разработки — особенно для разработки операционных систем, поскольку сбой разрабатываемой системы не приведет к сбою системы разработки.

Виртуальные машины лежат в основе облачных вычислений. Арендуйте место в облаке — и вы сможете запускать любой нужный набор операционных систем.

Операционная система, поддерживающая виртуальные машины, часто называется *гипервизором*.

Портативные устройства

Как и в случае с облачными вычислениями, совершенствование технологий связи и оптимизация соотношения цены и производительности оборудования привели к созданию высокомощных и функциональных портативных устройств. В одном современном сотовом телефоне больше вычислительной мощности и памяти, чем во всех компьютерах, существовавших несколько десятилетий назад. Помимо портативности, в этих устройствах нет ничего нового или волшебного, и все они имеют свою экосистему и инструменты.

При программировании портативных устройств большая сложность заключается в обеспечении управления питанием. Поскольку портативные устройства питаются от батареи, необходимо свести к минимуму тяжелые операции вроде обращений к памяти, так как они потребляют энергию и разряжают батарею.

Среда разработки

Программирование за деньги — это не то же самое, что работа над личными или учебными проектами. Работать программистом означает получать указания от других, самому давать указания и работать с людьми. Об этом мало говорят

во время учебы, если говорят вообще. Поэтому приходится учиться прямо на рабочем месте, набивая шишки.

Есть ли у вас опыт?

Итак, вы начинающий программист с небольшим опытом или вообще без него. Что такое опыт и как его приобрести?

Работодатели всегда ищут «опытных специалистов». Что это значит? Самое простое определение — у кандидата есть навыки, нужные работодателю. Но это определение не самое лучшее и не самое практическое. Например, в 1995 году мне позвонил рекрутер, который искал человека с пятилетним опытом программирования на Java. Пришлось объяснять, что даже у авторов Java на тот момент не было такого опыта, потому что язык Java только-только появился.

Одно из преимуществ программирования заключается в том, что вы учитесь делать то, чего никогда раньше не делали. Как заранее получить навыки, которые нельзя получить, не имея опыта? Существует ли хорошее определение опыта?

Прежде всего полагайтесь на основы. Если все, что вы знаете, — это как сделать веб-сайт, вы вряд ли внесете значительный вклад в проект создания хирургического робота. Но что еще более важно, опыт — это понимание, что вы можете сделать, а что нет. Как определить, что вы можете сделать, если вы еще этого не сделали? Ответ: нужно научиться оценивать, основываясь не просто на догадках, а на эвристике.

Учимся оценивать

Как член проектной группы, вы можете навредить проекту, не выполнив работу вовремя без предупреждения. Ключевое слово — *без предупреждения*. Не существует человека, который все всегда делает вовремя, но неожиданный срыв сроков вряд ли останется незамеченным другими членами команды.

Как научиться оценивать? Тренируйтесь. Начните с простого упражнения: прежде чем приступить к выполнению задачи, например домашнего задания, запишите, сколько времени это займет. Затем отслеживайте, сколько времени на самом деле заняла задача. Вскоре вы обнаружите, что стали оценивать лучше. Это хорошая практика, потому что в домашних заданиях, как и в программировании, вы всегда делаете что-то, чего не делали раньше.

Один из полезных методов управления, которым часто злоупотребляют, — отчет о состоянии. Периодически составляйте краткий список того, что было сделано с момента последнего отчета, какие проблемы возникли и каковы ваши планы на следующий отчетный период. Это всего лишь более формальный метод отслеживания домашних заданий. Если отчет о состоянии показывает, что планы

не выполнены, но проблем не возникло, это должно вас насторожить. Отчеты о состоянии позволяют корректировать оценки, сравнивая их с фактическими результатами.

Планируем проекты

Программные проекты, как правило, более сложны, чем домашние задания (возможно, за исключением домашних заданий с курсов программирования). Как дать оценку более сложному проекту?

Довольно простой метод — разбить проект на части. Разложите все части по трем корзинам подходящего размера — например, 1 час, 1 день и 1 неделя. Сложите полученные результаты. Вы, скорее всего, ошибетесь в большинстве предложений, но общая оценка окажется довольно близкой. Отчеты о состоянии здесь играют ключевую роль, потому что они показывают, что на одни задачи уходит больше времени, чем ожидалось, а на другие — меньше, что позволяет отслеживать первоначальную оценку.

Подобные подходы — важный компромисс, поскольку создание полного и точного графика работы над сложным проектом часто занимает больше времени, чем сама работа. И график все равно не будет учитывать такие ситуации, как сильный гололед.

С этим связано то, как на самом деле планируются программные проекты. Както я объяснял принципы оценивания, отвечая на вопрос аудитории на лекции ACM в 2004 году в Университете штата Орегон. Невозможно описать последовавшую за этим тишину и отвисшие челюсти студентов. Оказывается, во время учебы невозможно узнать все, что пригодится в работе. Итак, что происходит на самом деле. Например, вы прекрасно проявили себя в завершенном проекте. Ваш менеджер отведет вас в сторонку и скажет: «Привет, отличная работа! Мы подумываем выпустить кое-что новенькое. Скажи, сколько времени это займет и сколько будет стоить?» Вы почувствуете себя настолько польщенным, что на время откажетесь от личной жизни, чтобы тщательно все просчитать. Вы сделаете это, не зная (точнее, уже будете знать, потому что только что прочитали об этом), что до разговора с вами менеджер уже прикинул цифры, возможно, получив их от своего руководителя. Вы покажете свои выкладки менеджеру, и тот ответит: «Хм. Ну, если это так долго и так дорого, мы просто не будем этого делать». В вашей голове загорится лампочка, и вы спросите себя: «Нужна ли мне работа на следующей неделе?» Вы скажете: «Ну, это была оценка с запасом, можно подкорректировать ее тут и тут». И начинается самое интересное. Вы лжете менеджеру, который знает, что вы лжете. Он также знает, что ваши первоначальные цифры верны и что проект будет выполнен вовремя и в рамках бюджета, если их утвердят. Кроме того, менеджер понимает, что меньшие цифры, которые он вынудил вас назвать, приведут к задержке проекта и превышению бюджета. Но, к сожалению, именно так обычно и происходит.

В этот сценарий может быть трудно поверить, но вспомните, как популярны комиксы о *Дилберте*¹.

Как показывает этот пример, частая проблема планирования — это руководство, которое отказывается принимать графики и реальные затраты. Неинженеры часто считают, что графики — это то, о чем можно договориться. Менеджеры уверены, что инженеры слишком завышают оценки, и пытаются сократить расчетные сроки. Это почти всегда приводит к большим проблемам. Единственный законный способ сократить время — отказаться от разработки новых функций.

Принимаем решения

Большинство проектов можно выполнить самыми разными способами, выбрав языки программирования, структуры данных и многое другое. Инженеры славятся жаркими спорами о том, как «правильно» что-то делать. Иногда проекты не завершаются, а люди теряют работу именно потому, что не перестают спорить и не могут к ней приступить. Горячие дебаты часто неудобны руководству.

Ничем не примечательный менеджер научил меня некоторым приемам, очень полезным в решении подобных проблем. В начале проекта он приглашал нас всех в конференц-зал и рассказывал, как он работает. Он сказал, что решения будут приниматься в первую очередь из практических соображений. Но, по его словам, часто не существует реальных причин, чтобы делать что-то именно так, а не иначе. Он сказал, что в таких случаях можно спокойно сказать: «Я хочу сделать это так, потому что мне это нравится». Он объяснил, что согласится с выбором, если никто не предложит другой путь. Он не хотел выслушивать сложные псевдотехнические аргументы, которыми кто-то просто оправдывает свои предпочтения, прямо не говоря об этом. Тот, кто будет так делать, не только не добьется своего, но и, вероятно, потеряет работу. Мораль этой истории в том, чтобы отделять практические соображения от личных предпочтений.

Вы уже познакомились с таким поведением в главе 12, где узнали, что реальное логическое обоснование и преимущества промисов в JavaScript затуманены страхом рационализации пирамиды судьбы.

Работаем с разными людьми

Выше я сказал, что программирование обычно предполагает работу с другими людьми.

Многочисленные адепты обучения программированию утверждают, что «программирование — это удовольствие». Я с этим не согласен. Я придерживаюсь

¹ «Дилберт» («Dilbert») — серия комиксов Скотта Адамса об офисной жизни, названная по имени главного героя — инженера в технологической компании. — *Примеч. ред.*

мнения итальянского исследователя Вальтера Ваннини (Walter Vannini), которое он выразил в статье «Coding Is Not “Fun”. It’s Technically and Ethically Complex»¹. Вспомните двухэтапный процесс программирования, описанный во введении к книге. Второй шаг, объяснение трехлетнему ребенку (то есть собственно программирование), требует тщательного внимания к деталям. Вам, скорее всего, трудно поддерживать свою комнату в чистоте. И это не доставляет вам удовольствия. Я бы сказал, что программирование приносит удовлетворение. Самое интересное — это первый шаг, понимание проблемы.

Люди любой профессии обладают совершенно разными качествами, и не всех из них можно назвать «хорошо приспособленными». Программисты не исключение. Хотя многие программисты имеют уравновешенный характер, кто-то предпочитает технические навыки социальным. Ричард Столлман и Деннис Ритчи находятся на разных концах спектра, а Линус Торвальдс — где-то посередине. Это может привести к проблемам, особенно в наше время, когда люди очень чувствительны к словам.

В последнее время в СМИ много говорят об оскорбительном поведении на работе. Позвольте прояснить: такое поведение неприемлемо никогда, поэтому не будьте обидчиком и не позволяйте обижать себя. Но часто бывает трудно определить, что является оскорблением, а что нет: у людей разные взгляды на мир, и что-то, что хорошо для одного человека, может не подойти другому. Классический пример — основатель Apple Стив Джобс.

Вы можете подумать, что это решается несколькими простыми правилами. Так и есть, но придется учитывать компромиссы. Много лет назад я работал с руководителем, который очень хорошо это выразил. Он сказал, что может заставить людей в команде вести себя «хорошо» — например, меньше спорить, — но в результате потеряется большая часть творческой маниакальной энергии, ради которой он нанял этих людей. Он чувствовал, что значительная часть его работы заключается в том, чтобы сгладить личностные различия, тем самым повысив продуктивность.

Большой источник проблем заключается в том, что программисты, увлеченные своей работой, могут резко критиковать чужую. Трудный урок, который нужно усвоить, заключается в том, что это не переход на личности. Однажды у меня был сотрудник, который, как я в конце концов узнал, воспринимал любое указание на ошибку в его коде как личное оскорбление. Но, если тот же сотрудник с удовольствием указывал на ошибки в моем коде, я реагировал так: «Давайте исправим ошибку, потому что нам нужно, чтобы код работал». По сути, это вопрос самооценки. Постарайтесь вселить уверенность в членов команды, и тогда они с меньшей вероятностью примут все на свой счет.

¹ «Программирование — это не удовольствие. Это сложно как с технической, так и с этической точки зрения». — Примеч. ред.

Кстати, когда-то я работал на человека, который регулярно говорил мне, что я занимаюсь глупостями. В конце концов я сообразил, что происходит, и сказал: «Знаете, я наконец-то понял: когда вы говорите мне, что я делаю что-то глупое, вы имеете в виду, что не понимаете, что я делаю. Зная это, я буду стараться вас игнорировать. Но я обычный человек, поэтому каждый раз, когда вы говорите «глупо», следующие пару дней у меня все валится из рук. Так что, если хотите оправдать потраченные деньги, просто скажите, что ничего не понимаете».

Общение важно. Неуверенные в себе люди обычно пытаются заставить других чувствовать себя хуже. Они либо стремятся говорить о чем-то, что выше их уровня понимания, либо проявляют снисходительность. Задача уверенного человека — выяснить, как говорить с другими на их уровне. Например, я был на вечеринке на конференции SIGGRAPH (Special Interest Group on Computer Graphics, специальная группа по графическим и интерактивным методам) в 1989 году и услышал, как кто-то попросил другого человека объяснить что-то из статьи, написанной Лореном Карпентером (Loren Carpenter), первым гиком, получившим премию «Оскар». Тот терпеливо объяснил. После этого первый человек сказал: «Спасибо, это действительно помогло. Меня зовут Джо, а вас?» — на что другой ответил: «Я Лорен». Будьте как Лорен.

Еще одна вещь, о которой следует помнить, если вы окажетесь в трудной ситуации на работе: не ищите друзей в отделе кадров. Их работа не в том, чтобы защищать вас, а в том, чтобы защищать компанию от ответственности.

Создаем культуру поведения на работе

Каждое место работы обладает уникальной культурой. Ключ к успешной и приятной карьере — найти то, что подходит вам. Культура, основанная на результатах, и культура, основанная на личности, находятся на противоположных концах спектра.

Эми Вжесневски (Amy Wrzesniewski), Кларк Макколи (Clark McCauley), Пол Розин (Paul Rozin) и Барри Шварц (Barry Schwartz) в статье 1997 года «Jobs, Careers, and Callings: People's Relations to Their Work»¹ делят работу на три категории, вынесенные в название статьи: работа, карьера и призвание. Вкратце, люди получают финансовое вознаграждение благодаря работе, продвижение в карьере и удовольствие от призыва. Соответствие вашей категории и индивидуальности вашей работе — ключевое слагаемое успеха.

Работа и карьера лучше вписываются в культуры, основанные на личности. Эти культуры вознаграждают свободные от драматизма взаимодействия. Люди хорошо относятся друг к другу, по крайней мере при общении лицом к лицу.

¹ «Работа, карьера и призвание: наше отношение к труду». — Примеч. ред.

Призвание идет рука об руку с культурой, основанной на результатах. Награда — это успешное выполнение работы, даже если оно сопровождается горячими спорами и оживленными дискуссиями.

Например, мы с научным редактором этой книги целый месяц увлеченно спорили по поводу абзаца в главе 7. Мы оба были счастливы, что нашли отличное решение, и это счастье компенсировало все споры. Нас обоих раздражало, что решение так долго не удавалось найти, но так оно и есть: зачастую решения неочевидны. Если такой процесс и результат делают вас счастливыми, вы с удовольствием выйдете на работу, где такое поведение ценится.

Если решение долго не удается найти, стоит сделать шаг назад и переосмыслить проблему. Однако трудно вспомнить об этом в разгар ожесточенной дискуссии.

Делаем осознанный выбор

Вы, наверное, заметили, что мне нечего сказать о некоторых технологических сферах, таких как веб-разработка. Вероятно, вы задумаетесь, почему вы хотите работать в этой области. Многое зависит от того, что вы хотите получить от работы, как сказано выше. Имейте в виду, что у всех начинаний есть свои плюсы и минусы. Трезво оценивайте рабочие ситуации.

Часто приходится идти на компромисс между интересной работой и большим заработком. Люди с призванием предпочитают интересную работу и могут даже выполнять ее бесплатно при необходимости. Людям, сделавшим карьеру, часто хорошо платят за работу с громоздкой или проблемной технологией. Хороший пример — многие вспомнили, как программировать на COBOL, чтобы найти и исправить ошибки Y2K. Речь идет об ошибках в устаревшем коде, работавшем с датами, — он сохранял только две последние цифры года. Переход с 1999 на 2000 год сломал бы такой код, использовавшийся во многих важных инфраструктурах.

Методологии разработки

Кажется, что любая область деятельности рождает «экспертов по методологии». Программирование ничем не отличается, за исключением, возможно, того, что для нас больше подходит термин *идеология*, чем *методология*. И каждая методология, похоже, имеет свою униформу, прическу, терминологию и набор секретных сигналов. В основном все это просто помогает сторонникам какой-то идеи вычислять «неверных», в отличие от примера выше с Лореном Карпентером. И это может привести к казусам: я обсуждал методологию с клиентом, который в конце концов выпалил: «Все должно быть в порядке, пока мы используем Agile на основе Scrum».

Мой экспертный совет — не воспринимать идеологию слишком серьезно. Ни одна из них не работает в чистом виде. Отберите идеи и используйте те из них, которые имеют смысл для вашего проекта. Но как определить, что сработает? Рассмотрим различные этапы разработки (рис. 15.2).

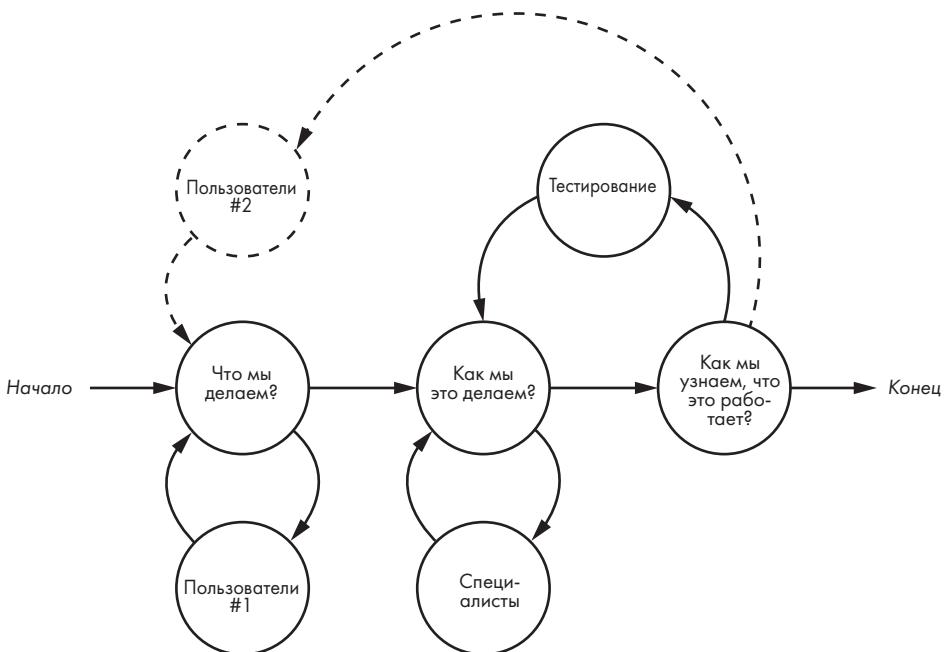


Рис. 15.2. Цикл разработки проекта

У нас есть три вопроса, с которых мы начали эту главу. Большое различие между идеологиями заключается в роли пользователя.

Вопреки распространенному мнению, программное обеспечение пишется не только для развлечений. Идеология, подходящая для веб-сайта или видеоигры, скорее всего, не сработает для спутника, электростанции, кардиостимулятора или автомобиля.

В проектах с высокой ценой ошибки важно точно знать, что делать, поэтому пользователи (#1) вовлекаются в процесс на ранней стадии, чтобы четко сформулировать задание. После этого можно перейти к написанию кода, который обычно — и желательно — проверяется коллегами на соответствие заданию.

Если цена ошибки низка, заранее продумывать четкое задание необязательно. Обычно в этих случаях занимают позицию «Узнаем, когда увидим». Пользователи (#2) играют более важную роль в проверке результатов и принятии решения о том, достигнут ли результат. Тестирование для проверки работоспособности

кода часто путают с тестированием, определяющим, нравится ли пользователям текущее задание.

Лень и некомпетентность — плохие методологии развития. Многие люди не пишут спецификации, потому что не знают, как это сделать. Выберите методологию, которая подходит в первую очередь для проекта, а во вторую — для людей.

Проектирование

Проект начинается с идеи. Идея может быть или лично ваша, или чья-то еще. Как превратить идею в код?

Можно, конечно, просто начать писать код. Для небольших личных проектов это может сработать. Но что-то значительное требует наличия процессов, которым стоит следовать, чтобы добиться лучших результатов.

Ведение записей

Сначала запишите идею. Вы удивитесь, но так вы выясните множество недостающих деталей.

Важно составлять документацию правильно. Говорите о том, что вы хотите сделать, а не о том, как это сделать.

Пример того, как делать не стоит: меня однажды попросили помочь с проектом разработки нового прибора для измерения артериального давления. Клиент прислал мне около 5000 страниц документации и попросил оценить стоимость, чего я сделать не смог. Оказывается, в попытке решить проблемы руководство компании постановило, что нельзя писать код без документации. Звучит неплохо, но они игнорировали тот факт, что никто из сотрудников не умел составлять документацию и никто их этому не обучил. Поэтому инженеры сначала писали код, не сообщая об этом руководству, а затем описывали его от руки на английском языке. Нигде даже не упоминалось, что они создают тонометр.

Другой пример — веб-сервер Apache. Хорошая программка. Тонны документации с описанием, как задать тот или иной параметр конфигурации. Но ни слова о том, что это веб-сервер, ни описания связей между его частями.

Быстрое прототипирование

Быстрое прототипирование — одна из методологий разработки, заслуживающих упоминания. Оно включает в себя создание частично рабочей версии проекта. Как и записывание, прототипирование помогает глубже разобраться в идее. Прототип также помогает объяснить идею другим.

Остерегайтесь следующих ловушек:

- Не принимайте прототип за рабочий код. Выбросьте его и напишите новый код, используя то, что вы узнали при создании прототипа.
- Не старайтесь составить жесткий график для разработки прототипа. В конце концов, вы создаете прототип главным образом потому, что у вас недостаточно знаний, чтобы составить реалистичный график.
- Самое сложное: не позволяйте руководству принять прототип за готовый к поставке продукт.

Одна из отличительных черт опубликованного кода прототипа — отсутствие согласованности. В книге «The Stuff of Thought»¹ Стивен Пинкер (Steven Pinker) обсуждает разницу между работой с блоками и работой с принципами, управляющими поведением этих блоков. В основном во время прототипирования вы работаете с блоками. Важно сделать шаг назад, после того как прототип станет функциональным, чтобы соблюсти эти руководящие принципы, а затем написать код повторно, чтобы последовательно эти принципы использовать.

Разработка интерфейса

Ваш проект займет какое-то место в *стеке программного обеспечения*, изображенном на рис. 1 на с. 33. ПО — это начинка в бутерброде, которая взаимодействует с продуктами выше и ниже себя. Приложение использует интерфейсы — их можно представить как нижний кусок хлеба. Ваша задача — определить верхнюю часть.

Системное программирование находится между оборудованием и приложениями. Системные программы взаимодействуют с оборудованием, используя любую комбинацию регистров и битов, подробно описанную в спецификациях производителя устройства. Но, помимо этого, системные программы должны взаимодействовать с приложениями. Граница между ними называется *программным интерфейсом приложения* (application program interface, API). API называется *пользовательским интерфейсом* (user interface, UI), если его используют люди, а не другие программы. Существует множество API, поскольку программы создаются послойно: внизу может располагаться операционная система с API, который используется библиотеками, которые, в свою очередь, используются в приложениях. Как устроен API и когда его можно считать хорошим?

Для начала неплохо задокументировать *варианты использования* — ситуации, в которых API используется для выполнения какой-либо задачи или набора задач. Можно собирать варианты использования, опрашивая конечных пользователей программы. Но пользователи часто дают недальновидные ответы,

¹ Пинкер С. «Субстанция мышления. Язык как окно в человеческую природу».

потому что они уже что-то используют. Многие их отзывы, как правило, звучат как «сделайте так, изменив вот это». Можно не получить четкого результата из-за массы дискретных требований.

Просто делая, что просят пользователи, вы какое-то время продержитесь на плаву. Но чтобы API имел продолжение, стоит абстрагироваться от пользовательских требований и найти элегантное решение. Рассмотрим несколько примеров.

Первоначальный API Apple Macintosh был опубликован в 1985 году в виде трехтомника под названием *Inside Macintosh* (Addison-Wesley). В томах было более 1200 страниц, и сейчас они полностью устарели — современные (на базе UNIX) Mac ничего из этого не используют. Почему этот проект API не прижился?

API Mac можно назвать очень широким и неглубоким. Он имел огромное количество функций, каждая из которых делала что-то одно. Можно возразить, что этот интерфейс недолговечен, потому что он слишком специфичен. Из-за отсутствия абстракций или обобщений расширение по мере появления новых вариантов использования было невозможно осуществить. Конечно, можно было добавить больше функций, сделать API еще шире, но это было бы не очень практично.

Напротив, версия 6 операционной системы UNIX была выпущена на 10 лет раньше, в 1975 году, с руководством в 321 страницу. Она воплощала в себе совершенно другой подход — узкий и глубокий API. Узость и глубина стали возможны благодаря хорошему набору абстракций. Абстракция — это обобщающая категория. Например, вместо того чтобы говорить по отдельности о кошках, собаках, лошадях, коровах и т. д., можно использовать абстракцию «животные». Эти абстракции были очевидны не только в системных вызовах (см. «Пространство системы и пользователя» на с. 178), но и в приложениях.

Например, вы, вероятно, знакомы с концепцией файла как места для хранения данных. Многие операционные системы имели разные системные вызовы для каждого типа файла. В UNIX использовался файл одного типа с несколькими системными вызовами. Например, системный вызов `creat` может создать файл любого типа. (Когда Кена Томпсона спросили, сделал бы он что-нибудь иначе, если бы можно было спроектировать систему UNIX заново, он ответил: «Я бы написал `creat` с буквой `e`¹».) В рамках файловой абстракции в качестве файлов рассматривались даже устройства ввода/вывода, как вы видели в главе 10.

Сравните эту абстракцию с утилитой `rip` (Peripheral Interchange Program, программа периферийного обмена) в существовавших в то же время системах DEC. Это был чрезвычайно сложный и неуклюжий инструмент со специальными командами, которые позволяли пользователям копировать файлы. Существовали отдельные команды для копирования файлов на ленты, принтеры

¹ Create — «создавать». — Примеч. ред.

и многое другое. Напротив, в UNIX использовалась единственная команда `cp` (сору, копировать), которую можно было запускать для копирования файлов независимо от их типа и месторасположения. Копировать файл в порт ввода/вывода, подключенный к принтеру, оказалось так же легко, как копировать файл из одного места в другое.

Абстракции UNIX поддерживали новую философию программирования:

- Каждая программа должна делать что-то одно и хорошо. Создайте еще одну программу, выполняющую что-то новое, вместо того чтобы усложнять старые.
- Создавайте программы для совместной работы. Выходные данные программ должны использоваться в качестве входных данных для других программ. Делайте сложные вещи, соединяя простые программы, вместо того чтобы писать огромные монолиты.

И API UNIX, и большое количество исходных приложений по-прежнему широко используются сегодня, более 40 лет спустя, что указывает на качественное проектирование. Мало того, многие библиотеки все еще используются практически без изменений, хотя их функциональность была скопирована в другие системы. А книгу «The UNIX Programming Environment»¹ Брайана Кернигана и Роба Пайка до сих пор стоит прочитать, хотя ей уже несколько десятков лет.

Более тонкое преимущество этого модульного подхода заключается в том, что новые программы не только имеют внутреннюю ценность, но и повышают ценность всей экосистемы.

Немного отвлечемся от темы. Я упоминал, что пользовательский интерфейс – это API для пользователей, а не для других программ. В книге 2004 года «The Art of Unix Usability»² Эрик Реймонд (Eric Raymond) приводит интересный пример общей системы печати Unix (Common Unix Printing System, CUPS), из которого вы узнаете, как не следует проектировать пользовательские интерфейсы.

Создать отличный интерфейс сложно. Вот о чем следует помнить:

- API не должен раскрывать внутреннюю реализацию и не должен зависеть от конкретной реализации.
- API должны демонстрировать *концептуальную тяжесть* (то же самое, что иметь хорошие абстракции).
- API можно *расширять* или адаптировать к будущим потребностям. Опять же, вспомните хорошие абстракции.

¹ Керниган Б., Пайк Р. «Unix. Программное окружение».

² Реймонд Э. «Искусство программирования для Unix».

- API должны быть *минимальными*, то есть они не должны быть перегружены множеством способов выполнения одной и той же задачи.
- Хорошо, если вы применяете *модульность*: если API предоставляет связанные наборы функций, сделайте их как можно более независимыми. Это также упрощает разбиение проекта на части, чтобы несколько человек могли работать над ним одновременно.
- Функциональность должна быть *компонуемой*, чтобы мы могли легко комбинировать части полезным образом. (Не путайте с *компостируемой*. В мире и так слишком много гниющих, плохо спроектированных интерфейсов.) Например, если у вас есть интерфейс, который возвращает отсортированные результаты поиска, имеет смысл разделить поиск и сортировку, чтобы использовать их как по отдельности, так и совместно.

Если вы не проспали, то могли заметить, что я поклонник философии UNIX. Это потому, что она работает, а не потому, что она модная. А еще она соответствует всем вышеуказанным пунктам.

Как мы уже обсуждали, одна из функций UNIX, которая теперь также доступна во многих других системах, — это файловая абстракция. Большинство операций над файлами выполняются без использования имени файла. Вместо него имя файла преобразуется в дескриптор (*файловый дескриптор*). Эта абстракция позволяет пользователям выполнять файловые операции, например подключение к чему-либо по сети, над сущностями, которые технически не являются файлами.

Как мы видели в главе 10, когда программа запускается в UNIX, ей передается пара файловых дескрипторов, называемых *стандартным вводом* и *стандартным выводом*. Программу можно представить как водяной фильтр в трубе: нефильтрованная вода поступает в стандартный ввод, а фильтрованная льется из стандартного вывода. Одна из замечательных особенностей UNIX заключается в том, что стандартный вывод одной программы можно подключить к стандартному вводу другой, получив *конвейер*, или *пайpline* (pipeline). Например, имея программу фильтрации воды и программу нагревателя, можно соединить их, чтобы получать нагретую фильтрованную воду без необходимости писать для этого специальную программу. UNIX — это своеобразный ящик, наполненный случайными инструментами и частями, из которых можно строить объекты.

Забавным образом эта философия была проиллюстрирована в 1986 году, когда Дон Кнут (Don Knuth) (почетный профессор компьютерных наук Стэнфордского университета и автор серии книг «The Art of Computer Programming»¹, экземпляр которой должен быть и у вас) написал для журнала *Communications of the ACM* статью, в которую добавил более 10 страниц кода для умного решения конкретной проблемы. За этим последовал критический анализ от Дуга Макилроя (Doug

¹ Кнут Д. «Искусство программирования».

McIlroy) (начальника Кена Томпсона и Денниса Ритчи в Bell Laboratories), показывающий, как все решение можно записать одной строкой из шести конвейерных команд UNIX. Мораль этой истории в том, что хорошие и связываемые общие инструменты лучше, чем одноразовые специальные решения.

Одна из особенностей конвейерной обработки заключалась в том, что программы в основном работали с текстом и, следовательно, имели общий формат. Они не полагались на большую структуру данных, кроме строки текста или полей, разделенных символом. Кое-кто утверждает, что это работало только потому, что, когда «времена были проще», текстовый формат был широко распространен. Но опять же, API имеют продолжение. Пакеты программ наподобие ImageMagick поставляют сложные конвейеры для обработки изображений. Также существуют программы для обработки данных с более сложной структурой, например XML и JSON.

Использовать сторонний код или писать собственный?

Хотя определение интерфейса как «витрины проекта» имеет решающее значение для проекта, трудные решения придется принимать, выбирая и то, что находится в подсобке. На какой код (помимо собственного) опираться?

Ваша программа, скорее всего, использует библиотеки (см. «Запуск программ» на с. 183), составленные другими людьми. Библиотеки хранят функции, которые можно использовать, вместо того чтобы писать собственные. Как узнать, когда применить библиотечную функцию, а когда стоит написать что-то самостоятельно?

Это проблема того же уровня, что и поиск хорошего ПО с открытым исходным кодом, как мы обсуждали в разделе «ПО с открытым исходным кодом» на с. 497. Если у библиотеки нет стабильного API, то вполне вероятно, что будущие выпуски приведут к проблемам с кодом. Умножьте все это на количество библиотек, и станет ясно, что все ваше время будет уходить на исправление ошибок, а не на написание собственного кода. Слишком большое количество библиотек может сделать код ненадежным. Например, недавно в Node.js сломался пакет, от которого зависели многие другие пакеты, что в результате затронуло множество программ.

Иногда приходится использовать библиотеки, потому что они реализуют что-то, что требует действительно узких знаний, которых у вас нет. Хороший тому пример — криптографические библиотеки OpenSSL.

Некоторые утверждают, что лучше использовать библиотеки, чем писать код самому, потому что популярные библиотеки уже отлажены. К сожалению, это не всегда так — вспомните ту же OpenSSL.

В обычной ситуации я бы сказал, что не стоит использовать библиотеку, если количество строк кода для включения библиотеки больше, чем количество

строк кода самостоятельно написанной функции — как в случае с `glibc` для реализации односвязных списков. Однако также необходимо учесть среду, в которой используется библиотека. `glibc` встречается в огромном количестве программ и, скорее всего, находится в памяти в виде разделяемой библиотеки, что позволяет библиотеке эффективно получать код, не занимая места в памяти.

Часто полезные библиотеки найти очень сложно. В недавней статье упоминалось, что существует более 350 000 пакетов Node.js. Вероятно, быстрее написать собственный код, чем отыскать нужную иголку в таком гигантском стоге сена.

Разработка

Добравшись до этого места, вы, скорее всего, можете создать спецификацию проекта и график реализации. Как это сделать?

Попробуйте программировать в Linux или другой производной UNIX. Их можно использовать по-разному. Если вы работаете с Mac, у вас все готово, потому что вам уже доступен вариант UNIX. Можно установить Linux на свой компьютер. Если это нецелесообразно, запустите *живой образ*, что означает запуск с DVD, ничего не меняя на жестком диске ПК. Лучший вариант — запустить Linux на *виртуальной машине*, которая представляет собой программу, позволяющую запускать другую ОС в отдельном окне на компьютере. Например, можно установить *VirtualBox* на компьютер с Windows, а затем запустить там Linux.

Серьезный разговор

Что ж, пришло время поговорить. Может быть, ваши родители слишком стеснялись, может быть, они надеялись, что вы услышите об этом в школе. Или, может быть, они считают, что вы найдете нужную информацию в интернете. Все это несерьезно. Если вы собираетесь всерьез заняться программированием, с компьютерами нужно иметь взрослые отношения. Отложите мышь в сторону и научитесь пользоваться текстовым редактором.

Взрослые отношения с компьютерами

До сих пор ваши отношения с компьютерами были довольно незрелыми. Вы указывали, щелкали, тыкали или как-то еще дергали компьютер и наблюдали, как он хихикает в ответ. Для программирования это не годится.

Программирование предполагает довольно тесное взаимодействие с компьютером. Вы будете делать гораздо больше, чем просто набирать текст или смотреть видео. Настолько больше, что вам придется повысить свою продуктивность. Это значит, что пришло время научиться пользоваться полезными инструментами.

Многие из них загадочны, и их не так просто освоить, а жаль. Научившись ими пользоваться, вы никогда уже от них не откажетесь, потому что сможете делать гораздо больше с гораздо меньшими усилиями. Итак, стисните зубы и приступайте к работе заранее — это оккупится!

Терминалы и оболочки

Помните все, что мы узнали о терминалах в главе 6? И угадайте что? Настоящие программисты до сих пор их используют. Терминалы больше не шумят и не мигают зеленым светом. Они даже не находятся на отдельном устройстве — теперь это часть программ, работающих на компьютере.

Все десктопные компьютерные системы имеют терминалы, даже если их трудно найти. По умолчанию терминалы запускают *интерпретаторы команд*. Вы увидите командную строку. Логично, что в командной строке вводят команды. Системы, основанные на UNIX, такие как продукты Apple, Linux и FreeBSD, имеют интерпретатор команд, или *оболочку*, под названием *bash*. Конечно, в Windows есть собственные инструменты, но и в ней можно установить bash.

ОБОЛОЧКА BASH

Одна из оригинальных оболочек UNIX под названием *sh* была написана Стивеном Борном (Stephen Bourne). С годами были созданы другие оболочки, имевшие больше возможностей. К сожалению, эти новые функции были совершенно несовместимы с *sh*. В конце концов, была написана новая версия *sh*, куда была добавлена совместимость с этими дополнительными функциями. Эта версия была названа *bash* — сокращение от Bourne-again shell (снова оболочка Борна). Благодаря сохранению идентичности Борна *bash* получила неоспоримое превосходство среди оболочек.

Многие команды имеют абсолютно непонятные названия, например *grep* (global regular expression printer, глобальный принтер регулярных выражений). Из-за этого прослеживается сходство с анатомией, где многие части тела названы в честь чего-то, на что они похожи, или в честь человека, который впервые их обнаружил. Например, команда *awk* была названа в честь ее авторов: Альфреда Ахо, Питера Вайнбергера (Peter Weinberger) и Брайана Кернигана. Программистов, обсуждающих эти команды, бывает трудно отличить от пещерных людей, издающих нечленораздельные звуки.

Изучить эти загадочные команды стоит ради автоматизации. Мощное преимущество оболочек заключается в том, что можно поместить команды в файл и создать программу, которая их запускает. Если вы часто повторяете одни и те же действия, вы можете просто задать в команде их автоматическое выполнение — это

намного продуктивнее, чем сидеть в затейливом графическом редакторе, нажимая кнопки и ожидая результатов.

Текстовые редакторы

Текстовые редакторы — это программы, позволяющие создавать и изменять стандартные данные ASCII, из которых создаются программы (я совершенно не имею права комментировать языки программирования, в которых используются не-ASCII-символы, такие как китайский язык). Основное преимущество текстовых редакторов заключается в том, что они работают с командами, это намного эффективнее, чем вырезание и вставка элементов мышью — по крайней мере, после того как эти команды изучить.

Есть два популярных текстовых редактора: *vi* и *Emacs*. Научитесь использовать один (или оба). У каждого из них свои фанаты (рис. 15.3).



Рис. 15.3. *vi* и *Emacs*

Eclipse и Visual Studio — это примеры изощренных инструментов программирования, известных как *интегрированные среды разработки*, или *IDE* (integrated development environment). (Проверьте дату их выпуска и остерегайтесь мартовских IDE¹.) Хотя IDE отлично подходят для распутывания чужого, плохо написанного кода, если они вам понадобились, то вы уже сошли с пути. Перечитайте введение книги и изучите основы, прежде чем заблудиться в затейливых инструментах. Кроме того, вы обнаружите, что все эти инструменты довольно медленные, и намного эффективнее делать то же самое при помощи простых, но мощных альтернатив. Например, пока инструмент запускается, вы успеете отредактировать программу в текстовом редакторе и пересобрать ее.

¹ «Beware the IDEs of March» — отсылка к фразе «Берегись мартовских ид!» («Beware the ides of March») из пьесы Шекспира «Юлий Цезарь». Этой фразой предсказатель предупредил Цезаря о смертельной опасности. — Примеч. ред.

Переносимый код

Хотя вы, возможно, не собираетесь использовать часть программы в другом месте, удивительно, как часто это происходит. И если ваш проект с открытым исходным кодом, другие люди могут использовать его (или его части) где-нибудь еще. Как написать код, чтобы его не было слишком сложно переносить? Если вкратце: по возможности избегайте жестких привязок.

Как вы уже узнали из этой книги, аппаратное обеспечение различается по множеству параметров, таких как порядок битов и байтов и размер слова. Помимо этого, существуют различия в том, как языки программирования представляют аппаратное обеспечение программисту. Например, проблема в С и С++ заключается в том, что языковые стандарты не определяют, является ли символ знаковым или беззнаковым. Обходной путь должен быть явным в коде.

Можно использовать оператор `sizeof` в С, чтобы определить количество байтов в типе данных. К сожалению, для определения битового и байтового порядка приходится писать небольшие программы. Многие языки включают способы определения, например, наибольшего и наименьшего числа, которые могут храниться в определенном типе данных.

Наборы символов — еще одна проблемная область. Использование UTF-8 позволяет избежать многих сложностей.

Многие программы используют внешние библиотеки и другие средства. Как изолировать что-то наподобие сравнения строк от системных различий? Один из способов — придерживаться стандартной функциональности. Например, такие стандарты, как POSIX, определяют поведение библиотечных функций.

Вы встретите различия между целевыми средами, с которыми будет нелегко справиться. Поместите как можно больше таких зависимостей в одно место, а не разбрасывайте их по всему коду. Это позволяет другим программистам легко вносить необходимые изменения.

Тот факт, что код может быть создан для другой системы, не значит, что это в принципе хорошая идея. Классический пример — система X Window. В начале 1980-х аспирант Стэнфордского университета Энди Бехтольшайм (Andy Bechtolsheim) разработал специальный персональный терминал, похожий на рабочую станцию, для работы в сети университета. Стэнфорд лицензировал конструкцию оборудования, которая легла в основу линейки продуктов Sun Workstation компании SUN Microsystems. Профессора Стэнфорда Дэвид Черитон (David Cheriton) и Кейт Лантц (Keith Lantz) разработали операционную систему V, которая работала на SUN. Она отличалась очень быстрым синхронным механизмом межпроцессного взаимодействия, то есть программы очень быстро взаимодействовали друг с другом. Пол Асенте (Paul Asente) и Брайан Рид (Brian Reed) разработали оконную систему W, которая была включена в V.

Этот код в конце концов попал в Массачусетский технологический институт, где был перенесен в UNIX и переименован в X. Но в UNIX не было быстрого синхронного межпроцессного взаимодействия (IPC, inter-process communication). Вместо этого использовалось более медленное асинхронное IPC, разработанное для зарождающегося интернета. Производительность X была хуже ужасной, и потребовалось серьезно ее переделать, чтобы сделать просто ужасной.

Управление версиями

Программы меняются: вы дополняете их, совершенствуете, исправляете ошибки и т. д. Как отслеживать старые версии? Важно иметь возможность вернуться в прошлое, чтобы увидеть, что изменилось, если в новой версии была найдена ошибка.

Пришло время добавить UNIXизмов. Даг Макилрой (Doug McIlroy) создал в начале 1970-х годов программу под названием *diff*, которая сравнивала два файла и генерировала список различий. Эта программа могла при необходимости производить вывод в форме, которую можно было передавать в текстовый редактор, чтобы пользователи создавали измененный файл на основе исходного файла и списка различий при помощи компоновки. Марк Рочкинд (Mark Rochkind) использовал эту идею для создания *системы управления исходным кодом* (Source Code Control System, SCCS). Вместо того чтобы хранить полную копию каждого измененного файла, SCCS сохраняла оригинал и список изменений для каждой версии. Это позволяло пользователям запрашивать любую версию файла, которая создавалась на лету.

У SCCS был неудобный пользовательский интерфейс, и она работала медленно, потому что по мере накопления версий приходилось применять больше наборов изменений для восстановления. Уолтер Тичи (Walter Tichy) выпустил *систему управления версиями* (англ. Revision Control System, RCS) в 1982 году. RCS получила лучший пользовательский интерфейс и использовала обратное дифференцирование (вместо прямого, как в SCCS). Это означает, что RCS сохраняла последнюю версию кода вместе с изменениями, необходимыми для создания более старых версий. Поскольку пользователям чаще всего требовалась текущая версия, система возвращала ответ намного быстрее.

SCCS и RCS хорошо работали, только будучи установленными на один компьютер. Дик Грюн (Dick Grune) разработал *систему параллельного управления версиями* (Concurrent Versioning System, CVS), которая, по сути, обеспечивала сетевой доступ к RCS-подобным функциям, а также была первой системой, использующей слияния вместо блокировок.

Первые инструменты SCCS и RCS плохо масштабировались, потому что они полагались на блокировку файлов. Пользователи «извлекали» файл, редактировали его, а затем «регистрировали». Извлеченный файл не могут изменять другие

пользователи. Поэтому, если вдруг кто-то заблокировал файл и ушел в отпуск, возникали серьезные проблемы. В ответ на это ограничение были созданы *распределенные* системы, такие как Subversion, Bitkeeper и Git. Эти инструменты заменяют проблему блокировки проблемой слияния. Любой пользователь может редактировать файлы, но при возврате он должен согласовать свои изменения с изменениями, сделанными другими.

Используйте одну из этих программ для отслеживания кода. RCS очень проста и удобна, если вы работаете над проектом в одиночку в своей собственной системе. Сейчас наибольшей популярностью для распределенных проектов пользуется Git. Изучите его.

Тестирование

Невозможно быть уверенным, что программа работает, пока не протестируете ее. Разрабатывайте набор тестов вместе с программой. (Некоторые методологии рекомендуют начинать с тестов.) Храните тесты в системе контроля версий. Опять же, одна из замечательных особенностей автоматизации UNIX заключается в том, что можно создать одну команду, запускающую полный набор тестов. Часто может пригодиться ночная сборка, когда сборка программы запускается ежедневно в определенное время и выполняются тесты. *Регрессионное тестирование* — это термин, используемый для описания процесса проверки того, что изменения кода не нарушили ничего, что работало раньше. *Регресс* в этом контексте означает «движение назад»; регрессионное тестирование помогает убедиться, что исправленные ошибки не появляются вновь.

Есть несколько программ, которые помогут провести тестирование. Существуют сложные фреймворки, которые позволяют тестировать пользовательские интерфейсы, печатая и нажимая на кнопки.

По возможности просите других специалистов генерировать тесты для вашего кода. Человек, пишущий код, подсознательно не заметит известных проблем и будет избегать написания тестов для них.

Создание отчетов и отслеживание багов

Пользователи найдут баги в вашем коде независимо от того, как вы его протестирували. Вам понадобятся способы сообщать о них и отслеживать исправления.

Опять же, для этого существует множество инструментов.

Рефакторинг

Рефакторинг — это процесс переписывания кода без изменения поведения или интерфейсов, похожий на замедленное быстрое прототипирование. Зачем он

вообще нужен? В первую очередь затем, что, завершив работу над кодом, вы замечаете, что он превратился в мешанину, — и у вас появляется план, как это исправить. Рефакторинг может снизить затраты на обслуживание. Однако стоит позаботиться о хорошем наборе тестов и убедиться, что переписанный код работает так, как должен. Кроме того, всякий раз, когда что-то переписывается, возникает соблазн добавить новые функции — не поддавайтесь ему. Время рефакторинга отлично подходит для того, чтобы пересмотреть принципы уже готового кода, как упоминалось в разделе «Быстрое прототипирование» на с. 513.

Обслуживание

Один неочевидный факт о программировании заключается в том, что для любого серьезного фрагмента кода стоимость обслуживания значительно превышает стоимость его разработки. Имейте это в виду. Избегайте милых шалостей, которые могут произвести впечатление на коллег. Помните: если бы те, кто обслуживает ваш код, были так же умны, как вы, они занимались бы не обслуживанием, а проектированием.

В главе 12 вы ознакомились с несколькими способами написания асинхронного кода на JavaScript. Некоторые из этих способов хранят все в одном месте, в то время как другие отделяют настройку от выполнения. Требуется больше времени на поиск и исправление багов, если предварительно приходится отслеживать все части.

Некоторые программисты считают, что программа — это произведение искусства, которое нужно понять во всей его полноте, прежде чем к нему прикасаться. Эта философия звучит прекрасно. Но на самом деле важнее, чтобы любой мог открыть произвольный фрагмент кода и быстро понять, что тот делает. Красивый код, который невозможно обслуживать, часто приводит к провалу. Найдите красоту в создании кода, который легко понять.

Что действительно помогает вести обслуживание, если код взаимодействует с оборудованием, так это добавление в код ссылок на спецификации оборудования. Если вы обращаетесь к какому-то журналу, укажите номер(а) страницы спецификации, где этот журнал описан.

Позаботьтесь о стиле

Люди часто учатся программированию, не понимая среды, в которой оно существует. В этом отношении нужно обратить внимание на несколько моментов.

Возможно, вы никогда не задумывались о системе образования. Прямо сейчас она извергает на вас знания, и некоторые из них вы на самом деле усваиваете. Откуда пришло это знание? Его открыли другие люди. В какой-то момент, особенно если вы хотите получить ученую степень, наступит ваша очередь открыть для

себя то, что когда-нибудь узнают другие. Одна из замечательных особенностей проектов с открытым исходным кодом заключается в том, что вы можете внести в них свой вклад. Даже если вы не готовы программировать, многим из этих проектов нужна помощь с документацией, поэтому принимайте в них участие, если это программа, которую вы используете или которая вас интересует. Это отличный способ познакомиться с людьми, а еще такой опыт станет бонусом при поступлении в колледж и при приеме на работу. Будьте осторожны, так как многие программисты не очень социализированы. Не будьте обидчивыми.

Создавая программные продукты, пишите код четко и тщательно документируйте. Убедитесь, что другие понимают, что происходит, иначе помочь вам никто не сможет. Получите «гарантию занятости», хорошо выполнив свою работу и поддерживаю соответствующую репутацию, а не убедившись, что никто, кроме вас, не сможет работать с вашим кодом. Я уже говорил, что стоимость поддержки продукта значительно превышает стоимость его разработки.

Старайтесь создавать ПО с открытым исходным кодом. Откройте миру ту часть своей работы, на которую можете положиться.

Научитесь писать по-английски (или на любом человеческом языке, на котором хотите) связно и правильно. Пишите настоящую документацию для своего кода. Избегайте инструментов для документации, таких как Doxygen. Вы, возможно, заметили, что они прекрасно подходят для создания массы бесполезной макулатуры.

Документация должна описывать, что делает код. Опишите структуру данных и то, как код управляет ими. Я впервые стал работать кодером в Bell Telephone Laboratories, когда учился в старшей школе. Вот это мне повезло! Мой начальник сказал, что к каждой строке кода должен быть комментарий. Будучи не слишком умным в то время, я сделал так:

```
lda foo ; загрузить foo в аккумулятор
add 1   ; добавить 1
sta foo ; снова сохранить результат в foo
```

Как вы, наверное, видите, эти комментарии были совершенно бесполезны. Лучше было бы написать так:

```
; В foo хранится количество гремлинов, сидящих за углом.
; Увеличить счетчик на 1, поскольку мы нашли еще одного.
lda foo
add 1
sta foo
```

Еще в 1985 году у меня возникла идея, что было бы здорово извлекать документацию из файлов с исходным кодом, особенно затем, чтобы изменять ее в том же месте, где был изменен код. Я написал инструмент под названием *xman* (extract manual, извлечь документацию), который генерировал руководства

в формате `troff` из исходного кода. Он использовал специальный комментарий в C, который начинался с `/**`, чтобы представить документацию. Примерно тогда же было принято мое предложение прочитать курс на конференции SIGGRAPH 1986. Мне нужны были дополнительные докладчики, и я связался с Джеймсом Гослингом, впоследствии изобретателем Java. Я продемонстрировал ему `xtrap`, и вскоре мы от него отказались, потому что стало ясно, что этот инструмент может создавать кипы красивой и неправильной документации. Хотя корреляция не демонстрирует причинно-следственную связь (и Гослинг этого не помнит), в Java был включен `Javadoc`, способ добавления документации в исходные файлы, а документация вводилась комментариями `/**`. Впоследствии эту технику скопировали многие другие инструменты. Так что, возможно, в этом беспорядке виноват я.

Автоматическая документация, как правило, относится к категории «добавить 1». Существуют массивы документов, где хранятся только имена функций, а также имена и типы аргументов. Если, просто взглянув на код, вы этого не видите, программирование не для вас! Лишь малая часть этой документации описывает то, что делает функция, как она это делает и как она связана с остальной частью системы. Мораль такова: не думайте, что модные инструменты гарантируют хорошую документацию. Пишите ее сами.

Последнее замечание по поводу документации: добавляйте то, что вам очевидно, то, о чем вы вообще не задумываетесь. Люди, которые будут это читать, не знают очевидных для вас вещей. В UNIX версии 6, где и так комментариев было немного, есть знаменитый экземпляр, в котором говорится: «Мы не ждем, что вы это поймете». Что ж, лучше не стало!

Чините, а не создавайте заново

Вселенная программ, особенно часть с открытым исходным кодом, усеяна на половину рабочими продуктами и продуктами, которые делают почти одно и то же. Не создавайте таких продуктов.

Старайтесь доводить до конца и свои, и чужие проекты. Если вы не смогли закончить свой проект, по крайней мере оставьте его в достаточно хорошей форме, чтобы другим было легко продолжать разработку. Помните: речь идет о повышении ценности.

Выводы

Теперь вы узнали, что программирование — это не просто знание аппаратной и программной части. Это сложная и полезная работа, связанная с большим объемом знаний в разных областях. Вместе мы проделали большой путь. Вы увидели, как представлять сложную информацию и работать с ней при помощи

битов. Вы узнали, почему вообще используются биты и как они встраиваются в оборудование. Мы изучили основные строительные блоки аппаратного обеспечения и то, как из этих блоков собираются компьютеры. Мы рассмотрели дополнительные функции, созданные для повышения удобства использования компьютеров, и различные технологии подключения компьютеров к внешнему миру. Затем обсудили, как организовать данные, чтобы использовать преимущества архитектуры памяти. Мы разобрали процесс преобразования языков программирования в инструкции, понятные для компьютеров, и узнали о веб-браузерах и о том, как они организуют данные и языки обработки. Мы сравнивали приложения высокого уровня с низкоуровневыми системными программами. Далее поговорили об интересных приемах решения задач на примере нескольких фотографий кошек и обсудили некоторые проблемы многозадачности. Затем рассмотрели передовые темы безопасности и машинного интеллекта, в которых нам встретилось еще больше кошек. Надеюсь, вы заметили, что основные строительные блоки и приемы используются снова и снова в разных комбинациях. Наконец, мы узнали, что, кроме аппаратной и программной частей, программирование подразумевает участие в нем людей.

Это лишь начало. Книга дает представление обо всем, что может пригодиться в программировании, и закладывает основу для дальнейшего обучения. Не останавливайтесь — впереди еще очень много нового.

Возможно, вы помните, как еще во введении я говорил, что нам нужно понять целую Вселенную. Но ни один человек не в состоянии объять необъятное. Например, мне так и не удалось выяснить, как красиво завершить книгу. Итак, это все. На этом ставим точку. Конец.