И. Ш. Хабибуллин

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ C/C++

- *Базовые компьютерные алгоритмы*
- *Структуры данных*
- *Принципы объектно-ориентированного программирования*
- *Жизненный цикл программного продукта*
- *Стандартная библиотека шаблонов STL*

УЧЕБНОЕ ПОСОБИЕ

bhv®

И. Ш. Хабибуллин

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ C/C++

Рекомендовано УМО по университетскому политехническому образованию
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлению 654600
«Информатика и вычислительная техника»

Санкт-Петербург

«БХВ-Петербург»

2006

УДК 681.3.068+800.92(075.8)
ББК 32.973.26.018.1я73
Х12

Хабибуллин И. Ш.

Х12 Программирование на языке высокого уровня. C/C++. — СПб.: БХВ-Петербург, 2006. — 512 с.: ил.

ISBN 5-94157-559-9

Учебное пособие написано на основе одноименного учебного курса и посвящено технологии программирования на языках высокого уровня. Рассматриваются элементы современных языков программирования с примерами их реализации на языке C/C++. Большое внимание уделяется стилю программирования. Разобраны базовые алгоритмы и основные структуры данных, принципы объектно-ориентированного программирования, работа со стандартной библиотекой шаблонов STL, а также этапы и современные методы разработки надежного программного обеспечения. Приемы программирования и применения алгоритмов и структур данных иллюстрируются фрагментами программ. Может использоваться как справочник по языкам C и C++, так как содержит схемы, таблицы, описания стандартных библиотек функций и библиотеку шаблонов классов STL.

Для студентов технических вузов

УДК 681.3.068+800.92(075.8)
ББК 32.973.26.018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Константин Костенко*
Компьютерная верстка	*Татьяны Олоновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Рецензенты:

В. Г. Трусов, д.т.н., профессор, заведующий кафедрой «Программное обеспечение ЭВМ и информационные технологии» Московского государственного технического университета им. Н. Э. Баумана, заместитель председателя УМК по специальности 220400 «Программное обеспечение ЭВМ»;

Ф. М. Аблаев, д.ф.-м.н., профессор, член-корреспондент Академии наук Республики Татарстан, заведующий кафедрой теоретической кибернетики факультета вычислительной математики и кибернетики Казанского государственного университета.

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 26.05.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 41,28.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-559-9

© Хабибуллин И. Ш., 2006
© Оформление, издательство "БХВ-Петербург", 2006

Оглавление

Предисловие **1**

Введение **3**

Составные части компьютера 4
Логическое строение памяти компьютера 5
Машинные команды 6
Поколения языков программирования 7
Алгоритмические языки 9
Функциональные языки 10
Логические языки 10
Языки низкого и высокого уровня 11
Компилируемые и интерпретируемые языки 11

Глава 1. Алгоритмы и их запись **15**

Понятие алгоритма 16
Массовость 17
Конечность 18
Однозначность 18
Исполнитель алгоритма 18
Алгоритм Евклида 19
Определение алгоритма 20
Способы записи алгоритма 20
Псевдокод 22
Блок-схема 24
Контрольный вопрос 25
Применение блок-схем 25
Алгоритмические языки 26

Структурные части алгоритма 27
Линейный алгоритм 27
Разветвление 29
Циклы 30
Упражнения 32
Стандартные алгоритмы 32
Действия с целыми числами 33
Упражнения 35
Суммирование и умножение 35
Упражнения 38
Вычисление многочлена по схеме Горнера 38
Работа с текстом 39
Упражнения 42
Перевод десятичного числа в двоичную форму 42
Упражнение 43
Упражнение 44
Контрольные вопросы 44

Глава 2. Простые типы данных 47

Числовые типы данных 48
Целые типы 48
Вещественные типы 50
Символьный тип 51
Шестнадцатеричная система счисления 52
Упражнения 53
Стандартные кодировки символов 53
Упражнения 60
Unicode 60
Упражнение 60
Символьный тип в языке C 60
Константы 61
Целые константы 61
Упражнение 61
Вещественные константы 62
Символы 62
Упражнения 63
Строковые константы 63
Имена объектов 64
Определение переменных 64
Операции 66
Арифметические операции 66
Приведение типов 67
Сравнения 69
Логические операции 69

Побитовые операции 70
Сдвиги 71
Операции присваивания 72
Условная операция 73
Операция "запятая" 73
Выражения 73
Приоритет операций 74
Упражнения 75
Контрольные вопросы 75

Глава 3. Основные структуры управления 77

Операторы 77
Блок....78
Операторы присваивания 79
Условный оператор 79
Упражнения 84
Первая программа на языке С 85
Функция вывода *printf()* 87
Упражнения 90
Функция ввода *scanf()* 90
Упражнение 91
Нахождение корней квадратного уравнения....91
Операторы цикла....93
Упражнения 96
Упражнения 99
Оператор *continue*....99
Оператор *break* 100
Оператор варианта *switch* 100
Оператор перехода *goto* 102
Упражнения 103
Контрольные вопросы 103

Глава 4. Массивы и указатели 105

Массивы 106
Упражнения 108
Многомерные массивы 108
Упражнения 110
Указатели....111
Адресная арифметика 113
Связь массивов и указателей 115
Работа со строками....118
Упражнения 120
Функции обработки строк 121

Работа с отдельными символами 123
Динамическое выделение памяти 124
Бинарный поиск в массиве 125
Упражнения 127
Методы сортировки массива 127
Сортировка вставкой 128
Сортировка выбором 130
Сортировка методом "пузырька" 130
Метод просеивания 133
Метод Шелла 134
Обзор методов сортировки 135
Контрольные вопросы 136

Глава 5. Процедурное программирование 137

Парадигмы программирования 137
Функции языка С 140
Оператор *return* 141
Передача аргументов функции 143
Упражнения 145
Передача массивов в функцию 145
Упражнения 146
Аргументы функции *main()* 147
Функции преобразования строк в числа 148
Процедуры 149
Указатели на функцию 150
Упражнения 152
Локальные и внешние переменные 152
Прототипы функций 156
Рекурсия 157
Быстрая сортировка 158
Контрольные вопросы 160

Глава 6. Работа с файлами 161

Функции буферизованного ввода-вывода 162
Упражнения 166
Потоки ввода-вывода 167
Форматированный ввод-вывод 168
Прочие функции ввода-вывода 169
Упражнения 174
Интерактивный ввод-вывод 174
Работа с каталогом 175
Контрольные вопросы 176

Глава 7. Типы данных, определяемые пользователем 177

Перечислимые типы 177
Упражнения 180
Структуры 180
Структуры как аргументы и возвращаемые значения функций 185
Упражнения 188
Объединения 189
Упражнения 191
Битовые поля 191
Упражнения 193
Контрольные вопросы 193

Глава 8. Препроцессор 195

Включение файла директивой *#include* 196
Подстановка *#define* и *#undef* 196
Макросы 197
Переменные препроцессора 198
Условная компиляция *#ifdef* 199
Нумерация строк *#line* 201
Сообщение *#error* 201
Прагма *#pragma* 201
Упражнения 202
Контрольные вопросы 202

Глава 9. Расширения языка C в языке C++ 203

Комментарии 203
Строгая типизация 204
Логический тип 204
Объявление переменных внутри блока 205
Операция разрешения видимости *::* 206
Определение констант 206
Ссылки 208
Упражнение 210
Прототипы функций 210
Функции с переменным числом аргументов 211
Встраиваемые функции 212
Явное приведение типов 212
Типы перечислений, структур, объединений 213
Операции выделения памяти 213
Упражнение 214
Перегрузка функций 214

Шаблоны функций 215
Перегрузка операций 217
Упражнения 221
Контрольные вопросы 222

Глава 10. Принципы ООП 223

Принципы объектно-ориентированного программирования 224
Абстракция 224
Иерархия 232
Полиморфизм 235
Ответственность 236
Модульность и инкапсуляция 237
Принцип KISS 240
Наследование или вложение 241
Упражнения 243
Контрольные вопросы 243

Глава 11. Реализация ООП в языке C++ 245

Описание класса 253
Упражнение 253
Указатель на себя *this* 254
Описание полей класса 254
Упражнение 254
Описание методов класса 255
Упражнение 256
Пример класса 256
Упражнение 258
Класс комплексных чисел 258
Упражнение 263
Класс с деструктором 263
Упражнение 267
Наследование классов 267
Инициализация полей классов 270
Конструкторы, деструкторы и наследование 271
Обращение к переопределенному методу суперкласса 272
Доступ к членам суперкласса 272
Реализация полиморфизма 273
Абстрактные классы 275
Множественное наследование 277
Друзья класса 280
Пространства имен 282
Уточнение имени 282
Директива *using namespace* 283

Вложение пространств имен ... 283
Объявление *using* ... 284
Неименованное пространство имен ... 284
Псевдонимы пространства имен ... 285
Упражнения ... 285
Контрольные вопросы ... 285

Глава 12. Обработка исключительных ситуаций ... 287

Блоки перехвата исключения ... 289
Часть *throw* заголовка функции ... 293
Оператор *throw* ... 296
Порядок обработки исключений ... 296
Создание собственных исключений ... 298
Упражнения ... 299
Контрольные вопросы ... 299

Глава 13. Дополнительные конструкции языка C++ ... 301

Шаблоны классов ... 301
Упражнения ... 305
Друзья шаблона ... 306
Члены класса — реализации шаблонов ... 307
Наследование шаблонов ... 308
Упражнение ... 310
Специализированные версии шаблона ... 310
Упражнение ... 311
Информация о типе на этапе выполнения ... 311
Контрольные вопросы ... 313

Глава 14. Динамические структуры данных ... 315

Стек ... 315
Упражнения ... 328
Дек ... 328
Упражнения ... 329
Очередь ... 329
Упражнения ... 330
Кольцо ... 330
Упражнение ... 331
Линейный список ... 331
Двунаправленный список ... 335
Упражнение ... 336
Бинарное дерево ... 336
Обход бинарного дерева ... 341
Упражнение ... 343

В-дерево ... 343
Упражнения ... 345
Контейнеры и итераторы ... 345
Контрольные вопросы ... 347

Глава 15. Работа со строками ... 349

Типы отдельных символов ... 350
Шаблон класса *basic_string* ... 351
Классы *string* и *wstring* ... 352
Как создать строку ... 352
Сцепление строк ... 354
Действия со строками ... 354
Как узнать длину строки ... 354
Как выбрать символы из строки ... 355
Как просмотреть строку ... 356
Как выбрать подстроку ... 356
Как сравнить строки ... 357
Как найти символ или подстроку в строке ... 358
Как заменить подстроку ... 360
Как заменить символ ... 360
Как добавить подстроку ... 360
Как вставить подстроку ... 361
Как удалить подстроку ... 362
Как удалить символ ... 363
Как переставить строки ... 363
Ввод и вывод строк ... 363
Упражнения ... 363
Контрольные вопросы ... 364

Глава 16. Потоки ввода-вывода ... 365

Консольный ввод-вывод ... 368
Особенности ввода ... 369
Перегрузка операций >> и << ... 372
Форматирование вывода ... 373
Методы класса *basic_ios* ... 373
Флаги ... 374
Манипуляторы ... 376
Состояние потока ... 377
Файловый ввод-вывод ... 377
Создание, открытие и закрытие файловых потоков ... 378
Перемещение по потоку ... 379

Чтение из файла и запись в файл 380
Прямой доступ к файлу 382
Строковые потоки 383
Упражнения 385
Контрольные вопросы 385

Глава 17. Классы-контейнеры 387

Шаблон класса *vector* 387
Как создать вектор 388
Как добавить элемент в вектор 389
Как заменить элемент 390
Как узнать размер вектора 390
Как обратиться к элементу вектора 390
Как удалить элементы вектора 392
Шаблон класса *stack* 392
Шаблон класса *deque* 394
Как добавить элемент в дек 395
Как удалить элементы из дека 395
Как узнать размер дека 396
Прочие методы дека 396
Шаблон класса *queue* 396
Шаблон класса *priority_queue* 397
Шаблон класса *list* 398
Как добавить элементы в список 399
Как удалить элементы из списка 399
Как заменить элемент 400
Как узнать размер списка 400
Как обратиться к элементу списка 400
Сортировка списка 401
Слияние списков 402
Ассоциативный массив *map* 402
Как создать ассоциативный массив 403
Как заполнить ассоциативный массив 403
Как узнать размер ассоциативного массива 404
Как получить значение по ключу 404
Как найти элемент ассоциативного массива 405
Как получить все значения массива 405
Как удалить элементы 405
Шаблон класса *multimap* 406
Шаблон класса *set* 407
Упражнения 408
Контрольные вопросы 408

Глава 18. Утилиты стандартной библиотеки 409

Сортировка 409
Поиск 413
Бинарный поиск 416
Подсчет числа элементов 418
Копирование 419
Замена 420
Удаление 421
Объекты-функции 421
Алгоритмы 423
Упражнения 424
Контрольные вопросы 425

Глава 19. Жизненный цикл программы 427

Программный продукт 428
Определение требований к продукту 429
Требования к функционированию продукта 429
Требования к надежности продукта 431
Условия эксплуатации продукта 432
Требования к техническим средствам 432
Требования к установке продукта 433
Техническое задание 433
Упражнения 434
Проектирование программного продукта 434
Анализ 435
Разработка эскизного проекта 436
Разработка технического проекта 437
Рабочий проект 438
Упражнения 438
Разработка программного продукта 439
Отладка 440
Тестирование 441
Сопровождение продукта 442
Модификация продукта 443
Цикличность разработки продукта 443
Контрольные вопросы 444

Глава 20. Методы отладки и тестирования программы 445

Обнаружение ошибки 447
Программа не дает результатов 448
Программа дает неверные результаты 449
Программа дает правдоподобные результаты 450

Локализация ошибки 450
Устранение ошибки 451
Средства отладки 452
Упражнения 453
Тестирование 453
Unit-тестирование 454
Методики тестирования 456
Функциональное тестирование 457
Тестирование обращений к базам данных 459
Тестирование бизнес-логики программы 459
Нагрузочное тестирование 459
Стрессовое тестирование 460
Тестирование интерфейса пользователя 460
Тестирование безопасности и прав доступа 460
Тестирование инсталляции программного продукта 461
Наборы тестов 461
Процесс тестирования 462
Особенности тестирования объектно-ориентированных программ 463
Средства тестирования 464
Обеспечение качества программного продукта 465
Упражнения 465
Контрольные вопросы 466

Глава 21. Верификация программы 467

Верификация алгоритма 468
Метод индуктивных утверждений 469
Пример 471
Использование верифицированных фрагментов 472
Упражнения 472
Доказательство завершения цикла 473
Метод Флойда 473
Метод счетчиков 474
Упражнения 474
Автоматизация верификации 475
Достоверность верификации 475
Модельный подход к верификации 476
Контрольные вопросы 477

Литература 479

Предметный указатель 481

Предисловие

Книга, которую вы держите в руках, посвящена введению в современное состояние алгоритмических языков и технологии программирования на языках высокого уровня, главным образом, на объектно-ориентированных языках. Она нацелена на студентов младших курсов технических специальностей, изучающих дисциплину "Алгоритмические языки и программирование", но может использоваться всеми программистами, не применявшими в своей работе языки C/C++ и объектно-ориентированное программирование, но желающими быстро и в полном объеме овладеть ими.

Ну и что, скажете вы? Таких книг море. Есть масса учебников для всех желающих. Книг, посвященных языку C/C++, в каждом магазине лежит добрый десяток. Среди них есть подробнейшие тысячестраничные "кирпичи" и для профессионалов, и для начинающих программистов. Зачем нужен еще один учебник по C/C++?

Во-первых, я сам с большим удивлением обнаружил, что среди великого многообразия книг по программированию нет учебника по курсу "Алгоритмические языки и программирование", отвечающего действующему ныне государственному образовательному стандарту. Информационные технологии развиваются очень быстро, и стандарты меняются раз в несколько лет. Выпуск учебников отстает даже от выпуска образовательных стандартов.

Во-вторых, книга не акцентирует внимание на синтаксических конструкциях языка программирования. В современные средства разработки встроена подробная интерактивная справка (Help), из которой всегда можно узнать синтаксические правила, да и компилятор укажет на неправильные конструкции в тексте программы. Язык C/C++ выбран только потому, что он остается основным алгоритмическим языком программирования в современных разработках программного обеспечения.

В-третьих, для чтения книги не нужны никакие предварительные познания в программировании, ее может читать любой человек, знакомый с компьютерами на уровне "продвинутого" пользователя и помнящий некоторые сведения из школьного курса основ информатики и вычислительной техники.

В-четвертых, сейчас все большее распространение получает свободно распространяемое программное обеспечение. Книга выводит читателя на уровень его разработки. Программист, прочитавший эту книгу, сможет сразу же принимать участие в разработке Linux, FreeBSD, XFree, OpenOffice и другого открытого программного обеспечения.

Эти отличительные особенности повлияли на манеру написания книги. Ее первые главы написаны очень просто, их материал может понять каждый логически мыслящий человек. По мере подачи материала изложение становится более сжатым. Это диктуется усложнением тематики последующих глав и конечной целью книги — вывести читателя на передний край современного программирования.

Основное внимание в книге уделяется стилю программирования. За время работы с компьютерами мне пришлось использовать множество языков, поработать во многих операционных системах и изучить множество технологий создания компьютерных программ. Каждая технология диктует свои правила программирования, но два правила остаются неизменными.

Первое правило заключается в неуклонном следовании принципу KISS. Эта аббревиатура расшифровывается так: "Keep It Simple, Stupid". Здесь в шутливой и грубоватой форме выражена рекомендация: программировать как можно проще, не применять вычурных конструкций и вынужденных трюков, которые затрудняют чтение текста программы и могут привести к трудноисправимым ошибкам в ней.

Второе правило гласит: программировать надо стильно. У каждого языка, у каждой технологии программирования есть свой стиль. Очень важно, знакомясь с языком, сразу же проникнуться его стилем и следовать ему в своей работе. Это предотвратит многие ошибки и даст возможность работать в любой команде профессионалов и свободно общаться с другими разработчиками.

В книге много упражнений и вопросов, взятых из жизни. С такими задачами и вопросами я сталкивался сам, изучая языки программирования. Такие задачи и вопросы ставили передо мной ученики. Если вы сумеете ответить на них, значит, вы хорошо усвоили материал раздела и можете смело переходить к следующему пункту.

В книге много листингов фрагментов реальных программ. Все они проверены компилятором GCC 3.4.2 с библиотекой функций и классов glibc 2.3.5 и выполнены в программной среде Linux 2.6.9, собранной в дистрибутиве Fedora Core 3. Все фрагменты программ написаны с использованием только стандартных средств языков C и C++, в них не применены никакие особенности конкретных компиляторов.

Для выполнения упражнений достаточно иметь под рукой текстовый редактор, компилятор с библиотекой функций и классов и программную среду, в которой будут выполняться готовые программы. Выбор компилятора обычно диктуется операционной системой, для которой пишется программа. Если вы пишете программу под UNIX, то используйте компилятор cc, имеющийся в среде разработки вашей системы. Если вы работаете в Linux или FreeBSD, то пользуйтесь компилятором GCC. Для .NET лучше всего подойдет Visual Studio for .NET. У профессионалов есть свои соображения и предпочтения, но к компилятору надо привыкнуть, ознакомиться с его особенностями и возможностями, а это требует некоторого времени.

Введение

История электронной вычислительной техники насчитывает уже более шестидесяти лет. Сначала электронные вычислительные машины собирались на триодах и пентодах, потом на транзисторах, затем на микросхемах. Микросхемы становились все более и более емкими и компактными. Сейчас микросхемы настолько усложнились, что изменилось даже их название, специалисты стали называть их чипами. Электронные устройства, похоже, исчерпали свои возможности, и, говоря о будущем вычислительной техники, ученые все увереннее называют другие физические принципы. Ожидаются оптические компьютеры, квантовые компьютеры, говорят о сверхпроводимости при низких температурах.

Поэтому, вводя понятие компьютера, сейчас не упоминают об электронике и вообще не говорят о его физическом устройстве. Компьютер определяется просто как *устройство для хранения, обработки и передачи информации.*

Элементная база машин часто меняется, но одно остается неизменным — электронные элементы всегда имеют два устойчивых состояния. Конечно, можно собрать электронные устройства с большим числом устойчивых состояний, но надежность таких устройств будет невелика, поскольку различать большое число состояний очень трудно.

У двух устойчивых состояний может быть самая разная физическая природа: магнитный сердечник может быть намагничен по часовой стрелке или против, на концах проводника может быть разность потенциалов или ее отсутствие, постоянный ток может идти в прямом или обратном направлении, конденсатор может быть заряжен или разряжен и т. д. Чтобы управлять этими состояниями, для них надо ввести какие-то обозначения. Можно было бы обозначить одно устойчивое состояние плюсом, а другое — минусом, можно было бы использовать диез и бемоль, "инь" и "янь", но специалисты выбрали ноль и единицу. Это было сделано не случайно, а для того чтобы в действиях с числами можно было применить двоичную систему счисления (*см. гл. 1*).

Итак, *физически* компьютер может быть собран из самых разных электронных элементов, но с точки зрения программирования, как говорят,

логически, компьютер состоит из огромного множества миниатюрных устройств, принимающих значения 0 и 1. Все сведения и все команды, которые мы хотим передать компьютеру, должны состоять только из нулей и единиц. Например, вот такая строка нулей и единиц хорошо понятна компьютеру:

00000001000011010000111000001010

Для некоторых типов машин это команда сложения двух целых чисел. Человеку такая запись ничего не говорит. Более того, человек не может ее запомнить, ему трудно даже переписать ее без ошибок. Что ж, очевидно, человек и компьютер говорят на совершенно разных языках.

У первых компьютеров на передней панели располагался ряд из 16 или 32, по числу разрядов машины, тумблеров. Включенное положение тумблера означало единицу, выключенное — ноль. Оператор машины набирал команду, состоящую из нулей и единиц, ставя в нужное положение соответствующие тумблеры, и нажимал кнопку ввода команды. После этого можно было набирать следующую команду, изменяя состояние тумблеров. После ввода всех команд оператор нажимал кнопку запуска, и программа начинала выполняться. Сейчас все эти манипуляции кажутся смешными и нелепыми, но в то время даже такая работа электронной вычислительной машины вызывала немалое восхищение.

Результат работы программы распечатывался на длинном рулоне бумаги, а промежуточные результаты высвечивались рядом небольших лампочек. Это было очень неудобно, но ученые были чрезвычайно довольны.

Составные части компьютера

Исходя из определения компьютера как устройства для хранения, обработки и передачи информации, следует заключить, что он состоит из трех основных частей:

- устройства для хранения информации или памяти (store);
- устройства для обработки информации — центрального процессора (Central Processor Unit, CPU);
- устройств для передачи информации — устройств ввода-вывода (Input/Output units, I/O).

Процессор называется центральным, потому что в каждом компьютере есть еще несколько небольших специализированных процессоров, не играющих самостоятельной роли. Они выполняют операции ввода-вывода, стандартные преобразования информации, другие рутинные действия, и совершенно незаметны программисту.

Логическое строение памяти компьютера

Память компьютера разделена на несколько уровней: регистры, кэш-память, оперативная память, внешняя память. Эти уровни различаются емкостью, быстродействием чтения и записи, ценой. Регистры встроены в процессор и используются им для хранения промежуточных результатов вычислений. Это самая быстрая память, но и самая дорогая. Кэш-память расположена между процессором и оперативной памятью и служит буфером между ними, ускоряющим обмен данными. Основным уровнем является оперативная память.

Оперативная память современного компьютера собрана на чипах, состоящих из миниатюрных элементов, имеющих, как говорилось выше, два устойчивых состояния. Эти элементы хранят *биты* (bit — сокращение слов binary digit) информации. Такое название, в буквальном переводе "двоичная цифра", выбрано потому, что в двоичной записи один бит — это один разряд двоичного числа: 0 или 1.

Бит — очень мелкая единица хранения информации. Допустим, нам надо хранить тексты в виде последовательностей букв, цифр и других печатных символов. Мы можем обозначить символ "А" нулем, символ "Б" — единицей и тем самым исчерпать возможности одного бита.

Два бита дают уже четыре варианта: 00, 01, 10, 11. Три бита дадут возможность хранения восьми символов, например, можно обозначить "А" — 000, "Б" — 001, "В" — 010, "Г" — "011", "Д" — 100, "Е" — 101, "Ё" — 110, "Ж" — 111. Вообще, N битами можно закодировать 2^N символов. Поскольку обычные тексты состоят из 150—200 символов, включая знаки препинания, математические обозначения и прочее, достаточно взять 8 битов, которыми можно закодировать 256 символов. Число 8 удобно еще и тем, что это третья степень двойки, а машина "любит" двоичный счет.

Набор из 8 битов называется *байтом* (byte). Правда, сначала байт состоял из семи битов, но это было очень давно. Во многих компьютерах к каждому байту добавляют девятый, контрольный, бит, но, поскольку он недоступен для программирования и не содержит полезной информации, его не включают в число битов, составляющих байт. Байт — это самая мелкая единица хранения информации в памяти компьютера. Все байты в ней пронумерованы, начиная от 0. К каждому байту можно обратиться по его номеру, как говорят, *адресу*, и прочитать хранящуюся в нем информацию или записать новую информацию.

Байт оказался слишком мелкой единицей хранения информации, ведь в нем можно записать всего один символ. Поэтому чаще используется *килобайт* — тысяча байтов, сокращенно KB, *мегабайт* — тысяча килобайтов, сокращенно MB, *гигабайт* — тысяча мегабайтов, GB, *терабайт* — тысяча гигабайтов, TB. Специалисты по сетевым технологиям, называющие байт

октетом (octet), предпочитают употреблять не байт, а бит. Они пользуются *килобитами*, сокращенно Kb, *мегабитами*, Mb и т. д.

В прошлом веке греческие приставки кило, мега, гига, тера необоснованно означали множитель, кратный не 1000, а $1024 = 2^{10}$, но в декабре 1998 г. Международной электротехнической комиссией (International Electrotechnical Commission, IEC) был принят стандарт, по которому этим приставкам вернули их исконный смысл. Единицы информации с множителем 1024 получили новые названия *кибибайт* (kibibyte, KiB), равный 1024 байтам, *мебибайт*, (mebibyte, MiB), равный 1024 кибибайтам, *гибибайт* (gibibyte, GiB), равный 1024 мебибайтам, *тебибайт* (tebibyte, TiB) и т. д. Новые названия еще не прижились, но старые названия с новым смыслом уже используются производителями. Например, если производитель жесткого диска утверждает, что его емкость составляет 160 Гбайт, то будьте уверены, что это именно 160 000 000 000 байтов и ни одним байтом больше.

Кроме этих точных единиц измерения объема памяти, существует еще понятие ячейки памяти. *Ячейка памяти* — это некоторое количество байтов: 2, 4, 8, 100, необходимое для хранения одного числа, одного слова, одной машинной команды. *Адресом* ячейки считается адрес ее первого байта, а *длиной* — количество байтов, ее составляющих. Говорят, что "для хранения целого числа выделяется ячейка длиной в 4 байта", что "переменная x хранится в ячейке с адресом 2048" и т. д.

Машинные команды

Машинные команды, исполняемые центральным процессором — это повелительные утверждения вида: "Прибавить к содержимому ячейки 2020 содержимое ячейки 2024", "Перенести содержимое ячейки 4000 в ячейку 4800", "Сравнить на равенство содержимое ячеек 1600 и 1604". Они состоят из двух частей: кода операции "прибавить", "перенести", "сравнить" и адресов операндов. В некоторых случаях команды содержат еще дополнительные информационные биты.

Обычно процессоры могут выполнять около сотни команд, поэтому для записи кода операции достаточно выделить один байт. Чаще всего так и делается, хотя иногда код операции занимает больше или меньше места в памяти. Адреса операндов занимают от одного до десяти байтов. Для каждого типа процессоров определяется самая распространенная длина команды. Она называется *машинным словом* или просто *словом* (word). У большинства популярных современных процессоров машинное слово составляет 4 байта, 32 двоичных разряда, но сейчас идет активный переход к 64-разрядным процессорам, у которых машинное слово равно 8 байтам.

Величина машинного слова определяет все остальные характеристики процессора. Он обрабатывает информацию порциями величиной в одно ма-

шинное слово, поэтому шины, которые связывают оперативную память и процессор, имеют число проводников, совпадающее с машинным словом. Как уже говорилось выше, внутри процессора есть небольшое количество ячеек очень быстрой памяти, называемых *регистрами*. Обычно их число колеблется от 16 до 64 ячеек. Регистры используются процессором для хранения промежуточных значений. Длина большинства регистров также совпадает с машинным словом.

Машинные команды хранятся в оперативной памяти, так же, как и те данные, которые они обрабатывают. Процессор по очереди извлекает команды из оперативной памяти, переносит их в свои регистры, там по коду операции узнает, что ему надо делать, и выполняет команду. В процессе выполнения команды процессор извлекает из оперативной памяти нужные операнды, а после выполнения заносит в оперативную память результаты команды.

Какие же команды можно давать процессору? Хотя их довольно много, они составляют всего несколько групп:

- команды пересылки информации из оперативной памяти в регистры процессора и обратно;
- арифметические команды сложения, вычитания, умножения и деления;
- команды сравнения по величине двоичных чисел;
- логические команды конъюнкции, дизъюнкции, отрицания, инверсии;
- команды сдвига двоичных разрядов;
- прочие команды, в числе которых команды передачи управления и несколько команд ввода-вывода.

Как видите, машинные команды очень примитивны, они выполняют мелкие действия над отдельными ячейками. Представьте себе, сколько нужно команд типа "загрузить содержимое ячейки с адресом 2048 в регистр 05" и "сдвинуть на два разряда влево содержимое регистра 12", чтобы запрограммировать задачу "найти в телефонном справочнике номер абонента с какой-то лошадиной фамилией, переехавшего в этот район в ноябре или декабре прошлого года". Представили? Тогда вы можете оценить гигантский труд программистов, проделанный ими всего за каких-нибудь полвека.

Поколения языков программирования

Двоичная запись информации крайне неудобна для человека, поэтому довольно быстро от двоичной записи перешли к восьмеричной записи, а затем к шестнадцатеричной. О шестнадцатеричной системе счисления мы поговорим позже (*см. гл. 2*), а пока я приведу шестнадцатеричную запись той же машинной команды сложения, которая была показана в двоичной записи:

010D0E0A

Как видите, запись команды осталась такой же непонятной, но, по крайней мере, в ней труднее ошибиться. Главное же заключается в том, что такая запись непонятна и машине! Для нее это просто ничего не значащий набор символов.

Пришлось написать небольшую программу, переводящую шестнадцатеричную запись в двоичную. После этого процесс выполнения программы усложнился. В машину сначала вводится непонятный для нее текст программы, написанный в шестнадцатеричных кодах. Потом этот текст обрабатывается программой, преобразующей каждый символ в двоичный набор нулей и единиц. И только после этого преобразованная программа начинает выполняться.

Шестнадцатеричная или восьмеричная запись команд стала первым языком программирования, требующим перевода на машинный язык. Такие языки были названы языками программирования *первого поколения*, *1GL* (1 Generation Language).

Следующим шагом стал переход от шестнадцатеричной записи информации к символьной. Раз уж машина не понимает вводимый в нее текст, то почему бы не сделать его хотя бы понятным человеку? Договорились обозначать команды сокращенной записью выполняемых ими действий, а ячейки памяти — какими-нибудь именами. Например, команду сложения целых чисел обозначили `ADD` — английским глаголом "прибавить". В этих обозначениях предыдущий пример выглядит так:

```
ADD A, B, C
```

Такая запись уже приемлема для человека. Разумеется, она совершенно непонятна машине, и понадобилось написать программу, переводящую эту запись в двоичный код. Такая программа была названа *ассемблером*, а язык — *языком ассемблера*. Языки ассемблера составили *второе поколение* языков программирования, *2GL*.

Наконец, через 10 лет развития вычислительной техники удалось разработать языки, позволяющие записывать команды машине почти как математические формулы. Например, предыдущую команду на таком языке можно записать в виде

```
C = A + B
```

Это был революционный прорыв. Теперь программисту не надо было думать о том, какие машинные команды будут выполняться в его программе. Он может вообще не знать систему команд компьютера. Более того, он может даже не знать тип компьютера, для которого пишет программу. Такими языками (*языками третьего поколения*, *3GL*) мы пользуемся до сих пор. Первым языком третьего поколения был язык вычисления формул FORTRAN, потом появился Algol и множество других, среди которых ныне

здравствующие — Pascal, C, C++, Java, C#. Программа, переводящая запись на машинный язык, называется *компилятором*. Конечно, компиляторы гораздо сложнее ассемблеров, это большие и сложные программы, создаваемые целыми коллективами программистов.

Для каждого языка третьего поколения создан не один компилятор. Во-первых, разные компиляторы создаются разными фирмами, конкурирующими между собой. Во-вторых, различаются отладочные и оптимизирующие компиляторы, создающие разный, черновой или окончательный, машинный код. В-третьих, для каждого типа компьютеров разрабатываются свои компиляторы, ведь набор машинных команд может сильно различаться. Выбор подходящего компилятора — это одна из важнейших задач программиста, приступающего к реализации своего проекта.

На этом счет поколениям языков программирования не остановился. Были созданы *декларативные* языки, составившие *четвертое поколение* языков программирования, *4GL*. Основная идея декларативных языков заключается в том, чтобы записывать не действия, приводящие к результату выполнения программы, а указания машине, по которым она сама проделает необходимые операции. Языки четвертого поколения говорят, *что* надо сделать, а не то, *как* это сделать.

Самым ярким примером языка четвертого поколения стал язык SQL (Structured Query Language), на котором происходит обращение к базам данных. На этом языке мы можем записать команду, гласящую: "Выбери из базы данных сведения обо всех Петровых, родившихся в январе 1985 года и не получивших среднее образование". Компилятор сам сгенерирует необходимые для выполнения этой команды действия и запишет их на машинном языке.

Попытки создать языки пятого поколения, основанные на голосовом вводе обычных для человека указаний типа: "Реши-ка мне дифференциальное уравнение", оказались неудачными и незаметно были сняты с повестки дня. Неожиданные новшества в создании новых языков прекратились. Теперь языки программирования развиваются постепенно, без заметных скачков.

Итак, на современных языках программирования записываются не машинные команды, а *операторы*, требующие предварительного перевода на машинный язык. Каждый оператор преобразуется компилятором в несколько машинных команд. По типу и составу операторов языки программирования делятся на алгоритмические или процедурные, функциональные и логические.

Алгоритмические языки

Программа, записанная на алгоритмическом языке программирования, представляет собой запись алгоритма решения задачи, ради которого написана программа. Она состоит из совокупности последовательных действий,

приводящих, в конце концов, к определенному результату. Такие языки программирования и являются предметом всей данной книги, поэтому здесь мы долго говорить о них не будем.

К алгоритмическим языкам относятся и объектно-ориентированные языки программирования. Алгоритмические языки программирования часто называют *процедурными* языками, но после появления объектно-ориентированных языков это название может ввести в заблуждение, ведь на объектно-ориентированных языках пишут не процедуры, а классы и входящие в них методы.

Функциональные языки

Функциональные языки программирования, такие как Lisp, Рефал, APL, Haskell, SMI, Scheme и другие, подходят к решению поставленных перед ними задач совершенно иначе, чем алгоритмические языки. Они представляют каждое выполняемое в программе действие в виде функции. Например, операция суммирования двух чисел записывается как функция `sum(a, b)` от двух переменных-слагаемых. Более сложные выражения записываются в виде набора функций, вызывающих друг друга. Например, выражение $\sqrt{a^2+b^2}$ будет на функциональном языке выглядеть примерно так: `sqrt(sum(sqr(a), sqr(b))).`

Функциональные языки не признают присваиваний и передач управления. Циклы вычисляются с помощью рекурсии, разветвления реализуются логическими функциями. Такая запись удобна при обработке символьной информации, длинных списков, ее выгодно применять для поиска и замены некоторых участков текста. Поэтому на функциональных языках часто пишут компиляторы, программы поиска, программы автоматического доказательства теорем.

Логические языки

Логические языки программирования, такие как Prolog, Mandala и другие, смотрят на результат программы как на некоторое утверждение, которое надо вывести из исходных данных. Программа рассматривается как доказательство этого утверждения. Доказательство проводится по правилам математической логики, откуда и взято название этих языков.

В программе, написанной на логическом языке, вообще нет действий. В ней задаются исходные данные как аксиомы и правила вывода истинных утверждений из этих аксиом. Программа по этим правилам проводит логические рассуждения и делает выводы, приводящие к нужному результату.

При этом для ускорения и упрощения работы программа использует накопленную ранее базу данных, содержащую уже сделанные логические выводы.

Логические языки программирования наиболее удобны для решения задач искусственного интеллекта, медицинской диагностики, машинного перевода — везде, где нужно делать логические заключения. Они применяются и для составления сложных расписаний, "добычи знаний" из больших баз данных, прогнозирования в экономике.

Языки низкого и высокого уровня

Еще одно популярное деление языков программирования — это деление на низкоуровневые и высокоуровневые языки программирования. Низкоуровневые языки близки к машинным языкам, они содержат набор машинных команд, записанных по правилам этого языка, а также директивы, макросы и другие элементы. Это, прежде всего, языки ассемблера и различные псевдокоды.

Программы, написанные на языках низкого уровня, наиболее близки к машине. На них можно в точности написать последовательность машинных команд, составляющих программу. Компиляторы низкоуровневых языков, чаще всего это ассемблеры, сравнительно невелики, работают быстро и надежно. Но программа, написанная на языке низкого уровня, получается очень длинной, трудной для понимания и выявления ошибок.

Высокоуровневые языки программирования, такие как языки третьего поколения, напротив, далеки от машинных команд. Программа получается относительно короткой, понятной и удобной для восприятия. Но компиляторы высокоуровневых языков велики, громоздки, работают медленно и зачастую неэффективно.

Особняком стоит язык С, который мы будем изучать в первой части книги. По своему синтаксису это типичный язык высокого уровня, но в его состав входят операции, позволяющие работать с отдельными байтами и битами, как это делают языки низкого уровня. Поэтому многие специалисты относят его к языкам низкого уровня. Иногда его называют даже "переносимым ассемблером".

Компилируемые и интерпретируемые языки

До сих пор мы говорили о том, что программу, написанную на языке высокого уровня, перед выполнением следует откомпилировать, т. е. перевести на машинный язык. Компилятор — это тоже компьютерная программа, на

вход которой подается файл с исходным текстом, написанным на языке высокого уровня. Этот файл во многих операционных системах называется *исходным модулем* (source module). Компилятор переводит программу на машинный язык и записывает ее в другой файл, называемый *объектным модулем* (object module).

Содержимое объектного модуля в самых простых случаях можно загрузить в оперативную память и выполнить. Но чаще всего объектный модуль перед выполнением надо еще *скомпоновать* (link) с другими объектными модулями, содержащими дополнительные модули программы, системные функции, служебные сведения и прочие данные, необходимые для выполнения программы.

Компоновку выполняет специальная программа, так и называемая *компоновщиком* (linker) или *редактором связей*. На ее вход подаются файлы с объектными модулями, а на выходе получается *исполнимый модуль* (executable module) — файл с полностью готовой к выполнению программой. Этот файл загружается в оперативную память и выполняется.

Очень часто компиляцию и компоновку объединяют в одно действие. Для этого компилятору дается специальное указание, и он сразу же после своей работы сам вызывает компоновщик. Программисту все это представляется единым процессом, только на экране дисплея появляются сообщения об окончании компиляции и начале компоновки.

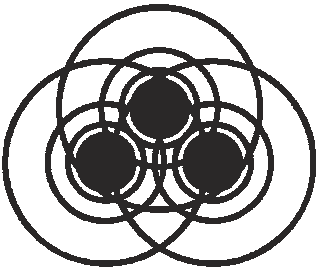
Все описанное выше характерно для *компилируемых* языков программирования. Другой класс языков составляют *интерпретируемые* языки программирования, не требующие компиляции и компоновки. Это тоже языки высокого уровня, поэтому программы, написанные на них, часто называемые *скриптами* (script) или *сценариями*, требуют перевода на машинный язык. Для этого создается программа-интерпретатор операторов, записанных в программе. В отличие от компилятора программа-интерпретатор переводит на машинный язык не весь текст целиком, а каждый оператор по отдельности, и тут же выполняет полученный машинный код.

Интерпретация программы происходит медленнее, чем выполнение исполнимого модуля, из-за того, что каждый оператор сначала преобразуется в машинные коды. Кроме того, в процессе интерпретации могут неожиданно возникнуть ошибки, ведь исходный текст не просматривается заранее в поисках ошибок. Тем не менее, интерпретируемые языки широко распространены из-за удобства создания программ. Программу можно написать и сразу выполнять без предварительных преобразований. Если работа программы кажется неудовлетворительной, то ее текст можно изменить и тут же выполнить программу заново без всякой перекомпиляции.

В современном программировании разница между компилируемыми и интерпретируемыми языками стирается. Это слияние происходит по двум на-

правлениям. С одной стороны, для интерпретируемых языков создаются компиляторы. Классическим интерпретируемым языком всегда считался язык BASIC, но сейчас для него написано много компиляторов. У программиста есть выбор. Программу, написанную на BASIC, можно интерпретировать и сразу выполнить, а можно предварительно откомпилировать в исполнимый модуль и выполнить в другое, более удобное время.

С другой стороны, интерпретаторы научились сохранять машинный код уже проинтерпретированных и выполненных операторов. При повторном выполнении этих операторов, например, в циклах, интерпретатор использует готовые машинные команды, что значительно ускоряет работу. Такие интерпретаторы называются *JIT-интерпретаторами* (Just-In-Time). Они работают значительно быстрее классических интерпретаторов и поэтому приобретают все большее распространение.



Глава 1

Алгоритмы и их запись

Компьютер работает очень быстро. Он выполняет сотни миллионов операций в секунду. Человек, даже тренированный, способен сделать за секунду только несколько осмысленных действий. Если изменить масштаб времени так, чтобы компьютер выполнял не больше десяти команд в секунду, то человек в этом масштабе будет совершать одно движение в 10—20 лет! Ясно, что при таком соотношении скоростей нельзя управлять компьютером как автомобилем, во время его работы. В таком случае компьютер практически все время будет простаивать, ожидая следующего действия пользователя. Надо дать компьютеру все инструкции или большинство инструкций до начала его работы, и он будет выполнять их со своей сумасшедшей скоростью. Вся программа действий компьютера должна быть заранее записана и введена в его оперативную память.

Допустим, мы выбрали один из языков программирования высокого уровня и решили написать на нем программу для компьютера. Что же она будет содержать? А что вообще содержат всякие инструкции и указания, которыми мы руководствуемся в жизни? Каждый из вас легко приведет множество примеров.

- Кулинарный рецепт: "Довести до кипения 5 стаканов подсоленного молока, тонкой струйкой всыпать 1 стакан манной крупы и, непрерывно помешивая, варить на слабом огне 10—15 минут до загустения. После этого положить 1 столовую ложку сахара и размешать".
- Совет прохожего: "Идите прямо два квартала, на перекрестке поверните направо, дойдите до магазина и войдите во двор. Зайдите во второй подъезд и поднимитесь на третий этаж".

- Дозировка и применение лекарства: "Принимать по одной таблетке 3—4 раза в день после еды в течение недели. При необходимости повторить курс лечения через 2—3 недели".
- Инструкция к стиральной машине: "Загрузите белье в машину, распределив равномерно крупные и мелкие вещи, и плотно закройте дверцу. Засыпьте стиральный порошок в выдвижной ящик. Нажмите кнопку Питание, чтобы включить машину. Выберите необходимые условия работы, поворачивая программатор по шкале и нажимая на каждую кнопку. Нажмите кнопку Старт".

Эти инструкции может выполнить любой человек, и не только человек, но даже машина. Другие инструкции требуют специальной подготовки.

- Из инструкции по окраске полов: "Профилированные галтели олифят. После высыхания шпаклюют резиновой пластинкой на сдир и шлифуют". Даже чтобы понять эти фразы, нужны соответствующие знания.
- Ремонт велосипеда: "Установить ротор на цапфу коленчатого вала, а кулачок на ротор так, чтобы штифты вошли в прорези ротора. Поставить и закрепить наковальню и конденсатор двумя винтами, при этом щеточка должна быть заведена за кулачок". Да, тут без инструктора не обойдешься.

Инструкции могут быть записаны специальными обозначениями, понятными только посвященным людям.

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Известная со школы формула — это инструкция по вычислению корней квадратного уравнения. Для человека, не изучавшего математику, это какая-то непонятная шифровка, бессмысленный набор символов.

Аналогичные указания следует дать компьютеру. При этом надо учесть, что компьютер не обладает никакими предварительными знаниями, позволяющими человеку додумывать то, что не написано явно в инструкции. Компьютер — это "быстродействующий идиот", выполняющий то, что ему предписано. Даже людьми самые подробные инструкции понимаются поразному. Это иногда приводит к ошибкам, что хорошо знают военные, стремящиеся доводить выполнение команд до автоматизма. Что уж говорить о компьютере. Команды для него надо писать подробно и точно. Компьютеру дают не инструкцию, а алгоритм решения поставленной задачи.

Понятие алгоритма

Алгоритм или, по старинке, алгорифм — очень общее понятие и, как всякое общее понятие, его трудно определить. Попробуйте в двух-трех фразах

объяснить, что такое метод, наука, культура. Существует целая теория алгоритмов, которая вводит не одно, а даже несколько определений, доказывает их эквивалентность и детально изучает алгоритмы исходя из строгого определения. Теория алгоритмов доказала много интересных теорем. Оказывается, например, что область их применения ограничена — есть алгоритмически неразрешимые классы задач. У нас другой учебный курс, и нам не нужно точное определение. Для нас достаточно описать основные свойства: массовость, конечность, однозначность.

Массовость

Ценность алгоритма возрастает, если его можно применить к решению не одной, а целого класса задач. Такое свойство алгоритма называется его *массовостью*. Известные из арифметики правила сложения и умножения чисел "столбиком" — массовые алгоритмы, применимые к любым слагаемым или сомножителям.

Приведенная выше формула нахождения корней квадратного уравнения массовая, она применима к любому квадратному уравнению, все они могут быть решены одинаково, по одному алгоритму. Единственное ограничение — коэффициент a при второй степени неизвестного не должен обращаться в ноль. Но тогда это уже не квадратное, а линейное уравнение. Представьте себе, как трудно было бы делать расчеты, если бы для каждого квадратного уравнения пришлось бы придумывать свой метод решения! Впрочем, так приходится делать для большинства нелинейных уравнений, а если это не удается, применять приближенные методы нахождения их корней.

Кулинарные рецепты и инструкции по применению того или иного товара, как правило, не обладают массовостью, они составляются для каждого конкретного случая. Однако компьютерный алгоритм, разработанный для решения только одной частной задачи, чаще всего бесполезен, потому что за время, потраченное на его создание, кодирование и ввод в компьютер, можно было бы решить эту задачу вручную.

Разработчики алгоритмов стремятся расширить область их применения, сделать их как можно более массовыми. Идеально было бы найти универсальный алгоритм решения всех задач, но это, к счастью, невозможно, хотя бы потому, что есть алгоритмически неразрешимые классы задач. К счастью — ибо программисты никогда не останутся без работы.

С другой стороны, массовые алгоритмы, в силу своей общности, не учитывают конкретные особенности каждой решаемой ими задачи. Поэтому для особо важных задач приходится разрабатывать специальные алгоритмы, использующие все характеристики конкретной задачи. Этим объясняется великое разнообразие алгоритмов решения компьютерных задач.

Конечность

Алгоритм должен решить задачу за конечное число шагов. Мы не можем вечно ждать результата. Если же алгоритм по каким-то причинам не может решить задачу, он должен сообщить об этом и завершить работу. Несмотря на очевидность этого требования, иногда приходится прикладывать значительные усилия для того, чтобы предотвратить зацикливание алгоритма. Часто для этого нужно уменьшить массовость алгоритма, ограничив область его применения.

Иногда время решения задачи составляет несколько часов или даже несколько суток. В таком случае в алгоритм обычно вводят инструкции, которые время от времени передают сообщения пользователю, сигнализирующие о нормальной, как говорят, штатной работе алгоритма. Даже программа, работающая непрерывно всего несколько минут, как правило, выводит на экран графические индикаторы своего выполнения: песочные часы, "градусник", различные вращающиеся и бегущие предметы.

Однозначность

Действия, выполняемые алгоритмом, должны следовать одно за другим. После окончания каждого действия компьютер должен точно знать, что делать дальше, каково следующее предписание. Ему не скажешь: "Купи что-нибудь: колбасу или ветчину". Посылая компьютеризированного робота за покупками (почему бы нет?), надо дать ему точную, однозначную инструкцию: "Если колбаса свежее ветчины, купи колбасу, в противном случае купи ветчину". Даже такая подробность может оказаться недостаточной, если в магазине не окажется или, наоборот, встретится несколько сортов колбасы.

Требование однозначности обычно учитывается уже при разработке алгоритмического языка. Сами конструкции языка исключают неоднозначное их толкование и выполнение. Но на стадии обдумывания программы следует помнить об однозначности применяемого алгоритма. Часто в программе встречаются несколько вариантов продолжения работы алгоритма. В таком случае нужно тщательно продумать условия выбора одного и только одного варианта.

Исполнитель алгоритма

При разработке алгоритма обязательно надо четко представлять себе, кто или что, какой специалист, или какое устройство будет выполнять алгоритм, что оно умеет делать, какие действия может совершать, какие команды способно выполнять. Если алгоритм будет выполнять человек, например, повар будет готовить блюдо по рецепту, написанному вами, то следует учитывать его профессиональные навыки. Если алгоритм будет выполнять организация, какой-

нибудь коллектив, то надо знать опыт работы и направление деятельности организации. Если алгоритм предназначен для робота или компьютера, надо знать набор выполняемых ими операций. Часто при смене исполнителя приходится перерабатывать или даже заменять алгоритм.

Покажем это на примере нахождения наибольшего общего делителя (НОД) двух натуральных чисел. При ручном счете для нахождения НОД каждое число разлагается на простые множители, а затем перемножаются общие для обоих чисел множители, взятые в меньшей степени. На компьютере разложить число на множители труднее, поэтому применяется алгоритм Евклида. Познакомимся с ним.

Алгоритм Евклида

Даны два натуральных числа M и N. Пусть для определенности число M больше числа N. Для нахождения НОД(M, N) по алгоритму Евклида выполняются следующие действия:

1. Число M делится на число N. При этом получается какое-то частное Q_1 и какой-то остаток R_1.
2. Число N делится на число R_1 — на первый остаток. При этом получается частное Q_2 и второй остаток R_2.
3. Первый остаток R_1 делится на второй остаток R_2. При этом получается следующее частное Q_3 и следующий остаток R_3.
4. Деление повторяется, причем каждый раз остаток от предыдущего деления R_n делится на следующий остаток, R_{n+1}, $n = 1, 2, 3, \ldots$, до тех пор, пока остаток не обратится в ноль. Когда-нибудь это произойдет, потому что остатки R_n строго уменьшаются, оставаясь неотрицательными. Последний не равный нулю остаток или, что то же самое, последний делитель, и будет наибольшим общим делителем НОД(M, N).

Пусть, например, $M = 384$, а $N = 276$. Алгоритм Евклида даст такую последовательность вычислений:

$$384 = 276 \cdot 1 + 108,$$

$$276 = 108 \cdot 2 + 60,$$

$$108 = 60 \cdot 1 + 48,$$

$$60 = 48 \cdot 1 + 12,$$

$$48 = 12 \cdot 4 + 0.$$

Итак, НОД(384, 276) равен 12. Заметьте, что для алгоритма Евклида важны только остатки от деления, частные он не использует. Значит, он особенно удобен для исполнителя, умеющего вычислять остаток от деления двух целых чисел за одно действие.

Пусть теперь в нашем распоряжении оказался исполнитель, который не выполняет ни вычисление остатка, ни даже деление целых чисел, а только сложение и вычитание. Тогда алгоритм Евклида можно записать в таком виде:

1. Если число A равно числу B, то НОД(A, B) равен A и вычисления заканчиваются.
2. Если число A больше числа B, то надо применить алгоритм к числам $A - B$ и B.
3. Если число B больше числа A, то надо применить алгоритм к числам A и $B - A$.

Вот какая последовательность пар чисел будет получена по этому алгоритму для тех же чисел $M = 384$ и $N = 276$: (384, 276), (108, 276), (108, 168), (108, 60), (48, 60), (48, 12), (36, 12), (24, 12), (12, 12). Этот вариант алгоритма Евклида интересен тем, что в процессе работы он целиком выполняется несколько раз для разных исходных чисел. Такие алгоритмы называются рекурсивными (*см. гл. 5*).

Как видите, алгоритм Евклида принимает совершенно разные формы в зависимости от своего исполнителя. Понятие исполнителя настолько важно для программирования, что книга Кушниренко А. Г. и Лебедева Г. В. "Программирование для математиков" ставит его во главу написания программ.

Определение алгоритма

Теперь, после рассмотрения основных свойств алгоритма, можно сформулировать его определение. Оно не отличается строгостью, но пригодно для целей нашего курса. Алгоритм — это точное пошаговое описание в системе команд исполнителя последовательности действий, однозначно приводящей к результату за конечное число шагов. Это определение просто свело вместе отмеченные выше свойства. Оно отвечает не на вопрос: "что такое алгоритм", а на вопрос: "что делает алгоритм". В практическом программировании второй вопрос возникает гораздо чаще и ответ на него для программиста-практика гораздо важнее.

Способы записи алгоритма

Кулинарные рецепты и инструкции по применению различных товаров записываются на обычном языке. Как видно из приведенных выше примеров, такая запись часто понятна только специалистам, но специалисты и так знают, что им делать. Все мы знаем, как трудно объяснить словами что-либо непосвященному человеку. Слов часто не хватает, и мы сразу же начинаем активно жестикулировать, пытаемся имитировать некоторые действия или начертить какие-то поясняющие схемы.

Попробуем описать словами процесс нахождения корней квадратного уравнения. У нас получится примерно такая запись.

1. Возведите в квадрат коэффициент при первой степени неизвестного и вычтите из него учетверенное произведение двух других коэффициентов.
2. Если получился ноль, то корни уравнения будут совпадать и будут равны числу, противоположному результату деления коэффициента при первой степени неизвестного на удвоенный коэффициент при второй степени.
3. Если полученное на первом шаге число неотрицательно, то извлеките из него квадратный корень.
4. В этом случае для получения первого корня возьмите число, противоположное коэффициенту при первой степени неизвестного, вычтите из него полученный на предыдущем шаге корень и разделите результат на удвоенный коэффициент при второй степени.
5. Для получения второго корня выполните предыдущий шаг, заменив вычитание сложением.
6. Если же полученное на первом шаге число отрицательно, то извлеките квадратный корень из его абсолютной величины.
7. В этом случае корни будут комплексными числами. Действительная часть первого корня будет равна числу, противоположному результату деления коэффициента при первой степени неизвестного на удвоенный коэффициент при второй степени. Мнимая часть будет равна числу, противоположному результату деления корня, полученного на предыдущем шаге, на удвоенный коэффициент при второй степени.
8. Второй комплексный корень вычисляется аналогично предыдущему пункту, только у мнимой части будет противоположный знак.

Как видите, это описание не только гораздо длиннее формулы, но и менее понятно. Оно чересчур многословно, за потоком слов теряется смысл каждой фразы и ее приходится перечитывать несколько раз. Кроме того, описание даже такого простого вычисления не обходится без специальных терминов: коэффициент, степень, неизвестное. Что же говорить о более сложных областях науки и техники? Там всякое словесное описание пестрит специальными терминами, освоение которых требует немалых усилий.

Как же описывать алгоритмы? Математика выбрала крайний путь — она выработала чрезвычайно сложный язык обозначений, использующий специально придуманные символы. Для того чтобы привыкнуть к ним, требуется не один год упорных занятий, но зато квалифицированный математик одним взглядом на формулу может охватить сложнейшие соотношения, заключенные в ней.

В большинстве других областей науки и техники вырабатывается какой-нибудь свой язык описания алгоритмов, приближающийся либо к словес-

ному описанию, либо к кодированию малопонятными значками. Первый шаг на этом пути — создание псевдокода.

Псевдокод

В процессе обучения у каждого человека вырабатывается своя система обозначений. Все ученики делают сокращения в записях уроков и лекций, причем со временем эти сокращения принимают вид целой системы. Эту систему человек применяет всю дальнейшую жизнь, оттачивая ее до совершенства. Точно так же ученые, обдумывая очередную проблему, создают в ходе дискуссий и записей своеобразный язык, помогающий найти решение. Дадим такое промежуточное описание нахождения корней квадратного уравнения.

1. Дано квадратное уравнение $ax^2 + bx + c = 0$. Вычислим число $d = b^2 - 4ac$.
2. Если d == 0, то x1 = x2 = −c / b и вычисления заканчиваются.
3. Если d > 0, то полагаем d = sqrt(d).
4. x1 = 0.5(−b − d) / a.
5. x2 = 0.5(−b + d) / a. Вычисления закончены.
6. Если d < 0, то полагаем d = sqrt(fabs(d)).
7. x1 = −0.5b / a − d / ai.
8. x2 = −0.5b / a + d/ ai. Вычисления закончены.

Такое описание гораздо нагляднее, но требует некоторых знаний. На третьем шаге использована запись, типичная для языков программирования: переменная *d* присваивает значение квадратного корня, вычисляемое в правой части равенства. При этом ее значение меняется — если на первом шаге значение *d* было равно 9, то на четвертом оно будет равно 3. Тут же сделана функциональная запись квадратного корня sqrt(*d*). Функциональная запись и круглые скобки, окружающие аргумент функции, — вторая особенность языков программирования. В пункте 6 записана еще одна функция fabs(*d*), вычисляющая абсолютную величину действительного числа.

Поскольку знак равенства обозначает не равенство двух величин, а присваивание, для сравнения с нулем в пункте 2 пришлось записать два знака равенства. Такая запись используется во многих языках программирования.

Внимание!

Всюду в этой книге один знак равенства обозначает присваивание, а удвоенный знак равенства — сравнение двух известных величин.

В приведенном псевдокоде осталось совсем немного русских слов. Осталось записать условия по-английски:

```
if (d == 0) x1 = x2 = - c / b;
```

```
if (d > 0) d = sqrt(d);

if (d < 0) d = sqrt(fabs(d));
```

и получим почти готовый язык программирования. Попытавшись сделать такую же запись для алгоритма Евклида, мы столкнемся с неожиданным затруднением.

1. $M = NQ_1 + R_1$.
2. $N = R_1 Q_2 + R_2$.
3. R_1 = R2Q3 + R3.
4. Повторяем: $R_n = R_n{+}1 Q_n{+}2 + R_n{+}2$, n = 1, 2, ..., пока $R > 0$. Пусть $R_n{+}2 = 0$. Тогда НОД = R_{n+1}.

Затруднение заключается в том, что мы не знаем, сколько раз надо выполнить деление. Для описания этой неопределенности приходится изобретать неуклюжие словесные конструкции и обозначения. Попробуем улучшить это описание. Заметим, что последовательность делений обладает определенной регулярностью: на каждом следующем шаге бывший делитель становится делимым, а бывший остаток — делителем, делитель и остаток перемещаются влево на новые места, как показано на рис. 1.1.

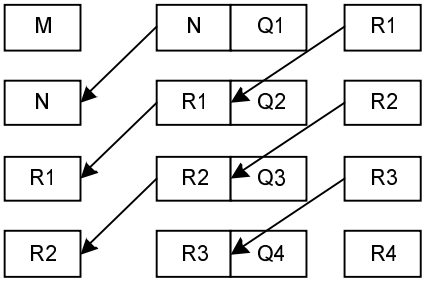


Рис. 1.1. Процесс делений в алгоритме Евклида

Заметив эту особенность, введем еще две переменные: обозначим буквой P делимое, а буквой Q — делитель, и будем менять их значения после каждого деления.

1. $P = M$, $Q = N$.
2. $R = P \% Q$.
3. Если $R = 0$, то полагаем НОД = Q и заканчиваем вычисления.
4. В противном случае полагаем $P = Q$, $Q = R$ и переходим к пункту 2.

Во втором пункте мы учли то, что для алгоритма Евклида не нужны частные, и обозначили знаком процента новую операцию — взятие остатка от деления целого числа, стоящего слева от знака процента, на целое число, стоящее справа.

Описание алгоритма приобрело, наконец-то, простоту и стройность, но ему не хватает наглядности. Гораздо большей наглядностью обладает следующий способ описания алгоритма — блок-схема.

Блок-схема

Блок-схема описывает алгоритмы с помощью небольшого числа простых графических символов.

- Овалом на блок-схеме отмечаются начало и конец алгоритма.
- Вычисления заносятся в прямоугольник.
- Ввод исходных данных и вывод результатов записываются в параллелограмм.
- Условия вида "если U то A; иначе B" выражаются ромбом, внутри которого записывается условие U. Из ромба выходят две стрелки. Одна стрелка, отмеченная плюсом, предназначена для действия A, происходящего при выполнении условия U, другая стрелка, отмечаемая минусом, — для действия B, которое происходит, если условие U не выполнено.

Вот и все обозначения. Несмотря на их минимальное количество, с помощью этих графических символов можно описать очень сложные алгоритмы. На рис. 1.2 показана блок-схема алгоритма Евклида. На ней хорошо виден весь процесс нахождения наибольшего общего делителя двух целых чисел: вход в алгоритм, последовательность вычислений и выход из алгоритма.

Отметим общепринятые требования к хорошо спроектированному алгоритму.

- У него только одно начало, как говорят, *точка входа* в алгоритм, и только один выход.
- В каждый блок, кроме условия, есть только один вход и один выход. Обычно, но не обязательно, вход в блок чертится сверху, а выход — снизу.
- У условия есть только один вход и ровно два выхода, поскольку условие может либо выполняться, либо не выполняться.
- Хорошо спроектированный алгоритм содержит все необходимые проверки, не оставляя исполнителю никакой возможности ошибиться.

Во исполнение последнего условия на рис. 1.2 на всякий случай взяты абсолютные величины чисел M и N, и проверено их соотношение, чтобы число P оказалось больше числа Q и потом большее число делилось на меньшее, а не наоборот.

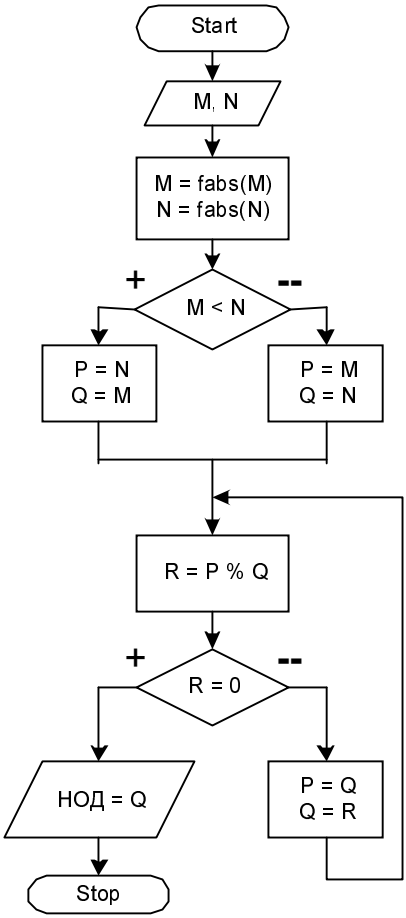


Рис. 1.2. Блок-схема алгоритма Евклида

Контрольный вопрос

Какую еще проверку необходимо сделать перед делением чисел?

Применение блок-схем

Блок-схемы наглядны и удобны для обсуждения алгоритма, но у программистов всегда было двоякое отношение к ним — с одной стороны, разработчика всегда просят показать блок-схему спроектированного алгоритма, а с другой стороны, никто не любит сам их чертить — это довольно скучное и трудоемкое занятие. Главная же причина, по которой блок-схемы сейчас почти не применяются, заключается в том, что при описании больших и

запутанных алгоритмов они сразу теряют свою наглядность. Слежение за хитросплетением линий и нагромождением блоков становится трудным, глаза разбегаются по листу бумаги и теряют ход вычислений.

Конечно, графическое представление очень удобно для человека и он никогда не откажется от блок-схем. Просто сейчас они заменяются структурными и потоковыми диаграммами. Эти диаграммы используются при проектировании крупных программных комплексов в так называемых CASE-системах.

Алгоритмические языки

Современные языки программирования хорошо приспособлены для описания алгоритмов. Первый язык, специально ориентированный на запись алгоритмов, — язык Algol — появился еще в 1960 г. Об этом говорит само его название, происходящее от сокращения слов Algorithmic Language (алгоритмический язык). Он широко применялся не только в программировании, но и в патентных заявках и даже в научных статьях. В середине 80-х годов прошлого столетия его заменил в этом качестве язык Pascal, на котором удобно оформлять алгоритмы в виде процедур и функций. Сейчас его заменяет язык Java, на котором удобно описывать не только алгоритмы, но и объекты.

За полвека развития программирования было создано несколько сотен языков программирования. Было время, когда чуть ли не каждый уважающий себя программист писал программы на собственном языке и переводил их в машинный код собственным компилятором. Конечно, такое богатство языков появилось не из-за амбиций программистов и не в результате вавилонского столпотворения. Новые языки создавались для решения новых задач и для более точного решения старых. Например, язык FORTRAN удобен для вычислений по формулам и широко применяется в этом качестве уже пятьдесят лет. На столь же древнем языке COBOL удобно делать экономические расчеты. Язык Perl предназначен для синтаксического анализа и грамматического разбора текстов. Командные языки помогают задать сложные алгоритмы работы операционной системы.

Время от времени программисты пытались уйти от специализации и создать универсальный язык программирования, пригодный для записи любых алгоритмов. Всякий раз такой универсальный язык оказывался слишком большим и трудным для изучения, а его компилятор — слишком громоздким и медленным. Кроме того, на универсальном языке не удавалось описывать особенности конкретных задач и создать эффективный машинный код. Вследствие этого универсальные языки програмирования вышли из употребления.

Структурные части алгоритма

Компьютер — очень мощное устройство с большими возможностями, но все его способности выявились не сразу. В первые годы своей истории компьютер, называемый тогда ЭВМ (Электронная Вычислительная Машина), был просто быстрой вычислительной машиной, заменившей логарифмическую линейку и арифмометр. Сначала он умел только находить численные решения дифференциальных уравнений и других сложных математических и механических задач, требующих длинных расчетов. Это дало толчок к развитию оптики, авиации, космонавтики и других областей науки и техники.

Потом компьютеры освоили обработку текстов, и их основной задачей стала работа с документами, что требует решения задач анализа текстов, их синтаксической проверки, перевода с одного языка на другой. Все больше и больше документов переводилось в электронную форму, возникли электронные таблицы для экономических расчетов, документы пересылались по компьютерным сетям, что значительно ускорило деловые отношения. В результате появились огромные базы данных, возник Интернет — всемирное хранилище информации. На первый план вышла задача поиска нужной информации в этой громадной хаотической куче сведений.

Затем появились звуковые карты, мощные видеокарты, возникло понятие мультимедиа, и неожиданно для своих создателей компьютеры превратились в средство развлечения. Скоро будет сделан следующий шаг — компьютер станет центром управления всей домашней бытовой техникой.

Все эти изменения прямо отражаются в алгоритмах решения соответствующих задач. От вычислительных алгоритмов перешли к алгоритмам обработки текстов, от них к алгоритмам поиска, потом стали актуальными алгоритмы сжатия, шифрования, отбора информации.

Современные программы — это большие приложения, использующие в своей работе сотни алгоритмов разной сложности. Но структура алгоритмов остается неизменной — все алгоритмы содержат линейные части, разветвления и циклы.

Линейный алгоритм

Предписания могут выполняться последовательно одно за другим как в рецепте приготовления манной каши, приведенном выше. Алгоритмы с такой последовательной структурой называются *линейными алгоритмами*. Линейные алгоритмы не характерны для компьютеров, поскольку написание всей последовательности действий займет слишком много времени по сравнению с их выполнением. Тем не менее, в каждом алгоритме обязательно есть линейные участки: вычисления формул, определения переменных, перечисления.

На блок-схеме каждая линейная часть алгоритма видна очень хорошо — это последовательность блоков, такая как на рис. 1.3. В старых языках программирования, таких как FORTRAN или BASIC, каждое вычисление записывалось на отдельной строке, что соответствовало последовательной записи блоков. Но запись получалась очень длинной, и в более новых алгоритмических языках решили разделять вычисления не переводом строки, а другим символом, обычно точкой с запятой. В таком случае можно записать каждую линейную часть алгоритма на одной строке. Запись становится компактной, но при этом плохо просматривается последовательность вычислений, которая часто бывает очень важной. Поэтому специалисты рекомендуют записывать каждое вычисление на отдельной строке, даже если язык позволяет все записать в одну строчку. Но иногда такая запись, наоборот, способствует ошибке.

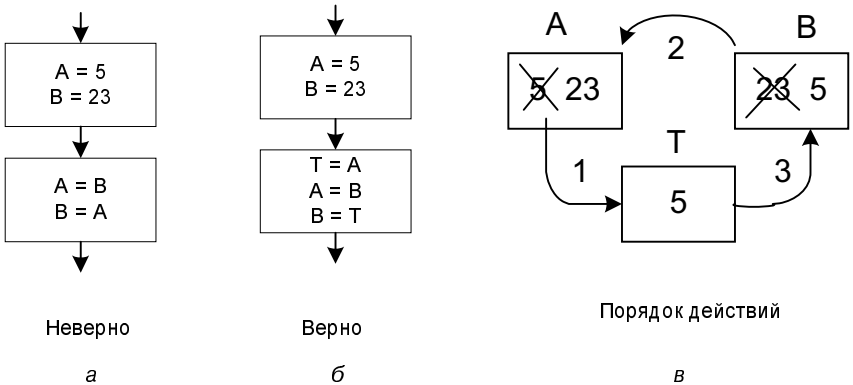


Рис. 1.3. Линейный алгоритм

Поясним это на простом примере. Пусть надо взаимно поменять значения двух переменных *A* и *B*. Скажем, было $A = 5$, $B = 23$, а надо сделать так, чтобы $A = 23$, $B = 5$. Простая последовательность действий $A = B$, $B = A$, показанная на рис. 1.3, *а*, приведет к тому, что обе переменные будут равны 23. Дело в том, что в ячейке памяти, выделяемой для переменной, может храниться только одно значение. При изменении значения переменной старое значение просто теряется. После первого действия переменная *A* уже будет равна 23. Поэтому старое значение надо предварительно сохранить в какой-нибудь временной переменной *T*, как сделано на рис. 1.3, *б*. Теперь перестановка значений будет выглядеть так: $T = A$, $A = B$, $B = T$.

При этом очень важна последовательность действий. Она показана на рис. 1.3, *в*. Попробуйте записать действия в другом порядке, и вы получите неверный результат. На блок-схеме или при записи действий на отдельных строках это не видно, а при линейной записи на одной строке мы приставляем равенства друг к другу как костяшки домино и никогда не ошибемся в последовательности присваиваний.

Разветвление

Выполнение почти всех алгоритмов связано с выбором одного из двух или нескольких вариантов. Вернемся к нахождению корней квадратного уравнения. Алгоритм, выраженный обычной формулой, действует только в том случае, когда коэффициент a не равен нулю. При обычных вычислениях мы не обращаем внимания на это условие, потому что если оно не выполнено, то мы имеем дело не с квадратным, а с линейным уравнением. Но квадратное уравнение может возникать в середине алгоритма, а его коэффициенты вычисляться по ходу выполнения алгоритма, и мы не можем быть уверены, что коэффициенты не равны нулю. Правильно спроектированный алгоритм должен учитывать все возможные ситуации. При $a = 0$ надо проверить коэффициент b, т. к. при решении линейного уравнения также выполняется деление. Если же оба коэффициента a и b равны нулю, то надо проверить еще и третий коэффициент c. Если c равен нулю, то уравнение вырождается в тождество $0 = 0$ и его решением по формальным правилам может служить любое число, в противном случае уравнение не выполнимо ни при каком значении неизвестного. Все эти рассуждения приводят к алгоритму, представленному на блок-схеме рис. 1.4.

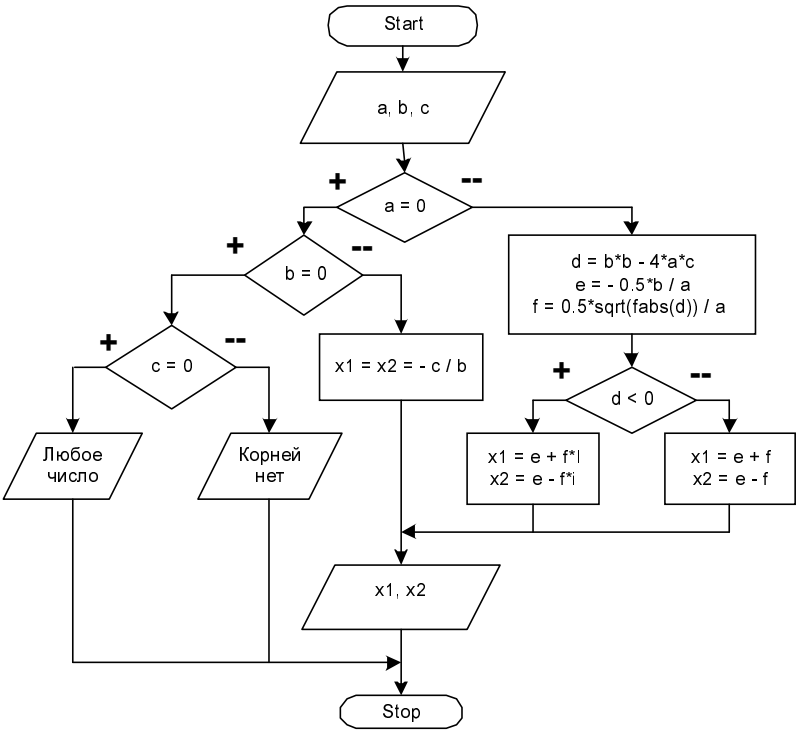


Рис. 1.4. Блок-схема нахождения корней квадратного уравнения

Даже в таком простом алгоритме сделаны четыре проверки. В настоящих производственных программах их еще больше. Очень часто программа на две трети состоит из проверок исходных данных, условий выполнения операций и проверок, сделанных для предотвращения критических ситуаций.

В разветвлении не обязательно должны выполняться обе ветви. Часто условие ставится так: "если выполнено условие *A*, то выполнять действие *P*". Здесь ничего не говорится о том, что делать, если условие не выполнено. Примеры таких разветвлений показаны на рис. 1.6.

С другой стороны, очень часто в алгоритме приходится сделать выбор из большого числа ситуаций, например, выбрать число месяца или день недели. Даже в решении квадратного уравнения можно отдельно обработать три ситуации: дискриминант отрицателен, положителен или равен нулю. Такой выбор можно сделать несколькими разветвлениями, записанными одно за другим, как показано на рис. 1.5, но это сильно усложняет алгоритм. В большинстве алгоритмических языков для записи такого множественного выбора имеются специальные операторы варианта.

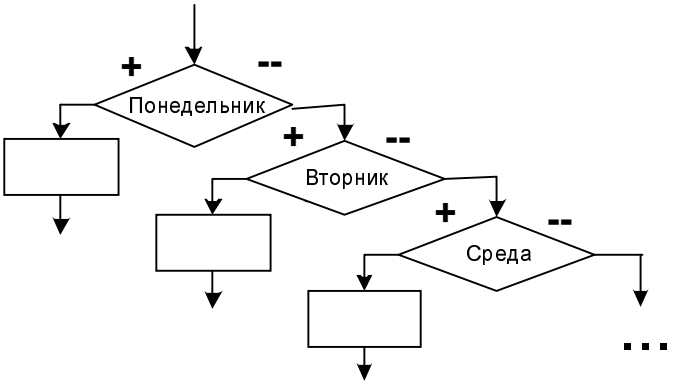


Рис. 1.5. Множественный выбор

Циклы

Основу любого алгоритма, предназначенного для компьютера, составляет циклическое повторение действий. Именно циклы позволяют выявить всю мощь компьютера. Мы записываем свои предписания только один раз, включаем их в цикл, а компьютер с огромной скоростью повторяет указанные действия тысячи раз.

Конечно, бессмысленно повторять абсолютно одинаковые действия. При каждом выполнении цикла, как говорят, при каждой *итерации*, обрабатываемые данные должны меняться. Кроме того, на каждой итерации следует

проверять условие продолжения или завершения цикла, чтобы предотвратить зацикливание. Это условие можно проверять перед началом итерации цикла, см. рис. 1.6, *а*, или в конце итерации, как показано на рис. 1.6, *б*.

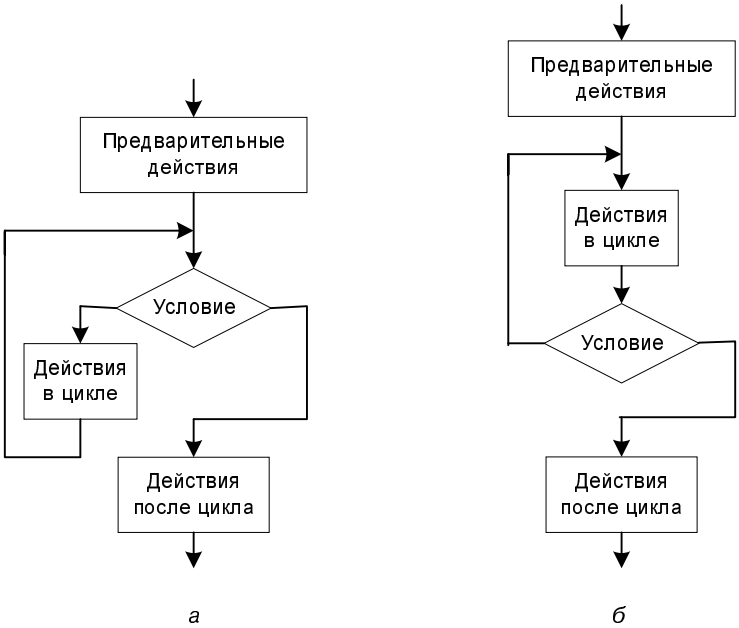


Рис. 1.6. Разновидности циклов

Во многих ситуациях безразлично, делать проверку в начале цикла или в его конце. Принципиальное различие между двумя разновидностями цикла заключается в том, что цикл с проверкой в конце обязательно будет выполнен хотя бы один раз, а цикл с проверкой в его начале может не выполниться ни разу, если условие его выполнения сразу же будет нарушено. Чтобы не получить зацикливание, лучше заранее тщательно проверить все условия, при которых будет выполняться цикл. Поэтому цикл с проверкой в начале считается в программировании основным и его следует предпочитать, если неважно, как написать цикл. Некоторые языки программирования содержат только один оператор цикла, выполняющий проверку в начале каждой итерации.

Во многих задачах заранее известно число итераций цикла, например, надо обработать список, содержащий ровно 1000 значений. В этом случае для перечисления итераций обычно определяется специальная переменная, называемая *счетчиком цикла*. Перед циклом она получает какое-то *начальное значение*. На каждой итерации она изменяется на какое-то постоянное или переменное значение, называемое *шагом* переменной цикла. Чаще всего шаг равен 1. Цикл завершается, когда переменная цикла примет свое *конечное*

значение. Если обозначить переменную цикла буквой *i*, ее начальное значение буквой *M*, шаг ее изменения буквой *H*, а конечное значение — буквой *N*, причем $M < N$, то цикл с известным заранее числом повторений можно записать так, как показано на рис. 1.7. Ввиду большой распространенности таких циклов во многих языках программирования есть специальный оператор для записи циклов с известным числом повторений.

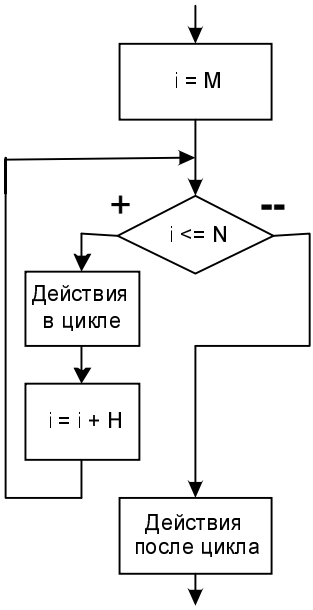


Рис. 1.7. Цикл с известным числом повторений

Упражнения

1. Начертите блок-схему цикла с известным числом повторений, в котором начальное значение больше конечного.
2. Начертите блок-схему цикла с известным числом повторений с проверкой условия в конце итерации.

Стандартные алгоритмы

На протяжении всей книги мы будем рассматривать самые разные алгоритмы, наиболее часто применяемые в программировании. Многие из этих алгоритмов признаны лучшими способами решения своих задач, для некоторых алгоритмов это строго доказано. Другие алгоритмы, может быть, и не

лучшие, но общепризнанны и стандартно применяются для решения практических задач. В багаже каждого программиста должен быть набор таких алгоритмов.

Действия с целыми числами

Задачи, возникающие при обработке целых чисел, обычно требуют выделения и анализа отдельных цифр числа. Целые числа хранятся в оперативной памяти компьютера в двоичной форме и обычно занимают два, четыре или восемь байтов. В такой записи десятичные цифры никак не выделяются, их приходится вычислять арифметическими операциями.

Рассмотрим типичную задачу. Дано целое число M, скажем, $M = 12\ 345$. Надо вычислить число N, цифры которого идут в обратном порядке, в нашем примере получится $N = 54\ 321$. Для решения применим такой алгоритм. Будем последовательно, в цикле, выделять цифры младших разрядов числа M и ставить их в старшие разряды числа N. Это только идея, которую надо как-то реализовать.

Для реализации этой идеи надо уметь выделять последние цифры целого числа M, а затем сделать эти цифры коэффициентами при старших степенях числа N. Выделить самую последнюю цифру D просто — это остаток от деления числа на 10. Другие цифры выделить нелегко, но после того как мы уже получили последнюю цифру D, можно разделить число на 10 нацело и следующая справа цифра окажется последней. Получаем такие действия: $D = M \% 10$, $M = M / 10$.

В этой записи мы опять воспользовались знаком процента для обозначения операции взятия остатка от деления двух целых чисел. Кроме того, мы воспользовались наклонной чертой для записи операции деления. Но это не все. Мы использовали специфическое свойство языков программирования, заключающееся в том, что результат арифметической операции с целыми числами снова будет целым числом. Для сложения, вычитания и умножения это всегда выполнено, а вот при делении дробная часть просто отбрасывается. Такой непривычный результат как раз и нужен для нашей цели — сделать следующую цифру последней цифрой числа. После выполнения этих действий в нашем примере получим в первый раз: $D = 5$, $M = 1234$, во второй раз: $D = 4$, $M = 123$ и т. д.

Вторая часть алгоритма — сделать выделяемые цифры старшими цифрами формируемого числа N. Для этого их надо умножать на соответствующие степени числа 10. В нашем примере число 5 надо умножить на 10^4, число 4 — на 10^3 и т. д. Это сложно сделать хотя бы потому, что надо заранее знать степень, в которую возводится каждое следующее число. Конечно, можно подсчитать количество цифр в первоначальном числе M, но это отдельная задача такой же сложности, какую мы сейчас решаем.

Поступим проще: будем умножать формируемое число N в каждой итерации на 10, и нужная степень накопится сама собой: $N = N \cdot 10$, $N = N + D$. Эти две операции можно совместить: $N = N \cdot 10 + D$. Здесь, как принято в языках программирования, мы обозначили умножение звездочкой. В нашем примере сначала получаем $N = 5$, на втором шаге $N = 50 + 4$, на третьем шаге $N = 540 + 3$ и т. д.

Итак, идея реализована. Остается дать начальное значение переменной $N = 0$, задать условие окончания цикла и можно записать алгоритм. Его блок-схема показана на рис. 1.8.

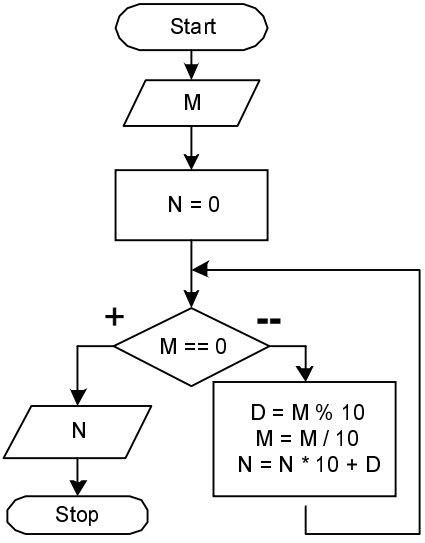


Рис. 1.8. Перестановка цифр целого числа

После просмотра этого алгоритма остается какая-то неуверенность в его правильности. Проверим алгоритм на нашем примере $M = 12\ 345$, проделав все действия вместо машины, сделав так называемую *прокрутку*. Сведем вычисления в таблицу (табл. 1.1).

Таблица 1.1. *Прокрутка алгоритма перестановки цифр целого числа*

Номер итерации	D	M	N
	–	12 345	0
1	5	1234	5
2	4	123	54

Таблица 1.1 *(окончание)*

Номер итерации	D	M	N
3	3	12	543
4	2	1	5432
5	1	0	54 321

Для нашего примера алгоритм работает правильно. Возникает некоторая уверенность, но неплохо проверить алгоритм для отрицательного числа, нуля, единицы и еще каких-нибудь значений.

С помощью прокрутки можно выявить ошибки, но доказать правильность алгоритма нельзя. Впрочем, это сложная тема (*см. гл. 21*), а пока отметим только полезность прокрутки не для доказательства правильности алгоритма, а для выявления ошибок. При этом особое внимание следует обратить на правильность начала цикла и его окончания.

Упражнения

3. Найдите алгоритм, подсчитывающий количество цифр заданного целого числа.
4. Найдите алгоритм, дающий сумму цифр заданного целого числа.
5. Целое шестизначное число считается "счастливым", если сумма трех его старших цифр равна сумме трех младших цифр. Создайте алгоритм, определяющий, будет заданное целое число "счастливым" или нет.

Суммирование и умножение

Суммирование и умножение большого количества чисел — старая и почетная задача программирования, и первая работа, выполненная компьютером. Алгоритм суммирования очень прост. Пусть надо просуммировать N чисел a_1, a_2,, a_N. Выделяем для хранения суммы отдельную переменную, назовем ее S. Сначала полагаем $S = 0$. Затем перебираем слагаемые в цикле, на каждой итерации прибавляя к сумме следующее слагаемое. Можно представлять себе переменную S как ящик, в который на каждой итерации добавляется следующее слагаемое. Блок-схема алгоритма представлена на рис. 1.9.

Обратите внимание на две важные вещи. Во-первых, переменная S получила начальное значение 0. Если этого не сделать, то результат будет непредсказуем. Дело в том, что при объявлении переменной S ей выделяется ячейка оперативной памяти. Эта ячейка не очищается, в ней находится какая-то величина, оставшаяся от предыдущих вычислений, как говорят, "мусор". Если не задать начальное значение переменной, то к этому "мусору" будут

прибавляться все слагаемые. Чтобы избежать этого, мы *обнулили* переменную, хотя можно было дать ей другое значение, например, $S = a[1]$, а цикл начинать со второго слагаемого.

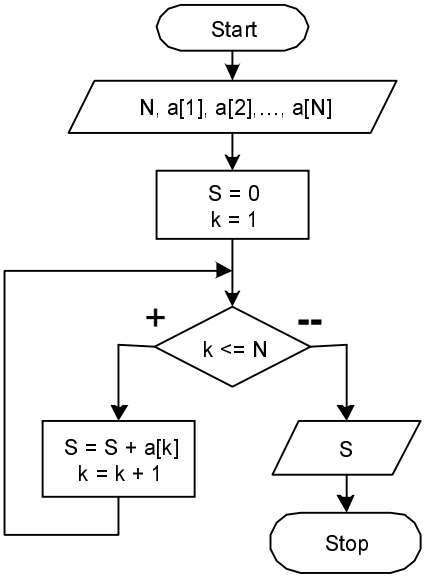


Рис. 1.9. Блок-схема суммирования чисел

Во-вторых, программа, читающая алгоритм, не умеет читать подстрочные и надстрочные индексы, поэтому мы записали номера слагаемых в квадратных скобках. Почему скобки квадратные? Да просто потому, что в круглых скобках записываются аргументы функции, и если мы используем $a(1)$, то это будет воспринято как вызов функции a с аргументом 1, а поскольку такой функции нет, программа сделает сообщение об ошибочном использовании несуществующей функции.

Произведение чисел вычисляется аналогично, только начальное значение переменной P, выделенной для хранения произведения, нельзя сделать нулевым. Его можно положить равным 1 или первому сомножителю.

Алгоритм умножения можно улучшить. Ведь если хотя бы один сомножитель равен нулю, то вычисления можно прекратить, т. к. произведение будет равно нулю. Эта особенность учтена на блок-схеме рис. 1.10.

Здесь введено новое обозначение. Двумя символами "!=" записано неравенство, как это делается во многих языках программирования.

При умножении чисел может возникнуть неприятность, вызванная тем, что произведение больших чисел будет очень большим числом, а у каждого компьютера есть наибольшая величина, при превышении которой происхо-

дит так называемое *переполнение*. Оно заключается в том, что компьютер не может хранить такое большое число, программа перемножения сообщает о переполнении и немедленно завершает свою работу. Хорошо спроектированный алгоритм должен учитывать возможность переполнения. Как же это сделать?

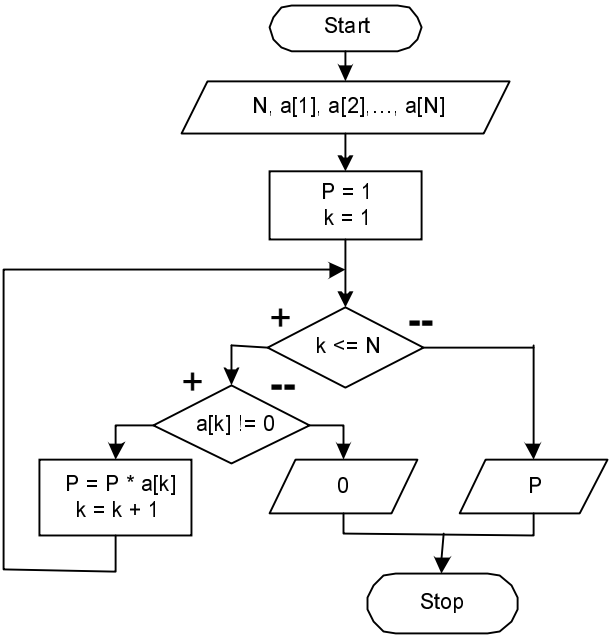


Рис. 1.10. Блок-схема умножения чисел

Среди большого количества чисел всегда найдутся и маленькие и большие числа. Если умножить сначала самое большое число на самое маленькое, потом перемножить следующее по величине большое и маленькое число и действовать до конца в том же духе, то можно будет избежать переполнения. При таком умножении надо учесть, что количество сомножителей может быть нечетным и тогда надо сделать дополнительное умножение на среднее число. На рис. 1.11 показан измененный алгоритм.

Обратите внимание на то, как проверяется четность или нечетность числа N — остаток от его деления на 2 будет нулевым или ненулевым.

Теперь алгоритм учитывает возможность переполнения, но он стал значительно сложнее, ведь сомножители надо предварительно расположить в возрастающем порядке. Сортировка по возрастанию большой последовательности чисел — задача гораздо более сложная, чем их умножение, и надо хорошенько подумать, получили мы какой-нибудь выигрыш или нет.

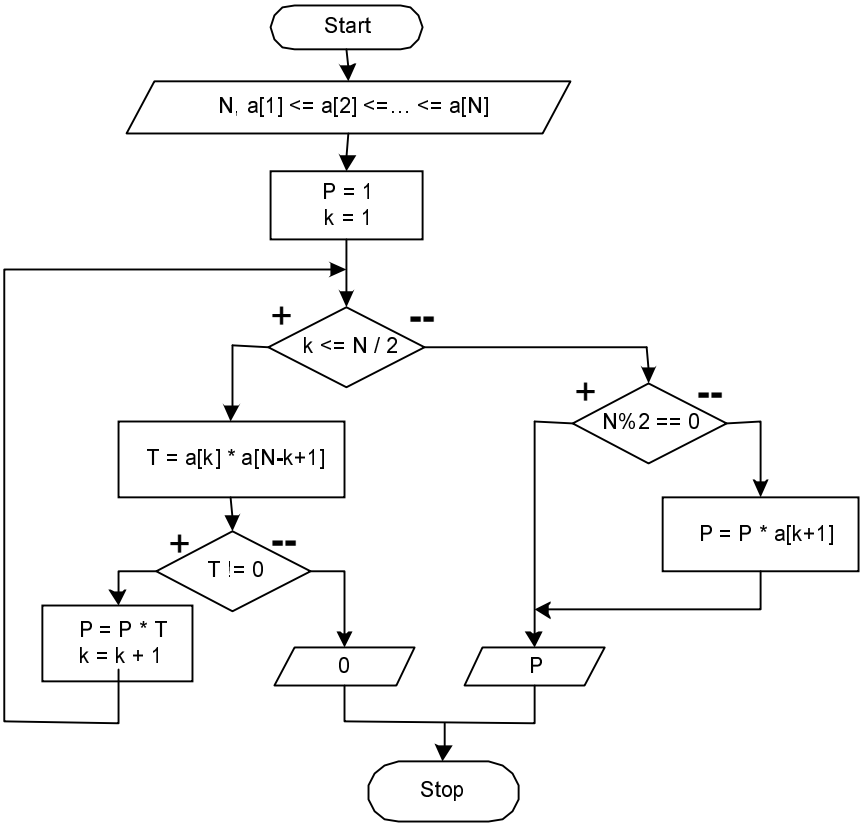


Рис. 1.11. Улучшенная блок-схема умножения чисел

Упражнения

6. Найдите алгоритм, подсчитывающий сумму квадратов первых N натуральных чисел.
7. Найдите алгоритм, суммирующий величины, обратные первым N натуральным числам.
8. Как будет выглядеть алгоритм, вычисляющий сумму величин, обратных факториалам первых N натуральных чисел?

Вычисление многочлена по схеме Горнера

Как мы обычно находим значение заданного многочлена $P = a_0 x^n + a_1 x^{n-1} + \ldots + a_n$ в заданной точке x_0? Возводим число x_0 в степень n, умножаем на коэффициент a_0, возводим число x_0 в степень $n-1$, умножаем на коэффициент a_1, прибавляем к предыдущему результа-

ту, возводим, умножаем, прибавляем. Оказывается, все можно сделать гораздо проще, если воспользоваться схемой Горнера.

Схема Горнера основана на том, что многочлен можно записать в таком виде: $P = ((a_0 x + a_1)x + a_2)x + a_3)x + ... + a_n$. Действительно, раскрыв скобки, вы получите обычную запись. По схеме Горнера мы начинаем вычислять многочлен в точке x_0 с внутренних скобок, и видим, что вычисление сводится к однообразным действиям: умножаем коэффициент a_0 на x_0, прибавляем следующий коэффициент a_1, результат умножаем на x_0, прибавляем следующий коэффициент a_2 и т. д. Получаем такие действия, похожие на обычное суммирование: $P = a_0$, $P = P \cdot x_0 + a_k$, где $k = 1, 2, ..., n$. Полностью схема Горнера показана на рис. 1.12.

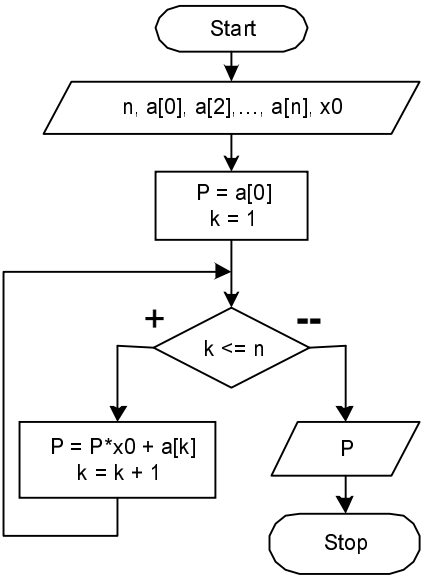


Рис. 1.12. Схема Горнера вычисления многочлена

В теории сложности доказано, что схема Горнера — самый экономичный алгоритм, он требует меньше всего умножений и совсем не использует возведение в степень.

Работа с текстом

Современные компьютеры уже трудно назвать вычислительными машинами — бóльшую часть времени они занимаются не вычислениями, а обработкой

различных документов: редактированием, переводом, поиском каких-то фрагментов текста. При этом текст представляется последовательностью символов, из которых он состоит. Все символы нумеруются, начиная с нуля, включая и пробелы и "невидимые" символы, такие как знак табуляции, символ перевода строки и другие управляющие символы, которые могут встретиться в документе. Мы всегда можем обратиться к произвольному символу по его порядковому номеру, который называют *индексом* символа в последовательности.

Рассмотрим типичную задачу — подсчитать количество слов в заданном тексте. Первый вопрос, который возникает при обдумывании решения задачи: что считать словом? После некоторого раздумья ответим на него так: это последовательность символов, не содержащая символов — разделителей слов. Но какие символы разделяют слова? Это конечно пробел, символ табуляции, обозначаемый HT (Horizontal Tabulation), символ перевода строки LF (Line Feed), символ возврата каретки CR (Carriage Return). Будут ли разделителями слов запятая, точка, двоеточие, точка с запятой, восклицательный знак, вопросительный знак? Вопрос спорный. По правилам русского языка после этих символов надо оставлять пробел, но, к сожалению, это часто нарушается. Будет ли разделителем дефис? Конечно, "какой-то" — это одно слово, а как насчет выражения "водитель-экспедитор"?

Не будем вникать в тонкости грамматики. Договоримся, что символы, разделяющие слова, так называемые *пробельные символы* — это пробел, символ горизонтальной табуляции HT, символ перевода строки LF и символ возврата каретки CR. Эта договоренность ограничивает массовость нашего алгоритма, но упрощает его.

Казалось бы, достаточно подсчитать количество пробельных символов, оставленных между словами, прибавить к нему единицу и мы получим количество слов в тексте. Сложность заключается в том, что между словами может оказаться несколько пробелов или пробел и символ табуляции и так может получиться любое сочетание любого количества пробельных символов.

Тогда поступим так: будем подсчитывать количество ситуаций "конец слова". Такая ситуация возникает, когда после непробельного символа сразу же идет пробельный символ. На псевдокоде ее можно описать так: "Если k-й символ $S[k]$ текста S не пробельный, а $(k+1)$-й символ $S[k+1]$ — пробельный, то увеличиваем количество слов на единицу". Но такой алгоритм может упустить последнее слово, т. к. после него пробельных символов может не оказаться. Алгоритм придется дополнить проверкой: "если последний символ $S[n-1]$ не пробельный, то увеличиваем количество слов на единицу".

Полученный алгоритм показан на рис. 1.13. В нем количество символов — длину текста S — вычисляет функция strlen(S). Поскольку символы нумеруются, начиная с нуля, индекс последнего символа равен $n-1$. Пробельные

символы на блок-схеме обозначены буквой *B*. Как делается сложное сравнение с каждым из пробельных символов, мы узнаем в следующей главе. Там же разъяснен смысл удвоенного амперсанда "&&", использованного для объединения двух проверок.

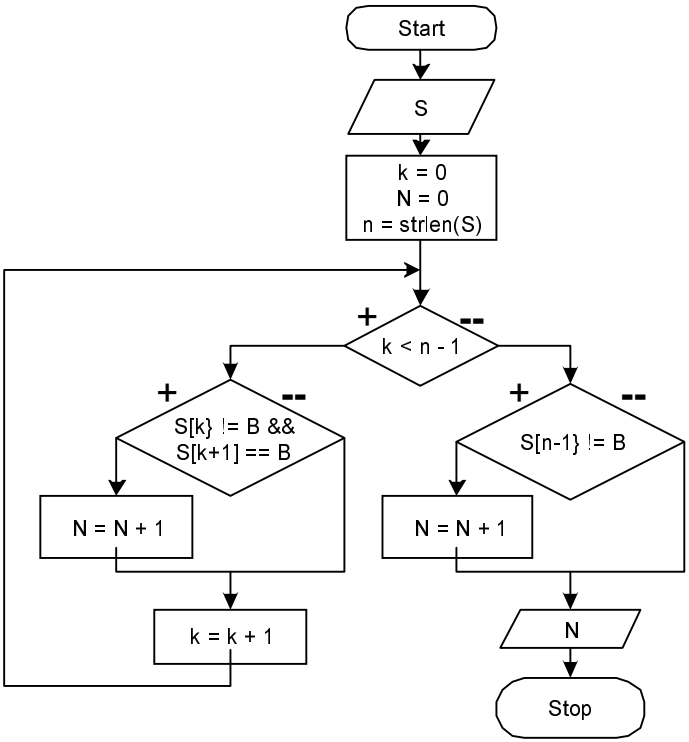


Рис. 1.13. Подсчет количества слов в тексте

Алгоритм, отмечающий ситуацию "конец слова", несовершенен. Ему требуется дополнительная проверка окончания текста. Конечно, можно отмечать ситуацию "начало слова", но тогда придется проверять начало текста. Давайте проверять ситуацию "внутри слова".

Введем переменную *F*, которая будет принимать значение true — истина, если *k*-й символ находится внутри слова, и значение false — ложь, если *k*-й символ — один из пробельных символов. Как только слово заканчивается или начинается новое слово, переменная *F* меняет свое значение. Это отражено на блок-схеме алгоритма, начерченной на рис. 1.14.

На рис. 1.14 введено еще одно обозначение. Знаком "++" обозначено увеличение переменной на 1. Вместо $N = N + 1$ мы пишем N++. Такая операция введена во многие языки программирования потому, что она выполняется всего одной машинной операцией.

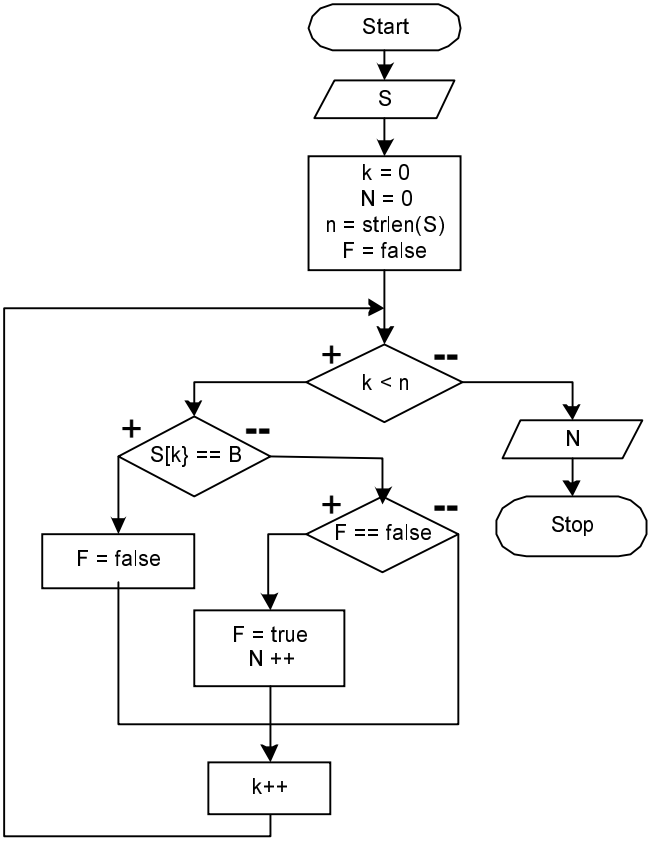


Рис. 1.14. Подсчет количества слов в тексте

Упражнения

9. Запишите алгоритм, использующий ситуацию "начало слова".
10. Запишите алгоритм, использующий ситуацию "вне слова".

Перевод десятичного числа в двоичную форму

Человек пользуется десятичной системой счисления, для компьютера, собранного из электронных элементов, принимающих два значения, "родной" является двоичная система счисления. В десятичной системе счисления каждое количество записывается в виде многочлена по степеням 10. Например, количество триста сорок шесть целых двадцать пять сотых записывается так:

$$3 \cdot 10^2 + 4 \cdot 10^1 + 6 \cdot 10^0 + 2 \cdot 10^{-1} + 5 \cdot 10^{-2}.$$

Мы настолько к этому привыкли, что выписываем только коэффициенты этого многочлена, отделяя целую часть числа от дробной части точкой (в программировании принята точка, а не запятая): 346.25. Если какая-то степень многочлена отсутствует, то в записи числа на ее месте появляется ноль. Для компьютера гораздо удобнее запись по степеням числа 2. В такой записи то же количество будет записано так:

$$1 \cdot 2^8 + 1 \cdot 2^6 + 1 \cdot 10^4 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^{-2}.$$

Вычислите записанный выше многочлен, и вы получите десятичную запись числа. Полная запись многочлена с нулевыми коэффициентами будет выглядеть следующим образом:

$$1 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 10^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + \\ + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}.$$

Поэтому двоичная запись нашего числа 246.25 выглядит так: 101011010.01. Итак, для перевода десятичного числа в двоичную форму надо записать его в виде многочлена по степеням числа 2. Коэффициентами этого многочлена будут служить остатки от деления десятичного числа на 2.

Упражнение

11. Убедитесь в том, что цифры двоичного числа — это остатки от последовательного деления исходного десятичного числа на 2.

Мы уже умеем находить остатки от деления, подобные действия показаны на блок-схеме рис. 1.8. Надо только заменить делитель 10 на делитель 2. Задумаемся теперь над представлением результата. Алгоритм рис. 1.8 из одного целого числа M получает другое целое число N. Если поступить сейчас так же, то у полученного двоичного числа окажется слишком много цифр, чтобы хранить его в виде целого числа. Компьютер вместо ожидаемого результата даст нам сообщение о переполнении.

Поэтому будем хранить двоичное число в виде последовательности символов — цифр 0 и 1. Вместо операции $N = N \cdot 10 + D$, показанной на рис. 1.8, запишем так называемую *операцию сцепления*. Она заключается в том, что к одной строке символов приписывается другая последовательность символов без пробелов и без всяких других разделителей. Во многих языках программирования операция сцепления записывается плюсом или удвоенной вертикальной чертой. Получаем алгоритм перевода целого действительного числа в двоичную форму, представленный на рис. 1.15.

В этом алгоритме N — строка символов. Вначале она должна быть пуста, не содержать ни одного символа. Это обеспечивается записью $N =$ "" — парой кавычек, между которыми ничего нет, даже пробела. Ведь пробел — это тоже символ! В конце блока вычислений запись $N = D + N$ дает сцепление очередного остатка D с предыдущей строкой N, состоящей из нулей и

единиц. Обратите внимание на то, что новый остаток приписывается слева, сцепление вовсе не коммутативно, хотя и обозначается плюсом.

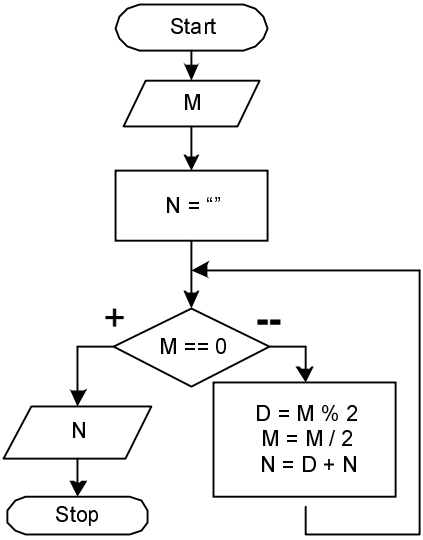


Рис. 1.15. Перевод целого числа в двоичную форму

Алгоритм рис. 1.15 работает только с целыми числами. Дроби представляются коэффициентами при отрицательных степенях числа 2. Значит, для перевода в двоичную форму дробей вместо деления на 2 надо выполнять умножение на 2, а вместо остатка надо брать целую часть полученного после умножения числа. При этом вычисления заканчиваются, когда в результате получится 1. Этого чаще всего не происходит, двоичная дробь получается бесконечной и вычисления надо обрывать на каком-то двоичном знаке.

Упражнение

12. Запишите алгоритм, переводящий правильные десятичные дроби в двоичную форму.

Контрольные вопросы

1. Как вы можете сформулировать понятие "алгоритм"?
2. Как меняется алгоритм в зависимости от его исполнителя?

3. Каковы главные свойства алгоритма, предназначенного для компьютера?
4. Какие составные части алгоритма вы можете перечислить?
5. Как можно записать разветвление на три ветви?
6. Как удобнее реализовать цикл с известным числом повторений: сделать проверку до или после цикла?
7. Приведите примеры зацикливания.
8. Как проверить с помощью алгоритма Евклида взаимную простоту двух целых чисел?
9. Что надо изменить в алгоритме перевода чисел в двоичную систему, чтобы переводить числа в восьмеричную систему?

Глава 2

Простые типы данных

Работа любой, простой или сложной, программы заключается в обработке исходных данных. Блок-схема каждой программы в самом общем виде выглядит так, как показано на рис. 2.1.

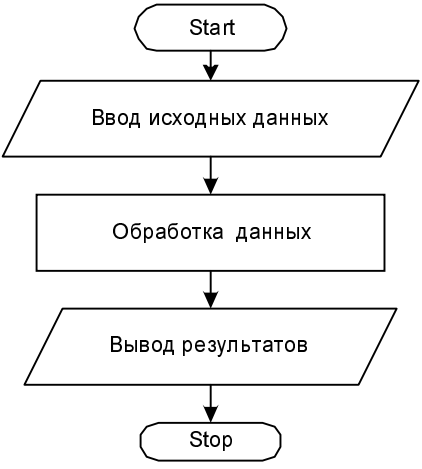


Рис. 2.1. Блок-схема программы в общем виде

Конечно, данные могут поступать на обработку в процессе выполнения программы, их количество может меняться в зависимости от условий, заложенных в алгоритм обработки, но пока для нас важно только одно — обработка информации сильно зависит от формы представления исходных данных. Поэтому очень важно знать, как информация хранится в оперативной памяти компьютера и как она описывается выбранным вами языком программирования.

Какую же информацию надо хранить и обрабатывать? В первую очередь это числа, отдельные символы, строки символов, машинные команды, адреса оперативной памяти. Кроме того, надо хранить звуковые и видеофайлы, а также различную сжатую, зашифрованную и заархивированную информацию. По-своему хранят информацию системы управления базами данных, редакторы текстов. Короче говоря, у многих программ есть свой формат записи данных.

В этой главе мы будем иметь дело только с самыми простыми исходными данными: числами, символами и перечислимыми типами. Подробно рассмотрим способы их хранения и работу с ними.

Числа и символы хранятся в оперативной памяти совершенно по-разному. Более того, отличается запись целых и действительных чисел, например, для компьютера 5 и 5.0 не равны друг другу, т. к. относятся к разным *типам данных*. В понятие типа данных входит диапазон значений, принимаемых данными этого типа, и набор операций, выполняемых с этими значениями.

Посмотрим внимательнее, какие простые типы данных обычно хранятся и обрабатываются в компьютере, и какие операции с ними можно выполнять.

Числовые типы данных

Как уже было сказано выше, целые и вещественные числа — это разные типы данных, отличающиеся величиной числа и точностью его записи в оперативной памяти компьютера.

Целые типы

Как правило, целые числа хранятся в компьютере в двоичной форме. Разумеется, при вводе мы записываем их в привычной для нас десятичной системе счисления, но компьютер сразу же переводит его в двоичную систему по алгоритму, подобному тому, что изображен на рис. 1.15. При этом многие вычислительные системы соблюдают еще три правила.

Во-первых, для хранения целого числа выделяется фиксированное число байтов: один, два, четыре или восемь. Целое число не может занять три или пять байтов. Соответственно получается четыре типа целых чисел: `byte`, `short`, `int`, `long`.

Во-вторых, один, старший, двоичный разряд отводится под обозначение знака числа. Ноль в старшем разряде означает, что хранится неотрицательное число, единица означает, что число отрицательное. Таким образом, для хранения двоичных цифр числа остается 7, 15, 31 или 63 бита.

В-третьих, у отрицательных чисел хранятся не их двоичные цифры, а хранится *дополнительный код* отрицательного целого числа. Остановимся на этом подробнее. Дополнительный код двоичного числа получается так: двоичные единицы заменяются нулями, нули — единицами, а затем к получен-

ному двоичному числу прибавляется 1. Прибавление происходит по правилам двоичной арифметики, по которым 1 + 1 = 10, т. е. происходит перенос двоичной единицы в старший разряд. Например, двоичная запись числа −94 равна −1011110. Дополнительным кодом числа 1011110 будет число 0100001 + 1 = 0100010. Добавив слева 1 в знаковый разряд, получим 10100010. Именно эту комбинацию нулей и единиц создаст компьютер в оперативной памяти, когда вы введете −94 и укажете, что это число типа `byte`.

Обратите внимание на то, что единица в знаковом разряде возникает автоматически при переводе неотрицательного числа в дополнительный код. Если мы сразу сделаем двоичную запись числа 94 в восьми разрядах: 01011110, то, переведя ее в дополнительный код, получим все восемь разрядов дополнительного кода.

Зачем нужна дополнительная операция по переводу двоичной записи в дополнительный код? Давайте сложим число 94 с его дополнительным кодом по правилам двоичной арифметики: 01011110 + 10100010 = 100000000. Сложение выполняем поразрядно справа налево, выполняя перенос единицы в старший разряд после сложения единиц. Получили единицу с восьмью нулями. Но единицу негде хранить, ведь для числа выделен только один байт! Она пропадает, мы получаем в результате сложения ноль. Итак, в этой странной арифметике сложение числа D со своим дополнительным кодом D' дает ноль. Дополнительный код D' оказывается числом, противоположным данному числу D, т. е. $D' = -D$. Раз так, то:

$$C - D = C + D'.$$

Операция вычитания заменяется операцией сложения с дополнительным кодом вычитаемого, в процессоре не нужно устройство для вычитания, достаточно сумматора и инвертора для получения дополнительного кода. Так и сделано во многих процессорах.

С учетом этих трех правил получаем диапазоны каждого целого типа, представленные в табл. 2.1.

Таблица 2.1. *Целые типы*

Тип	Разрядность	Диапазон
byte	1 байт	от –128 до 127
short	2 байта	от –32 768 до 32767
int	4 байта	от –2 147 483 648 до 2 147 483 647
long	8 байтов	от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807

Наличие всех четырех типов или отсутствие каких-либо из них зависит от языка программирования. Кроме того, такая строгая типизация характерна для современных языков программирования.

В языке C, созданном более 30 лет назад, тип `byte` отсутствует, т. к. его во многом заменяет символьный тип `char`. Кроме того, разрядность основного целого типа `int` в языке C не фиксируется. Она совпадает с величиной машинного слова. Сейчас компьютеры в основном 32-разрядные и числа типа `int` занимают 4 байта. На 64-разрядных машинах тип `int` составляет 8 байтов. Два других целых типа получаются как *модифицикация* типа `int`. Это подчеркивается полным названием типа, состоящим из модификатора `short` или `long` и имени типа `int`, при этом слово `int` можно не писать.

Тип `short int` или просто `short` занимает или половину длины типа `int`, или столько же байтов, сколько занимает тип `int`. Тип `long int` или просто `long` занимает в два раза больше места в оперативной памяти, чем тип `int`, или столько же места. Многие компиляторы языка C реализуют только два целых типа: обязательно тип `int` и дополнительно один из типов `short` или `long`.

У языка C есть еще одна особенность. Он вводит беззнаковые целые типы. Типы `unsigned short`, `unsigned int`, `unsigned long` выделяют все биты под двоичные цифры числа. Знак числа не хранится, число считается неотрицательным. Это позволяет увеличить в два раза максимальное значение типа. Но главное отличие беззнаковых типов то, что для них не бывает переполнения, все вычисления выполняются по модулю 2^n, где n — разрядность типа. Вместо словосочетания `unsigned int` можно писать просто `unsigned`.

Вещественные типы

Вещественные числа хранятся в оперативной памяти в форме с плавающей точкой. Эта форма записи вида $0.35 \cdot 10^{12}$ по-другому называется "научной". Коэффициент при 10 называется мантиссой, степень числа 10 — порядком числа. В компьютере мантисса записывается в двоичной форме и применяется степень числа 2, а не 10. При этом хранятся только знак числа, мантисса и характеристика числа, число 2 не хранится и никакого умножения на степень двойки не выполняется. Поскольку запись с плавающей точкой неоднозначна, $0.35 \cdot 10^{12} = 0.035 \cdot 10^{13} = 3.5 \cdot 10^{11}$, договорились выбрать нормализованную запись, при которой целая часть мантиссы равна нулю, а дробная часть начинается с 1.

У нормализованной мантиссы хранятся только двоичные цифры дробной части. Поскольку вследствие нормализации в такой записи старший бит всегда равен 1, он не хранится. Это так называемый *скрытый бит*. Скрытый бит всегда автоматически добавляется к мантиссе при извлечении ее из оперативной памяти.

Двоичный порядок числа, т. е. степень числа 2, преобразуется в *характеристику* числа. Характеристика получается прибавлением к порядку числа 127 или 1023 в зависимости от длины типа. Зачем это нужно? Порядок числа может быть положительным или отрицательным, значит надо где-то хранить знак порядка. Вместо этого решили сдвинуть порядок вправо по числовой оси в неотрицательную область. Характеристика вещественного числа — это всегда неотрицательное целое число.

Эти и другие правила описываются стандартом IEEE 754, принятым в большинстве языков программирования. По стандарту IEEE 754 выделяются три вещественных типа (см. табл. 2.2).

Таблица 2.2. *Вещественные типы*

Тип	Разрядность	Диапазон	Точность
`float`	4 байта	$3.4 \cdot 10^{-38} < \lvert x \rvert < 3.4 \cdot 10^{38}$	7–8 цифр
`double`	8 байтов	$1.7 \cdot 10^{-308} < \lvert x \rvert < 1.7 \cdot 10^{308}$	17 цифр
`long double`	10 байтов	$3.4 \cdot 10^{-4932} < \lvert x \rvert < 3.4 \cdot 10^{4932}$	21 цифра

Как получаются эти величины? Тип `float` распределяет свои 32 бита следующим образом: под кодирование знака числа отводится один, старший бит. Ноль в этом разряде означает, что число неотрицательно, единица — что число отрицательно. Следующие 8 битов отводятся под характеристику числа. Таким образом, характеристика принимает значения от 0 до 255. Порядок числа на 127 меньше, он может меняться от −127 до 128. Значит, степень двойки меняется от 2^{-127} до 2^{128}. После учета мантиссы и перевода в степень числа 10 получаются величины, приведенные в третьем столбце табл. 2.2.

Для мантиссы в типе `float` остается 23 бита, с учетом скрытого бита — 24 бита. Двадцати четырем двоичным разрядам соответствует 7—8 десятичных разрядов. У типа удвоенной точности `double` распределение битов под знак, характеристику и мантиссу 1 + 11 + 52, у типа `long double` — 1 + 15 + 64. Расширенный тип `long double` не обязателен, он встречается не на всех компьютерах. Этот тип был поначалу введен для процессоров фирмы Intel, математический сопроцессор которых работает с регистрами величиной 10 байтов.

Символьный тип

Символьный тип составляют отдельные символы: буквы, цифры, знаки препинания, знаки арифметических операций, пробельные символы и прочие символы, используемые в документах. В то время, когда создавался язык C,

для хранения одного символа выделялся один байт, в котором хранился числовой *код символа* — какое-то, в общем произвольное, число. Один байт — это комбинация из 8 нулей или единиц. Таких комбинаций может быть всего $2^8 = 256$, следовательно, ими можно закодировать не более 256 символов. Этого явно недостаточно и приходилось создавать различные *кодовые таблицы*. Это неудобно — кроме числового кода символа приходится указывать еще и кодовую таблицу. Тем не менее, байтовые кодировки широко распространены и полный отказ от них произойдет еще не скоро.

Теоретически каждому символу можно сопоставить любую комбинацию из 8 нулей и единиц и получить великое множество различных кодировок. На практике выработаны стандартные кодировки, которых придерживаются специалисты всех стран. Прежде чем перейти к стандартным кодировкам посмотрим, как записываются коды в шестнадцатеричной системе счисления.

Шестнадцатеричная система счисления

Мы уже говорили о системах счисления и отметили, что цифры обычного десятичного числа — это коэффициенты многочлена по степеням основания десятичной системы — числа 10. Основание двоичной системы — число 2 и цифры двоичного числа — коэффициенты при степенях двойки.

Количество цифр должно совпадать с основанием системы счисления. В обычной десятичной системе мы пользуемся десятью арабскими цифрами 0, 1, ..., 9. Для построения двоичной системы достаточно двух цифр 0 и 1. В шестнадцатеричной системе нужно 16 символов для обозначения цифр. Десять символов есть — это те же арабские цифры. Нужно взять еще 6 символов. Было решено не создавать новые символы, а просто взять первые 6 символов латинского алфавита. Число десять в шестнадцатеричной системе обозначается буквой A, одиннадцать — буквой B и т. д. Получается запись вида 1C2AB00F, при разложении на многочлен она приобретает следующий вид:

$$1 \cdot 16^7 + C \cdot 16^6 + 2 \cdot 16^5 + A \cdot 16^4 + B \cdot 16^3 + F \cdot 16^0.$$

Заменив буквы десятичными числами, получим:

$$1 \cdot 16^7 + 12 \cdot 16^6 + 2 \cdot 16^5 + 10 \cdot 16^4 + 11 \cdot 16^3 + 15 \cdot 16^0.$$

Вычислив его, получим десятичную запись числа. Регистр букв в записи шестнадцатеричного числа не различается, можно записывать и заглавные и строчные буквы, например, 1c2ab00f, но заглавные буквы лучше различимы.

Чем же удобна шестнадцатеричная система для записи кодов символов? Поскольку $16 = 2^4$, облегчается перевод шестнадцатеричных чисел в двоичную форму записи и обратно. Надо просто каждую шестнадцатеричную цифру

перевести в двоичную форму, записав ее двоичное представление ровно четырьмя двоичными цифрами. Например, предыдущее шестнадцатеричное число в двоичной форме будет записано так:

1 1100 0010 1010 1011 0000 0000 1111.

Для удобства просмотра число разбито на группы по 4 двоичные цифры. Итак, достаточно знать только двоичное представление первых шестнадцати чисел: 0 — 0000, 1 — 0001, 2 — 0010, 3 — 0011, 4 — 0100, 5 — 0101, 6 — 0110, 7 — 0111, 8 — 1000, 9 — 1001, A — 1010, B — 1011, C — 1100, D — 1101, E — 1110, F — 1111.

Для перевода двоичного числа в шестнадцатеричную форму надо проделать обратные действия: разбить число на группы по 4 двоичные цифры, влево и вправо от двоичной точки, и каждую четверку записать одной шестнадцатеричной цифрой. Например:

$$111011000010001.011 = 0111\ 0110\ 0001\ 0001.0110 = 7611.6_{16}.$$

Для того чтобы не спутать последнюю запись с записью десятичного числа, нижним индексом указано основание системы счисления.

Особенно удобна шестнадцатеричная запись для показа содержимого байтов оперативной памяти. Ведь байт состоит из 8 битов, которые записываются ровно двумя шестнадцатеричными цифрами. Например, 10001010 = 8A, 10110010 =B2 и т. д. Поэтому, если вы захотите посмотреть содержимое оперативной памяти, то компьютер покажет вам его именно в шестнадцатеричной записи.

Упражнения

1. Переведите в шестнадцатеричную систему счисления числа 678, 345, −543.
2. Найдите алгоритм перевода чисел из десятичной системы в шестнадцатеричную аналогично алгоритму, изображенному на рис. 1.15.
3. Переведите число 2F3C05 в двоичную систему счисления, из двоичной системы в десятичную, а из нее снова в шестнадцатеричную. Получилось то же самое число?

Стандартные кодировки символов

Проще всего обстоит дело с кодированием специальных символов и букв английского языка. Стандартная кодировка ASCII (American Standard Code for Information Interchange, американский стандартный код обмена информацией) общепринята, она не вызывает никаких возражений и давно используется всем компьютерным миром. Коды ASCII приведены в табл. 2.3. Чтобы получить шестнадцатеричный код символа, надо сложить числа,

записанные против строки и столбца, в которых расположен этот символ. Например, код символа "F" равен 40 + 06 = 46 = 01000110.

Таблица 2.3. *Коды ASCII*

	00	**01**	**02**	**03**	**04**	**05**	**06**	**07**
00	NUL	DLE	SP	0	@	P	‘	p
01	SOH	DC1	!	1	A	Q	a	q
02	STX	DC2	"	2	B	R	b	r
03	ETX	DC3	#	3	C	S	c	s
04	EOT	DC4	$	4	D	T	d	t
05	ENQ	NAK	%	5	E	U	e	u
06	ACK	SYN	&	6	F	V	f	v
07	BEL	ETB	‘	7	G	W	g	w
08	BS	CAN	(	8	H	X	h	x
09	HT	EM	)	9	I	Y	i	y
0A	LF	SUB	*	:	J	Z	j	z
0B	VT	ESC	+	;	K	[	k	{
0C	FF	FS	,	<	L	\	l	|
0D	CR	GS	–	=	M	]	m	}
0E	SO	RS	.	>	N	^	n	~
0F	SI	US	/	?	O	_	o	DEL

Первые два столбца занимают коды управляющих символов. Они придуманы еще в эпоху телеграфа для управления принимающим телеграфным устройством. Сокращением SP обозначен обычный пробел. Его код, как видите, равен 00100000. Цифра 0, понимаемая не как число, а как символ, получила код 00110000, а вовсе не 00000000, как можно было подумать. Нулевой код дан пустому *нуль-символу*, обозначенному NUL. Обычно нуль-символ показывает отсутствие всяких символов. Итак, последовательность символов "It’s a string" будет записана в оперативной памяти

компьютера в таком виде: 0100100101110100001001110111001100100000 0110000100100000011100110111010001110010011010010110011101100111.

Как можно заметить из этой таблицы, коды ASCII меняются от 0 до 127. Буквы других национальных алфавитов и графические символы получают коды от 128 до 255. Здесь появляются различные кодовые страницы, для которых разработан международный стандарт ISO 8859.

Буквы немецкого, французского, испанского и некоторых других алфавитов составили кодовую страницу Latin 1, ее официальное название ISO 8859-1. Коды других западноевропейских алфавитов составили кодовые страницы Latin 2 или ISO 8859-2, Latin 3 или ISO 8859-3 и т. д. Кириллица включена в кодовую страницу ISO 8859-5 (табл. 2.4).

Таблица 2.4. *Коды кириллицы ISO 8859-5*

	B0	**C0**	**D0**	**E0**	**F0**
00	А	Р	а	р	Ё
01	Б	С	б	с	ё
02	В	Т	в	т	
03	Г	У	г	у	
04	Д	Ф	д	ф	
05	Е	Х	е	х	
06	Ж	Ц	ж	ц	
07	З	Ч	з	ч	
08	И	Ш	и	ш	
09	Й	Щ	й	щ	
0A	К	Ъ	к	ъ	
0B	Л	Ы	л	ы	
0C	М	Ь	м	ь	
0D	Н	Э	н	э	
0E	О	Ю	о	ю	
0F	П	Я	п	я	

Этот международный код используют иностранцы, когда хотят написать что-нибудь на кириллице. В России же применяются другие кодировки. Со времен MS-DOS осталась CP 866 (табл. 2.5).

Таблица 2.5. *Коды кириллицы CP 866*

	80	**90**	**A0**	**B0**	**C0**	**D0**	**E0**	**F0**
00	А	Р	а				р	Ё
01	Б	С	б				с	ё
02	В	Т	в				т	
03	Г	У	г				у	
04	Д	Ф	д				ф	
05	Е	Х	е				х	
06	Ж	Ц	ж				ц	
07	З	Ч	з				ч	
08	И	Ш	и				ш	
09	Й	Щ	й				щ	
0A	К	Ъ	к				ъ	
0B	Л	Ы	л				ы	
0C	М	Ь	м				ь	№
0D	Н	Э	н				э	
0E	О	Ю	о				ю	
0F	П	Я	п				я	

В этой кодировке коды малых букв разрываются: после кода 10101111 буквы "п", десятичное число 175, сразу идет код 11100000 буквы "р", число 224. Так получилось потому, что в середине таблицы располагались символы псевдографики, использовавшиеся во многих программах, например, в Norton Commander.

С появлением графической системы Microsoft Windows надобность в псевдографике отпала, ее нет на новых кодовых страницах. На смену CP 866 пришла кодовая страница CP 1251 (см. табл. 2.6).

Таблица 2.6. *Коды кириллицы CP 1251*

	80	**90**	**A0**	**B0**	**C0**	**D0**	**E0**	**F0**
00			NBSP		А	Р	а	р
01					Б	С	б	с
02					В	Т	в	т
03					Г	У	г	у
04					Д	Ф	д	ф
05					Е	Х	е	х
06					Ж	Ц	ж	ц
07					З	Ч	з	ч
08			Ё	ё	И	Ш	и	ш
09				№	Й	Щ	й	щ
0A					К	Ъ	к	ъ
0B					Л	Ы	л	ы
0C					М	Ь	м	ь
0D					Н	Э	н	э
0E					О	Ю	о	ю
0F					П	Я	п	я

Сокращением NBSP (No-Break Space) обозначен неразрывный пробел. Кроме этих кодировок есть еще интересная кодировка KOI8-R, в которой коды идут не по алфавиту (табл. 2.7).

Таблица 2.7. *Коды кириллицы KOI8-R*

	80	**90**	**A0**	**B0**	**C0**	**D0**	**E0**	**F0**
00					ю	п	Ю	П
01					а	я	А	Я
02					б	р	Б	Р
03					ц	с	Ц	С
04					д	т	Д	Т
05					е	у	Е	У
06					ф	ж	Ф	Ж
07					г	в	Г	В
08					х	ь	Х	Ь
09					и	ы	И	Ы
0A					й	з	Й	З
0B					к	ш	К	Ш
0C					л	э	Л	Э
0D					м	щ	М	Щ
0E					н	ч	Н	Ч
0F					о	ъ	О	Ъ

На первый взгляд таблица кажется совершенно произвольной. Приглядевшись, можно заметить, что расположение букв фонетически соответствует буквам латиницы. Например, если в кодировке 11000100 буквы "д" заменить старший бит нулем, то получим кодировку 01000100 латинской буквы "D". Такая замена, к сожалению, часто случается при передаче информации по

сетям. В результате вместо "Здравствуйте" получим "zDRAVSTVUJTE", что, в общем-то, можно прочитать. Если бы слово "Здравствуйте" было записано в кодировке CP 1251, то в этих же обстоятельствах мы получили бы совершенно нечитаемое "Gdp`bqrbsire". По этой причине кодировка KOI8-R часто применяется в Интернете.

Наконец, надо привести еще и кодировку Mac Cyrillic, применяемую в компьютерах Apple Macintosh (табл. 2.8).

Таблица 2.8. *Коды кириллицы Mac Cyrillic*

	80	**90**	**A0**	**B0**	**C0**	**D0**	**E0**	**F0**
00	А	Р					а	р
01	Б	С					б	с
02	В	Т					в	т
03	Г	У					г	у
04	Д	Ф					д	ф
05	Е	Х					е	х
06	Ж	Ц					ж	ц
07	З	Ч					з	ч
08	И	Ш					и	ш
09	Й	Щ					й	щ
0A	К	Ъ					к	ъ
0B	Л	Ы					л	ы
0C	М	Ь				№	м	ь
0D	Н	Э				Ё	н	э
0E	О	Ю				ё	о	ю
0F	П	Я				я	п	

Малая буква "я" оказывается расположенной раньше буквы "а". Это неудобно для сортировки по алфавиту, как мы увидим в дальнейшем, но в других ситуациях полезно. Дело в том, что многие устройства понимают байт с кодом 11111111 (десятичное число 255), как управляющий сигнал и вместо того чтобы передавать букву "я" в CP 1251 или твердый знак в KOI8-R,

выполняют какие-то, часто нежелательные, действия. Программистам, сталкивающимся с таким устройством, приходится идти на всякие хитрости, чтобы передать букву с кодом 255. Например, в кодировке KOI8-R твердый знак часто заменяют апострофом.

Упражнения

4. Каковы коды буквы "Ж" в разных кодировках кириллицы?
5. Какие буквы кириллицы обозначены кодом 200 в разных кодировках?

Unicode

В последние годы происходит переход на двухбайтовую кодировку символов, называемую Unicode. Два байта дают 65 536 комбинаций. Этого пока хватает для кодирования всех символов, придуманных человечеством. Никакие кодовые таблицы для Unicode не нужны, но размер документа, записанного в Unicode, увеличивается вдвое.

Первые 256 кодов Unicode от 0000_{16} до $00FF_{16}$ занимают коды Latin 1. Далее идут коды других европейских алфавитов, в частности, греческого алфавита. Кириллица кодируется первым байтом 04_{16}. Заглавная буква "Ё" получила код 0401_{16}. Коды от 0410_{16} до $042F_{16}$ даны заглавным буквам от "А" до "Я". Следующие коды от 0430_{16} до $044F_{16}$ отведены малым буквам от "а" до "я". Код 0451_{16} дан малой букве "ё".

Упражнение

6. Каков код буквы "Ж" в Unicode?

Символьный тип в языке C

В языке C стандартный символьный тип `char` представляет байтовую кодировку. Английский алфавит и специальные символы записываются кодами ASCII. Кодировка кириллицы определяется русификацией компьютера, на котором выполняется программа, хотя это правило всегда можно изменить.

Одной из особенностей языка C является то, что символы можно использовать в арифметических выражениях, что мы часто увидим в дальнейшем. В этом случае в вычислениях участвует код символа, понимаемый как целое число. Эта особенность подчеркивается тем, что в язык C введен тип `signed char` — символьный тип со знаком, совпадающий с типом `byte` других языков. В этом типе старший бит отводится под знак, и значения типа изменяются от −128 до 127, а не от 0 до 255. Для кодировки ASCII это неважно, она вся укладывается в диапазон от 0 до 127, а для кириллицы вызывает дополнительные трудности. Поэтому надо всегда четко представлять

себе, какой тип символов — `signed char` или `unsigned char` — применяет ваш компилятор по умолчанию или явно указывать модификатор типа.

Константы

При программировании все время приходится записывать константы и переменные разных типов. Разные языки предлагают для этого разные способы. В языке С можно записывать константы в нескольких видах. Перечислим их.

Целые константы

Целые константы можно записывать в трех системах счисления:

- в обычной десятичной системе: `+5`, `-7`, `12345678`;
- в восьмеричной системе счисления, начиная их с нуля: `027`, `-0326`, `0777`; в записи таких констант недопустимы цифры 8 и 9;

Внимание!

Целое число, начинающееся с нуля, записано в восьмеричной системе, а не в десятичной!

- в шестнадцатеричной системе счисления, начиная их с нуля и латинской буквы `x` или `X`: `0xff0a`, `0xFC2D`, `0X45a8`, `0X77FF`; здесь строчные и прописные буквы не различаются.

Целые константы хранятся в формате типа `int`, если они не слишком велики. Очень большие по модулю целые константы хранятся в формате `long`. Если число положительно и не помещается в типе `long`, то для ее записи выбирается тип `unsigned long`.

Нестандартный тип можно выбрать явно. В конце целой константы всегда можно записать букву `L` или строчную букву `l`, тогда константа будет сохраняться в длинном формате типа `long`: `+25L`, `-037l`, `0xffL`, `0XDFDFl`.

Рекомендация

Не используйте при записи длинных целых констант строчную латинскую букву `l`, ее легко спутать с единицей!

Неотрицательные константы можно закончить буквой `u` или `U`, если надо хранить их в беззнаковом типе `unsigned int` или `unsigned long`. Например, `256u`, `345609467LU`, `45902455768UL`.

Упражнение

7. Чему равно выражение `2C3 + 067L - 234U`?

Вещественные константы

Вещественные константы записываются только в десятичной системе счисления в двух формах:

- с фиксированной точкой: `37.25`, `-128.678967`, `+27.035`;
- с плавающей точкой: `2.5e34`, `-0.345e-25`, `37.2E+4`; можно писать строчную или прописную латинскую букву `E`; пробелы, скобки недопустимы.

Вещественные константы всегда хранятся в формате типа `double`. Если надо хранить вещественную константу в формате типа `float`, то в конце ее надо написать букву `f` или `F`, например, `2.5F`, `0.35e-4F`. Если же для константы надо задать тип `long double`, то в конце числа записывается строчная или заглавная буква `L`, например, `-3.45L`.

Символы

Для записи одиночных символов используются три формы.

- Печатные символы можно записать в апострофах: `'a'`, `'N'`, `'?'`.
- Управляющие символы и некоторые другие записываются в апострофах с обратной наклонной чертой. Несмотря на то, что мы пишем два символа, включая обратную наклонную черту, компилятор воспринимает эту запись как один символ, выделяя для него один байт. Вот набор таких символов:
  - `\n` — символ перевода строки LF с кодом ASCII 10;
  - `\r` — символ возврата каретки CR с кодом ASCII 13;
  - `\f` — символ перевода страницы FF с кодом ASCII 12;
  - `\b` — символ возврата на шаг BS с кодом ASCII 8;
  - `\t` — символ горизонтальной табуляции HT с кодом ASCII 9;
  - `\v` — символ вертикальной табуляции VT с кодом ASCII 11;
  - `\a` — звуковой сигнал BEL с кодом ASCII 7;
  - `\\` — обратная наклонная черта;
  - `\"` — кавычка;
  - `\'` — апостроф;
  - `\0` — нуль-символ.
- Код любого символа с десятичной кодировкой от 0 до 255 можно задать, записав его не более чем тремя восьмеричными цифрами в апострофах после обратной наклонной черты: `\123` — буква "S", `\346` — буква "Ж" в кодировке CP 1251. Не рекомендуется использовать эту форму записи

для печатных и управляющих символов, перечисленных в предыдущем пункте, поскольку большинство компиляторов сразу же переведет восьмеричную запись в указанную выше форму. Наибольший код `\377` — десятичное число 255.

- Код любого символа можно задать в шестнадцатеричной форме, записав после обратной наклонной черты букву "x" или "X" и затем несколько шестнадцатеричных цифр. Например, `\xFF`, `\x77`, `\X1D`.

Символы хранятся в формате типа `char`.

Упражнения

8. Чему равно выражение: `'A' + 'B' - 'C'`?
9. Чему равна разность `'9' - '0'`?

Строковые константы

Строки символов записываются в кавычках. Управляющие символы и коды записываются в строках точно так же, с обратной наклонной чертой, но, разумеется, без апострофов, и оказывают то же действие. Строки без дополнительных символов могут располагаться только на одной строке исходного кода, нельзя открывающую кавычку поставить на одной строке, а закрывающую — на следующей. Дополнительный символ, о котором шла речь в предыдущей фразе, — это обратная наклонная черта, которую можно поставить в конце первой строки. Вот некоторые примеры:

```
"Это строка символов с переносом \n на другую строку".

"\"Загрядник\" - Чемпион!".

"Строка символов, записанная на \
двух строках исходного кода."
```

Внимание!

Строки символов нельзя начинать на одной строке исходного кода программы, а заканчивать на другой, не оставив обратный слэш в конце первой строки!

Подряд идущие строки символов сцепляются и считаются одной строковой константой. Компилятор языка C всегда дополняет строковую константу еще одним символом: строка заканчивается нуль-символом. Поэтому строковая константа занимает в оперативной памяти на один байт больше, чем число символов в ней. Даже если строка состоит из одного символа, она займет два байта, например, строка `"a"` кодируется как 0110000100000000. Символ же `'a'` займет один байт 01100001.

Имена объектов

Переменные в программе различаются своими именами. *Имена* (names) переменных и других объектов носят общее название — *идентификаторы* (identifiers). Идентификаторы в языке С составляются из букв, цифр и знаков подчеркивания, причем первым символом идентификатора не может быть цифра. (Действительно, как понять запись `2e3`: как число 2000.0 или как имя переменной?)

Вот примеры правильных идентификаторов: `a1`, `my_var`, `var3_5`, `_var`, `veryLongVarName`, `aName`, `theName`, `a2Vh36kBnMt456dX`. Обратите внимание на то, что пробелы в имени недопустимы. Нельзя давать объектам имена, совпадающие со служебными словами языка: `int`, `char`, `if`, `for` и т. д. Язык С различает заглавные и строчные буквы, поэтому `var1`, `Var1` и `VAR1` — совершенно разные переменные!

В именах лучше не использовать строчную букву "l", которую легко спутать с единицей, и букву "O", которую легко спутать с нулем. Количество символов в идентификаторе стандартом не ограничивается, имя может быть длинным, но многие компиляторы различают имена только по первым символам. Раньше учитывались первые 8 символов, теперь чаще всего первые 32 символа, но не менее 31 символа.

Определение переменных

Как видите, нигде в идентификаторе не отмечается тип переменной, которую он обозначает. Поэтому перед использованием переменной обязательно надо явно указать ее тип. Для этого записывается название типа, и после пробела, через запятую, перечисляются имена переменных, относящихся к этому типу. Тут же, через знак равенства, можно дать начальное значение переменной, причем его можно записать не только константой, но и выражением. Конечно, все элементы этого выражения должны быть известны. Компилятор вычислит выражение и даст переменной полученное значение. Завершается определение точкой с запятой. Например:

```
short det = 0, ind = 1;

int i = -100, j = 100, k = 9999;

long big = 50, veryBig = 2147483648L;

char c1 = 'A', c2 = '?', newLine = '\n', c3 = 67;

float x = 0.001, y = -34.789, z = 3.45e-5;

double z1 = -16.2305, z2;

int n = 0, m = 1, k = m + 100;
```

Порядок определений произволен, названия типов могут повторяться, но одна переменная не может быть определена дважды.

Обратите внимание на то, что мы может задать начальное значение символьной переменной типа `char`, указав числовой код символа. Заметьте также, что мы можем в той же строке использовать только что определенную переменную для определения следующих переменных. Язык С позволяет дать символическое имя числовой или символьной константе. Для этого в начале определения символического имени надо поставить слово `const`. Например:

```
const int MAX_VAL = 9999;

const char NEW_LINE = '\n';
```

После такого определения компилятор будет следить за тем, чтобы значения идентификаторов `MAX_VAL` и `NEW_LINE` не менялись. Символические имена констант наглядны и очень часто применяются в программировании. У них есть еще одно ценное качество. В случае изменения числового или символьного значения константы достаточно изменить только одно место программы — ее определение — и значение изменится всюду. Например, на компьютерах Apple Macintosh новая строка отделяется от старой строки не символом `\n`, как принято в UNIX, а символом `\r`. Если сделано определение константы `NEW_LINE`, то при переносе программы на Apple Macintosh надо будет только изменить значение константы.

У программистов всего мира давно установлено общепринятое соглашение — записывать имена констант заглавными буквами, соединяя слова знаком подчеркивания. Это соглашение позволяет сразу увидеть константы в тексте программы. От старых версий в языке С осталось еще одно определение символических имен — директива препроцессора `#define`. В ней первое слово, идущее после `#define`, дает символическое имя, остальные слова, до перевода строки — определение константы. Например:

```
#define MAX_VAL 9999

#define NEW_LINE '\n'
```

Увидев эти директивы, препроцессор сразу же заменит символические имена их значениями и передаст компилятору только числа и символы, обозначенные константами. Директива `#define` удобна тем, что она, в отличие от определения константы, позволяет записать под одним именем несколько слов. Но зато компилятор не может узнать и проверить тип обозначенной директивой константы. Препроцессор вообще не различает типы, он делает простую подстановку. Поэтому типизированные константы лучше определять модификатором `const`.

Операции

В языке C реализовано очень много операций. Они позволяют обрабатывать отдельные значения простых типов данных, а также их составляющие и даже отдельные биты. Все операции можно разделить на следующие группы.

Арифметические операции

К арифметическим операциям относятся:

- сложение — `+` (плюс);
- вычитание — `-` (минус);
- умножение — `*` (звездочка);
- деление — `/` (наклонная черта, слэш);
- взятие остатка от деления целых чисел (деление по модулю) — `%` (процент);
- инкремент (увеличение на единицу значения целочисленной переменной) — `++` (два знака плюс);
- декремент (уменьшение на единицу значения целочисленной переменной) — `--` (удвоенный знак минус).

Между сдвоенными плюсами и минусами нельзя оставлять пробелы. Сложение, вычитание и умножение целых значений выполняются как обычно, а вот деление целых значений в результате дает опять целое (так называемое *целое деление*), например, выражение `5/2` даст в результате 2, а не 2.5, а `5/(-3)` даст −1. Дробная часть попросту отбрасывается, происходит усечение частного. Это поначалу обескураживает, но потом оказывается удобным для усечения чисел.

Внимание!

В языке C для целых типов принято целое деление!

Это странное для математики правило естественно для программирования: если оба операнда имеют один и тот же тип, то и результат получает тот же тип. Для вещественных типов результат деления будет вещественным числом. Достаточно написать `5/2.0` или `5.0/2` или `5.0/2.0` и получим `2.5` как результат деления вещественных чисел. Операция *деление по модулю* применяется только к целым типам и определяется так:

```
a % b = a - (a / b) * b;
```

Например, `5%2` даст в результате 1, а `5%(-3)` даст 2, т. к. $(-3) \cdot (-1) + 2 = 5$, но `(-5)%3` даст −2, т. к. $3 \cdot (-1) - 2 = -5$.

Операции *инкремент* и *декремент* означают увеличение или уменьшение значения целочисленной переменной на единицу и применяются только к переменным, но не к константам или выражениям, нельзя написать `5++` или `(a + b)++`. Например, после приведенных выше описаний `i++` даст −99, а `j--` даст 99. Эти операции кажутся надуманными, но на самом деле они естественны для компьютера и выполняются одной машинной командой.

Интересно то, что эти операции можно записать и перед переменной: `++i`, `--j`. Разница проявится только в выражениях: при первой форме записи (*постфиксной*) в выражении участвует старое значение переменной и только потом происходит увеличение или уменьшение самой переменной. При второй форме записи (*префиксной*) сначала изменится переменная, ее новое значение будет участвовать в выражении.

Например, после приведенных выше описаний `(k++) + 5` даст в результате 10004, а переменная `k` примет значение 10000. Но в той же исходной ситуации `(++k) + 5` даст 10005, а переменная `k` станет равной 10000.

Приведение типов

В языке C результат арифметической операции с целыми типами имеет тип `int`, кроме того случая, когда один из операндов типа `long`. В этом случае результат будет типа `long`. Результат никогда не будет иметь тип `short` или `char`. Перед выполнением арифметической операции всегда происходит *повышение* (promotion) типов `short`, `char`. Они преобразуются в тип `int`, а может быть, и в тип `long`, если другой операнд типа `long`. Операнд типа `int` повышается до типа `long`, если другой операнд типа `long`.

Конечно, для типов `short` и `long` числовое значение и знак операнда при этом не меняются, дополнительные биты заполняются тем же числом 0 или 1, какое находилось в старшем, знаковом разряде. Как говорят, распространяется знаковый бит. По-другому обстоит дело с повышением типа `char`. Если старший бит равен 0, например, 01010001, то при повышении типа дополнительные биты просто заполняются нулями: 00000000 00000000 00000000 01010001. Но если старший бит равен 1, например, 10100001, то компилятор может, как и выше, заполнить дополнительные биты нулями: 00000000 00000000 00000000 10100001, а может распространить знаковый разряд и заполнить их единицами: 11111111 11111111 11111111 10100001. Многие компиляторы допускают настройку этого поведения.

Обычно мы понимаем значения символьного типа как кодировку от 0 до 255, т. е. считаем их неотрицательными. В таком случае лучше указать явно тип `unsigned char`. Если же мы хотим, чтобы символы могли участвовать в арифметических операциях как отрицательные числа, надо определить их

как `signed char`. В операциях присваивания часто приходится *сужать* (narrow) тип. Например:

```
int k = 234.567;
```

Здесь целочисленная переменная `k` пытается присвоить значение типа `double`. В таких случаях компилятор просто отбрасывает дробную часть и редко сообщает об этом. При сужении `double` до `float` может происходить и округление, и усечение числа в зависимости от реализации компилятора. Сужение целых типов, например, `long` в `char`, осуществляется просто отбрасыванием старших битов, что необходимо учитывать для больших значений целых чисел. Во всех случаях можно выполнить явное приведение типа. Оно осуществляется специальной операцией явного приведения, которая записывается перед приводимым значением в виде имени типа в скобках. Например:

```
short k = (short)(b1 + b2);
```

При явном приведении типа компилятор создает временную переменную, в которую записывает приведенное значение. Таким же образом можно сделать и явное *расширение* (widening) типа, если в этом есть необходимость. Например, стандартные математические функции `sin(x)`, `sqrt(x)` и другие работают с аргументом типа `double`. Если написать:

```
int k = 10;
double y = sqrt(k);
```

то результат может оказаться неверным. В таких случаях приходится либо записывать прототип функции, либо писать:

```
double y = sqrt((double)k);
```

Если результат целой операции над беззнаковыми целыми выходит за диапазон своего типа `unsigned int` или `unsigned long`, то автоматически происходит приведение по модулю, равному длине этого диапазона, и вычисления продолжаются, переполнение никак не отмечается. Поскольку к вещественным типам применимы арифметические операции, целые и вещественные значения можно смешивать в операциях. При этом правило приведения типов дополняется такими условиями:

- если в операции один операнд имеет тип `long double`, то и другой приводится к типу `long double`;
- в противном случае, если в операции один операнд имеет тип `double`, то и другой приводится к типу `double`;
- в противном случае, если один операнд имеет тип `float`, то и другой приводится к типу `float`;
- в противном случае действует правило приведения целых значений.

Сравнения

В языке C шесть обычных операций сравнения величин:

- больше — `>`;
- меньше — `<`;
- больше или равно — `>=`;
- меньше или равно — `<=`;
- равно — `==`;
- не равно — `!=`.

Сдвоенные символы записываются без пробелов, их нельзя переставлять местами, запись `=>` будет неверной.

Числа сравниваются по величине, у символов сравниваются их коды. Результат сравнения — целое число 0 или 1 типа `int`. Если сравнение верно, то получаем 1, если неверно — 0. Такой результат можно использовать в арифметических выражениях, например, можно написать `x * (a < b)`. В результате, если `a` меньше `b`, то получим значение `x`, в противном случае получим 0.

Для записи сложных сравнений следует привлекать логические операции. Например, в вычислениях часто приходится делать проверки вида `a < x < b`. Подобная запись на языке C не приведет к сообщению об ошибке, поскольку первое сравнение, `a < x`, даст 0 или 1 и это число будет сравниваться со значением переменной `b`. Но результат, скорее всего, будет неверен, а мы можем этого даже не заметить. Например, и сравнение `1 < 2 < 3` и сравнение `3 < 2 < 1` дадут в результате 1.

В этом случае следует написать выражение `(a < x) && (x < b)`, причем здесь скобки можно опустить, написать просто `a < x && x < b`, но об этом далее.

Логические операции

Логические операции:

- отрицание (NOT) — `!` (восклицательный знак);
- конъюнкция (AND) — `&&` (удвоенный амперсанд);
- дизъюнкция (OR) — `||` (удвоенная вертикальная черта).

Логические операции выполняются над целыми операндами, их результатом будет целое число 0 или 1 типа `int` (табл. 2.9).

Таблица 2.9. *Логические операции*

b1	b2	!b1	b1 && b2	b1 || b2
!= 0	!= 0	0	1	1
!= 0	0	0	0	1
0	!= 0	1	0	1
0	0	1	0	0

Словами эти правила можно выразить так:

- отрицание меняет нулевое значение операнда на 1, ненулевое — на ноль;
- конъюнкция похожа на умножение, она дает 1, только если операнды ненулевые, во всех остальных случаях получается ноль;
- дизъюнкция похожа на сложение, она дает ноль, только если операнды равны 0.

Замечание

Если бы Шекспир был программистом, фразу: "To be or not to be", он написал бы так: "2b || ! 2b".

Удвоенные знаки амперсанда и вертикальной черты следует записывать без пробелов. Логические операции вычисляются слева направо, причем компиляторы выполняют сокращение их числа. Сокращение заключается в том, что правый операнд вычисляется только в том случае, если от него зависит результат операции, т. е. если левый операнд конъюнкции имеет значение `true`, или левый операнд дизъюнкции имеет значение `false`.

Это правило очень удобно и очень ловко используется, например, можно записывать выражения `(n != 0) && (m/n > 0.001)` или `(n == 0) || (m/n > 0.001)`, не опасаясь деления на ноль. Конечно, нужно быть уверенным в том, что компилятор сделает сокращение. Некоторые старые компиляторы требуют предварительной настройки на выполнение сокращенных логических операций.

Побитовые операции

Иногда приходится изменять значения отдельных битов в целочисленных данных. Это выполняется с помощью побитовых (bitwise) операций. В языке C есть четыре побитовые операции:

- дополнение (complement) — `~` (тильда);
- побитовая конъюнкция (bitwise AND) — `&` (амперсанд);

- побитовая дизъюнкция (bitwise OR) — `|` (вертикальная черта);
- побитовое исключающее ИЛИ (bitwise XOR) — `^`.

Они выполняются поразрядно, после того как оба операнда будут приведены к одному типу `int` или `long`, так же как и для арифметических операций, а значит, и к одной разрядности. Операции над каждой парой битов выполняются согласно табл. 2.10.

Таблица 2.10. *Побитовые операции*

n1	n2	~n1	n1 & n2	n1 | n2	n1 ^ n2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Приведем пример. Пусть определены две переменные:

```
int b1 = 50, b2 = -99;
```

Двоичное 32-разрядное представление для `b1` — 000110010, для `b2` — 110011101. В результате побитовых операций получаем:

- `~b2 == 98`, двоичное представление — 001100010;
- `b1 & b2 == 16`, двоичное представление — 000010000;
- `b1 | b2 == -65`, двоичное представление — 110111111;
- `b1 ^ b2 == -81`, двоичное представление — 110101111.

Двоичное представление каждого результата занимает 32 бита. Заметьте, что дополнение `~x` всегда эквивалентно `(-x) - 1`.

Сдвиги

В языке C есть две операции сдвига двоичных разрядов целого числа:

- сдвиг влево — `<<` (удвоенный знак меньше);
- сдвиг вправо — `>>` (удвоенный знак больше).

Эти операции своеобразны тем, что левый и правый операнды в них имеют разный смысл. Слева стоит значение целого типа, а правая часть показывает, на сколько двоичных разрядов сдвигается значение, стоящее в левой части.

Например, операция `b1 << 2` сдвинет влево на два разряда предварительно повышенное значение 000110010 переменной `b1`, что даст в результате 0011001000, десятичное 200. Освободившиеся справа разряды заполняются

нулями, левые разряды, находящиеся за 32-м битом, теряются. Операция `b2 << 2` сдвинет повышенное значение 110011101 на два разряда влево. В результате получим 11001110100, десятичное значение −396. Заметьте, что сдвиг влево на *n* разрядов эквивалентен умножению числа на двойку в степени *n*, если только не наступает переполнение.

Операция `b1 >> 2` даст в результате 000001100, десятичное число 12, а `b2 >> 2` — результат 111100111, десятичное число −25, т. е. слева распространяется старший бит, правые биты теряются. Это так называемый *арифметический сдвиг*. Некоторые компиляторы всегда ставят слева на освободившиеся места нули, осуществляя *логический сдвиг*. Есть компиляторы, которые надо предварительно настроить на арифметический или на логический сдвиг.

Операции присваивания

Простая операция присваивания (simple assignment operator) записывается знаком равенства, слева от которого стоит переменная, а справа выражение, совместимое с типом переменной:

```
x = 3.5,
y = 2 * (x - 0.567) / (x + 2),
b = x < y, bb = x >= y && b.
```

Операция присваивания действует так: выражение, стоящее после знака равенства вычисляется и приводится к типу переменной, стоящей слева от знака равенства. Результатом операции будет приведенное значение правой части. Операция присваивания имеет еще одно, побочное, действие: переменная, стоящая слева, получает приведенное значение правой части, старое ее значение теряется.

В операции присваивания левая и правая части неравноправны, нельзя написать `3.5 = x`. После операции `x = y` изменится переменная `x`, став равной `y`, а после `y = x` изменится `y`. Кроме простой операции присваивания есть еще десять *составных* операций присваивания (compound assignment operators): `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`. Символы записываются без пробелов, нельзя переставлять их местами. Все составные операции присваивания действуют по одной схеме: `x op= a` эквивалентно `x = (тип x)(x op a)`.

Пусть переменная `ind` типа `short` определена со значением `1`. Присваивание `ind += 7.8` даст в результате число 8, то же значение получит и переменная `ind`. Эта операция эквивалентна простой операции присваивания `ind = (short)(ind + 7.8)`.

Условная операция

У этой своеобразной операции три операнда. Вначале записывается произвольное целочисленное выражение, затем знак вопроса, потом еще два произвольных выражения, разделенных двоеточием, например,

```
x < 0 ? 0 : x;

x > y ? x - y : x + y;
```

Условная операция выполняется так. Сначала вычисляется первое выражение. Если получилось ненулевое значение, то вычисляется второе выражение, записанное сразу после вопросительного знака, и его значение будет результатом всей операции. Последнее выражение при этом не вычисляется. Если же в результате вычисления первого выражения получился ноль, то вычисляется только последнее выражение, его значение будет результатом операции. Это позволяет написать выражение `n == 0 ? m : m / n`, не опасаясь деления на ноль. Условная операция поначалу кажется странной, но она очень удобна для записи небольших разветвлений и очень часто применяется в языке С.

Операция "запятая"

Несколько выражений, перечисленных через запятую, считаются одним выражением с операцией "запятая". Например, `a = 34, b = a + 5, c = 8`. Результатом такого выражения будет результат последнего из перечисленных выражений, в примере 8. Результаты предыдущих выражений потеряются, если это только не присваивание, как записано в примере. Конечно, это искусственная операция, она применяется главным образом в операторе цикла с перечислением.

Выражения

Из констант и переменных, операций над ними, вызовов функций и скобок составляются *выражения* (expressions). Разумеется, все элементы выражения должны быть совместимы. В языке С все операции, кроме присваиваний, выполняются слева направо. Сложение `x + y + z` будет вычисляться как `(x + y) + z`, сравнение `x < y < z` — как `(x < y) < z`, но выражение `x = y = z` будет вычисляться как `x = (y = z)`.

Порядок вычисления операндов в выражении не определен. Значения переменных `z` и `y`, стоящих справа, могут быть вычислены раньше, чем `x`. Это правило может приводить к расхождениям при вычислении выражения разными компиляторами. Пусть, например:

```
int b = 5;
```

Тогда результатом выражения `b + (b = 3)` может быть и число `6` и число `8`. То же относится к выражению `b += (b = 3)`. Единственный способ избежать таких расхождений — вычислить спорные места выражения заранее, например, обозначить `t = (b = 3)`; и использовать в выражении `b + t`.

Внимание!

Не записывайте выражения, зависящие от порядка операндов.

Выражения могут иметь сложный и запутанный вид. В таких случаях возникает вопрос о приоритете операций, о том, какие операции будут выполнены в первую очередь. Естественно, умножение и деление выполняются раньше сложения и вычитания. Остальные правила перечислены в следующем разделе.

Порядок вычисления выражения всегда можно отрегулировать скобками, их можно ставить сколько угодно. Но здесь важно соблюдать золотую середину. При большом количестве скобок снижается наглядность выражения и легко ошибиться. Если выражение со скобками выглядит корректно, то компилятор может отследить только парность скобок, но не правильность их расстановки.

Приоритет операций

Операции перечислены в порядке убывания приоритета. Операции на одной строке имеют одинаковый приоритет.

1. Постфиксные операции `++` и `--`.
2. Префиксные `++` и `--`, дополнение `~` и отрицание `!`.
3. Приведение типа `(тип)`.
4. Умножение `*`, деление `/` и взятие остатка `%`.
5. Сложение + и вычитание `-`.
6. Сдвиги `<<`, `>>`.
7. Сравнения `>`, `<`, `>=`, `<=`.
8. Сравнения `==`, `!=`.
9. Побитовая конъюнкция `&`.
10. Побитовое исключающее ИЛИ `^`.
11. Побитовая дизъюнкция `|`.
12. Конъюнкция `&&`.
13. Дизъюнкция `||`.
14. Условная операция `?:`.

15. Присваивания `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`.
16. Операция "запятая".

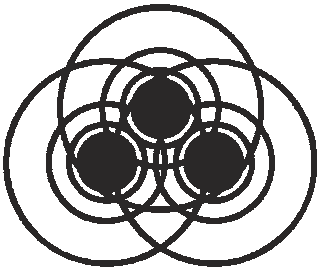
Здесь перечислены не все операции языка C, список будет дополняться по мере изучения новых операций.

Упражнения

10. Пусть `int x = 5`. Найдите значение `x` после присваивания `x = -3 * x % -5 / 4`.
11. Чему будет равно значение `x` после присваивания `x *= x+=x==8`?
12. Найдите результат выражения `++x |++x & x++`.
13. Найдите значение выражения `1 <= x <= 10`.
14. Чему равно значение выражения `x <<= x < 9 ? x++ : x < 2`?

Контрольные вопросы

1. Перечислите все целые типы языка C.
2. Каков тип константы 0xFF2C?
3. Какие вещественные типы есть в языке C?
4. Каков тип констант 1.0, 98975.340568908e-12?
5. Какая кодировка кириллицы принята в Интернете?
6. Почему символы записываются в апострофах, а строки — в кавычках?
7. Можно ли взять остаток от деления вещественных чисел?
8. Можно ли применить логические операции к вещественным типам?
9. Можно ли применить побитовые операции к данным символьного типа?
10. Можно ли применить побитовые операции к вещественным типам?
11. Применяются ли сдвиги к вещественным типам?
12. Какой тип будет у результата сложения слагаемого типа `short` и слагаемого типа `char`?
13. Каков тип у результата деления данного типа `int` на делитель типа `float`?
14. Какой тип получится в результате присваивания переменной типа `short` выражения типа `float`?



Глава 3

Основные структуры управления

Первая задача каждого языка программирования — обеспечить удобную запись самых разнообразных алгоритмов обработки исходных данных: чисел, символов, строк и других структур, создание которых обеспечивают средства языка. Для этого в язык вводятся управляющие конструкции, называемые *операторами* языка. В английской литературе принят термин statement (предложение). Его буквальный перевод не отражает активность управляющих конструкций, поэтому в русской литературе принят другой термин.

Операторы

Как вы уже знаете — любой алгоритм, предназначенный для выполнения на компьютере, можно разработать так, что в нем будут только линейные вычисления, разветвления и циклы. Записать алгоритм можно в разных формах: в виде блок-схемы, на псевдокоде, на обычном языке, как мы записываем кулинарные рецепты, или как-нибудь еще. Но ведь нужен еще компилятор, переводящий эту запись в машинный код, а его не так-то просто разработать. Обычный разговорный язык слишком расплывчат, содержит множество неопределенных формулировок и излишних эмоциональных оценок, недоступных компьютеру. Поэтому алгоритмы записывают на специально разработанных языках программирования.

Всякий язык программирования должен иметь средства записи алгоритмов. Их называют операторами языка, минимальный набор которых должен содержать оператор для записи линейных вычислений, условный оператор для записи разветвлений и оператор цикла. Обычно для удобства записи алгоритмов в язык включаются несколько операторов: цикла, варианта, перехода и описания объектов.

Набор операторов языка С включает:

- операторы описания переменных и других объектов (*см. гл. 2*);
- операторы-выражения;
- операторы присваивания;
- условный оператор `if`;
- три оператора цикла `while`, `do-while`, `for`;
- оператор варианта `switch`;
- операторы перехода `goto`, `break`, `continue` и `return`;
- блок `{}`;
- пустой оператор — просто точка с запятой.

Здесь приведен не весь набор операторов С, он будет дополняться по мере изучения языка.

Всякий оператор завершается точкой с запятой. Можно поставить точку с запятой в конце любого выражения, и оно станет оператором (expression statement). Но в этом есть смысл только для операций присваивания, инкремента, декремента и вызовов функций. Например:

```
a = 6; x++; --y;
```

В остальных случаях превращение выражения в оператор бесполезно, потому что вычисленное значение выражения потеряется. В языке С линейное выполнение алгоритма обеспечивается последовательной записью операторов. Например:

```
int t, a = 10, b = 20;
t = a + b / (a + b); a += b * t;
b /= t;
```

Записав операторы один за другим, мы обеспечим их последовательное выполнение. Переход со строки на строку в исходном тексте не имеет никакого значения для компилятора, он осуществляется только для наглядности и читаемости текста. Компилятору весь исходный текст, записанный на языке С, представляется одной строкой символов. Символ перевода строки `\n`, а в некоторых системах два символа `\r\n`, появляющиеся при нажатии клавиши <Enter>, — для компилятора просто разделители, как и пробелы.

Блок

Блок операторов заключает в себя ноль или несколько операторов с целью использовать их как один оператор в тех местах, где по правилам языка можно записать только один оператор. Например, `{x = 5; y = 7;}`. Обратите внимание на то, что перед закрывающей фигурной скобкой стоит точ-

ка с запятой — это последний символ оператора присваивания `y = 7;`. После фигурной скобки точка с запятой не ставится. Впрочем, иногда вы увидите точку с запятой после фигурной скобки, но у нее совсем другой смысл.

Можно записать и *пустой блок*, просто пару фигурных скобок `{}`. Можно окружить оператор фигурными скобками, получив блок из одного оператора, например, `{a += 2;}`. Блоки операторов часто используются для ограничения области действия переменных и просто для улучшения читаемости текста программы.

Операторы присваивания

Точка с запятой в конце любой операции присваивания превращает ее в *оператор присваивания* (assignment statement). Побочное действие операции — присваивание — становится в операторе основным. Разница между операцией и оператором присваивания носит лишь теоретический характер. Присваивание гораздо чаще используется именно как оператор, а не операция. Очень часто в операторе присваивания используется несколько операций присваивания, идущих подряд:

```
x = y = z = 1;
```

Это позволяет задать одно и то же значение сразу нескольким переменным. Можно написать даже такой оператор:

```
x += y += z;
```

Вы часто увидите и такие экзотические присваивания:

```
x = a < b;
```

Здесь переменная `x` получит значение 0 или 1 в зависимости от результата сравнения.

Условный оператор

Условный оператор (if-then-else statement) в языке C записывается лаконично:

```
if (Выражение) Оператор1 else Оператор2
```

Скобки, окружающие выражение, записанное после слова `if`, обязательны. Пробелы вокруг скобок необязательны, скобки сами считаются разделителями. Оператор действует следующим образом. Сначала вычисляется выражение в скобках. Если его значение не равно нулю (выражение верно), то выполняется `Оператор1` и на этом действие условного оператора завершается, `Оператор2` не выполняется, далее будет выполняться следующий за `if` оператор. Если значение выражения равно нулю (выражение неверно), то действует `Оператор2`, при этом `Оператор1` не выполняется. Итак, выполняется блок-схема, показанная на рис. 3.1.

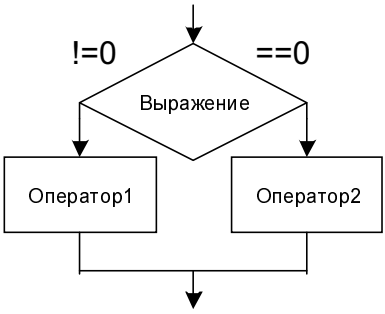


Рис. 3.1. Действие условного оператора

Условный оператор может быть сокращенным:

```
if (выражение) Оператор1
```

В случае нулевого выражения не выполняется ничего, как показано на рис. 3.2.

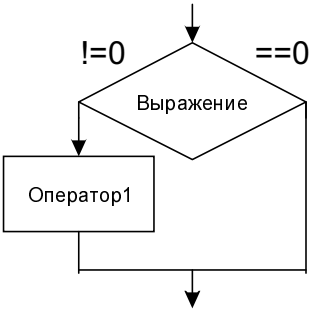


Рис. 3.2. Действие сокращенного условного оператора

Вот как можно, например, вычислить абсолютное значение вещественного числа `x`:

```
if (x < 0.0) x = -x;
```

Синтаксис языка не позволяет записывать несколько операторов ни после скобки, ни в ветви `else`. При необходимости составляется блок операторов, т. е. операторы окружают фигурными скобками. Пусть, например, переменной `x` надо дать большее из двух значений `a`, `b`, а переменной `y` — меньшее значение. Это можно сделать одним условным оператором:

```
if (a > b) {x = a; y = b;}
else {x = b; y = a;}
```

Многие специалисты рекомендуют всегда использовать фигурные скобки и размещать оператор на нескольких строках с отступами, как в следующем примере:

```
if (a < x){
    x = a + b;
} else {
    x = a - b;
}
```

Такое же правило принято во многих программистских коллективах. Это обеспечивает наглядность и облегчает добавление операторов в каждую ветвь при изменении алгоритма. Кроме того, многие текстовые редакторы позволяют одним нажатием клавиши перейти от открывающей фигурной скобки к соответствующей ей закрывающей скобке и обратно, и охватить весь блок целиком. Мы не будем строго следовать этому правилу, чтобы не увеличивать объем книги.

В каждой ветви условного оператора можно записать любые операторы, в том числе и условные, например:

```
if (n == 0){
    sign = 0;
} else if (n < 0){
    sign = -1;
} else {
    sign = 1;
}
```

Такая вложенность может быть продолжена, практически, без ограничений. Точнее говоря, стандарты языков программирования обычно не ставят пределы вложенности. У компиляторов такой предел есть, но они допускают более сотни вложенных друг в друга условных операторов. На практике такие вложенные условия никто писать не будет, т. к. это сильно уменьшает наглядность программы и замедляет ее выполнение. Более того, может возникнуть такая неопределенная ситуация (dangling else):

```
int ind = 5, x = 100;
if (ind >= 10) if (ind <= 20) x = 0; else x = 1;
```

Сохранит переменная `x` значение `100`, станет равной `0` или станет равной `1`? Будет выполняться блок-схема рис. 3.3 или блок-схема рис. 3.4?

И та, и другая схема имеет право на существование. Здесь нужно волевое решение, и общее для большинства алгоритмических языков, в том числе и языка С, правило таково: ветвь `else` относится к ближайшему слева условию `if`, не имеющему своей ветви `else`.

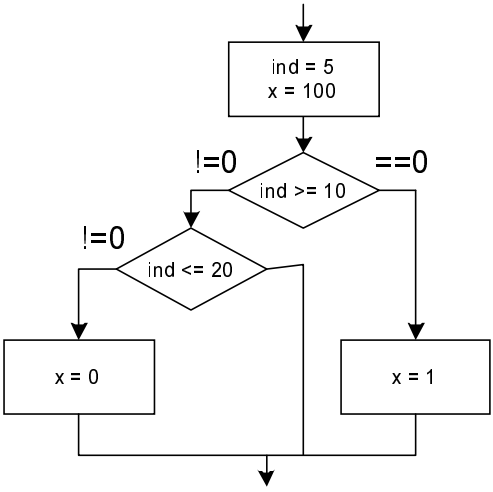


Рис. 3.3. Так не принято в языке C

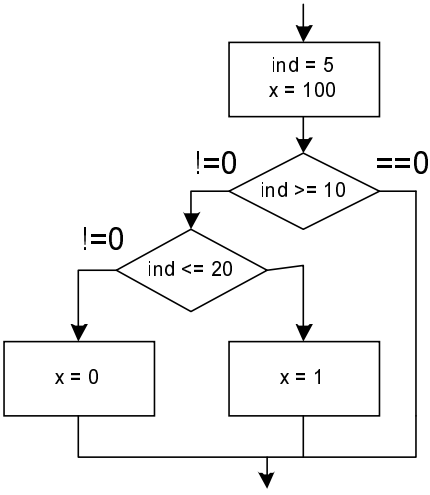


Рис. 3.4. Так принято в языке C

В нашем примере переменная `x` останется равной `100`. Изменить этот порядок можно с помощью блока:

```
if (ind >= 10) {if (ind <= 20) x = 0;} else x = 1;
```

Вообще говоря, не стоит увлекаться сложными вложенными условными операторами. Проверки условий занимают сравнительно много времени. По возможности лучше использовать логические операции.

Рассмотрим такую задачу. Пусть (x, y) — координаты точки M на плоскости. Величина z должна получить значение 1, если точка M находится внутри ромба с вершинами в точках (1, 1), (0, 1), (−1, 0), (0, −1), показанного на рис. 3.5, и значение 0, если точка М на границе ромба или вне него.

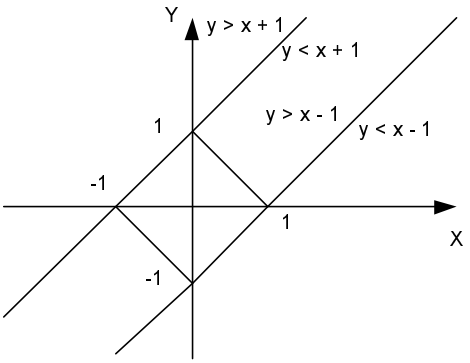


Рис. 3.5. Условие задачи

Приступим к решению этой задачи. Ромб образован отрезками четырех прямых, проходящих через вершины ромба. Их уравнения равны: $y = x + 1$, $y = x - 1$, $y = -x + 1$, $y = -x - 1$. Каждая прямая делит плоскость на две полуплоскости. В одной из полуплоскостей значение координаты y больше значения правой части соответствующего уравнения, в другой полуплоскости значение y меньше значения правой части. На рис. 3.5 показаны полуплоскости, на которые делят плоскость прямые $y = x + 1$ и $y = x - 1$. Поэтому первое, что приходит в голову, — написать вложенные условные операторы, каждый из которых проверяет, в какой полуплоскости расположена точка:

```
int z = 0;
if (y < x + 1)
   if (y < -x + 1)
      if (y > x - 1)
         if (y > -x - 1) z = 1;
```

Легко видеть, что все условия должны выполняться вместе, а значит, их можно объединить конъюнкцией:

```
int z = 0;
if (y < x + 1 && y < -x + 1 && y > x - 1 && y > -x - 1) z = 1;
```

Оператор стал проще, он будет выполняться быстрее. Не остановимся на этом и постараемся упростить условие. Заметим симметрию условий: коэф-

фициенты при x и 1 отличаются только знаком. Два уравнения, отличающиеся одним знаком, выделяют на плоскости полосу, пересечение двух полос и дает ромб. Показанная на рис. 3.5 полоса выделяется условиями: $x-1<y<x+1$. Эти два условия можно переписать следующим образом: $-1<y-x<1$.

Это двойное неравенство можно записать в виде одного условия, если применить абсолютную величину вещественного числа: $|y-x|<1$. Вторая пара условий таким же образом записывается в виде неравенства: $|y+x|<1$. После этого условный оператор можно записать так:

```
int z = 0;
if (fabs(y - x) < 1 && fabs(y + x) < 1) z = 1;
```

Оператор стал проще, но это еще не все. Теперь мы замечаем симметрию внутри абсолютной величины: и разность и сумма координат должны быть меньше единицы по абсолютной величине. Поскольку координаты могут быть и положительными и отрицательными, достаточно записать: $||y|+|x||<1$. Внешний модуль, разумеется, можно убрать, получив окончательно: $|y|+|x|<1$. Итак, достаточно проверить одно условие, записав окончательно:

```
int z = 0;
if (fabs(y) + fabs(x) < 1) z = 1;
```

Конечно, такое значительное упрощение получилось из-за особенности этой задачи, но о возможности упрощения условий надо думать всегда.

Упражнения

1. Решите эту же задачу для квадрата с вершинами в точках (1, 1), (−1, 1), (−1, −1), (1, −1) с наименьшим числом проверок.
2. Дана точка M с координатами (x, y). Надо величине z дать значение 0, если точка M на оси координат, значение 1, если точка M находится в первом квадранте, значение 2, если точка M во втором квадранте, значение 3, если точка M в третьем квадранте, и значение 4, если точка M в четвертом квадранте. Сделайте это за возможно меньшее число проверок.
3. Даны три числовые переменные x, y, z. Поменяйте их значения за возможно меньшее число перестановок, так, чтобы в результате $x <= y <= z$.
4. Запишите условие: точка M с координатами (x, y) находится в треугольнике с вершинами в точках (x_1, y_1), (x_2, y_2), (x_3, y_3).

Первая программа на языке С

Пришло время записывать полные программы на языке С, компилировать их, выполнять и отслеживать результаты выполнения программы. Это нетрудно сделать, если в вашем распоряжении есть любой текстовый редактор и компилятор с языка С. Надо только проследить за тем, чтобы текстовый редактор сохранял текст программы в байтовой кодировке, а не в графическом формате.

Все приведенные в этой книге программы были откомпилированы свободно распространяемым компилятором gcc, созданным сообществом GNU, и были выполнены в операционной системе GNU/Linux 2.4.24.

Всякая программа, записанная на языке С, — это одна или несколько функций, между которыми можно вставить определения внешних по отношению к функциям переменных и директивы препроцессора. Одна из функций должна называться `main()`, с этой функции начинает выполняться вся программа, из этой функции вызываются по мере необходимости другие функции. Как принято в литературе на языке С, я всегда буду после имени функции писать скобки, чтобы подчеркнуть, что это имя функции, а не простой переменной.

Содержимое всякой функции записывается после ее заголовка в фигурных скобках. С учетом этих правил простейшая программа на языке С выглядит так:

```
main(){}
```

Это реальная программа, ее можно откомпилировать и выполнить. Файлы, содержащие программы на языке С, как правило, снабжаются расширением .c. Пусть приведенная выше программа записана в файл prog.c, лежащий в текущем каталоге. Подаем файл на вход компилятора:

```
$ gcc prog.c
$
```

Знак доллара — это приглашение операционной системы, появляющееся в командной строке. Компилятор ничего не сообщает, по окончании его работы мы видим только новое приглашение операционной системы, но на выходе компилятор создает исполнимый файл со странным именем a.out. Этот файл компилятор записывает в текущий каталог. Отправляем его на выполнение:

```
$ ./a.out
$
```

Опять видим только новое приглашение операционной системы, поскольку программа ничего не делает. Давайте заставим программу выводить какой-то текст, это будет ставшая уже традиционной, фраза "Hello, world!"

В стандартной библиотеке языка С есть несколько функций вывода. Наиболее часто применяется функция форматированного вывода `printf()`. У нее переменное число аргументов, значения которых функция выводит на консоль. Аргументы могут быть выражениями разных типов, но первый аргумент — всегда указатель на строку символов. Об указателях речь пойдет в следующей главе, а пока воспользуемся тем, что строковая константа рассматривается компилятором как указатель и ее можно записать в качестве первого аргумента функции `printf()`.

Раз мы решили воспользоваться функцией из стандартной библиотеки, надо дать компилятору описание этой функции, так называемый *прототип*, и указать библиотеку, в которой лежит применяемая функция. Описание функции `printf()` записано в файл с именем stdio.h (standard input-output header file, файл заголовков стандартных функций ввода-вывода). Файлов с описаниями констант, макросов и функции, *файлов заголовков* довольно много, они обычно получают расширение h, их стандартное расположение — подкаталог usr/include каталога, в который инсталлирован С.

Для того чтобы компилятор увидел описание функции `printf()`, надо подключить к нашей программе содержимое файла stdio.h. Это делает вторая директива препроцессора — `#include`. В ней, в кавычках, надо написать полный путь к подключаемому файлу, но если файл лежит в стандартном каталоге, то достаточно написать его краткое имя, при этом надо записать имя в угловых скобках.

Имя библиотеки функций задается параметром `-l` командной строки компилятора. Для стандартной библиотеки со стандартным именем libc этот параметр можно не указывать, компилятор всегда знает имя и расположение стандартной библиотеки.

Вот теперь, после всех этих предварительных разъяснений, можно написать простейшую программу, выводящую на консоль приветствие "Hello, world!"

```
#include <stdio.h>
main(){
    printf("Hello, world!\n");
}
```

Записываем этот текст в файл с именем, допустим, hello.c, компилируем:

```
$ gcc hello.c
$
```

Теперь выполняем полученный исполнимый файл:

```
$ ./a.out
Hello, world!
$
```

Обратите внимание на действие символа перевода строки `\n`, стоящего в конце приветствия. Без него мы получили бы последнее приглашение операционной системы не на новой строке, а сразу же после выведенного значения.

Если вам не нравится, что все откомпилированные программы попадают в один и тот же файл a.out, в результате чего в этом файле хранится только программа, откомпилированная последней, дайте компилятору параметром `-o имя_файла` другое имя исполнимого файла. В следующем примере исполнимый файл получает имя hello без всякого расширения:

```
$ gcc hello.c -o hello
$ ./hello
Hello, world!
$
```

Функция вывода *printf()*

Вывод значений различных выражений с помощью функции `printf()` происходит немного сложнее. Сами выражения записываются вторым, третьим и так далее параметром через запятую, а в первом параметре — указателе на строку — специальными символьными выражениями — *спецификациями* — записывается формат вывода значения каждого выражения. Это выглядит примерно так:

```
#include <stdio.h>
main(){
   char c = '?';
   int k = 15;
   printf("c = %c, код c = %d, k = %d, k + c = %6d\n", c, c, k, k + c);
}
```

Спецификация начинается со знака процента и заканчивается одной из следующих букв, показывающих вид представления данного, формируемый при выводе:

- `c` — (character) — один символ;
- `d`, `i` — (decimal, integer) — целое число в десятичной системе счисления;
- `u` — (unsigned) — целое число без знака в десятичной системе счисления;
- `o` — (octal) — целое число без знака в восьмеричной системе счисления;
- `x`, `X` — (hexadecimal) — целое число без знака в шестнадцатеричной системе счисления. Если указан параметр `x`, то шестнадцатеричное число будет записано малыми буквами, например, `2f34cd`, если `X`, то выводимое число будет записано заглавными буквами: `2F34CD`;

- `f` — (float) — вещественное число будет выведено с фиксированной точкой с шестью цифрами в дробной части;
- `e`, `E` — (exponent) — вещественное число будет выведено с плавающей точкой с шестью цифрами в дробной части мантиссы. Число нормализуется так, что в целой части выводится одна ненулевая цифра. Если указан параметр `e`, то при выводе будет записана малая буква, например, `0.12345e-02`; если `E`, то получим `0.12345E-02`;
- `g`, `G` — (global) — будет выведено шесть значащих цифр вещественного числа с фиксированной или с плавающей точкой; если его порядок меньше −4 или больше его точности;
- `s` — (string) — строка символов, заканчивающаяся нуль-символом.

В написанном выше примере четыре спецификации — `%c`, `%d`, `%d`, `%6d`. Они применяются к выводимым данным в том порядке, в котором записаны в первом аргументе функции `printf()`. Первая спецификация `%c` применяется к символьной переменной `c` и формирует вместо себя вопросительный знак, вторая спецификация `%d` применяется к следующему аргументу `c` и формирует десятичный код 63 вопросительного знака. Третья спецификация `%d` применяется к целой переменной `k` и формирует десятичное число 15, а четвертая — `%6d` — к последнему аргументу — выражению `a + c`. Она дает число 78 в десятичной системе счисления.

Если число спецификаций меньше числа аргументов, то последние аргументы не выводятся. Если их число больше числа аргументов, то лишние спецификации выведут мусор, лежащий в следующих ячейках оперативной памяти.

На месте спецификаций в выводимой строке появляются сформированные ими значения, причем пробелы между значениями не формируются автоматически, их надо указывать специально. Остальные символы выводятся так, как записаны в первом аргументе функции `printf()`. Итак, приведенная выше программа даст следующий вывод:

```
c = ?, код c = 63, k = 15, a + c =    78
```

Последняя спецификация `%6d` немного отличается от остальных. Она содержит число 6. Это число позиций на консоли, выделяемых для выводимого значения. Если значение короче, скажем, в нашем примере число 78 занимает только две позиции, то оно сдвигается вправо, слева остаются пробелы. Если записать в числовой спецификации ноль, например, `%06d`, то вместо пробелов будут выведены нули, мы получим `000078`. Если записать дефис, `%-6d`, то значение будет сдвигаться влево, в нашем примере пробелы останутся после числа 78.

Число позиций можно указать не только с помощью `%d`, но и в любой другой спецификации. Оно задает минимальную ширину поля, в которое по-

мещается выводимое значение. Если выводимое значение требует больше позиций, чем указано в спецификации, то поле расширяется до необходимых размеров.

После числа позиций через точку можно записать еще одно число, причем первое число можно и не записывать. Для целых значений число после точки будет означать минимальное количество выводимых цифр, например, `%10.5d`, `%4.3u`, `%.2o`, `%6.6x`, `%.6X`. Если значение короче, то слева будут выведены незначащие нули.

Для вещественных спецификаций (`%f`, `%e`, `%E`) число после точки означает количество выводимых цифр в дробной части. Например, `%5.2f`, `%10.5e`, `%.5E`. Для спецификации `%g`, `%G` оно означает количество выводимых значащих цифр, например, `%8.2g`, `%.8G`.

Если число после точки указано в спецификации `%s`, то выводятся только первые символы строки, например, при выводе

```
printf("%.6s\n", "Первые символы");
```

получим только слово `Первые`.

У чисел со знаком обычно выводится только минус, плюс не выводится. Если в спецификации поставить плюс, например, `%+8d`, то числа будут показаны со своим знаком. Для нечисловых значений плюс игнорируется. Если же спецификацию начать с пробела, `% d`, то перед положительным числом не ставится плюс, но останется пробел, что удобно для выравнивания столбцов в таблице чисел.

Для вывода длинных целых значений типа `long` перед буквами `d`, `i`, `o`, `u`, `x`, `X`, завершающими спецификацию, надо записать малую букву `l`, иначе функция `printf()` возьмет только первые байты длинного значения и сформирует из них выводимое число. Для вывода значений типа `double` перед буквами `f`, `e`, `E`, `g`, `G` надо поставить малую букву `l`, иначе функция `printf()` возьмет только первые байты вещественного числа с удвоенной точностью. Для вывода значений типа `long double` перед буквами `f`, `e`, `E`, `g`, `G` надо поставить заглавную букву `L`, иначе функция `printf()` возьмет только первые байты.

Наконец, если в начале спецификации поставить знак номера (#), например, `%#o`, `%#5x`, `%#7.2f`, `%#E`, `%#g`, то восьмеричные числа будут выводиться с начальным нулем, ненулевые шестнадцатеричные числа будут начинаться с `0x` или `0X`, у вещественных чисел будет ставиться точка после целой части, даже если в дробной части нет цифр. Спецификации `%g`, `%G`, кроме того, выводят не значащие хвостовые нули.

Итак, знак процента в первом аргументе функции `printf()` имеет особое значение. Он означает начало спецификации. После него компилятор будет искать одну из букв, завершающих спецификацию. Если ее нет, то результат

трудно предугадать. Если же в тексте встречается знак процента, то его надо удвоить. Например:

```
printf("Если ratio < 80%%, то увеличьте кеш\n");
```

Упражнения

5. Напишите простенькую программу, содержащую переменные разных типов и функцию `printf()` со всевозможными спецификациями, выводящую значения этих переменных и выражений с ними. Посмотрите внимательно на полученный вывод, особенно для значений типа `long`, `double` и `long double`.
6. Попробуйте записать в функции `printf()` подряд несколько аргументов, содержащих выражения `++k`, и убедитесь, что они не обязательно вычисляются слева направо.

Функция ввода *scanf()*

Теперь научимся вводить исходные данные с клавиатуры. В стандартной библиотеке функций ввода-вывода, описанной в файле stdio.h, есть несколько функций, предназначенных для ввода исходных данных. Мы их изучим постепенно, а пока будем пользоваться функцией форматированного ввода `scanf()`.

Почему надо форматировать ввод? С точки зрения компьютера все, что вы вводите с клавиатуры, — это строка символов. Даже когда вы набираете число 12 345, программа получает после нескольких преобразований пять байтов с кодами нажатых клавиш, в этом примере: 00110001001100100011001100110101000011010 1. Эти пять байтов надо преобразовать в четыре байта, хранящих целое число типа `int`. Этим-то и занимается функция `scanf()`.

Функция `scanf()` выглядит аналогично функции `printf()`. Ее первый аргумент тоже указатель на строку формата, содержащую те же спецификации, но следующие аргументы — не переменные или выражения, а адреса переменных, в которые будут записаны отформатированные исходные данные. Об адресах мы будем говорить в следующей главе, а пока для того чтобы получить адрес переменной, просто ставьте перед ней амперсанд. Например:

```
main(){
   int a, k = a;
   float x;
   scanf("%d%d%f", &a, &k, &x);
}
```

Как видите, в строке формата между спецификациями нет никаких разделителей. В таком случае вводимые данные надо разделять пробелами. Функция начнет выполняться и читать набранные на клавиатуре символы после нажатия клавиши <Enter>.

Всякий символ, записанный между спецификациями, считается разделителем и должен вводиться с клавиатуры. Например, записана функция:

```
scanf("%d,%d,%f", &a, &k, &x);
```

В этом случае вводимые данные надо разделять запятыми и необязательными пробелами. Разделять данные можно, регулируя ширину поля, выделенного для них. Пусть, например, написана функция:

```
scanf("%2d%3d%5.2f", &a, &k, &x);
```

Мы набрали цифры 1234567890. После этого переменная `a` получит значение 12, в переменную `k`, будет записано значение 345, а переменная `x` получит значение 678.9.

Так же, как и в функции `printf()`, при вводе значений длинных переменных типа `long` и `double` перед буквой соответствующей спецификации надо поставить букву `l`, а при вводе значений типа `long double` — букву `L`. Некоторые сложности возникают при вводе символьных данных. Ведь всякий разделитель — это символ, который будет считаться вводимым значением. В данном примере набирать символы надо подряд без всяких разделителей:

```
char ch1, ch2, ch3;
scanf("%c%c%c", &ch1, &ch2, &ch3);
```

Если записать следующую функцию, то всякий разделитель целых значений попадет в символьную переменную `ch1`:

```
scanf("%d%c%d", &k &ch1, &k);
```

Упражнение

7. Напишите простенькую программу, содержащую переменные разных типов, и функцию `scanf()` со всевозможными спецификациями, заносящую значения в эти переменные. Выведите значения переменных функцией `printf()` и посмотрите внимательно на полученный вывод. Убедитесь в том, что спецификации должны точно соответствовать типу переменной.

Нахождение корней квадратного уравнения

Теперь, вооружившись функциями `scanf()` и `printf()`, приведем полный пример программы, содержащей многочисленные проверки. В листинге 3.1 вычисляются корни квадратного уравнения $ax^2 + bx + c = 0$ для любых коэффициентов, в том числе и нулевых. Эта программа реализует блок-схему рис. 1.4.

Листинг 3.1. Вычисление корней квадратного уравнения

```
#include <stdio.h>
#include <math.h>
main(){
   float a = 0.5, b = -2.7, c = 3.5, d, eps=1e-5;
   printf("Введите точность вычислений: ");
   scanf("%f", &eps);
   printf("\nВведите коэффициенты уравнения через пробелы: ");
   scanf("%f%f%f", &a, &b, &c);
   if (fabs(a) < eps)
      if (fabs(b) < eps)
        if (fabs(c) < eps) printf("\nРешение - любое число.\n");
        else printf("\nРешений нет.\n");
      else printf("x1 = x2 = %f\n". -c / b);
   else          // Коэффициенты не равны нулю
      if ((d = b*b - 4*a*c)< 0.0){ // Комплексные корни
         d = 0.5 * sqrt(-d) / a;
         a = -0.5 * b/ a;
         printf("\nx1 = %f +i%f, x2 = %f - i%f\n", a, d, a, d);
      } else {      // Вещественные корни
         d = 0.5 * sqrt(d) / a;
         a = -0.5 * b / a;
         printf("\nx1 = %f, x2 = %f\n", a + d,  a - d);
      }
}
```

В этой программе использована функция вычисления модуля `fabs()` вещественного числа и функция извлечения квадратного корня `sqrt()` из вещественного числа. Это функции из стандартной библиотеки математических функций языка С. Поскольку все вычисления с вещественными числами производятся приближенно, мы считаем, что коэффициент уравнения равен нулю, если его модуль меньше 0.00000001. Обратите внимание на то, как операция присваивания при вычислении дискриминанта вложена в сравнение с нулем.

Компилятору необходимо указать в параметре `-l` имя использованной библиотеки математических функций. Ее имя libm. Для таких стандартных имен, начинающихся с lib, достаточно написать последнюю букву. Итак, если программа листинга 3.1 записана в файл listing31.c, то откомпилировать ее можно, набрав командную строку

```
$ gcc listing31.c -o listing31 -lm
```

Операторы цикла

Основной оператор цикла — оператор `while` — в языке C выглядит чрезвычайно лаконично:

```
while (выражение) оператор
```

Скобки, окружающие выражение, обязательны, пробелы, как мы уже говорили, необязательны, потому что скобки — это тоже разделители.

Оператор работает так. Вначале вычисляется выражение в скобках. Если его значение не равно нулю (выражение верно), то выполняется `оператор`, образующий цикл. Затем снова вычисляется `выражение` и действует `оператор`, и так до тех пор, пока выражение не станет равным нулю (станет неверным). Если выражение изначально равно нулю, то `оператор` не будет выполнен ни разу. Таким образом, оператор `while` реализует блок-схему рис. 3.6.

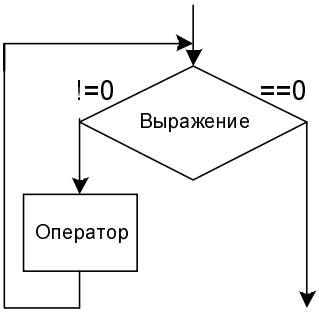


Рис. 3.6. Действие оператора цикла `while`

Предварительная проверка обеспечивает безопасность выполнения цикла, позволяет избежать переполнения, деления на ноль и других неприятностей. Поэтому оператор `while` является основным, а в некоторых языках программирования и единственным оператором цикла.

Оператор в цикле может быть и пустым, например:

```
int i = 0;

double s = 0.0;

while ((s += 1.0 / ++i) < 10);
```

Этот фрагмент кода вычисляет количество сложений `i`, которые необходимо сделать, чтобы гармоническая сумма `s` достигла значения 10. Такой стиль характерен для языка C. Не стоит им увлекаться, чтобы не превратить текст

программы в шифровку, на которую вы сами через пару недель будете смотреть с недоумением.

Можно организовать и бесконечный цикл — `while (1) оператор`. Конечно, из такого цикла следует предусмотреть какой-то выход, например, оператором `break`, как в листинге 3.2. В противном случае программа зациклится и вам придется прекращать ее выполнение "комбинацией из трех пальцев" <Ctrl>+<Alt>+<Del> в Windows 95/98/ME, комбинацией <Ctrl>+<C> в UNIX или с помощью Task Manager в Windows NT/2000/XP/2003.

Если в цикл надо включить несколько операторов, то следует образовать блок операторов `{}`.

Второй оператор цикла — оператор `do-while` — имеет вид:

```
do оператор while (выражение);
```

Здесь сначала выполняется оператор, а потом происходит вычисление выражения. Цикл выполняется, пока `выражение` остается не равным нулю (верным). Блок-схема этих действий показана на рис. 3.7.

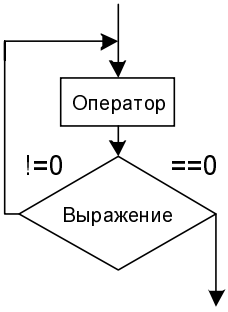


Рис. 3.7. Действие оператора цикла `do-while`

Существенное различие в организации циклов этими двумя способами в том, что в последнем оператор обязательно выполнится хотя бы один раз. Например, пусть задана какая-то непрерывная функция $f(x)$, имеющая на отрезке $[a,b]$ ровно один корень. В листинге 3.2 приведена программа, вычисляющая этот корень приближенно методом деления пополам. Этот же метод часто называют методом *бисекции*, используя английский термин, или, по-гречески, методом *дихотомии*.

Суть метода в том, что заданный отрезок делится пополам и выясняется, в какой половине находится корень. Это можно сделать проверкой значений функции на концах каждой половины. Там, где лежит корень, значения непрерывной функции имеют разные знаки. После этого выбранная половина

отрезка снова делится пополам, и все продолжается до тех пор, пока не будет достигнута нужная точность.

Поскольку деление пополам надо сделать хотя бы один раз, в листинге 3.2 выбран цикл do-while.

Листинг 3.2. Нахождение корня нелинейного уравнения методом бисекции

```
#include <stdio.h>

double f(double x){
    return x*x*x - 3*x*x + 3; // Если хотите, напишите
                              // что-нибудь другое…
  }

main(){
    double a = 0.0, b = 1.5, c, y, eps = 1e-8;

    do{
      c = 0.5 *(a + b); y = f(c);

      if (fabs(y) < eps) break;
      // Корень найден. Выходим из цикла.

      // Если на концах отрезка [a, c]
      // функция имеет разные знаки:
      if (f(a) * y < 0.0) b = c;
        // значит, корень здесь. Переносим
        // точку b в точку c.

        // В противном случае:
      else a = c;
```

```
            // переносим точку a в точку c.

            // Продолжаем, пока отрезок [a, b] не станет мал.
        } while(fabs(b-a) >= eps);

        printf("x = %f,  f(%f) = %f\n", c, c, y);
    }
```

Эта программа сложнее предыдущих примеров: в ней кроме функции `main()` есть еще функция `f()`. Эта функция очень проста: она вычисляет значение многочлена и возвращает его в качестве значения функции, причем все это выполняется одним оператором:

`return выражение;`

В функции `main()` появился еще один новый оператор `break`, который просто прекращает выполнение цикла, если мы по счастливой случайности наткнулись на приближенное значение корня.

Упражнения

8. Напишите программу, подсчитывающую количество цифр заданного целого числа.
9. Напишите программу, дающую сумму цифр заданного целого числа.
10. Напишите программу, переставляющую цифры заданного целого числа в обратном порядке. Вы можете воспользоваться алгоритмом на рис. 1.8.
11. Напишите программу, подсчитывающую сумму величин, обратных квадратам первых натуральных чисел $S = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots$, с заданной точностью.

Приведем еще один полный пример типичного вычисления. Известно, что точность вычисления суммы первых членов сходящегося знакопеременного ряда можно оценить абсолютной величиной последнего отброшенного члена. Применим эту теорему для вычисления синуса заданного числа с заданной точностью `eps` по известной формуле:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{(-1)^n x^{2n+1}}{(2n+1)!} + \ldots .$$

Эти вычисления проводятся в программе листинга 3.3. Они сравниваются со значением синуса того же аргумента, вычисленным функцией `sin(x)` из стандартной библиотеки языка C.

Обратите внимание на то, что перемена знака слагаемых обеспечивается тем, что мы просто ставим знак минус при вычислении каждого следующего слагаемого `p`. Не надо вычислять никакую степень минус единицы.

Вторая особенность программы заключается в том, что мы не вычисляем полностью следующее слагаемое `p`, а получаем его из предыдущего слагаемого умножением на квадрат аргумента `x` и делением на произведение следующих целых чисел `k*(k+1)`. Это вызвано тем, что аргумент `x` может быть довольно велик по абсолютной величине и возведение его в большую степень быстро вызовет переполнение. Это же неизбежно случится при вычислении факториала. Мы же вычисляем отношение сравнительно небольших величин, что не должно вызвать переполнение.

Листинг 3.3. Вычисление синуса с помощью отрезка ряда Тейлора

```
#include <stdio.h>

main(){
   float eps = 1e-8, x= 0.0;
   printf("Введите точность вычислений: ");
   scanf("%f", &eps);
   printf("Введите аргумент синуса: ");
   scanf("%f", &x);
   double s = 0.0, p = x;
   int k = 2;

   while (fabs(p) >= eps){
      s += p;
      p = -p * x * x / (k * (k + 1));
      k += 2;
   }
   printf("x = %f, s = %f, sin = %f"\n", x, s, sin(x));
}
```

Третий оператор цикла — оператор `for` — выглядит так:

```
for (списокВыр1; выражение; списокВыр2) оператор
```

Перед выполнением цикла вычисляется список выражений `списокВыр1`. Это ноль или несколько выражений, перечисленных через запятую, т. е. операция "запятая". Они вычисляются слева направо, и в следующем выражении уже можно использовать результат предыдущего выражения. Как правило, здесь задаются начальные значения переменным цикла.

Затем вычисляется второе выражение `выражение`. Если оно отлично от нуля, то действует `оператор`, потом вычисляются слева направо выражения из списка выражений `списокВыр2`. Далее снова проверяется `выражение`. Если оно не равно нулю, то выполняется `оператор` и `списокВыр2` и т. д. Как только `выражение` станет равным нулю, выполнение цикла заканчивается. Короче говоря, выполняется последовательность операторов:

```
списокВыр1;

while (выражение){

    оператор

    списокВыр2;

}
```

с тем исключением, что, если оператором в цикле является оператор `continue`, то `списокВыр2` все-таки выполняется.

Многие компиляторы допускают сделать в `списокВыр1` одно определение переменных обязательно с начальным значением. Такие переменные известны только в пределах этого цикла.

Любая часть оператора `for` может отсутствовать: цикл может быть пустым, выражения в заголовке тоже, при этом точки с запятой сохраняются. Можно задать бесконечный цикл: `for (; ;) оператор`. В этом случае в теле цикла следует предусмотреть какой-нибудь выход.

Хотя в операторе `for` заложены большие возможности, используется он, главным образом, для перечислений, когда их число известно, например:

```
int s = 0, k;

for (k = 1; k <= N; k++) s += k * k;
```

Этот фрагмент кода вычисляет сумму квадратов первых `N` натуральных чисел. Посмотрите, какими разными способами можно записать эти операторы.

```
int s = 0, k = 1;

for (; k <= N; k++) s += k * k;
```

Здесь все начальные действия вынесены из оператора цикла. На их месте в заголовке цикла осталась только точка с запятой.

```
int s = 0, k = 1;

for (; k <= N;) s += k * (k++);
```

Здесь изменение переменной `k` вынесено в цикл, в заголовке осталось только условие. Можно сделать наоборот и внести вычисления в заголовок:

```
int s = 0, k = 1;

for (; k <= N; s += k * (k++));
```

Можно воспользоваться операцией "запятая" и написать так:

```
int s = 0, k = 1;

for (; k <= N; s += k * k, k++));
```

Наконец, можно все действия внести в заголовок цикла:

```
int s, k;

for (s = 0, k = 1; k <= N; s += k * k, k++));
```

Не увлекайтесь такими необычными конструкциями. Самолетостроители говорят: "Красивый самолет лучше летает". В программировании действует такое же недоказанное мистическое правило: чем проще и красивее программа, тем лучше она работает. Старайтесь записывать операторы в том же виде, как это делали создатели компилятора. Будьте уверены: они проверили свои записи сотни раз и довели их до совершенства!

Упражнения

12. Напишите полную программу, в которой проверьте действие всех способов записи этого цикла.
13. Напишите программу, подсчитывающую сумму величин, обратных квадратам первых ста натуральных чисел.
14. Число $\frac{\pi}{4}$ можно вычислить как сумму знакопеременного ряда $1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\ldots$, но она очень медленно сходится. Какую точность даст сумма первой тысячи слагаемых этого ряда?

Оператор *continue*

Оператор `continue` используется только в операторах цикла. Он состоит только из одного слова и осуществляет немедленный переход к следующей итерации цикла. В следующем фрагменте кода данный оператор позволяет обойти деление на ноль:

```
int i, j = N / 2;

for (i = 0; i < N; i++){

   if (i == j) continue;

   s += 1.0 / (i - j);

}
```

Когда значение переменной `i`, возрастая, сравняется со значением переменной `j`, соответствующее слагаемое не будет вычисляться и прибавляться к `s`, а сразу же произойдет переход к следующей итерации, в которой `i` уже будет больше, чем `j`.

Оператор *break*

Оператор `break` используется в операторах цикла и операторе варианта для немедленного выхода из этих операторов. Это удобно, например, при поиске в цикле заданного значения. Как только оно будет найдено, нужно сразу прекратить выполнение цикла и выйти из него.

Оператор `break`, как и оператор `continue`, состоит всего из одного слова. Он используется, чаще всего, в условных операторах. Например:

```
double s = 0.0;
int k = 1;
while(1){
   s += 1.0 / k++;
   if (s > 10.0) break;
}
printf("k = %d\n", k);
```

Оператор варианта *switch*

Оператор варианта `switch` реализует разветвление на несколько ветвей. Каждая ветвь отмечается константой или константным выражением какого-либо целого типа и выбирается, если значение определенного выражения совпадет с этой константой. Вся конструкция выглядит так.

```
switch (Целое выражение){
   case КонстВыр1: Оператор1
   case КонстВыр2: Оператор2
      . . . . . . . . .
   case КонстВырN: ОператорN
   default: ОператорDef
}
```

Стоящее в скобках `целое выражение` может быть типа `char`, `short`, `int`, `long`. Оператор варианта выполняется так. Все константные выражения вычисляются заранее, на этапе компиляции, и должны иметь отличные друг от

друга значения. Сначала вычисляется целочисленное выражение `Целое выражение`. Если оно совпадает с одной из констант, то выполняется оператор, отмеченный этой константой. Затем выполняются (fall through labels) все следующие операторы, включая и `ОператорDef`, и выполнение оператора варианта заканчивается.

Если же ни одна константа не равна значению выражения, то выполняется `ОператорDef` и все следующие за ним операторы. Поэтому ветвь `default` должна записываться последней строкой оператора `switch`, хотя синтаксис не запрещает ей располагаться и между другими вариантами. Ветвь `default` может отсутствовать, тогда в этой ситуации оператор варианта вообще ничего не делает. Схематически работа оператора `switch` показана на рис. 3.8.

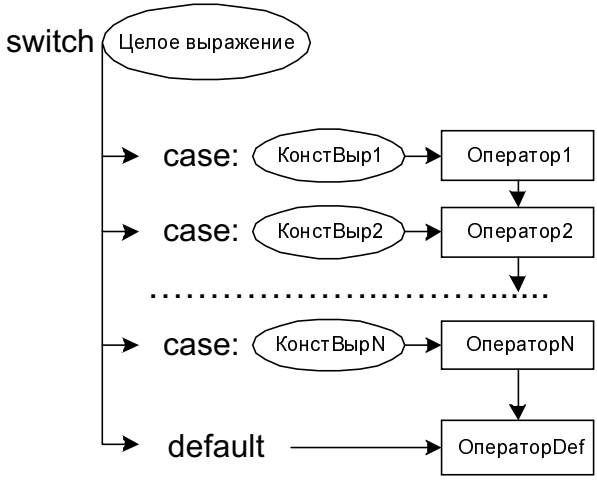


Рис. 3.8. Работа оператора варианта

Таким образом, константы в вариантах `case` играют роль только меток, точек входа в оператор варианта, а далее выполняются все оставшиеся операторы в порядке их записи. Чаще всего необходимо выполнить только одну ветвь операторов. В таком случае используется оператор `break`, сразу же прекращающий выполнение оператора `switch`. С другой стороны, может понадобиться выполнить один и тот же оператор в разных ветвях `case`. В этом случае ставятся несколько меток `case` подряд. Вот простой пример, в котором переменная `dayOfWeek` хранит день недели в виде целого числа от 1 до 7, а нам надо получить сообщение о том, рабочий это день или выходной.

```
switch (dayOfWeek){

  case 1: case 2: case 3: case 4:case 5:

      printf("Рабочий день\n"); break;
```

```
  case 6: case 7:
      printf("Выходной день\n"); break;
  default:
      printf("День задан неверно\n");
}
```

Внимание!

Не забывайте завершать варианты оператором `break`!

Оператор перехода *goto*

Оператор перехода `goto` выглядит очень просто:

```
goto метка;
```

Он вызывает немедленный переход к некоторому оператору, на который указывает `метка`. *Метка* записывается, как все идентификаторы, буквами, цифрами и знаками подчеркивания, но не требует никакого описания. Метка ставится перед оператором или открывающей фигурной скобкой и отделяется от них двоеточием. Так получается *помеченный оператор* или *помеченный блок*. Вот классический пример зацикливания, в котором переменная `x` будет бесконечно получать значение `0`:

```
M: x = 0;
goto M;
```

Можно написать даже так:

```
M: goto M;
```

Никакого нарушения синтаксиса здесь нет, компилятор ничего не сообщит, он не считает это ошибкой. Раз человек так написал, значит надо сгенерировать машинный код и выполнить его.

Подобные ошибки встречаются на удивление часто, конечно, не в таком простом виде, как написано выше. Никто в трезвом уме и добром здравии не напишет такой код, но если оператор `goto` и помеченный оператор находятся на достаточном удалении друг от друга, строк на 50—70, то подобную ошибку допустить очень легко, а найти и исправить очень трудно.

Кроме того, резкий переход к произвольному оператору нарушает плавное течение программы, затрудняет слежение за ее выполнением и способствует появлению ошибок. Все это делает оператор `goto` нежелательным для использования.

В 70-х годах прошлого века шла бурная дискуссия по поводу оператора `goto`. Противники ратовали за полное изгнание его из языков программи-

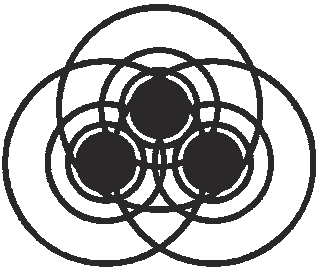
рования, они придумали для его замены операторы `continue` и `break`, доказывая, что их вполне достаточно для изменения хода программы. Они объявили, что квалификация программиста обратно пропорциональна количеству операторов `goto` в написанном им коде. Защитники приводили примеры ситуаций, в которых он необходим. Дискуссия закончилась компромиссом. Было решено, что оператор `goto` в некоторых ситуациях полезен, но лучше им не злоупотреблять.

Упражнения

15. Найдите алгоритм, подсчитывающий количество цифр заданного целого числа.
16. Найдите алгоритм, дающий сумму цифр заданного целого числа.
17. Целое шестизначное число считается "счастливым", если сумма трех его старших цифр равна сумме трех младших цифр. Создайте алгоритм, определяющий, будет заданное целое число "счастливым", или нет.

Контрольные вопросы

1. Как можно записать разветвление на три ветви с помощью условного оператора `if`?
2. Даны три переменные `a`, `b`, `c`. Как наименьшим числом условных операторов поменять их значения так, чтобы в результате `a < b < c`?
3. Можно ли в языке С записать оператор цикла `for (;;) x = 0;`?
4. Сколько раз будет выполнен цикл `while (k++ < 10) k++;` если перед циклом `k == 1`?
5. Каким образом можно прекратить выполнение цикла?
6. Приведите примеры зацикливания.
7. Обязательна ли конструкция `default: оператор;` в операторе варианта `switch`?
8. Что случится, если поставить конструкцию `default: оператор;` первым вариантом оператора `switch`?
9. Какую роль играют операторы `break` в операторе варианта `switch`?



Глава 4

Массивы и указатели

Многие задачи, решение которых находится с помощью компьютера, требуют просмотра и обработки большого количества однотипных данных. Такие данные в науке и технике обозначаются переменными с индексами: x_1, a_0, x_k, M_{ij} и т. д. Для ускорения просмотра однотипные данные лучше всего поместить в соседние ячейки оперативной памяти, чтобы не тратить время на их поиск. Так возникает понятие массива.

Массив — это совокупность переменных одного типа, хранящихся в смежных ячейках оперативной памяти.

Это определение приводит к следующим особенностям массива.

- Все переменные, образующие массив, одного типа, у них одинаковая длина. Для перехода к следующему значению достаточно передвинуться по оперативной памяти на фиксированное количество байтов, равное длине типа. Это очень быстрая операция.
- Поскольку тип всех элементов массива одинаков, они обозначаются одним и тем же именем, а различаются с помощью индексов.
- Для хранения массива надо выделить непрерывный участок оперативной памяти. Поэтому массив должен быть объявлен заранее; при объявлении надо указать тип элементов массива и их количество, чтобы можно было подсчитать размер выделяемого участка оперативной памяти.

Итак, количество элементов массива, называемое его *длиной*, должно быть известно заранее. Это довольно сильное ограничение. В большинстве задач количество элементов, которые надо обработать с помощью массива, неизвестно заранее. В таких случаях приходится создавать массив с наибольшей предполагаемой длиной, что приводит к перерасходу оперативной памяти. Некоторые языки программирования позволяют определять

массивы с переменной длиной, но в их основе всегда лежит выделение максимального количества оперативной памяти.

Массивы

В языке С массивы создаются обычными операторами определения типа вместе с обычными переменными или в отдельных операторах. При определении массива после его имени в квадратных скобках записывается его длина. Например,

```
double a[5], b[128], c1 = 0.0;
char c[1000];
int p[100], q[100];
double x = 1.0, y[1024];
```

Индексы массива всегда начинаются с 0, увеличиваясь на 1, так что последнее значение индекса меньше его длины на единицу. Так, переменная с наибольшим индексом в определенных выше массивах — это `a[4]`, `b[127]`, `c[999]`, `p[99]`, `q[99]`, `y[1023]`. Поначалу это сбивает с толку. Определенный выше массив `a` состоит из пяти переменных `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`. Элемента `a[5]` в массиве нет!

Дело усугубляется тем, что компилятор языка С не проверяет границы массива. Вы можете использовать в программе переменную `a[5]`. Компилятор не сообщит об ошибке. Просто для `a[5]` будет взято значение, хранящееся в следующей после `a[4]` ячейке. Как правило, там лежит какой-то "мусор", который и окажется значением переменной `a[5]`. Хорошо, если это значение резко отличается от всех других значений массива, и вы заметите ошибку. Хуже, если оно сравнимо с остальными значениями. Программа может долго работать и давать неверные результаты, прежде чем вы это увидите.

Индексы можно задавать любыми целочисленными выражениями, например, `a[i + j]`, `a[i % 5]`, `a[++i]`, `b['?']`, `x['9'-'0']`. При этом надо еще тщательнее следить за тем, чтобы индекс не оказался отрицательным и не превзошел объявленную границу массива.

Перед использованием массива надо дать всем его элементам начальные значения, например:

```
a[0] = 0.01; a[1] = -3.4; a[2] = 2.89; a[3] = 4.5; a[4] = -6.7;
for (int i = 0; i < 128; i++) b[i] = 1.0 / i;
for (int i = 0; i < 100; i++) p[i] = 2 * i + 1;
```

Если этого не сделать, то в элементах массива окажется мусор, взятый из соответствующих ячеек оперативной памяти. В некоторых случаях элементы массива автоматически обнуляются, но об этом мы поговорим позже.

При объявлении массива можно сразу задать начальные значения его элементов, записав их в фигурных скобках через запятую в виде констант или константных выражений. При этом даже необязательно указывать количество элементов массива, его длина будет равна количеству начальных значений:

```
double a[] = {0.01, -0.02, 0.89, 44.5e-3, -68.75};
```

Здесь определен массив `a` из 5 элементов — вещественных переменных удвоенной точности. Их начальные значения: `a[0] == 0.01`; `a[1] == -0.02`; `a[2] == 0.89`; `a[3] == 44.5e-3`; `a[4] == -68.75`;

Если же длина массива указана явно, то количество начальных значений не должно ее превышать. Можно задать меньше начальных значений, чем длина массива. Эти начальные значения присвоят первые элементы массива, а последние элементы получат нулевое значение. Вот, например, массив из 10 символьных переменных:

```
char c[10] = {'a', 'b', 'c'};
```

Здесь заданы значения `c[0] == 'a'`, `c[1] == 'b'` и `c[2] == 'c'`. Остальные элементы `c[3]`, `c[4]`,, `c[9]` будут равны `'\0'`.

Элементы массива — это обычные переменные своего типа и с ними можно выполнять все операции, входящие в этот тип. Например:

```
a[1] += 2 * a[k] - 3 * a[k + 1] / a[k - 1];
```

Обработка массива, как правило, производится в циклах. Вот как обычно вычисляется сумма всех элементов массива.

```
double s = 0.0;
int k;
for (k = 0; k < 128; k++) s += b[k];
```

В листинге 4.1 показано, как можно вычислить диапазон значений массива.

Листинг 4.1. Вычисление диапазона значений массива

```
#include <stdio.h>
#include <math.h>
#define LENGTH 100
main(){
    double a[LENGTH], x = 0.1234;
    int i;

    for (i = 0; i < LENGTH; i++) a[i] = i * sin(M_PI * i * x);

    double amin = a[0], amax = amin;
```

```
    for (i = 1; i < LENGTH; i++){
       if (a[i] < amin) amin = a[i];
       if (a[i] > amax) amax = a[i];
    }
    double range = amax - amin;

    printf("Диапазон значений = %f\n", range);
}
```

Здесь использована константа `M_PI`, определенная в файле math.h и дающая приближенное значение числа.

Очень часто надо знать не значение наибольшего или наименьшего элемента массива, а только его индекс. Вот как можно его найти:

```
int i, ind = 0;
for (i = 1; i < 5; i++)
   if (a[i] > a[ind]) ind = i;
```

Очень часто приходится выводить на консоль все значения какого-нибудь массива. При этом удобно вывести на одной строке несколько значений. Следующий пример показывает, как можно вывести массив `a` по пять значений в строке:

```
for (k = 0; k < LENGTH; k++){
  printf("a[%d] = %f ", k, a[k]);
  if (k % 5 == 4) printf("\n");
}
```

Упражнения

1. Найдите сумму абсолютных значений всех элементов массива.
2. Найдите наибольший и наименьший по абсолютной величине элемент массива, а также их индексы.
3. Подсчитайте количество элементов массива, превышающих по абсолютной величине наибольший из первых десяти элементов массива.
4. Найдите в массиве заданное значение `x`. Сообщите его индекс или −1, если такого значения в массиве нет.

Многомерные массивы

Элементами массивов могут быть не только переменные простых типов, но и снова массивы. Можно объявить:

```
char c[3][4];
```

Компилятор поймет эту запись как определение массива `c` из трех элементов, причем каждый элемент `c[0]`, `c[1]` и `c[2]` — это массив типа `char`, состоящий из четырех элементов. В частности, `c[0]` — это имя массива, элементы которого, как обычно, различаются индексами, записанными в квадратных скобках: `c[0][0]`, `c[0][1]`, `c[0][2]`, `c[0][3]`.

Количество индексов массива называется его *размерностью*. При определении многомерного массива можно задать начальные значения его элементам по тому же принципу, что и у одномерного массива:

```
int inds[][] = {{1, 2, 3}, {4, 5, 6}};
```

Определенный здесь двумерный массив `inds` состоит из двух массивов по три элемента каждый. После этого определения элемент `inds[0][0]` будет равен 1, элемент `inds[0][1]` равен 2, и т. д. Элемент `inds[1][1]` будет равен 5.

Как видите, двумерный массив полностью соответствует математическому понятию прямоугольной матрицы, только индексы матрицы обычно начинаются с 1, а индексы двумерного массива начинаются с нуля.

Из определения многомерного массива следует, что его элементы располагаются в оперативной памяти по строкам. Элементы одной строки хранятся в соседних ячейках, но элементы одного столбца могут находиться далеко друг от друга. Поэтому многомерные массивы быстрее просматриваются по строкам, а не по столбцам. Для просмотра многомерного массива приходится записывать вложенные циклы. Быстрый просмотр соседних ячеек будет обеспечен, если первый индекс будет изменяться во внешнем цикле, а последний индекс — во внутреннем цикле. Вот как можно вывести значения двумерного массива по строкам в виде таблицы чисел:

```
for (i = 0; i < FIRST_DIM; i++){
   for (j = 0; j < SECOND_DIM; j++)
        printf("%f  ", a[i][j]);
   printf("\n");
}
```

В листинге 4.2 приведен пример программы, вычисляющей первые `LINES` строк треугольника Паскаля, заносящей их в двумерный массив и выводящей его элементы на экран.

Листинг 4.2. Треугольник Паскаля

```
#include <stdio.h>
#define LINES 10
main(){
```

```
    int p[LINES][LINES], i, j;

    for (i = 0; i < LINES; i++)
      for (j = 0; j < LINES; j++)
          p[i][j] = 0; // Обнуляем массив.

    printf("%3d\n", p[0][0] = 1);

    p[1][0] = p[1][1] = 1;
    printf("%3d %3d\n", p[1][0], p[1][1]);

    for (i = 2; i < LINES; i++){
      printf("%3d ", p[i][0] = 1);
      for (j = 1; j < i; j++)
         printf("%3d ", p[i][j] = p[i-1][j-1] + p[i-1][j]);
      printf("%3d\n", p[i][i] = 1);
    }
}
```

Упражнения

1. Если вы знакомы с линейной алгеброй, то напишите программу умножения двух матриц.
2. Если вы хорошо знакомы с линейной алгеброй, то напишите программу обращения квадратной матрицы.
3. Если вы любите линейную алгебру, то опишите на языке C все действия с матрицами.

Точно так же можно определить трехмерный массив:

```
int a[2][3][2] =
    {{{1, 2}, {3, 4}}, {{{1, 2}, {3, 4}}, {{{1, 2}, {3, 4}},
     {{1, 2}, {3, 4}}, {{{1, 2}, {3, 4}}, {{{1, 2}, {3, 4}}};
```

Допустимо и большее число индексов. Стандарт языка C не накладывает ограничения на размерность массива, но каждый компилятор задает максимальное количество индексов массива. Оно обычно измеряется десятками и никогда не достигается в реальных программах. На практике редко встречаются массивы больше чем с тремя индексами.

Если при определении многомерного массива заданы начальные значения, то количество элементов по первому индексу можно не указывать. Компи-

лятор сможет восстановить его по следующим размерностям. Например, предыдущее определение можно записать так:

```
int a[][3][2] =
    {{{1, 2}, {3, 4}}, {{{1, 2}, {3, 4}}, {{{1, 2}, {3, 4}},
     {{1, 2}, {3, 4}}, {{{1, 2}, {3, 4}}, {{{1, 2}, {3, 4}}};
```

Указатели

До сих пор мы имели дело с переменными простых типов и массивами, составленными из таких переменных. В языке С очень часто применяются переменные, значения которых — адреса ячеек оперативной памяти. Такие переменные называются *указателями* (pointers).

Напомню, что *ячейкой* оперативной памяти мы называем совокупность смежных байтов, содержащих единицу информации: число, символ, строку. *Длина* ячейки, т. е. количество байтов, составляющих ячейку, может быть различной: один, два, четыре, восемь байтов в зависимости от типа информации, хранящейся в ячейке. *Адрес* ячейки — это порядковый номер первого из ее байтов.

Чаще всего в ячейке хранится значение какого-нибудь типа, поэтому в языке С говорят не просто "указатель", а "указатель на такой-то тип". Это отражается на определении, которое выглядит так:

```
char *pc;
double *pd;
```

Звездочка помечает указатель, чтобы отличить его от обычной переменной. Ведь в одном определении могут встретиться и указатели, и обычные переменные:

```
char *pc1, c0 = '?', *pc2;
int *pi, k = 5;
```

В некоторых случаях заранее невозможно задать тип для указателя. В таких случаях делается запись:

```
void *p;
```

Такой указатель не знает структуру и размер ячеек памяти. После определения его значения он просто будет содержать номер некоторого байта. Перед его использованием надо сделать явное приведение к определенному типу.

Приведенные выше операторы определяют только указатели, например:

```
int *pi;
```

В этом случае компилятор выделяет память для указателя `pi`. В ней будет храниться адрес ячейки, на которую будет ссылаться указатель, но никакого адреса пока еще нет — указатель ни на что не указывает. Каким-нибудь

образом ему надо дать значение — адрес какой-то ячейки. Можно записать адрес явно: `pi = (int *)245040`; но делать так никогда не приходится, ведь оперативная память распределяется операционной системой, мы не знаем, какой адрес у нужной нам ячейки.

Гораздо чаще указателю дается адрес ячейки памяти, занятой какой-то уже определенной до этого переменной простого типа. Для этого надо знать этот адрес, а его можно получить операцией *взятия адреса* переменной. Эта операция обозначается амперсандом, записываемым перед переменной. Мы можем написать присваивание указателем адреса переменной `k`:

```
pi = &k;
```

Теперь указатель `pi` полностью определен, он показывает на ячейку памяти, занятую переменной `k`, содержит ее адрес. Это схематически показано на рис. 4.1, где для примера взят конкретный адрес 200.

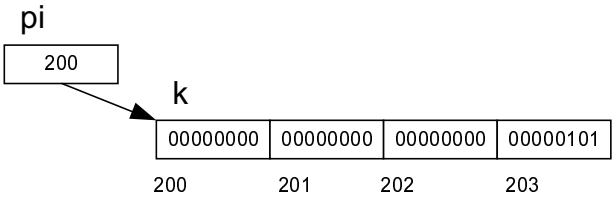


Рис. 4.1. Определение указателя

Мы получили то, что в программировании называется *косвенной адресацией*. По указателю определяется не значение переменной, а только ее адрес, по которому находится переменная. К такому двойному определению надо привыкнуть. Поначалу бывает трудно разобраться в хитросплетениях указателей, тем более что многие программисты грешат этим.

Мы можем извлечь содержимое ячейки, адрес которой содержится в указателе, воспользовавшись операцией *взятия переменной по адресу*. Для этой операции не придумано специального обозначения, перед указателем опять-таки записывается звездочка.

```
int n = *pi;
```

После этого переменная `n` станет равной 5. Операция "звездочка" создает полноценную переменную. Вы можете считать, что имя переменной просто начинается со звездочки, хотя на самом деле это результат операции, а не имя. Такую запись можно использовать в выражениях: `*pi / 2 + k`. Можно сделать присваивание: `*pi = 10` и все другие операции, доступные этой переменной.

Вот поэтому-то звездочка записывается в определении указателя. Строку определения `int *pi;` следует читать так: "`int` — это тип результата вычисления выражения `*pi`".

Итак, в языке С есть операция взятия адреса и есть операция взятия содержимого ячейки по адресу. Эти две операции взаимно обратные, как сложение и вычитание. Они выполняются справа налево и их можно применить вместе. Мы можем написать:

```
char ch = *&c0;
```

При этом мы получим адрес, а потом значение символьной переменной, или написать:

```
pc2 = &*pc1;
```

В этом случае мы получим сначала значение по адресу, а потом снова адрес. Надо подчеркнуть, что эти операции применяются только к переменным, но не к константам или выражениям, нельзя написать `&(a + b)`, потому что результат выражения заносится во временную ячейку, не имеющую постоянного адреса.

При работе с указателями следует внимательно следить за тем, чтобы у них было определенное значение. После начального определения — `int *pi;` — указатель содержит мусор, оставшийся в выделенной для него ячейке после предыдущих вычислений. Хорошо, если это содержимое не представляет никакого адреса оперативной памяти. Гораздо хуже, если оно случайно окажется адресом ячейки, важной для работы компьютера. Тогда любым присваиванием вида `*pi = x;` мы изменим содержимое этой ячейки и нарушим работу программы. Чтобы избежать такой ситуации, называемой "висящим указателем", в язык С введено понятие *нулевого указателя*, значение которого — такой адрес, которого заведомо нет в оперативной памяти. Это значение записывается константой `NULL`. Определяя указатель, которому мы не можем сразу же дать точное значение, следует всегда давать ему значение `NULL`:

```
int *pi = NULL;
```

Вместо константы `NULL` можно записать просто 0:

```
int *pi = 0;
```

Не думайте, что это будет адрес нулевого байта. Компилятор преобразует такую запись по своим правилам и может занести в ячейку, выделенную для указателя `pi` другое значение, не соответствующее ни одной ячейке оперативной памяти.

Адресная арифметика

Из предыдущего пункта видно, какую свободу дает язык С программисту в работе с оперативной памятью. Программист может обратиться к любой ячейке и попытаться изменить ее значение. На самом деле такая попытка чаще всего окажется неудачной. Защитой оперативной памяти занимается операционная система. Она следит за тем, чтобы программа могла изменить только выделенную для нее память и не изменяла содержимое важных для

работы компьютера участков. Но в оперативной памяти есть общие области, разделяемые несколькими программами. Их работу неразумное обращение к оперативной памяти может нарушить.

Кроме этого, язык С предлагает набор арифметических операций с указателями, позволяющих перемещаться по оперативной памяти и облегчающих непосредственную работу с ее ячейками.

- К указателю можно прибавить целое значение, положительное или отрицательное. Прибавление к указателю 1, т. е. операция `pi + 1`, или увеличение указателя на 1, т. е. операции `pi++`, `++pi`, приводит к переходу указателя на следующее значение своего типа. Важно понять, что увеличение адреса на 1 означает не увеличение номера байта на 1, а прибавление к текущему номеру байта длины типа указателя. В примере рис. 4.1 значение `pi + 1` будет больше значения `pi` на 4 байта и станет показывать на байт с номером 204. Увеличение `pi` на 2, т. е. `pi + 2`, даст адрес 208 и т. д. Увеличение или уменьшение ровно на один байт будет сделано только для типа (`char *`). Операция прибавления к указателю целого значения коммутативна, можно записать как `pi + k`, так и `k + pi`.
- Из указателя с большим адресом можно вычесть указатель того же типа с меньшим адресом. Результатом будет число значений данного типа, лежащих между этими адресами. Например, если `p1` и `p2` — указатели на целый тип `int`, имеющий длину 4 байта, указатель `p1` содержит значение 200, а указатель `p2` — значение 208, то разность `p2 - p1` будет равна 2.
- Как уже замечено выше, к указателю можно применить операции инкремента и декремента, префиксные и постфиксные: `pi++`, `++pi`, `pi--`, `--pi`. Они дают в результате указатель на следующее или предыдущее значение своего типа.
- С указателями можно сделать присваивания вида =, += или -=, если в правой части присваивания выражение того же типа, что и указатель в левой части.
- Выражения, состоящие из указателей одного и того же типа, можно сравнивать межлу собой операциями ==, !=, <, <=, >, >=. Указатели считаются равными, если содержат один и тот же адрес, показывают на одну и ту же ячейку памяти. Больший указатель тот, который содержит больший адрес. Всякий указатель можно сравнить с нулевым указателем на равенство или неравенство.
- К указателю можно применить операцию отрицания `!pi`. Она переводит не нулевой указатель в нулевой и обратно.

Сразу же надо предупредить, что адресная арифметика — это источник множества ошибок, сделанных в свое время в тысячах программ. Записывая операции с указателями, надо полностью отдавать себе отчет об их результатах. В частности, вычитание и сравнение указателей следует выполнять только в пределах одного массива.

Участие указателей в выражениях ставит вопрос о приоритете новых операций взятия адреса `&` и взятия переменной по адресу `*`. Это операции с довольно высоким приоритетом. Они выполняются после инкремента и декремента, но до явного преобразования типа и умножения/деления. Как уже было сказано, они выполняются справа налево. Здесь надо быть внимательным. В программах, написанных на языке С, вы часто увидите такую запись:

```
*pi++;
```

Здесь берется значение по адресу, хранящемуся в указателе `pi`, а потом значение указателя увеличивается. Почему, ведь инкремент выполняется раньше взятия значения по адресу? Рассмотрим это подробнее. Пусть записаны операторы:

```
int a = 5, *p = &a;
int b = *p++;
```

После этого значение переменной `b` равно 5. Порядок операций во втором операторе таков: `*(p++)`, но инкремент постфиксный, поэтому увеличение адреса происходит после вычисления всего выражения.

Связь массивов и указателей

В языке С имя массива всегда является указателем на начало массива, адресом его первого элемента, например:

```
int a[10];
```

При таком определении компилятор неявно определяет указатель с именем, совпадающим с именем массива, и хранит в нем адрес массива, как показано на рис. 4.2, где для примера взято конкретное значение адреса 200.

Рис. 4.2. Определение массива `int a[10]`

На самом деле конкретные адреса определяет операционная система.

Именем массива можно свободно пользоваться как указателем. Вместо `a[0]` можно написать `*a`, вместо `a[1]` — написать `*(a + 1)`, вместо `a[k]` — `*(a + k)`. Такую запись можно делать в выражениях:

```
*(a + 3) += *a - *(a + 1) / *(a + 2);
```

Более того, компилятор так и поступает, он сразу же переводит запись с индексами в запись с указателями и затем работает с этой записью. Теперь можно понять, почему в языке С индексы элементов массива всегда начинаются с нуля. Индекс — это просто смещение по оперативной памяти от начала массива. Поскольку прибавление целого значения к указателю — коммутативная операция, справедлива каждая из следующих записей:

```
a[2] == *(a + 2) == *(2 + a) == 2[a]
```

Проверьте, любое из этих выражений допустимо!

Есть одно существенное различие между именем массива и указателем, определенным звездочкой: имя массива — это указатель-константа. Массив размещается в памяти операционной системой, она определяет его адрес, и этот адрес нельзя изменить. Нельзя написать, например, `a = &x`. При необходимости работы с указателем-переменной можно определить ее и связать с массивом:

```
int *p = a;
```

После такого определения указатель `p` показывает на ту же ячейку памяти, что и `a`, значит, `*p == a[0]`, `*(p + 1) == a[1]` и т. д. Интересно то, что можно записывать `p[0]`, `p[1]`, хотя мы не определяли никакого массива `p`! Это допустимо, потому что компилятор механически переведет индексную запись в запись с указателями.

Далее в программе можно сделать новое определение:

```
int b[1000];
```

Если перебросить указатель на новый массив `p = b`, то `*p == p[0] == b[0]`.

Указатели очень часто применяются для работы с массивами. Вот как можно просуммировать элементы массива с помощью указателей:

```
double a[100], s = 0.0, *p = NULL;
// Каким-то образом определяем элементы массива a.
int k = 0;
for (p = a; k < 100; p++, k++) s += *p;
```

До сих пор мы определяли указатель на массив. В случае необходимости можно определить массив указателей, например:

```
int k = 22;
int *p[5] = {&k, &k, &k, &k, &k};
```

Здесь определен массив из пяти указателей. Все указатели содержат адрес переменной `k`. Еще более интересная картина возникает при определении многомерного массива:

```
int a[2][3];
```

При этом возникает схема, показанная на рис. 4.3.

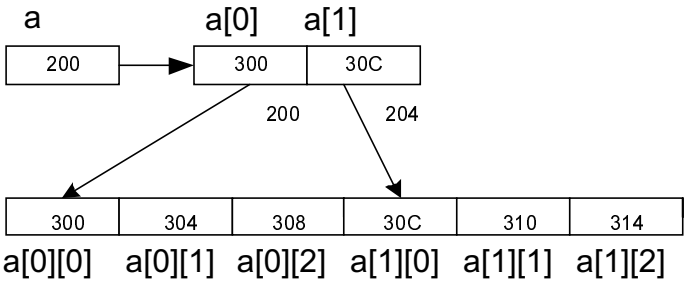


Рис. 4.3. Определение двумерного массива

Здесь имя массива `a` — это указатель-константа на массив из двух элементов `a[0]` и `a[1]`. Это массив указателей-констант, каждый его элемент — указатель на массив из трех элементов. Всеми указателями можно пользоваться; более того, здесь возникает новое понятие — *указатель на указатель*. В самом деле, указатель `a` показывает не на переменную обычного простого типа, а на указатель `a[0]`. Значит, `*a` — это `a[0]`, но `*a[0]` — это `a[0][0]`. Отсюда `**a` — это `a[0][0]`. Итак:

- `a` — указатель на начало массива указателей `a[0]`;
- `*a == a[0]` — указатель на элемент `a[0][0]`;
- `**a` — сам элемент `a[0][0]`.

Далее:

- `a + 1` — указатель на элемент `a[1]`;
- `*(a + 1)` — сам элемент `a[1]` — указатель на элемент `a[1][0]`;
- `*(a + 1) + 2` — указатель на элемент `a[1][2]`;
- `*(*(a + 1) + 2)` — сам элемент `a[1][2]`.

Значит, элемент массива `a[i][j]` компилятор запишет как `*(*(a + i) + j)`. В листинге 4.3 показано, как все эти адреса можно вывести на экран дисплея.

Листинг 4.3. Адреса оперативной памяти

```
#include <stdio.h>
main(){
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

    printf("Адрес массива a: %d\n", a);
    printf("Адрес элемента a + 1: %d\n", a + 1);
```

```
    printf("Адрес элемента *(a + 1): %d\n", *(a + 1));
    printf("Адрес элемента *(a + 1) + 2: %d\n", *(a + 1) + 2);
    printf("Элемент *(*(a + 1) + 2): %d\n", *(*(a + 1) + 2));
}
```

Переменную типа "указатель на указатель на какой-то тип" можно определить и явно:

```
int **p;
```

Мы получаем *двойную косвенную адресацию*. Указатель `p` содержит адрес не простой переменной, а другого указателя, который содержит адрес переменной. Потом можно дать указателю `p` значение уже определенного двойного указателя, например, `p = a`, и работать с матрицей `a` с помощью указателя `p`.

Итак, в языке С можно работать с двойными, тройными и более указателями, реализуя множественную косвенную адресацию, можно в любом сочетании определять массивы указателей и указатели на массивы, выполняя сложную обработку ячеек оперативной памяти. Не увлекайтесь этими возможностями, чтобы не усложнять программу.

Работа со строками

Мы уже встречались со строковыми константами вида `"Строка символов"`. Компилятор рассматривает строку как массив символов, завершенный нуль-символом. Эта запись определяет простой массив символов:

```
char c[] = {'С', 'т', 'р', 'о', 'к', 'а', ' ',
            'с', 'и', 'м', 'в', 'о', 'л', 'о', 'в'};
```

А эта определяет строку:

```
char s[] = {'С', 'т', 'р', 'о', 'к', 'а', ' ',
            'с', 'и', 'м', 'в', 'о', 'л', 'о', 'в', '\0'};
```

Компилятор всегда генерирует на месте строковой константы ее адрес. Этот адрес может присвоить указатель, значит, можно записать:

```
char *p = "Строка символов";
```

Определение `s` создало указатель-константу на строку, а определение `p` — указатель-переменную, которой дан адрес строковой константы. В остальном эти указатели равноправны. Мы могли бы определить строку `s` и так:

```
char s[] = "Строка символов";
```

Нет никакой разницы между двумя определениями строки `s`, но между определениями `p` и `s` есть та же разница, что была при определении массивов: `p` — это указатель-переменная, `s` — указатель-константа.

Стандарт языка С не позволяет менять строковую константу через указатель на нее. На практике компилятор обычно располагает константу в области памяти, доступной вашей программе только для чтения, но не для записи. С помощью указателя `p` вы не сможете изменить строку. Для строки `s`, определенной как массив с нуль-символом в конце, такой компилятор выделит память в области вашей программы и вы вольны распоряжаться строкой по своему усмотрению.

Тем не менее, указатели очень широко применяются для работы со строками. Вот как можно подсчитать количество символов в строке — ее длину:

```
int len = 0;
char *p = "Строка символов, длину которой мы подсчитываем";
for (; *p; p++) len++;
```

Цикл можно записать другим оператором:

```
while (*p) { p++; len++; }
```

Или даже так:

```
while (*p++) len++;
```

Такую же лаконичную запись можно сделать в операторе цикла `for`:

```
for (; *p++;) len++;
```

Можно воспользоваться вычитанием указателей и подсчитать длину строки так:

```
int len = 0;
char *p = "Строка символов, длину которой мы подсчитываем";
char *q = p;
while(*q++);
len = q - p;
```

Во всех случаях завершающий нуль-символ не учитывается при подсчете, как только `*p` станет равной `\0`, выполнение цикла прекратится, мы получим длину `len`, равную 46.

В программах на языке С вы увидите примеры очень лаконичной записи операций с указателями, больше похожие на шифровку. Вот классический пример копирования всех символов строки, на которую указывает `q`, в строку `p`:

```
char p[1000];
char *q = "Копируемая строка";
while (*p++ = *q++);
```

Здесь в одном выражении происходит и перемещение указателей, и присвоение символов, и проверка окончания строки. Со строкой можно работать как с массивом, используя индексы. Так, в определенной выше строке `p[0] == 'С'`, `p[1] == 'т'`, `p[10] == 'в'`. Последний символ строки `м` —

это элемент `p[45]`. Завершающий нуль-символ — это элемент массива `p[46]`. Мы можем обратиться к любому символу по индексу или изменить любой символ простым присваиванием вида `p[k] = '?'`;

В качестве примера работы со строками приведем программу, подсчитывающую количество появлений каждого символа в заданной строке.

Листинг 4.4. Подсчет количества появлений каждого символа в строке

```
#include <stdio.h>
#include <string.h>

main(){
    char s[] = "Подсчитаем количество появлений каждого символа.";
    int ch[256], i;

    for (i = 0; i < 256; i++) ch[i] = 0;

    for (i = 0; i < strlen(s); i++) ch[s[i]]++;

    for (i = 32; i < 256; i++)
      printf("Символ %c появился %d раз\n", i, ch[i]);
}
```

Упражнения

4. Напишите программу, подсчитывающую количество пробелов в заданном тексте.
5. Напишите программу, подсчитывающую количество появлений заданного символа в заданном тексте.
6. Напишите программу, подсчитывающую количество слов в заданном тексте. Вы можете использовать алгоритм, изображенный на блок-схеме рис. 1.14.
7. Напишите программу, отыскивающую индекс заданного символа в заданном тексте. Если такого символа в тексте нет, то программа должна дать значение −1. Если символ встречается несколько раз, то можно написать два варианта программы, дающие индекс первого или последнего появления символа в тексте.
8. Напишите программу, отыскивающую индекс, с которого начинается заданное слово в заданном тексте. Если такого слова в тексте нет, то программа должна дать значение −1. Если слово встречается несколько

раз, то можно написать два варианта программы, дающие индекс первого или последнего появления слова в тексте.

Функции обработки строк

В стандарт языка С входит библиотека функций, предназначенных для работы со строками. В листинге 4.4 мы уже использовали функцию `strlen()` из этой библиотеки. Описание этих функций сделано в файле string.h, который надо включать в программу директивой `#include`, а сами функции, записанные в машинном коде, хранятся в библиотеке libc. Библиотека libc подключается к программе автоматически, никаких дополнительных указаний компилятору делать не надо.

Один из аргументов этих функций — указатель на уже определенную строку символов. Функция просматривает строку и дает соответствующий результат. Дадим описание стандартных функций.

Функция `strcat(s1, s2)` дополняет первую строку `s1` символами из второй строки `s2`. В качестве результата она возвращает адрес первой строки, т. е. значение указателя `s1`. В первой строке должно быть достаточно места для помещения всех символов второй строки. Пример:

```
char s[100];
strcat(s, "Начало строки. ");
strcat(s, "Окончание строки.");
```

Функция `strncat(s1, s2, n)` дополняет первую строку `s1` первыми `n` символами из второй строки `s2`. В остальном она действует так же, как функция `strcat()`. Функция `strcmp(s1, s2)` сравнивает первую строку `s1` со второй строкой `s2`. В качестве результата она возвращает 0, если строки совпадают, или разность кодов первых несовпавших символов. Пример:

```
int r = strcmp("Иванов", "Иванцов");
```

Получим разность кодов символов "о" и "ц". Для кодировок, в которых кириллица расположена по алфавиту, получим значение `r`, равное −8.

Функция `strncmp(s1, s2, n)` сравнивает первые `n` символов строк `s1` и `s2`. В остальном она действует так же, как функция `strcmp()`. Функция `strcpy(s1, s2)` копирует вторую строку `s2` в первую строку `s1`. Старое содержимое строки `s1` стирается. В качестве результата функция возвращает адрес первой строки, т. е. значение указателя `s1`. В первой строке должно быть достаточно места для помещения всех символов второй строки. Пример:

```
char s[100];
strcpy(s, "Эта строка копируется в строку s.");
```

Функция `strncpy(s1, s2, n)` копирует первые `n` символов второй строки `s2` в первую строку `s1`. В остальном она действует так же, как функция

strcpy(). Функция strlen(s) подсчитывает количество символов строки s без завершающего нуль-символа. Она устроена так, как было показано выше. Пример:

```
int n = strlen("Подсчет длины этой строки.");
```

Функция strchr(s, c) находит указатель на место первого появления символа c в строке s. Если такого символа в строке нет, функция дает NULL. Пример:

```
char *p = NULL;
if ( (p = strchr("Иванов, Петров, Сидоров.", 'П')) != NULL)
   printf("%s\n", p);
```

Функция strrchr(s, c) находит указатель на место последнего появления символа c в строке s. В остальном она действует так же, как функция strchr(). Функция strstr(s1, s2) находит указатель на место первого появления строки s2 в строке s1. Если такой подстроки в строке s1 нет, функция возвращает NULL. Пример:

```
char *p = NULL;
if ( (p = strstr("Иванов, Петров, Сидоров.", "Петров")) != NULL)
   printf("%s\n", p);
```

Функция strpbrk(s1, s2) возвращает указатель на место первого появления в первой строке s1 любого из символов второй строки s2 или NULL, если в строке s1 нет ни одного символа из строки s2. Пример:

```
if (strpbrk("(x + 2) * (y - 1)", "+-*/%") != NULL)
   printf("Арифметическое выражение\n");
```

Функция strspn(s1, s2) подсчитывает длину первой подстроки строки s1, целиком состоящей из символов строки s2, взятых в любом порядке. Пример:

```
if (strspn("Строка с пробелами", " \t\r\n") > 1)
   printf("Лишние пробельные символы\n");
```

Функция strcspn(s1, s2) подсчитывает длину первой подстроки строки s1, целиком состоящей из символов, отсутствующих в строке s2. Функция strtok(s1, s2) возвращает указатель на первое слово строки s1. Разделители слов перечислены в строке s2. Следующий вызов функции strtok(NULL, s2) даст указатель на второе слово той же строки, если ее первый аргумент равен NULL. Строка разделителей s2 может меняться от вызова к вызову. Если в строке уже нет слов, то возвращается NULL. Пример:

```
char *s = "Перебираем все слова этой строки";
char *p = strtok(s, " \t\r\n");
printf("Первое слово: %s\n", p);
while ( (p = strtok(NULL, " \t\r\n")) != NULL)
   printf("Следующие слова: %s\n", p);
```

Работа с отдельными символами

Во время работы с текстами очень часто приходится анализировать отдельные символы: отмечать переводы строк, выделять предложения, удалять лишние пробелы. В стандартную поставку языка C входит библиотека функций, делающих проверки и простейшие преобразования символов. Описание библиотеки сделано в файле ctype.h, который следует включить в программу директивой препроцессора `#include`.

У функции, проверяющей символы, заголовок записывается по принципу `int isXXX(int c)`, где `XXX` заменяется кратким обозначением класса символов, принадлежность к которому проверяет функция. Функции возвращают ненулевое значение, если символ `c` принадлежит указанному классу, и 0 в противном случае. Перечислим эти функции.

- `isalpha(c)` — символ `c` — буква.
- `isdigit(c)` — символ `c` — арабская цифра от 0 до 9.
- `isalnum(c)` — символ `c` — буква или арабская цифра.
- `isxdigit(c)` — символ `c` может участвовать в записи шестнадцатеричного числа, т. е. это цифры от 0 до 9 и буквы A, B, C, D, E, F, a, b, c, d, e, f.
- `isspace(c)` — символ `c` — пробельный, т. е. пробел, символ вертикальной или горизонтальной табуляции, символ перевода строки, возврата каретки, перевода страницы.
- `ispunct(c)` — символ `c` — знак пунктуации.
- `isprint(c)` — символ `c` — печатный, имеющий графическое представление, включая пробел.
- `isgraph(c)` — символ `c` — печатный, за исключением пробела.
- `iscntrl(c)` — символ `c` — управляющий, т. е. символ с кодом от 0 до 31 или символ с кодом 127.
- `isascii(c)` — символ `c` — символ ASCII, т. е. символ с кодом от 0 до 127.
- `isupper(c)` — символ `c` — заглавная буква.
- `islower(c)` — символ `c` — малая буква.

Эти функции применяются в тысячах программ обработки текста, в том числе почти во всех текстовых редакторах. Первоначально они были рассчитаны на английские тексты. Например, функция `isalpha()` считала буквой только символы с кодами от 65 до 90 и от 97 до 122. Для русских текстов и текстов, написанных на других языках, такая функция приведет к огромному числу ошибок. Поэтому при локализации C эти функции обязательно переписываются с учетом особенностей языка. Для русского

приходится учитывать и различные кодировки символов. В русифицированных программах часто появляются ошибки из-за того, что они применяются с не предназначенной для них библиотекой.

Вторая группа функций выполняет некоторые преобразования символов.

- `toascii(c)` — преобразует целое число `c` в символ ASCII, оставив в нем только 7 младших битов.
- `tolower(c)` — преобразует букву `c` в нижний регистр. Заглавные буквы становятся строчными, другие символы не изменяются.
- `toupper(c)` — преобразует букву `c` в верхний регистр. Строчные буквы становятся заглавными, другие символы не изменяются.

Динамическое выделение памяти

Все переменные и массивы, с которыми мы работали до сих пор, определялись операторами определения типа. Мы описывали их перед компиляцией, до выполнения программы. Для таких переменных оперативная память выделяется перед выполнением программы и сохраняется до ее окончания. Это *статическое* выделение памяти.

Во многих случаях заранее невозможно угадать, потребуется переменная или нет, или определить, сколько места в оперативной памяти понадобится для массива. В таких случаях надо определять переменную или массив *динамически*, во время выполнения программы, выделять ей память непосредственно перед ее использованием, и освобождать память, когда переменная уже не нужна.

В языке C основное средство динамического выделения памяти — функция `malloc(size)` (memory allocation). Аргументом `size` этой функции служит количество байтов, которое необходимо выделить в оперативной памяти. Функция возвращает указатель типа (`void *`) на первый байт выделенного участка памяти или `NULL`, если выделить память не удалось.

```
char *p;
if ( (p = (char *)malloc(1000)) == NULL) printf("Нехватка памяти\n");
else strcpy(p, new_text);
```

Здесь выделена память под символьный массив из 1000 элементов. Функция `malloc()` ничего не знает о нужном типе указателя и создает указатель неопределенного типа `(void *)`. Поэтому сделано явное преобразование типа. Еще один пример:

```
double *p;
if ( (p = (double *)malloc(800) == NULL){
    printf("Нехватка памяти\n");
    return;
}
```

```
int i;
for (i = 0; i < 100; i++) p[i] = (double)i;
```

Здесь память запрошена с расчетом, что длина типа `double` равна 8 байтов. Этот расчет не всегда верен. Гораздо лучше узнать длину типа, чтобы точно указать число байтов. Здесь помогает еще одна операция языка С — операция определения длины типа `sizeof (тип)`. Она возвращает длину заданного типа в байтах. С ее использованием предыдущий пример, наконец, будет выглядеть так, как он выглядит в большинстве программ:

```
double *p;
if ( (p = (double *)malloc(100 * sizeof(double)) == NULL){
    printf("Нехватка памяти\n");
    return;
}
int i;
for (i = 0; i < 100; i++) p[i] = (double)i;
```

При вызове функции `malloc()` надо учитывать, что она всегда выделяет непрерывную область памяти, отыскивая подходящий по размеру участок среди свободных блоков памяти. Поэтому не стоит без необходимости заказывать слишком много памяти за один раз. Это ускорит поиск свободного участка и позволит избежать отказов в виде возвращенного `NULL`.

Память, выделенная функцией `malloc()`, остается занятой до окончания программы, хотя необходимость в ней может отпасть раньше. Лучше освободить ее сразу же. Для освобождения памяти служит функция `free(pointer)`. Ей передается указатель `pointer`, ранее определенный функцией `malloc()`. По этому указателю функция `free()` узнает начало участка освобождаемой памяти и его размер.

Итак, динамическое выделение и освобождение участков оперативной памяти выполняется парой функций `malloc()`, `free()`. Вызвав первую функцию, надо сразу же подумать о том, в какой момент вызвать вторую, чтобы вовремя и полностью освободить память. Распространенная ошибка программирования, называемая "утечкой памяти" (memory leak), заключается в том, что память выделяется динамически в цикле, но не освобождается. Постепенно свободная память заканчивается, хотя есть много неиспользуемых участков, и программа завершается аварийно.

Бинарный поиск в массиве

Задача поиска информации по каким-то ее признакам — важнейшая задача информатики. Придумано много методов ее решения, разработаны остроумные алгоритмы, но объем информации все растет, и требуются все более быстрые алгоритмы поиска.

Методы поиска различны для разных способов хранения информации. В этой главе мы рассмотрим только поиск информации, находящейся в массиве.

Самый простой способ поиска информации в массиве — просто перебирать все его элементы с самого начала, пока не встретится нужный элемент или пока не закончится массив. Такой метод называется *линейным поиском*.

Пусть для простоты у нас есть целочисленный массив:

```
int a[LEN];
```

Он заполнен какими-то числами. Нам надо отыскать в этом массиве заданное число `x` и вернуть индекс `m` элемента, содержащего `x`, или число −1, если такого элемента в массиве нет. Поиск можно организовать так:

```
int m = 0;
while(m < LEN && a[m] != x) m++;
if (m == LEN) m = -1;
```

Такой алгоритм потребует в среднем `LEN / 2` итераций цикла.

Предположим, что массив `a` упорядочен по возрастанию или убыванию чисел. Тогда можно организовать *бинарный* поиск. Сравним число `x` со средним по величине элементом массива. Это будет элемент `a[LEN / 2]`. Если `x` меньше этого элемента, то его надо искать в первой половине массива, если больше — то во второй половине. Выбираем нужную половину и повторяем с ней предыдущие действия.

Именно так мы отыскиваем телефонный номер в справочнике по фамилии абонента: раскрываем справочник примерно посередине, смотрим, где окажется фамилия — ближе к началу справочника или ближе к концу, — открываем справочник посередине его первой или второй половины и т. д.

Как вы уже заметили, алгоритм бинарного поиска похож на метод бисекции для нахождения корня нелинейного уравнения, приведенный в листинге 3.2. Для его реализации введем меньший и больший индексы `left` и `right` текущего подмассива. Вначале мы рассматриваем весь массив, значит, их начальные значения будут `0` и `LEN`. Индекс искомого элемента `m` устанавливаем между ними: `left <= m < right`.

После каждого сравнения среднего элемента подмассива `a[m]` со значением `x` в зависимости от выбора или индекс `left` становится равным `m + 1`, или индекс `right` делается равным `m`. Тем самым подмассив уменьшается вдвое. Цикл повторяется до тех пор, пока индекс `left` не сравняется с индексом `right`. После него нужна дополнительная проверка. Все это реализовано в листинге 4.5.

Листинг 4.5. Бинарный поиск в массиве

```
#include <stdio.h>
#define LEN 10
```

```
main(){

   int a[] = {-10, -8, -3, 0, 2, 4, 7, 11, 15, 16};
   int x = 11;

   int left = 0, right = LEN, m = -1;

   while (left < right){

      m = (left + right)/2;
      if (x <= a[m]) right = m;
      else left = m + 1;
   }
   if (right == LEN || a[right] != x) m = -1;

    printf("x = %d, a[%d] = %d\n",
            x, m, (m != -1) ? a[m] : 9999);
}
```

Бинарный поиск гораздо быстрее простого линейного поиска. В среднем он требует числа повторений цикла, равного двоичному логарифму величины `LEN`. Если в массиве 100 000 элементов, то мы получаем средний выигрыш почти в 3000 раз! Ясно, что линейный поиск никуда не годится, но для применения бинарного поиска массив надо предварительно отсортировать.

Упражнения

9. Проверьте, будет ли работать алгоритм бинарного поиска, записанный в листинге 4.5, если в массиве несколько раз встречается искомый элемент.
10. Напишите программу бинарного поиска в упорядоченном по алфавиту массиве строк.

Методы сортировки массива

Существует несколько десятков способов сортировки и их число все время возрастает. Методам сортировки посвящено множество статей. Постоянно выходят целые книги, посвященные только сортировке. Мы познакомимся только с самыми простыми алгоритмами, иначе эта книга станет вдвое толще.

Все предлагаемые алгоритмы выполняют сортировку на месте, в области памяти, занимаемой массивом. Это обязательное условие для больших массивов, занимающих почти всю доступную оперативную память.

Сортировка вставкой

Метод сортировки вставкой применяется в тех случаях, когда массив только что создан, и его надо заполнять так, чтобы после вставки каждого нового элемента сохранилась его упорядоченность. Для этого осуществляется поиск подходящего для вставки места в уже заполненной части массива `a[0]...a[m]`. Поиск идет к началу массива. Место для нового элемента освобождается путем сдвига элементов массива к концу массива, как показано в листинге 4.6.

Листинг 4.6. Сортировка массива простой вставкой

```
#include <stdio.h>
#define LEN 10

main(){

   int a[LEN];
   int m = 0, x;



   scanf("%d", &x);

   while (m < LEN - 1 && x != 9999){

      int t = m;
      while (t >= 0 && x < a[t]){
         a[t + 1] = a[t];
         t--;
      }
      a[t + 1] = x;
      m++;
      scanf("%d", &x);
   }
}
```

В цикле `while` листинга 4.6 выполняется линейный поиск. Поскольку начало массива уже упорядочено, лучше организовать бинарный поиск, как сделано в листинге 4.7.

Листинг 4.7. Сортировка массива бинарной вставкой

```
#include <stdio.h>
#define LEN 10

main(){

   int a[LEN];
   int length = o,m, x;



   scanf("%d", &x);

   while (length < LEN && x != 9999){

       int left = 0, right = length;

       while (left < right){

          m = (left + right)/2;
          if (x <= a[m]) right = m;
          else left = m + 1;
       }
       int k;
       for (k = length; k >= right; k--)
          a[k + 1] = a[k];
       a[right] = x;
       length++;
       scanf("%d", &x);
   }
}
```

Даст ли бинарная вставка большой выигрыш в этом алгоритме? Спорный вопрос, поскольку модификация касается только поиска, а перемещение элементов по-прежнему линейно, для него организован дополнительный цикл.

Сортировка выбором

Метод сортировки выбором применяется в тех случаях, когда массив не упорядочен, но его элементы надо выводить в возрастающем порядке. В таком случае мы отыскиваем наименьший элемент, выводим его и на его место ставим какое-нибудь значение, заведомо большее всех элементов массива. Затем снова выводим наименьший элемент и т. д.

Такой метод годится только для вывода массива, поскольку массив уничтожается, его элементы постепенно заменяются каким-то большим значением. Тем не менее, его можно применить и для сортировки на месте, если не выводить наименьший элемент, а поменять его местами с первым элементом. При следующем поиске наименьшего элемента первый элемент массива уже не надо принимать во внимание. Так сделано в листинге 4.8.

Листинг 4.8. Сортировка массива выбором

```
#include <stdio.h>
#define LEN 10

main(){

   int a[] = {15, 12, 5, 7, 0, -2, 4, 8, -3, 10};
   int k, m;



   for (k = 0; k < LEN - 1; k++){

      int ind = k;

      for (m = k + 1; m < LEN; m++)
         if (a[m] < a[ind]) ind = m;

      int t = a[k]; a[k] = a[ind]; a[ind] = t;
   }
}
```

Как видите, сортировка выбором очень просто программируется, но она не эффективна, поскольку требует много сравнений и перестановок.

Сортировка методом "пузырька"

Приведем еще один малоэффективный, но характерный метод сортировки массива — *метод простого обмена*, чаще называемый методом *"пузырька"*

(bubble method). Он не применяется на практике, но обязательно изучается всеми программистами, поскольку прост и характерен для целой группы методов сортировки — сортировок обменом (листинг 4.9).

Данный метод заключается в сравнении пары соседних элементов и замене их местами, если элементы образуют инверсию. После этого сравнивается следующая пара и т. д. Просмотр всего массива приходится делать несколько раз, как показывает следующий пример.

Исходный массив:

7, 5, 4, 8, 2, 3

Первый просмотр массива и перестановки его элементов:

7, 5, 4, 8, 2, 3

5, 7, 4, 8, 2, 3

5, 4, 7, 8, 2, 3

5, 4, 7, 2, 8, 3

5, 4, 7, 2, 3, 8

Второй просмотр и перестановки:

5, 4, 7, 2, 3, 8

4, 5, 7, 2, 3, 8

4, 5, 2, 7, 3, 8

4, 5, 2, 3, 7, 8

Третий просмотр и перестановки:

4, 5, 2, 3, 7, 8

4, 2, 5, 3, 7, 8

4, 2, 3, 5, 7, 8

Четвертый просмотр и перестановки:

4, 2, 3, 5, 7, 8

2, 4, 3, 5, 7, 8

2, 3, 4, 5, 7, 8

Как видите, даже такой простой массив потребовал четыре просмотра. При первом просмотре наибольший элемент 8 всплыл как пузырек на свое место. При втором просмотре всплыл следующий по величине элемент 7 и т. д. Поэтому при каждом следующем просмотре мы можем ограничиться только первыми элементами массива, последние элементы встали на свои места при первых просмотрах массива.

Листинг 4.9. Сортировка массива методом "пузырька". Первый вариант

```
#include <stdio.h>
#define LEN 10

main(){

   int a[] = {15, 12, 5, 7, 0, -2, 4, 8, -3, 10};
   int k, m;

   for (k = LEN -1; k > 1; k--)

      for (m = 0; m < k; m++)

         if (a[m] > a[m + 1]){

            int t = a[m]; a[m] = a[m + 1]; a[m + 1] = t;
         }
}
```

Этот алгоритм можно улучшить. Количество просмотров массива в нем не зависит от его упорядоченности. Даже если массив изначально упорядочен, алгоритм листинга 4.9 будет просматривать его `LEN - 2` раза. Введем в алгоритм флаг упорядоченности — целочисленную переменную `flag`, которая примет значение 1, если массив уже упорядочен, т. е. при просмотре массива не было сделано ни одной перестановки.

Листинг 4.10. Сортировка массива методом "пузырька". Второй вариант

```
#include <stdio.h>
#define LEN 10

main(){

   int a[] = {15, 12, 5, 7, 0, -2, 4, 8, -3, 10};
   int k, m, flag = 0;

   for (k = LEN -1; k > 1; k--){

      int flag = 1;
```

```
      for (m = 0; m < k; m++)

         if (a[m] > a[m + 1]){

            int t = a[m]; a[m] = a[m + 1]; a[m + 1] = t;
            flag = 0;
         }

      if (flag) return;
   }
}
```

Метод просеивания

Метод "пузырька" можно улучшить так называемым *просеиванием* массива. Оно заключается в том, что после каждой перестановки мы продолжаем переставлять меньший элемент со всеми предыдущими элементами массива, пока он не встанет на свое место. Одни "пузырьки" всплывают, другие тонут. Посмотрим, как это происходит, на том же примере.

7, 5, 4, 8, 2, 3

5, 7, 4, 8, 2, 3

5, 4, 7, 8, 2, 3

4, 5, 7, 8, 2, 3

4, 5, 7, 2, 8, 3

4, 5, 2, 7, 8, 3

4, 2, 5, 7, 8, 3

2, 4, 5, 7, 8, 3

2, 4, 5, 7, 3, 8

2, 4, 5, 3, 7, 8

2, 4, 3, 5, 7, 8

2, 3, 4, 5, 7, 8

Как видите, в методе просеивания делается только один просмотр массива. Для реализации метода просеивания после перестановки элементов массива

надо вставить еще один цикл перестановок с предыдущими элементами, как сделано в листинге 4.11.

Листинг 4.11. Сортировка массива методом просеивания

```
#include <stdio.h>
#define LEN 10

main(){

   int a[] = {15, 12, 5, 7, 0, -2, 4, 8, -3, 10};
   int m;

   for (m = 0; m < LEN - 1; m++)

      if (a[m] > a[m + 1]){

         int t = a[m]; a[m] = a[m + 1]; a[m + 1] = t;

         int s = m;

         while (s > 0 && a[s] < a[s - 1]){

            t = a[s]; a[s] = a[s - 1]; a[s - 1] = t;
            s--;
         }
      }
}
```

Метод Шелла

В методах "пузырька" и просеивания переставляются соседние элементы массива. Сортировка существенно ускоряется, если делать сравнение и перестановку далеко отстоящих друг от друга элементов. Правда, при этом придется делать несколько просмотров массива, но общая скорость алгоритма при этом возрастает. В методе Шелла рекомендуется сначала сравнивать элементы `a[k]` и `a[k + h]`, отстоящие друг от друга на половину длины массива: `h = LEN / 2`. Затем шаг уменьшается вполовину: `h /= 2`. В последнем просмотре `h = 1`, т. е. выполняется обычное просеивание. Этот метод представлен в листинге 4.12.

Листинг 4.12. Сортировка массива методом Шелла

```
#include <stdio.h>
#define LEN 10

main(){

   int a[] = {15, 12, 5, 7, 0, -2, 4, 8, -3, 10};
   int m, h = LEN/2;

   while (h >= 1){

      for (m = 0; m < LEN - h; m++)

         if (a[m] > a[m + h]){

            int t = a[m]; a[m] = a[m + h]; a[m + h] = t;

            int s = m;

            while (s > h && a[s] < a[s - h]){

               t = a[s]; a[s] = a[s - h]; a[s - h] = t;
               s--;
            }
         }
      h /= 2;
   }
}
```

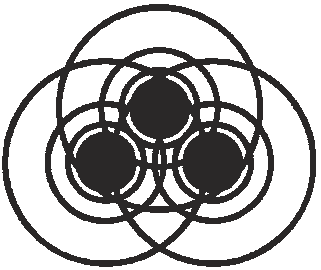
Обзор методов сортировки

Перечисленные выше методы сортировки — это только небольшая их часть. Мы занимались только сортировками обменом элементов массива. Кроме них, есть еще две большие группы методов сортировок: сортировки с помощью деревьев (*см. гл. 14*) и пирамидальные сортировки.

Даже среди сортировок обменом мы не рассмотрели самый быстрый метод, который так и называется — *быстрая сортировка* (quick sort). Этот метод основан на рекурсии, и мы его рассмотрим в следующей главе.

Контрольные вопросы

1. Можно ли в языке С использовать отрицательные индексы массивов?
2. Каково максимальное число индексов у массивов, определенных в языке С?
3. Что будет, если индекс массива оказался вещественным, а не целым значением?
4. Можно ли в качестве индексов массива использовать символы?
5. Чем опасен висящий указатель?
6. В чем разница между обычным указателем и именем массива?
7. Почему при определении указателя надо обязательно задавать тип?
8. Где в языке С появляются нетипизированные указатели и как нужно с ними работать?
9. Что такое строка в языке С?
10. Что подставляет компилятор на место строковой константы?
11. В какой области памяти хранят строковые константы некоторые компиляторы?
12. Сколько времени сохраняется память, выделенная статически?
13. Сколько времени сохраняется память, выделенная динамически?
14. Как освободить память, выделенную динамически?



Глава 5

Процедурное программирование

Вся история программирования, а может быть и история всей науки — это попытка совладать со сложностью окружающего мира. Перед программистами встают все более сложные задачи, способы их решения становятся все более громоздкими, информация, которую надо обработать, растет как снежный ком. Еще недавно обычными единицами измерения информации были килобайты и мегабайты, а сейчас уже говорят только о гигабайтах и терабайтах. Как только программисты предлагают более-менее удовлетворительное решение предложенных задач, тут же возникают новые, еще более сложные задачи. Программисты придумывают новые методы, создают новые языки. За полвека появилось несколько сотен языков, предложено множество методов и методик. Некоторые методы и стили программирования становятся общепринятыми и образуют на некоторое время так называемую *парадигму программирования*.

Парадигмы программирования

Первые, даже самые простые программы, написанные в машинных кодах, составляли сотни строк совершенно непонятного текста. Для упрощения и ускорения программирования придумали языки высокого уровня: FORTRAN, Algol и сотни других, возложив рутинные операции по созданию машинного кода на компилятор. Те же программы, переписанные на языках высокого уровня, стали гораздо понятнее и короче, но жизнь потребовала решения более сложных задач, и программы, написанные на языках высокого уровня, снова увеличились в размерах, стали необозримыми.

Возникла идея: оформить программу в виде нескольких по возможности простых процедур или функций, каждая из которых решает свою определенную задачу. Написать, откомпилировать и отладить небольшую процедуру

можно легко и быстро. Затем остается только собрать все процедуры в нужном порядке в одну программу. Кроме того, один раз написанные процедуры можно затем использовать в других программах как строительные кирпичики. *Процедурное программирование* быстро стало парадигмой. Во все языки высокого уровня включили средства написания процедур и функций. Появилось множество библиотек процедур и функций на все случаи жизни.

Встал вопрос о том, как выявить структуру программы, разбить программу на процедуры, какую часть программы выделить в отдельную процедуру, как сделать алгоритм решения задачи простым и наглядным, как удобнее связать процедуры между собой. Опытные программисты предложили свои рекомендации, названные *структурным программированием.* Структурное программирование оказалось удобным и стало парадигмой. Появились языки программирования, например, Pascal, на которых удобно писать структурные программы. Более того, на них очень трудно написать неструктурные программы. Язык С тоже написан с учетом идей структурного программирования. Вы видели в предыдущих главах, что операторы языка С позволяют выявить структуру программы, четко описать применяемые в ней алгоритмы.

Сложность стоящих перед программистами задач проявилась и тут: программы стали содержать сотни процедур, опять стали необозримыми. "Кирпичики" оказались слишком маленькими. Потребовался новый стиль программирования.

В это же время обнаружилось, что удачная или неудачная структура исходных данных может сильно облегчить или усложнить их обработку. Одни исходные данные удобнее объединить в массив, для других больше подходит структура дерева или стека. Никлаус Вирт даже назвал свою книгу: "Алгоритмы + структуры данных = программы".

Возникла идея объединить исходные данные и все процедуры их обработки в один модуль. Эта идея *модульного программирования* быстро завоевала умы и на некоторое время стала парадигмой. Программы составлялись из отдельных модулей, содержащих десяток-другой процедур и функций. Эффективность таких программ тем выше, чем меньше модули зависят друг от друга. Автономность модулей позволяет создавать и библиотеки модулей, чтобы потом использовать их в качестве строительных блоков для программы.

Для того чтобы обеспечить максимальную независимость модулей друг от друга, надо четко отделить те процедуры, которые будут вызываться другими модулями — *открытые* (public) процедуры, от вспомогательных, которые обрабатывают данные, заключенные в этот модуль, — *закрытых* (private) процедур. Первые перечисляются в отдельной части модуля — *интерфейсе* (interface), вторые участвуют только в *реализации* (implementation) модуля. Данные, занесенные в модуль, делятся на открытые, описанные в интерфейсе и доступные для других модулей, и закрытые, доступные только для

процедур того же модуля. В разных языках программирования это деление производится по-разному. В языке Turbo Pascal модуль специально делится на интерфейс и реализацию, в языке С интерфейс выносится в отдельные "заголовочные" (header) файлы. В языке С++, кроме того, для описания интерфейса можно воспользоваться абстрактными классами.

Так возникла идея о скрытии, *инкапсуляции* (incapsulation) данных и методов их обработки. Подобные идеи периодически возникают в дизайне бытовой техники. То телевизоры испещряются кнопками и топорщатся ручками и движками на радость любознательному телезрителю, господствует "приборный" стиль, то все куда-то пропадает, а на панели остаются только кнопка включения и ручка громкости. Любознательный телезритель берется за отвертку.

Инкапсуляция, конечно, производится не для того, чтобы спрятать от другого модуля что-то любопытное. Здесь преследуются две основные цели. Первая — обеспечить безопасность использования модуля, вынести в интерфейс, сделать общедоступными только те методы обработки информации, которые не могут испортить или удалить исходные данные. Вторая цель — уменьшить сложность, скрыв от внешнего мира ненужные детали реализации.

Опять возник вопрос, каким образом разбить программу на модули? Тут кстати оказались методы решения старой задачи программирования — моделирования действий искусственных и природных объектов: роботов, станков с программным управлением, беспилотных самолетов, людей, животных, растений, систем обеспечения жизнедеятельности, систем управления технологическими процессами.

В самом деле, каждый объект — робот, автомобиль, человек — обладает определенными характеристиками. Ими могут служить: вес, рост, максимальная скорость, угол поворота, грузоподъемность, фамилия, возраст. Объект может производить какие-то действия: перемещаться в пространстве, поворачиваться, поднимать, копать, расти или уменьшаться, есть, пить, рождаться и умирать, изменяя свои первоначальные характеристики. Удобно смоделировать объект в виде модуля. Его характеристики будут данными, постоянными или переменными, а действия — процедурами.

Оказалось удобным сделать и обратное — разбить программу на модули так, чтобы она превратилась в совокупность взаимодействующих объектов. Так возникло *объектно-ориентированное программирование* (object-oriented programming), сокращенно ООП (OOP) — современная парадигма программирования.

В виде объектов можно представить совсем неожиданные понятия. Например, окно на экране дисплея — это объект, имеющий ширину (width) и высоту (height), расположение на экране, описываемое обычно координатами левого верхнего угла окна, а также шрифт, которым в окно выводится текст,

скажем, Times New Roman, цвет фона (color), несколько кнопок, линейки прокрутки и другие характеристики. Окно может перемещаться по экрану методом `move()`, увеличиваться или уменьшаться в размерах методом `size()`, сворачиваться в ярлык методом `iconify()`, как-то реагировать на действия мыши и нажатия клавиш. Это полноценный объект! Кнопки, линейки прокрутки и прочие элементы окна — это тоже объекты со своими размерами, шрифтами, перемещениями.

Разумеется, считать, что окно само умеет выполнять действия, а мы только даем ему поручения: "свернись, развернись, передвинься", — это несколько неожиданный взгляд на вещи, но ведь сейчас можно подавать команды не только мышью и клавишами, но и голосом!

Идея объектно-ориентированного программирования оказалась очень плодотворной и стала активно развиваться. Оказалось удобным ставить задачу сразу в виде совокупности действующих объектов — возник *объектно-ориентированный анализ,* ООА. Решили проектировать сложные системы в виде объектов — появилось *объектно-ориентированное проектирование*, ООП (object-oriented design, OOD). *Подробную информацию о принципах объектно-ориентированного программирования см. в гл. 10.*

В этой главе мы остановимся на процедурном программировании.

Функции языка С

Мы уже говорили о том, что всякая программа, написанная на языке С, состоит из одной или нескольких функций. Если функция только одна, то она обязательно называется `main()`. Исполняющая система всегда начинает выполнение программы с вызова функции `main()`, потом из нее при необходимости вызываются другие функции.

Как у обычной математической функции, у функции языка С могут быть аргументы. Они перечисляются в скобках после имени функции через запятую. Аргументы представляют собой обычные идентификаторы, поэтому перед каждым аргументом надо записывать его тип. У функции может не быть ни одного аргумента, но скобки после ее имени все равно надо записывать.

Функция вычисляет, как говорят, *возвращает* в программу, одно значение. Его тип указывается перед именем функции. Все содержимое функции, как говорят, ее *тело*, записывается в фигурных скобках. Итак, структура функции такова:

```
тип имя(аргументы){
    тело функции
}
```

Если тип возвращаемого значения не указан, то подразумевается тип `int`. Так мы всегда записывали функцию `main()`. Некоторые функции не возвращают никакого значения. В таком случае вместо типа пишется слово `void`. Вот пример очень простой функции:

```
void salve(){
   printf("Приветствуем вас!\n");
}
```

У функции `salve()` нет ни аргументов, ни возвращаемого значения. Она просто вызывает функцию `printf()` с приветствием.

Использование функций языка С очень похоже на использование обычных математических функций. Они вызываются в любом выражении подходящего типа в том месте, где понадобилось значение функции.

Оператор *return*

Значение, возвращаемое функцией, записывается в операторе `return`. Его операндом служит любое выражение, тип которого должен совпадать с типом возвращаемого значения. Напишем простенькую функцию:

```
abscmp(double a, double b){
   return fabs(a) < fabs(b);
}
```

Функция `abscmp()` сравнивает абсолютные величины двух вещественных чисел `a` и `b` и возвращает целое значение — ноль или единицу, в зависимости от соотношения их величин своих аргументов. Тип этого возвращаемого значения `int`, поэтому он опущен. Такую функцию удобно вызывать в условном операторе:

```
if (!abscmp(x, 9999.99))
   printf("Выход за границы диапазона.");
```

Изменим эту функцию так, чтобы она возвращала большее по модулю значение:

```
double absmax(double a, double b){
   return fabs(a) < fabs(b) ? b : a;
}
```

Теперь пришлось записать тип возвращаемого значения `double`. Использовать эту функцию можно примерно так:

```
double x = 2.25 * absmax(a + b, sqrt(d));
```

Как принято в математике, функция возвращает только одно значение. Но этим значением может быть указатель на массив, строку или более сложный объект. Операторов `return` в одной функции может быть несколько. Предыдущую функцию `absmax()` можно переписать так:

```
double absmax(double a, double b){
   if(fabs(a) < fabs(b)) return b;
   else return a;
}
```

Каждый оператор `return` прекращает выполнение функции. Все следующие за ним операторы не будут выполнены. Поэтому операторы `return` чаще всего располагаются в конце функции, в условных операторах или в операторах варианта.

В функциях типа `void`, не возвращающих никакого значения, тоже могут встречаться операторы `return`, но в них не должно быть никакого выражения, только одно слово:

```
return;
```

Такой оператор `return` служит только для немедленного выхода из функции.

Функция `main()` возвращает целое значение. Мы этим никогда не пользовались, потому что завершение функции `main()` завершало всю программу, и целое значение, возвращенное функцией `main()`, негде было использовать. На самом деле это значение проверяет операционная система. Если оно равно нулю, операционная система считает, что функция завершилась удачно, проделав всю свою работу. Ненулевое значение означает завершение с какими-то более или менее серьезными ошибками.

Прочитав код завершения, операционная система может в зависимости от его значения предпринять какие-то действия. Чаще всего она посылает на консоль сообщение с указанием кода завершения. Программист по значению кода узнает, в каком месте программы произошел выход из нее. Правилом хорошего тона считается завершение функции `main()` оператором:

```
return 0;
```

Некоторые компиляторы требуют этого, другие посылают только предупреждение о необходимости возвращаемого значения, третьи компиляторы не требуют ничего. Если вы не хотите возвращать никакого значения из функции `main()`, то лучше всего дать ей тип `void`.

```
void main(){
  // Тело функции без оператора return.
}
```

При вызове функции следует обращать особое внимание на правильный тип аргументов и возвращаемого значения. Некоторые компиляторы следят за соответствием типов, посылая сообщения программисту. Тем не менее, стандарт языка C не требует таких проверок от компилятора, поэтому рассмотрим подробнее передачу аргументов в функцию.

Передача аргументов функции

При описании функции ее аргументы получают только тип и идентификатор и поэтому называются *формальными аргументами*. У них пока нет никакого значения, их можно рассматривать как хранилища значений, пока пустые.

При вызове функции на месте каждого аргумента записываются константы, переменные или даже выражения соответствующего типа. Это *фактические аргументы*, конкретные значения, передаваемые формальным аргументам. При вызове функции перед передачей значений формальным аргументам динамически выделяется память, в которую и записываются значения фактических аргументов. Память удерживается функцией до окончания ее работы и освобождается после ее окончания или после выполнения оператора `return`.

Такой способ передачи аргументов называется *передачей по значению*, поскольку передаваемые значения фактических аргументов копируются в ячейки, динамически выделенные для формальных аргументов. Его прекрасно описывает классический пример функции `swap()`, меняющей местами значения своих аргументов:

```
void swap(int a, int b){
   int t = a; a = b; b = t;
}
```

Вызовем эту функцию:

```
void main(){
   int x = 5, y = 7;
   printf("До обмена: x = %d, y = %d\n", x, y);
   swap(x, y);
   printf("После обмена: x = %d, y = %d\n", x, y);
}
```

Откомпилировав и выполнив эту программу, вы увидите, что значения переменных `x` и `y` не изменились! В самом деле, при вызове `swap(x, y)` было

выделено место в памяти для переменных `a` и `b`, в ячейку `a` попало значение 5, а в ячейку `b` — значение 7. Функция `swap()` честно поменяла значение этих ячеек, как показано на рис. 5.1, но значения ячеек `x` и `y` остались прежними! Функция `swap()` их не касалась, она даже не знает, где лежат фактические аргументы.

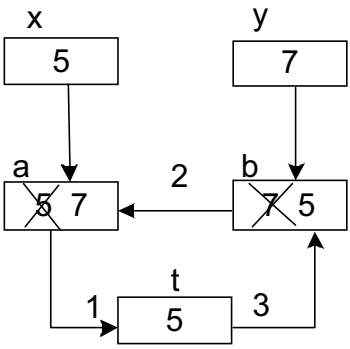


Рис. 5.1. Передача аргументов по значению

Если мы хотим поменять значения `x` и `y`, то передавать надо их адреса, а не значения. Перепишем для этого функцию `swap()`.

```
void swap(int *a, int *b){
   int t = *a; *a = *b; *b = t;
}
```

Теперь функции передаются адреса ячеек, и она меняет значения, находящиеся по этим адресам. При вызове функции надо записывать адреса переменных:

```
void main(){
   int x = 5, y = 7;
   printf("До обмена: x = %d, y = %d\n", x, y);
   swap(&x, &y);
   printf("После обмена: x = %d, y = %d\n", x, y);
}
```

Теперь `x` и `y` поменялись местами. Произошедшие перестановки показаны на рис. 5.2, где взяты условные адреса 200 и 204.

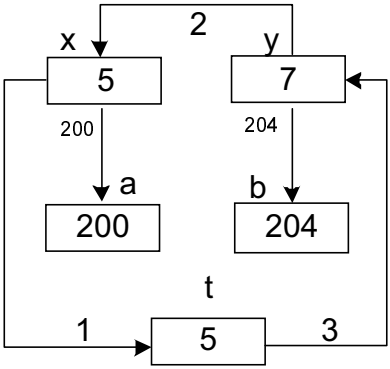


Рис. 5.2. Передача адресов по значению

Важно понять, что суть дела не изменилась. По-прежнему идет передача по значению, по-прежнему фактические аргументы копируются в ячейки, динамически выделенные для формальных аргументов, только теперь значениями аргументов будут адреса переменных, а не их величины.

Упражнения

1. Напишите функцию, возвращающую сумму своих аргументов.
2. Напишите функцию, копирующую значение своего второго аргумента по адресу, указанному первым аргументом.
3. Напишите функцию, вычисляющую дискриминант квадратного уравнения.
4. Напишите функцию, вычисляющую расстояние между двумя точками на плоскости или в пространстве.

Передача массивов в функцию

Поскольку имя массива — это указатель на его начало, написав при вызове функции имя массива в качестве фактического аргумента, мы передадим функции адрес массива. Это самый простой и естественный способ передачи массива в функцию.

В листинге 5.1 написана функция `sum()`, суммирующая все элементы заданного массива. В нее передаются имя массива `a` и количество элементов `len`, которые надо просуммировать. Функция `sum()` возвращает сумму первых `len` элементов переданного ей массива. Затем функция `sum()` вызывается в функции `main()` для конкретных массивов `x` и `y`. В конце листинга показано, как можно передать функции часть массива.

Листинг 5.1. Функция суммирования элементов массива

```
#include <stdio.h>

double sum(const double *a, int len){
   double s = 0.0;
   int k;
   for (k =0; k < len; k++) s += a[k];
   return s;
}

void main(){
   double x[] = {0.01, -3.45, 67.8, -23.5, 0.7, -23.7, 0.045};
   double y[] = {2.4, 3.65, -12.2, 3.4};
   double z = sum(x, 7);
   printf("Сумма всех элементов массива x = %f и массива y = %f\n",
       z, sum(y, 4));
   printf(
      "Сумма 5 элементов массива x, начиная с третьего = %f\n",
       sum(&x[2], 5));
}
```

Итак, при передаче массива в функцию по имени она работает непосредственно с ячейками памяти массива, изменяя сам массив, а не его копию. Иногда функция не должна изменять массив, а должна только читать его элементы. Такова функция `sum()` в листинге 5.1. Для того чтобы обезопасить массив от случайного изменения, формальный аргумент-указатель помечен словом `const`. После этого всякая попытка изменить массив будет пресечена и будет сделано сообщение об ошибке.

Упражнения

5. Напишите функцию, которая находит сумму абсолютных значений всех элементов массива.
6. Напишите функцию, вычисляющую диапазон значений массива.
7. Напишите функцию, подсчитывающую количество элементов массива, превышающих по абсолютной величине наибольший из первых десяти элементов массива.
8. Напишите функцию, отыскивающую в массиве заданное значение `x`. Верните его индекс или −1, если такого значения в массиве нет.

Аргументы функции *main()*

Мы всегда вызывали функцию `main()` без аргументов. Действительно, эта функция начинает выполнение программы, откуда же возьмутся фактические аргументы? Откуда вызвать функцию `main()`? Давайте проанализируем это.

Функцию `main()` вызывает одна из программ, входящих в состав операционной системы, — интерпретатор командной строки (shell). Он посылает нам приглашение, стандартный вид которого в UNIX-подобных системах состоит из знака доллара и пробела. Он вызывает программу на выполнение, и он может передать ей фактические аргументы.

Исходя из этой возможности, разработчики языка C предусмотрели для функции `main()` два или три аргумента. Первый аргумент целочисленный, типа `int`, второй — массив указателей на строки символов, типа `char **` или, эквивалентно, `char *[]`. Первому аргументу передается длина этого массива. Первый аргумент традиционно называется `argc` (arguments count), хотя имя, конечно, может быть любым, второй, как правило, называется `argv` (arguments vector).

Третий аргумент `arge` (arguments environment) типа `char **` используется редко, ему передаются массив переменных окружения и их значений. Итак, функцию `main()` можно записать следующим образом:

```
#include <stdio.h>


void main(int argc, char **argv){
    int k;
    for (k=0; k < argc; k++) printf("%s\n", argv[k]);
}
```

Эта простая функция выводит на консоль все свои аргументы. Пусть она записана в файл с именем mainargs.c. Откомпилируем:

```
$ gcc mainargs.c -o mainargs
```

Вызвать ее на выполнение можно так:

```
$ ./mainargs первый второй третий
```

При этом вызове значение `argc` будет равно 4. Аргумент `argv[0]` будет равен указателю на строку "`./mainargs`", аргумент `argv[1]` — указателю на строку "`первый`", `argv[2] == "второй@`, `argv[3] == "третий"`.

Как видите, в функцию `main()` в аргументе `argv[0]` всегда передается имя исполнимого файла, содержащего эту функцию. Это имя можно использо-

вать для проверки или для повторного рекурсивного вызова исполнимого файла.

Интерпретатор командной строки большинства операционных систем разделяет аргументы пробелами или табуляциями. Если надо передать аргумент, состоящий из нескольких слов, то их надо заключить в кавычки или апострофы. После вызова аргумент `argc` будет равен 2, `argv[0]`, попрежнему, будет содержать "`./mainargs`", а `argv[1]` будет равен указателю на строку "`первый второй третий`":

```
$ ./mainargs "первый второй третий"
```

Если же вызов будет выглядеть так:

```
$ ./mainargs первый "второй третий"
```

значение `argc будет 3`, `argv[0]` будет содержать "`./mainargs`", `argv[1]` — "`первый`", `argv[2]` — "`второй третий`". Таким образом, в программу всегда передаются строки символов, даже если при вызове были заданы числа.

```
$ ./mainargs 123 -456 25.08
```

При таком вызове `argv[1]` содержит "`123`", `argv[2]` — "`-456`", `argv[3]` — "`25.08`". Строки символов надо преобразовать в числа, как это сделано в листинге 5.2.

Функции преобразования строк в числа

Функции преобразования строк в числа описаны в файле stdlib.h. Функция с заголовком `int atoi(char *s)` (ascii to integer) возвращает целое число, полученное из строки, на которую показывает указатель `s`. Строка должна представлять целое значение, иначе результат будет непредсказуем.

Функция с заголовком `long atol(char *s)` (ascii to long) возвращает длинное целое число, полученное из строки, на которую показывает указатель `s`. Строка должна представлять целое значение, иначе результат будет непредсказуем.

Функция с заголовком `double atof(char *s)` (ascii to float) возвращает число с плавающей точкой удвоенной точности, полученное из строки, на которую показывает указатель `s`. Строка должна представлять число, иначе результат будет непредсказуем.

Листинг 5.2. Преобразование строк символов в числа

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[]){
```

```
    if (argc < 4){
        printf("Введите три числа\n");
        return -1;
    }
    int k = atoi(argv[1]), n = atoi(argv[2]);
    double x = atof(argv[3]);
    return 0;
}
```

Еще две функции носят общий характер. Функция с заголовком `long strtol(char *s, char **ptr, int base)` (string to long) возвращает длинное целое число, полученное из строки, на которую показывает указатель `s`. Указатель `ptr` показывает на первый символ строки, не входящий в число. Число, записанное в строке `s`, считается записанным в системе счисления, основание которой передается аргументом `base`. Если `base == 0`, то система счисления определяется по записи числа: начальные `0x`, `0X` означают шестнадцатеричную систему, начальный ноль — восьмеричную систему.

Функция с заголовком `double strtod(char *s, char **ptr)` (string to double) возвращает вещественное число удвоенной точности, полученное из строки, на которую показывает указатель `s`. Указатель `ptr` показывает на первый символ строки `s`, не входящий в число.

Процедуры

Процедуры, в отличие от функций, не возвращают никакого значения. Они обрабатывают значения своих аргументов внутри себя, выводя результаты в файл, на консоль, на принтер. Процедуры могут передавать свои результаты через аргументы. В этом случае аргументы делятся на *входные*, передающие исходные значения в процедуру, и *выходные*, передающие результаты работы процедуры. Входные фактические аргументы могут задаваться константами, переменными или выражениями, а выходные фактические аргументы должны быть только переменными.

Некоторые аргументы могут быть и входными, и выходными. При вызове процедуры они содержат входные значения. Потом они изменяются внутри процедуры, получая выходные значения.

Вызовы процедур невозможно записать в выражениях, поскольку на месте их вызова не формируются никакие возвращаемые значения. Процедуры вызываются как отдельные операторы — записывается имя процедуры, в скобках фактические аргументы, после закрывающей скобки — точка с запятой.

В языке C нет специальных конструкций для процедур. Они реализуются функциями типа `void` или функциями типа `int`. Функцию `main()` мы всегда

используем как процедуру. У функций `printf()`, `scanf()` мы никогда не использовали возвращаемое значение. Мы никогда не записывали эти функции в выражениях. Функция `swap()`, написанная выше, может служить классическим примером процедуры.

Выходные значения в функциях языка С, используемых как процедуры, передаются с помощью указателей. Так сделано, в частности, в функции `scanf()`, записывающей результаты ввода по адресам, указанным в ее аргументах.

Вообще говоря, стиль языка С предполагает использование функций, а не процедур. Даже в тех случаях, когда логика обработки данных приводит к написанию процедуры, функции языка С возвращают какое-нибудь значение, например, код завершения процедуры, позволяющий при необходимости отследить ход ее выполнения.

Указатели на функцию

Подобно массивам, имя функции понимается в языке С как адрес функции. Компилятор неявно создает указатель с именем, совпадающим с именем функции, и заносит в этот указатель адрес начала кода функции, по которому можно отыскать функцию в оперативной памяти и отправить ее на выполнение.

Более того, можно определить переменную типа "указатель на функцию, возвращающую определенный тип":

```
int (*f)();
```

Не путайте это определение с определением функции, возвращающей указатель на тип `int`:

```
int *f();
```

Перед использованием указателю на функцию надо дать значение:

```
f = sin;
```

После этого имя `f` станет синонимом функции вычисления синуса `sin` и вызов `f(0.1)` будет эквивалентен вызову `sin(0.1)`.

Возможность давать синонимы известным функциям любопытна, но мало полезна. Гораздо полезнее возможность передавать имя функции как аргумент другой функции.

В листинге 3.2 мы реализовали метод деления пополам для нахождения корня нелинейного уравнения. При попытке записать этот метод в виде функции `bisect()` мы столкнемся с необходимостью передать в нее функцию `f()` — левую часть уравнения `f(x) = 0`. Вот тут-то и полезен указатель на функцию. Мы передаем его в функцию `bisect()`. Это сделано в листинге 5.3. Затем, в функции `main()`, мы применяем функцию `bisect()` для на-

хождения нулей четырех функций — двух написанных тут же функций `func1()` и `func2()` и двух стандартных тригонометрических функций `sin()` и `cos()`.

Листинг 5.3. Нахождение корня нелинейного уравнения методом бисекции

```
#include <stdio.h>
#include <math.h>

double func1(double x){
    return x*x*x - 3*x*x + 3;
}

double func2(double x){
    return x*x*x*x - 2*x*x + 1;
}

double bisect(double a, double b, double eps, double (*f)()){

   double c;
   do{
      c = 0.5 *(a + b);
      double y = f(c);

      if (fabs(y) < eps) break;
         // Корень найден. Выходим из цикла.

         // Если на концах отрезка [a, c]
         // функция имеет разные знаки:
      if (f(a) * y < 0.0) b = c;
         // значит, корень здесь. Переносим
         // точку b в точку c.
```

```
            // В противном случае:
        else a = c;
            // переносим точку a в точку c.

            // Продолжаем, пока отрезок [a, b] не станет мал.
    } while(fabs(b-a) >= eps);

    return c;
}

main(){
    double a = 0.0, b = 1.5, eps = 1e-5;

    printf("x = %f\n", bisect(a, b, eps, func1));
    printf("x = %f\n", bisect(a, b, eps, func2));
    printf("x = %f\n", bisect(3.0, 3.5, eps, sin));
    printf("x = %f\n", bisect(1.5, 2.0, eps, cos));
}
```

Упражнения

9. Напишите функцию, которая вычисляет интеграл методом трапеций.
10. Напишите функцию, табулирующую заданную функцию в заданных точках.
11. Напишите функцию, возвращающую коэффициенты интерполяционного многочлена, построенного для заданной функции по заданной системе узлов.

Локальные и внешние переменные

Формальные аргументы функции, а также переменные и массивы, определенные внутри функции, являются *локальными* переменными. Они известны только в той функции, в которой определены. Другие функции ничего не

знают об этих переменных и не могут воспользоваться ими. Это удобно для реализации идеи процедурного программирования. Одно и то же имя можно использовать в каждой функции, никакой путаницы при этом не будет. Программист может создавать свои функции, записывая в них любые идентификаторы и не заботясь об их случайном совпадении. В листинге 5.2 формальные аргументы функций `func1()` и `func2()` называются одинаково `x`.

Локальные переменные можно определить не только в начале функции, но и в любом блоке сразу после открывающей фигурной скобки. Область действия такой переменной локализуется внутри блока. Вне блока она уже не видна и не может использоваться. Ее действие простирается на все вложенные блоки, если только в них не определена переменная с тем же именем.

Поскольку в функции может быть несколько уровней вложенности блоков, появляются несколько уровней определения переменных. На каждом уровне переменную можно определить только один раз. Переменные, определенные на каком-то уровне, действуют во всех вложенных блоках. Переменные, определенные во вложенных блоках, перекрывают переменные с теми же именами, определенные ранее во внешних блоках. Все это поясняет следующая схема:

```
/* Уровень вложенности 0. */
int f(int x, int y){ /* Уровень 0. */
   int a = 1;  /*Уровень 1. */
   int x = 3;   /* Уровень 1. */
   if (x < y){ /* Здесь x == 3 */
        int a = 2;  /* Уровень 2. */
        /*  Здесь a == 2. */
        static int count = 0;
        count++;
   } else {
     /* Уровень 2. Здесь a == 1. Переменная count неизвестна. */
      if (x == y){ /* Здесь x == 3. */
          int a = 3; /* Уровень 3. */
          /* Здесь a == 3. Переменная count неизвестна. */
      }
      /* Уровень 2. Здесь a == 1. Переменная count неизвестна. */
   }
   /* Уровень 1. Здесь a == 1. Переменная count неизвестна. */
}
/* Уровень 0. */
```

Некоторые компиляторы разрешают определять переменные не только в начале функции или блока, но и в любом месте блока перед ее использованием.

Локальные переменные существуют только во время выполнения блока. Ячейки оперативной памяти им выделяются только в точке их определения и освобождаются при выходе из блока.

Это правило можно изменить, записав при определении переменной слово `static`. Так определена переменная `count` в приведенном выше примере. *Статическим локальным* переменным память выделяется один раз сразу же при загрузке функции. Память сохраняется до окончания программы. Несмотря на это, статическая переменная остается локальной, она известна только внутри блока.

Хотя блок, в котором определена статическая переменная, может выполняться много раз, начальное значение дается локальной статической переменной только один раз, при начальной загрузке. В приведенном выше примере переменная `count` увеличивается на единицу при каждом выполнении блока, она подсчитывает, сколько раз выполняется блок.

Переменные и массивы можно определить и вне всякой функции. Это *внешние* переменные, определенные на уровне 0. Они видны во всех функциях, написанных ниже места своего определения, и могут перекрываться локальными переменными с теми же именами:

```
#include <stdio.h>

int f(){
   …..
}

#define LINES 1000
int year = 2005; /* Внешняя переменная */

int g(){
   int a = year++;
   …..
   while (*p){
      int year = 1994; /* Локальная переменная, в этом блоке она
                          перекрывает внешнюю переменную. */
      ……
   }
   …..
}

#include <math.h>
```

```
double h(){
   year++;
}

main(){
   printf("%d\n", year);
}
```

Внешние переменные, в отличие от локальных, при определении обнуляются, если для них не задано начальное значение.

Все функции всегда определяются на уровне 0, язык С не допускает вложенные определения функций. Обычно внешние переменные содержат какие-то глобальные значения, нужные нескольким или даже всем функциям, составляющим программу. Дело осложняется тем, что программу можно записать в несколько файлов. Пусть все написанное выше хранится в файле myprog1.c. В другом файле myprog2.c записаны еще две функции:

```
int year;  /* Неверно. */

char *p(){
   int a = year;
}

double *q(){
   . . .
}
```

Функции `p` и `q` тоже должны использовать ту же внешнюю переменную `year`. Поскольку файлы myprog1.c и myprog2.c могут компилироваться в разное время и в разных местах, компилятор ничего не знает о переменной `year`, и будет считать ошибкой ее использование. Определение переменной year, сделанное выше функции `p`, поможет избавиться от сообщений компилятора, но оно создает новую переменную с таким же именем, а не дублирует старое определение.

Для того чтобы указать компилятору на то, что `year` — это не новая внешняя переменная, а старая, определенная в другом файле, надо записать слово `extern`:

```
extern int year;

char *p(){
```

```
   int a = year;
}

double *q(){
   . . .
}
```

Такая запись дает не определение, а *описание* внешней переменной `year`. Увидев описание, компилятор не станет считать ошибкой использование переменной `year`, но и не будет выделять для нее ячейку памяти.

Итак, внешняя переменная должна быть определена в одном файле, при определении для нее резервируется ячейка оперативной памяти. Во всех других файлах эта переменная должна быть описана словом `extern`.

Словом `extern` можно пометить и локальную переменную. Эта пометка тоже будет означать, что переменная определена в другом файле, но она остается локальной, область ее видимости ограничивается блоком.

Внешнюю переменную можно определить со словом `static`. Смысл этого слова для внешних переменных совсем другой, чем для локальных переменных. Область видимости *статической внешней* переменной ограничивается остатком файла, в котором она определена. В других файлах статическая внешняя переменная неизвестна, в них мы можем определить другую внешнюю переменную с тем же именем.

Прототипы функций

При записи программы, состоящей из нескольких функций, возможно, разнесенных по нескольким файлам, возникает еще одна сложность. Некая функция может быть вызвана раньше, чем она записана в файле. Более того, описание функции может оказаться в другом файле, недоступном компилятору. В этих случаях компилятор не сможет проверить соответствие типов формальных и фактических аргументов. Он даже не сможет проверить совпадение количества аргументов. Отсутствие проверок приведет к появлению множества ошибок, которые будет очень трудно обнаружить и исправить.

Для того чтобы избежать такого рода ошибок, в язык C введено правило: если функция описана позднее ее вызова, то до описания функции, в которой она вызывается, следует написать прототип вызываемой функции. *Прототип* функции — это ее заголовок, включающий тип возвращаемого значения, имя функции и список аргументов в скобках. Сразу после закрывающей скобки ставится точка с запятой:

```
#include <stdio.h>

double sum(const double *a, const int len);
```

```
void main(){
   double x[] = {0.01, -3.45, 67.8, -23.5, 0.7, -23.7, 0.045};
   double z = sum(x, 7);
}

double sum(const double *a, const int len){
   double s = 0.0; int k;
   for (k =0; k < len; k++) s += a[k];
   return s;
}
```

Прототип содержит все сведения, необходимые компилятору для правильного вызова функции. Правила хорошего тона предписывают записывать в начале файла прототипы всех функций, имеющихся в нем. Это облегчает чтение программы. Открыв файл, вы сразу видите, какие функции он содержит, и можете понять по их именам и набору аргументов, что делают эти функции.

Прототипы функций наряду с определениями констант часто записывают в отдельный файл и подключают его в начало нужного файла директивой препроцессора `#include`.

Рекурсия

В математике часто встречается рекурсивное определение функции. Значение функции от аргумента n определяется через одно или несколько предыдущих значений $n-1$, $n-2$, ... Классический пример — определение факториала: $0!=1$, $n!=n\cdot(n-1)!$.

По этому определению 4! = 4 · 3!, но 3! = 3 · 2!, а 2! = 2 · 1!, 1! = 1 · 0! и мы дошли до конца рекурсии, раскрыли "матрешку" вложенных друг в друга определений. После этого движемся в обратном порядке, выполняя умножения: 1! = 1 · 1 = 1, 2! = 2 · 1 = 2, 3! = 3 · 2 = 6, наконец, 4! = 4 · 3 = 24.

Более сложная последовательность чисел Фибоначчи 1, 1, 2, 3, 5, 8, ... записывается рекурсивной формулой: $a_1 = a_2 = 1$, $a_n = a_{n-1} + a_{n-2}$.

Язык C допускает вызов функции внутри самой функции, поэтому рекурсию легко запрограммировать:

```
int fact(int n){
   if (n < 0) return -1; /* Неправильный аргумент. */
   if (n == 0) return 1;
   return n * fact(n - 1);
}
```

Вычисление n-го числа Фибоначчи ничуть не сложнее:

```
int fib(int n){
   if (n <= 0) return -1;
   if (n == 1 || n == 2) return 1;
   return fib(n - 1) + fib(n - 2);
}
```

Конечно, вычислять рекурсивно такие простые функции не стоит, рекурсия требует большого расхода ресурсов. Здесь достаточно простого цикла:

```
int fact(int n){
   if (n < 0) return -1; /* Ошибка. */
   int k, f = 1;
   for (k = 2; k <= n; k++) f *= k;
   return f;
}
```

Числа Фибоначчи тоже лучше вычислять в цикле:

```
int fib(int n){
   if (n <= 0) return -1;
   int k, a1 = 1, a2 = 1, f = 1;
   for (k = 3; k <= n; k++){
      f = a1 + a2;
      a1 = a2;
     a2 = f;
   }
   return f;
}
```

В более сложных случаях, например, при построении деревьев, рекурсия помогает написать ясную и короткую программу.

Создавая рекурсивную функцию, следует тщательно проследить за тем, чтобы рекурсия не оказалась бесконечной, чтобы "матрешка" раскрылась до конца и достигла конкретных начальных значений. Ошибки, вызванные бесконечной рекурсией, выявляются только во время выполнения программы, когда их трудно отследить и исправить.

Быстрая сортировка

В конце прошлой главы мы рассмотрели методы сортировки массива и отметили, что самый быстрый метод сортировки, так и называемая быстрая сортировка (quick sort), основан на рекурсии.

Метод быстрой сортировки заключается в том, что выбирается средний элемент массива, например, элемент `a[LEN/2]`, где `LEN` — длина массива. Левее этого элемента выбираются элементы, бóльшие `a[LEN/2]`, а правее — элементы, меньшие `a[LEN/2]`, после чего происходит перестановка этих элементов. В результате левее выбранного элемента `a[LEN/2]` окажутся все элементы, меньшие его, а правее — все элементы, бóльшие его. После этого алгоритм рекурсивно повторяется отдельно для левой половины массива и для правой половины.

Технически это оформлено в виде рекурсивной функции `quicksort(left, right)`, в которую передаются индексы левого `left` и правого `right` конца подмассива. В функции устанавливаются индексы `l` и `r`, которые передвигаются от концов к середине массива слева и справа. Индекс `l` останавливается на первом элементе массива, большем, чем средний элемент `m`, а индекс `r` — на первом справа элементе массива, меньшем, чем `m`. После этого элементы `a[l]` и `a[r]`, если такие элементы найдены, меняются местами, а индексы `l` и `r` продолжают продвигаться к середине массива.

После окончания этого цикла оказывается, что `l > r`. Если после этого `left < r` или `l < right`, то вызов функции надо повторить с новыми аргументами (листинг 5.4).

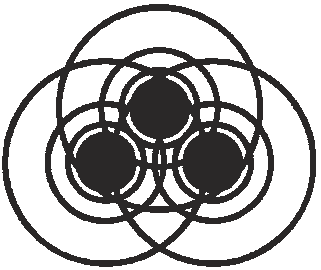
Листинг 5.4. Быстрая сортировка массива

```
#include <stdio.h>

#define LEN 10


int a[] = {15, 12, 5, 7, 0, -2, 4, 8, -3, 10};


void quicksort(int left, int right){


   int m = a[(left + right)/2];

   int l = left, r = right;


   while (l <= r){


      while (a[l] < m) l++;

      while (a[r] > m) r--;
```

```
            if (l <= r){
               int t = a[l]; a[l] = a[r]; a[r] = t;
               l++; r--;
            }
         }
         if (left < r)  quicksort(left, r);
         if (l < right) quicksort(l, right);
      }

      main(){
         quicksort(0, LEN - 1);
      }
```

Контрольные вопросы

1. Что такое парадигма программирования?
2. Что такое процедурное программирование?
3. Что такое модульное программирование?
4. Какая разница между процедурой и функцией?
5. Можно ли написать процедуру в языке C?
6. Каким образом вернуть два или более значений из функции?
7. Можно ли описать функцию внутри другой функции?
8. Можно ли создать внешнюю переменную внутри функции?
9. Можно ли записать текст одной функции в несколько файлов?
10. Обязательно ли записывать прототип для каждой функции языка C?
11. В каких случаях прототип необходим?
12. Рекурсия называется взаимной, если две функции вызывают друг друга. Можно ли в языке C создать взаимно рекурсивные функции?



Глава 6

Работа с файлами

В предыдущих главах мы вводили исходные данные с клавиатуры, а результаты выводили на экран дисплея. На практике гораздо чаще приходится связывать ввод и вывод программы с другими внешними устройствами: жесткими и гибкими дисками, принтерами, сканерами, картами флэш-памяти. На этих устройствах информация обычно хранится в файлах, с которыми программа должна уметь работать: отыскивать на диске, читать их, записывать в них информацию, создавать новые.

В стандартную поставку языка С входит обширная библиотека функций ввода-вывода информации. Она описана в заголовочном файле stdio.h (standard input-output). Мы уже использовали функцию форматированного ввода `scanf()` и функцию форматированного вывода `printf()` из этой библиотеки.

Функции, входящие в библиотеку ввода-вывода, рассматривают файл как последовательность байтов. Байты пронумерованы, начиная с 0, функции позволяют перемещаться по файлу в обоих направлениях и непосредственно обращаться к любому байту. Номер текущего байта в файле отслеживается при всех операциях с файлом. Он называется *текущей позицией* (current position) файла. Функции ввода-вывода могут прочитать из файла или записать в файл любое количество байтов, начиная с текущей позиции, после чего текущая позиция увеличивается на обработанное число байтов.

Перед работой с файлом он должен быть *открыт* (open). Операция открытия включает в себя поиск каталога, в котором описан файл, поиск файла на диске по информации, полученной из каталога, проверку блоков файла, создание в оперативной памяти специальной структуры и заполнение ее информацией о файле. Тип этой структуры `FILE`, он описан в заголовочном файле stdio.h. После открытия файла можно заняться *поиском* (seek) нужного байта в файле, как говорят по аналогии с магнитной лентой, перематывать файл.

Найдя начало нужной информации, можно *прочитать* (read) ее или *записать* (write) в нужное место новую информацию. При чтении файла информация сначала записывается в специально выделенную область оперативной памяти — *буфер* (buffer) ввода, а потом забирается оттуда и записывается в переменные и массивы программы, читающей файл. То же происходит и при выводе в файл. Информация из программы сначала накапливается в буфере вывода, а затем записывается на диск. Буферизация ввода-вывода значительно ускоряет операции с файлом. Для еще большего ускорения работы программа, как правило, создает не один, а несколько буферов ввода-вывода. В то время как один буфер заполняется, программа забирает информацию из другого.

Наконец, по окончании работы с файлом его надо *закрыть* (close). Операция закрытия сбрасывает содержимое буферов в файл, обновляет информацию о файле в каталоге и освобождает оперативную память, занятую буферами и структурой типа `FILE`. Во время завершения программы операционная система автоматически закрывает все открытые программой файлы, но чтобы не засорять оперативную память, лучше закрывать файлы сразу же после работы с ними.

Итак, вся работа с файлами заключается всего в нескольких действиях: "открыть", "отыскать", "прочитать/записать" и "закрыть". Каждое действие выполняется соответствующей функцией библиотеки ввода-вывода. Рассмотрим их подробнее.

Функции буферизованного ввода-вывода

Открытие файла выполняет функция `fopen()`. Ей передаются два аргумента: имя файла в виде указателя на строку и еще одна строка, задающая режим открытия файла. Функция среди прочих действий определяет структуру типа `FILE` и возвращает указатель на нее. Этот указатель затем используют другие функции, чтобы получить информацию об открытом файле. Если файл не удалось открыть, то возвращается константа `NULL`. Например:

```
#include <stdio.h>
FILE *f1 = fopen("C:\\myworks\\myfile1.txt", "r");
FILE *f2 = fopen("/home/khabib/myworks/myfile2.txt", "w");
```

В первой функции `fopen()` полное имя файла написано по правилам операционной системы MS-DOS. Каталоги разделяются обратной наклонной чертой, которую приходится удваивать, поскольку в языке C у обратной наклонной черты специальный смысл. Во второй функции полное имя файла написано по правилам операционных систем UNIX.

В классических операционных системах файл открывается в одном из двух режимов: на чтение или на запись. Файл, открытый на чтение, можно просматривать, считывать из него любую информацию, но его нельзя изменять.

Файл, открытый на запись, прежде всего очищается. Если в нем содержалась какая-то информация, то она удаляется, как говорят по аналогии с магнитной лентой, стирается. Потом происходит запись в пустой файл. В функции `fopen()` режим чтения задается строкой `r`, состоящей из одного символа, а режим записи — строкой `w`.

Внимание!

Файл, открытый на запись, стирается.

Если вы захотите прочитать только что записанную в файл информацию, то файл сначала надо будет закрыть, а потом открыть на чтение. Если файл, открываемый на чтение, не существует, то выдается сообщение об ошибке и выполнение программы прекращается. Если не существует файл, открываемый на запись, то он создается.

Поскольку операция записи не позволяет дополнить существующий файл новой информацией, был введен еще один режим открытия: открытие файла на дозапись. В этом режиме файл открывается на запись и сразу перематывается в конец, так что запись идет в конец файла. Прежнее содержимое файла изменить невозможно, даже перемотав файл на начало. В функции `fopen()` режим дозаписи задается строкой `a` (append).

В современных операционных системах файл, как правило, открывается сразу и на чтение, и на запись. Его можно перематывать, читать и записывать, не закрывая и не открывая вновь. Тем не менее, стандарт языка С рекомендует между операциями чтения и записи делать перемотку функциями `fseek()` или `rewind()`. Если на самом деле не надо перемещаться по файлу, то можно сделать ложную перемотку, перемотав файл на ноль байтов. В функции `fopen()` расширенные режимы открытия файла отмечаются плюсом: `r+`, `w+`, `a+`. Не забывайте, что в режиме `w+` содержимое файла, если оно было непустым, сотрется, а в режиме `a+` прежнее содержимое файла изменить нельзя.

Некоторые операционные системы различают текстовые файлы, в которых каждый байт считается символом, и бинарные файлы, содержащие изображение, звуки или машинные команды. В таких системах режим открытия уточняется параметром `t` (text) или `b` (binary), например, `rb`. Если такого уточнения нет, то по умолчанию понимается текстовый файл. В текстовом файле символы перевода строки `\n` и возврата каретки `\r` обрабатываются по-разному в разных операционных системах, поэтому надо соблюдать осторожность при работе с файлом, открытым как текстовый файл.

Итак, в функции `fopen()` можно задать один из шести режимов открытия файла. Заметьте, что режим задается строкой, а не символом, т. е. надо всегда записывать кавычки, а не апострофы, даже если режим записывается одной буквой. После открытия можно переместиться по файлу с помощью

функции `fseek()`. Ей передаются три аргумента: указатель на файл, *смещение* (offset) — число байтов, на которые надо передвинуться по файлу, и направление перемещения, задаваемое числом 0, 1 или 2.

Если направление указано числом 0, то перемещение идет от начала файла. В этом случае смещение должно быть неотрицательным. Если направление задано числом 1, то перемещение начинается от текущей позиции в файле. В этом случае смещение может быть и неотрицательным, и отрицательным. Наконец, если направление перемещения указано числом 2, то оно идет от конца файла к его началу, и смещение должно быть отрицательным. Во многих системах программирования в файле stdio.h определены константы `SEEK_SET`, `SEEK_CUR` и `SEEK_END`, соответствующие числам 0, 1 и 2.

Смещение задается длинным целым числом типа `long`. Поскольку компилятор не делает преобразований типа, его надо соблюдать даже в константах. Например:

```
#include <stdio.h>
FILE *f;
if ( (f = fopen("/home/khabib/myworks/myfile.txt", "r+")) != NULL){
   fseek(f, 200L, 0); /* перемотка на 200 байтов от начала файла. */
   fseek(f, 200L, SEEK_SET); /* То же самое. */
   fseek(f, -100L, SEEK_CUR); /* Перемотка на 100 байтов от
                                 текущей позиции к началу файла. */
   fseek(f, 0L, SEEK_END); /* Перемотка в конец файла */
   fseek(f, 0L, SEEK_SET); /* Перемотка в начало файла */
}
```

Функция `fseek()` возвращает 0 в случае удачного перемещения и ненулевое значение при ошибке, например, при попытке переместиться по еще не открытому файлу. При работе с текстовыми файлами функция `fseek()` может сделать неверную перемотку, вызванную разной интерпретацией символов перевода строки и возврата каретки.

Для удобства работы с файлами в язык С введена функция `rewind()`, перематывающая файл на начало. У нее один аргумент — указатель на открытый файл — и нет возвращаемого значения.

Текущая позиция файла хранится в структуре типа `FILE`, созданной при открытии файла. Она возвращается функцией `ftell()` в виде числа типа `long`. После того как установлена правильная позиция в файле, можно читать или записывать информацию. Чтение осуществляется функцией `fread()`. У нее четыре аргумента. Первый аргумент — указатель типа (`char *`) на буфер, в который будет записана прочитанная информация. Второй аргумент — количество байтов в одной порции читаемой информации, третий аргумент —

количество таких порций. Наконец, четвертый аргумент — указатель на структуру типа `FILE`, созданную при открытии файла. Например:

```
#include <stdio.h>

FILE *f;
char buf[4096];
if ( (f = fopen("/home/khabib/myworks/myfile.txt", "r+")) != NULL)
   fread(buf, 4096, 1, f);
```

Возникает вопрос, что лучше: прочитать один раз 4096 байтов, как написано выше, или 4096 раз по одному байту, как написано ниже?

```
fread(buf, 1, 4096, f);
```

Ответ прост и очевиден: поскольку операционная система читает и записывает информацию блоками, лучше переносить за один раз целую порцию байтов. Здесь помогает операция `sizeof()`, вычисляющая длину типа или массива. Например:

```
fread(buf, sizeof(buf), 1, f);
```

Но лучше всего читать информацию блоками операционной системы. Размер блока хранится в константе `BUFSIZ`, определенной в файле stdio.h. Приведенный выше пример можно записать так:

```
#include <stdio.h>

FILE *f;
char buf[BUFSIZ];
if ( (f = fopen("/home/khabib/myworks/myfile.txt", "r+")) != NULL)
   fread(buf, BUFSIZ, 1, f);
```

Функция записи информации в файл `fwrite()` выглядит так же, но переносит информацию в противоположном направлении — из буфера `buf` в файл `f`.

```
#include <stdio.h>

FILE *f;
char buf[] = "Информация, записываемая в файл.";
if ( (f = fopen("/home/khabib/myworks/myfile.txt", "w+")) != NULL)
   fwrite(buf, sizeof(buf), 1, f);
```

Функции `fread()` и `fwrite()` возвращают число фактически обработанных полных порций или 0, если чтение или запись не удались, например, встретился конец файла. После чтения или записи текущая позиция файла увеличивается на число прочитанных или записанных байтов. Проверить окончание файла можно функцией `feof()`, возвращающей ненулевое значение, если предыдущее чтение дошло до конца файла, и 0, если конец файла еще не достигнут.

После работы с файлом его надо закрыть функцией `fclose()`. При закрытии файла буфер очищается, но до этого может случиться, что последняя порция информации останется в буфере и не будет записана в файл. Для того чтобы предотвратить эту неприятность, надо принудительно очистить буфер, сбросив его содержимое в файл. Эту работу выполняет функция `fflush()`.

Полный цикл работы с файлами приведен в листинге 6.1.

Листинг 6.1. Функция ввода-вывода низкого уровня

```
#include <stdio.h>

main(){
   char buf[BUFSIZ];
   FILE *fr, *fw;
   if ( (fr = fopen("myoldfile.txt", "r")) == NULL){
       printf("Не удается открыть существующий файл.\n");
       return -1;
   }
   if ( (fw = fopen("mynewfile.txt", "w")) == NULL){
       printf("Не удается открыть новый файл.\n");
       return -1;
   }
   while (!feof(fr)){
      fread(buf, BUFSIZ, 1, fr);

      /* Работа с буфером buf */

      fwrite(buf, BUFSIZ, 1, fw);
   }
   fclose(fr);
   fclose(fw);
}
```

Упражнения

1. Прочитайте начало какого-нибудь из имеющихся у вас на диске файлов в буфер и выведите содержимое буфера на экран. Посмотрите, в каком виде хранятся байты этого файла.
2. Запишите массив чисел в файл. Прочитайте этот файл и восстановите массив.

3. В операционных системах семейства UNIX первые четыре байта большинства типов файлов — это "magic number" — число, показывающее тип файла. Напишите функцию, возвращающую первые четыре байта файла, заданного аргументом функции.
4. В операционной системе MS-DOS первые два байта EXE-файлов равны 0100110101011010. Это инициалы "MZ" создателя EXE-формата Марка Збиковски (Mark Zbikowski). Напишите функцию, проверяющую эти байты у файла, заданного ее аргументом, и возвращающую 1, если это EXE-файл, и 0 в противном случае.

Потоки ввода-вывода

С компьютером связано множество внешних устройств: принтеры, сканеры, компакт-дисководы, карты флэш-памяти, цифровые фотоаппараты и видеокамеры. Обмен информацией с каждым из них имеет свои особенности, которые приходится учитывать при программировании ввода-вывода. Более того, даже у разных моделей одного и того же устройства бывает разница в передаче информации. Вся история программирования пронизана стремлением сгладить эту разницу. Понятие файла — одна из больших удач на этом пути. Мы обращаемся к файлу по имени, не думая о том, на каком цилиндре диска он расположен. Мы рассматриваем файл как последовательность байтов, не зная о том, какие блоки диска он занимает, где расположены эти блоки и как записаны байты в этих блоках.

Следующим шагом на пути унификации работы с внешними устройствами стало понятие *потока* (stream) информации. Ввод информации рассматривается как *входной поток* байтов, "втекающий" откуда-то в компьютер (input stream), вывод — как *выходной поток* байтов, "вытекающий" из компьютера (output stream). При этом нам совершенно не важно, откуда поступает поток информации и куда он уходит. Это дело операционной системы. Мы пишем функции ввода-вывода, беря исходную информацию из входного потока и отправляя результаты в выходной поток. Перед выполнением программы операционная система связывает потоки с конкретными устройствами. Она может переназначить поток: сейчас информация выводится на принтер, через некоторое время она пойдет в файл или будет направлена в сеть.

Все стандартные функции ввода-вывода языка C рассчитаны на самом деле не на работу с файлами, а на работу с потоками. Указатель на структуру типа `FILE` — создаваемый при открытии файла поток. На другом его конце может оказаться не диск, а совсем другое устройство, лишь бы оно умело работать с файлами по их именам.

Перед началом работы каждой программы для нее автоматически открываются три стандартных потока: входной поток с номером 0 и именем `stdin`, выходной поток с номером 1 и именем `stdout`, выходной поток с номером

2 и именем stderr. Входной поток сразу же связывается с клавиатурой, а выходные потоки — с экраном дисплея. Потом эти устройства можно переназначить.

Выходной поток stderr рассчитан на передачу программой сообщений о своей работе. Чаще всего это предупреждения и сообщения об ошибках, откуда и произошло название "standard error stream". Из-за того что оба стандартных выходных потока направлены на одно и то же устройство, они часто смешиваются, что вы, наверное, могли наблюдать на консоли своего компьютера: среди результатов вашей программы неожиданно начинают появляться малопонятные сообщения о каких-то ошибках. Чтобы не происходило такого смешения, многие программы, особенно серверы, направляют поток stderr в какой-либо журнальный (log) файл, в котором накапливается полный отчет о работе программы. Все администраторы компьютерных систем хорошо знают о том, что надо регулярно просматривать журнальные файлы и время от времени очищать их.

Функция scanf(), которой мы пользовались в предыдущих главах, рассчитана на ввод из стандартного потока stdin, функция printf() — на вывод в стандартный поток stdout. Функции fread() и fwrite() можно связать со стандартными потоками:

```
fread(buf, BUFSIZ, 1, stdin);
fwrite(buf1, sizeof(buf1), 1, stdout);
fwrite(buf1, sizeof(buf1), 1, stderr);
```

При этом стандартные потоки открывать не надо, они уже открыты.

Потоки stdin, stdout и stderr — это константы, они не могут присвоить другой поток. Переназначить стандартный поток можно средствами операционной системы при запуске исполнимого файла:

```
$ ./mycoolprog >myoutputfile 2>mylogfile
```

Перед выполнением mycoolprog стандартный вывод stdout будет направлен в файл myoutputfile, а стандартный поток сообщений stderr будет направлен в файл mylogfile. Поскольку файлы myoutputfile и mylogfile открываются на запись, они предварительно будут очищены.

Этот способ очень распространен в операционных системах семейства UNIX. Программа для них очень часто пишется в расчете на стандартные потоки ввода-вывода, а при необходимости потоки перенаправляются операционной системой.

Форматированный ввод-вывод

В стандартной библиотеке ввода-вывода есть две функции, аналогичные функциям scanf() и printf(), но рассчитанные не только на стандартные

потоки. Это функции `fscanf()` и `fprintf()`. Предварительно открытый поток в них указывается первым аргументом, а остальные аргументы точно такие же, как у функций `scanf()` и `printf()`.

Строго говоря, все обстоит так: функция `scanf("format", arguments,...)` — это частный случай функции `fscanf()`, а именно: `fscanf(stdin, "format", arguments,...)`. А функция `printf("format", arguments,...)` — это функция `fprintf(stdout, "format", arguments,...)`.

Функцией `fprintf()` очень часто пользуются для записи сообщений об ошибках, направляя их в поток `stderr`:

```
if ( (f = fopen("myfile", "r")) == NULL){
    fprintf(stderr, "Невозможно открыть файл\n");
    return -1;
}
```

Функцией `fscanf()` пользуются реже, потому что она требует тщательной записи входного потока. Любая ошибка во входном потоке приведет к непредсказуемым результатам в ее работе.

Еще одна полезная функция из этого семейства заносит информацию не в выходной поток, а в строку символов. Это `sprintf()`. Первым ее аргументом служит не указатель на структуру типа `FILE`, а указатель на строку типа (`char *`). Эта функция удобна для записи чисел в виде строк, состоящих из цифр, десятичных точек, знаков плюс и минус. Например:

```
char s[100];
sprintf(s, "Сегодня %d градусов.", temperature);
```

В библиотеке есть и двойственная функция `sscanf()` форматированного ввода из строки символов по указанным форматам в переменные, адреса которых указаны следующими аргументами. Пример:

```
int k;
char s[10];
sscanf("126 ламп", "%d%s", &k, s);
```

Прочие функции ввода-вывода

Мы рассмотрели в этой главе функции ввода-вывода самого низкого уровня `fread()` и `fwrite()`, представляющие ввод-вывод данных в виде простого потока байтов. Эти функции не интересуются тем, какую информацию несут байты, числа это, символы или машинные команды. Они не могут проверить правильность передаваемой информации, не могут отследить искаженные или потерянные байты.

С другой стороны, мы пользовались функциями ввода-вывода самого высокого уровня `fscanf()` и `fprintf()`, представляющими информацию в виде значений простых типов данных. Они проверяют вводимые или выводимые данные, пытаются их преобразовать к указанному типу и сообщают о неудачных попытках.

Между этими двумя парами функций располагаются функции ввода-вывода, передающие отдельные символы или строки символов.

Функция с заголовком `int fgetc(FILE *stream)` вводит в программу один символ из потока, заданного своим аргументом, возвращая его как тип `int`. Тип `int`, а не `char` выбран потому, что в случае неудачи функция возвращает константу `EOF`, равную −1, которая не соответствует типу `char`. Сам символ в формате типа `unsigned char` будет занимать младший байт возвращаемого значения.

```
FILE *f;
int c;
if ( (f = fopen("myfile", "r+") == NULL){
   fprintf(stderr, "Ошибка открытия файла.");
   return -1;
}
if ( (c = fgetc(f) == EOF) {
   fprintf(stderr, "Ошибка чтения файла.");
   return -1;
}
char ch = (char)c;
```

Функция с заголовком `int fputc(char c, FILE *stream)` выводит символ `c`, указанный первым аргументом в выходной поток `stream`, заданный вторым аргументом. Функция возвращает в младшем байте выводимое значение или `EOF` в случае неудачи.

В листинге 6.2 приведена программа посимвольного копирования файлов. Она удобна для работы с символьными устройствами ввода-вывода. Имена файлов вводятся в командной строке при запуске программы на выполнение.

Листинг 6.2. Посимвольный ввод-вывод

```
#include <stdio.h>

main(int argc, char **argv){
   FILE *in, *out;
```

```
   int ch;
   if (argc != 3){
      printf("Usage: mycopy source target\n");
      return -1;
   }
   if ( (in = fopen(argv[1], "r")) == NULL){
       printf("Не удается открыть копируемый файл.\n");
       return -1;
   }
   if ( (out = fopen(argv[2], "w")) == NULL){
       printf("Не удается открыть новый файл.\n");
       return -1;
   }

   while ((ch = fgetc(in)) != EOF) fputc(ch, out);

   fclose(in);
   fclose(out);
}
```

Функции `int getchar()` и `int putchar(char c)` делают то же самое, но они жестко связаны со стандартными потоками `stdin` и `stdout` соответственно. Вызов `getchar()` эквивалентен вызову `fgetc(stdin)`, вызов `putchar(c)` эквивалентен вызову `fputc(c, stdout)`.

Вот как можно организовать эхо-отображение на экране символов, вводимых с клавиатуры:

```
int c;
while ((c = getchar()) != EOF) putchar(c);
```

Поскольку `stdout` — буферизованный поток, причем его буфер очищается только после своего заполнения или после попадания в него символа перевода строки, символы будут отображаться на экране только после нажатия клавиши <Enter>. Для окончания ввода с клавиатуры надо нажать комбинацию клавиш <Ctrl>+<D>, введя тем самым признак конца файла.

После работы всех этих функций текущая позиция файла перемещается на один символ вперед.

Передача за один раз по одному символу часто оказывается нецелесообразной. Во многих случаях удобнее вводить или выводить сразу целую строку. В таком случае используются функции `fgets()` и `fputs()`.

Функция с заголовком `char *fgets(char *s, int size, FILE *stream)` вводит максимум `size - 1` символов из входного потока `stream` и заносит

их в буфер, на который указывает `s`, добавляя в конец нуль-символ. Функция возвращает указатель на тот же буфер `s` или `NULL`, если ввод не удался. Ввод также заканчивается при переполнении буфера и при появлении в нем символа перевода строки `\n`.

Функция с заголовком `int fputs(char *s, FILE *stream)` выводит символы из буфера `s` до нуль-символа исключительно в выходной поток `stream`. Функция возвращает код последнего выведенного символа, в случае ошибки возвращается `EOF`.

В листинге 6.3 приведен еще один вариант копирования файлов.

Листинг 6.3. Построчный ввод-вывод

```
#include <stdio.h>

main(){
   FILE *infile, *outfile;
   char s[81];

   if ( (infile = fopen("myoldfile.txt", "r+")) == NULL){
       printf("Не удается открыть копируемый файл.\n");
       return -1;
   }
   if ( (outfile = fopen("mynewfile.txt", "w+")) == NULL){
       printf("Не удается открыть новый файл.\n");
       return -1;
   }
   while(!feof(infile)){
      fgets(s, sizeof(s), infile);
      fputs(s, outfile);
   }
   fclose(infile);
   fclose(outfile);
}
```

Для этих функций есть соответствующая пара функций ввода-вывода строк, связанных со стандартными потоками.

Функция с заголовком `char *gets(char *s)` вводит символы до символа перевода строки исключительно из стандартного потока `stdin` в буфер `s`, добавляя нуль-символ. Функция возвращает указатель на тот же буфер или `NULL`, если ввод не удался.

Функция `int puts(char *s)` выводит символы из буфера `s` до нуль-символа исключительно в стандартный выходной поток `stdout`. В конце добавляется символ перевода строки `\n`. В случае ошибки функция возвращает `EOF`.

Листинг 6.4. Построчный ввод-вывод на консоль с преобразованием

```
#include <stdio.h>
#include <stdlib.h>

main(){
   char name[81];
   char buf[81];

   puts("Введите фамилию, имя и отчество:");
   gets(name);

   puts("Введите сумму вклада:");
   int deposit = atoi(gets(buf));

   printf("%s %d\n", name, deposit);
}
```

Функция с заголовком `int ungetc (int c, FILE *stream)` возвращает один символ в буфер, связанный с входным потоком `stream`. Этот символ можно будет прочитать следующим вызовом функции `fgetc()`. Если значение `c` равно `EOF`, то ничего не меняется. Возвращаемое значение функции равно `c` или `EOF`. Эта функция кажется искусственной, но ее приходится часто применять при вводе. Например, вводя число и читая его цифры, мы должны остановиться в тот момент, когда встретился символ, отличный от цифры. Но для этого мы должны прочитать этот символ. Он может принадлежать следующему вводимому значению, поэтому его нельзя терять, а следует вернуть в поток для последующего чтения. Входной поток может быть недоступен, например, он связан с сетью, поэтому возврат происходит в буфер, связанный с потоком.

Вот простой тренировочный пример, возвращающий символ "G" в пустой стандартный входной поток `stdin`, и забирающий его оттуда:

```
#include <stdio.h>

main(){
   char ch;
```

```
    ungetc('G', stdin);
    ch = getchar();
    putchar(ch);
}
```

В библиотеке функций ввода-вывода есть функции, позволяющие работать не только с содержимым файла, но и с файлом целиком.

Функция `int rename(const char *oldname, const char *newname)` меняет имя файла с `oldname` на имя `newname`. Она возвращает 0 при удачном переименовании файла и −1 в случае неудачи. Например:

```
rename("C:\\works\\f1.txt", "D:\\archive\\f1.txt.arch");
```

Функция `int remove(const char *filename)` удаляет файл с именем `filename`. Она возвращает 0 при удачном удалении файла и −1 в случае неудачи. Например, `remove("f1.txt")`.

Упражнения

5. Напишите функцию `www_encode(FILE *f)`, читающую текстовый файл, заменяющую специальные символы тремя символами `%XX`, где `XX` — шестнадцатеричный код символа, и отправляющую преобразованный файл в стандартный выходной поток.
6. Напишите функцию `print_tab(char *filename, int tabsize)`, читающую текстовый файл с именем `filename` и выводящую его в стандартный выходной поток с заменой символа горизонтальной табуляции `\t` на пробелы, число которых задает аргумент `tabsize`.
7. В текстовом файле находится платежная ведомость с заголовком и таблицей с двумя столбцами: фамилией с инициалами и суммой выплаты. Найдите среднюю выплату и выведите ее в стандартный выходной поток.

Интерактивный ввод-вывод

Во многих системах программирования на языке C стандартная библиотека функций ввода-вывода дополнена функциями, работающими непосредственно с клавиатурой и экраном дисплея. Такие функции описаны в файле conio.h (console input-output).

Функция `int getch()` вводит символ с клавиатуры без эхо-отображения его на экране. Возвращает код введенного символа. Ввод не буферизуется, поэтому ввод не надо дополнять нажатием клавиши <Enter> или очисткой буфера. Эту функцию часто используют для приостановки выполнения программы до нажатия какой-нибудь клавиши, например, при просмотре на экране большого объема информации.

Функция `int getche()` вводит символ с клавиатуры и отображает его на экране. Она возвращает код введенного символа.

Функция `int putch(int ch)` выводит символ на экран дисплея. Она возвращает код выводимого символа.

```
#include <stdio.h>
#include <conio.h>

main(){
   char ch;
   while (!eof(stdin)){
      puts("Для продолжения нажмите любую клавишу.");
      ch = getch();
      putch(ch);
      puts("Для окончания нажмите комбинацию клавиш Ctrl-D");
   }
}
```

Функция `int ungetch(int c)` заносит символ `c` в буфер клавиатуры и возвращает его в качестве результата своей работы. Следующая функция, вводящая символы с клавиатуры, возьмет именно этот символ, как будто бы он был только что введен с клавиатуры.

Функция `char cgets(char *s)` формирует строку `s`, рассчитанную на использование в самых разных вычислительных системах. Элемент `s[0]` уже должен содержать максимальное число вводимых символов. В элемент `s[1]` функция заносит фактическое число введенных символов. Сами символы записываются, начиная с элемента `s[2]`. Строка завершается нуль-символом. Функция возвращает указатель на начало строки `s`. Пример:

```
char s[256];
s[0] = 253;
cgets(s);
printf("%s\n", s + 2);
```

Функция `int cputs(const char *s)` выводит строку `s` на экран дисплея. Ее отличие от функции `puts()` только в том, что она не добавляет в конец символ перевода строки.

Работа с каталогом

Некоторые системы программирования содержат функции, позволяющие работать с каталогами. Эти функции описаны в файле dir.h. Вот некоторые из этих функций.

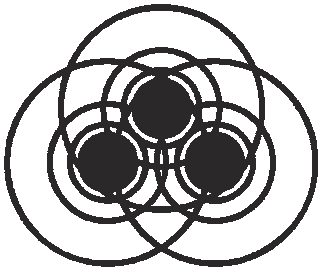
- Функция `int mkdir(const char *dirname)` создает каталог с именем `dirname` и возвращает 0 при удачном создании каталога и −1 в случае неудачи. Пример ее использования:

```
if (mkdir("C:\\myworks\\iodir")){
   fprintf(stderr, "Не удалось создать каталог");
   return -1;
}
```

- Функция `int chdir(const char *dirname)` делает каталог `dirname` текущим каталогом. Она возвращает 0 при удачном изменении каталога и −1 в случае неудачи.
- Функция `int rmdir(const char *dirname)` удаляет каталог `dirname`. Она возвращает 0 при удачном удалении каталога и −1 в случае неудачи.

Контрольные вопросы

1. Что такое файл в языке C?
2. Какая разница между бинарным и текстовым файлом?
3. Какие операции можно делать с файлом с помощью функций языка C?
4. Что входит в процедуру открытия файла?
5. К чему приведет повторное открытие уже открытого файла?
6. Могут ли несколько программ открыть один и тот же файл в одно и то же время?
7. Можно ли изменить содержимое файла, открытого в режиме `"a+"`?
8. Можно ли изменить содержимое файла, открытого в режиме `"r+"`?
9. Можно ли выйти за пределы файла функцией `fseek()`?
10. Сколько байтов может прочитать функция `fread()` за один вызов?
11. В каком виде должен быть записан файл, читаемый функцией `fscanf()`?
12. В каком виде запишет информацию в файл функция `fprintf()`?
13. Как очистить буфер вывода после записи всей информации в файл?
14. Какой функцией удобнее всего прочитать файл с текстом программы на языке C?



Глава 7

Типы данных, определяемые пользователем

Во всех предыдущих главах мы пользовались только простыми типами данных, встроенными в язык С, и массивами, построенными на их основе. Вообще говоря, этого достаточно для написания любой программы. В конце концов, любая информация представляется в оперативной памяти последовательностью байтов и программе достаточно обрабатывать отдельные байты. Но задачи, решаемые программированием, разнообразны. С одной стороны, при написании драйверов или сетевых программ приходится работать не с байтами, а с отдельными битами. С другой стороны, многие задачи заставляют обрабатывать более крупные единицы информации — регистрационные и учетные карточки, библиографические описания, анкетные данные, различные таблицы. Для таких документов было бы удобно определить специальные типы данных и работать с ними.

Большинство языков программирования, в том числе и язык С, позволяют программисту определить свои типы данных. Определение пользовательских типов базируется на стандартных типах языка С или на отдельных байтах. Пользовательские типы необязательны, но они помогают писать наглядные, хорошо читаемые программы и избегать ошибок.

Перечислимые типы

Для записи констант у нас есть числа, отдельные символы и строки символов. Этого достаточно для работы со всеми простыми типами, но иногда их использование неудобно и затеняет суть дела. Нам хотелось бы обозначить месяцы обычными словами — "январь", "февраль" и т. д., а не числами от 1 до 12. Сезоны лучше назвать "весна", "лето", "осень", "зима", а не обозначать числами 1, 2, 3, 4 или какими-нибудь символами, вроде "В", "Л", "О" или

"З". Конечно, можно воспользоваться строковыми константами, но они будут занимать неоправданно много места в оперативной памяти, да и выполнять операции с ними весьма затруднительно.

Язык С позволяет определить новый тип, просто перечислив все его значения. Каждое значение записывается своим идентификатором. Например, мы пишем программу управления светофором. У каждого светофора три состояния. Эти состояния выражены тремя цветами: красный, желтый, зеленый. Конечно, можно их обозначить числами 0, 1, 2 или −1, 0, 1, но тогда нам все время придется вспоминать, что обозначено единицей: зеленый цвет, желтый или красный. Мы хотим записывать обычные слова: "green" (зеленый), "yellow" (желтый), "red" (красный). Это можно сделать так:

```
enum traffic_light {green, yellow, red};
```

Первое слово `enum` (enumerate) говорит о том, что дальше записана *метка* (tag) перечислимого типа `traffic_light`. Потом в фигурных скобках перечислены все значения определяемого типа. С помощью метки можно определить переменные помеченного типа:

```
enum traffic_light tl1, tl2;
```

Переменные `tl1`, `tl2` отличаются от числовых и символьных переменных. Они принимают всего по три значения каждая и их значениями будут не числа или символы, а идентификаторы, перечисленные в скобках. Заметьте, идентификаторы, а не строковые константы! Далее мы можем писать такие выражения и операторы:

```
tl1 = red;
if (tl2 == green) move();
```

Обратите внимание на то, что значения переменных не записаны в кавычках, это не строковые константы. Можно записывать и русские слова, но при переносе программы на другую платформу придется думать о кодировке кириллицы. Точно так же можно описать месяцы и сезоны:

```
enum month1 {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
enum season0 {Spring, Summer, Autumn, Winter};
```

Надо только следить за тем, чтобы один и тот же идентификатор не оказался значением нескольких разных перечислимых типов.

До сих пор мы говорили о перечислимых типах, а определяли не типы, а метки `traffic_light`, `month1`, `season0` перечислимых типов. Сами типы оставались безымянными и присутствовали только в виде набора своих значений, записанного в фигурных скобках. Язык С позволяет дать имена типам с помощью оператора `typedef`:

```
typedef enum {green, yellow, red} light;
typedef enum {Jan, Feb, Mar, Apr, May, Jun,
              Jul, Aug, Sep, Oct, Nov, Dec} month;
typedef enum {Spring, Summer, Autumn, Winter} season;
```

Оператор `typedef`, вообще говоря, не определяет новый тип, он только указывает на то, что типу, записанному после слова `typedef`, будет дано имя. Само имя типа записывается в операторе последним словом. Но в нашем случае происходит и определение перечислимого типа, и наименование его. Многие реализации языка С в стандартных заголовочных файлах вводят определения:

```
typedef enum {false, true} boolean;

typedef enum {false, true} bool;
```

Появляется логический тип, названный `boolean` или `bool`.

Таким образом, у нас появились новые типы `light`, `month`, `season` и мы можем определять переменные этих типов, как обычно записывая имя типа и имя переменной:

```
light traffic_light;
month month1, m1, m2;
season season0, s1;
```

Перечислимые типы введены в язык С не для компьютера, а для людей. Они делают программу наглядней. Такую программу легче читать и понимать. Работу компилятора перечислимые типы не облегчают. Более того, компилятор, как правило, немедленно переводит значения перечислимого типа в неотрицательные целые числа, просто перенумеровав все значения слева направо, начиная от 0. У наших типов окажется, что `green` — это 0, `yellow` — это 1, `red` — это 2 и т. д. Значения `Jun` и `Spring` в определения месяцев и сезонов — тоже нули. Программист может изменить этот порядок и задать свои значения:

```
enum traffic_light {green = 10, yellow = 20, red = 1};
```

Некоторые значения могут совпадать:

```
enum sm1 {Jan = 4, Feb = 4, Mar = 1, Apr = 1, May =1,
          Jun = 2, Jul = 2, Aug = 2, Sep = 3, Oct = 3, Nov = 3, Dec = 4};
```

Здесь различаются только месяцы, относящиеся к разным сезонам. Значения перечислимого типа могут быть и символьными:

```
enum white_symbol {BLANK = ' ', TAB = '\t', NEW_LINE = '\n',
   CARRIAGE_RETURN = '\r'};
```

Наконец, можно задать только первое значение:

```
enum traffic_light {green = 1, yellow, red};
```

Тогда следующие значения будут равны 2, 3, т. е. увеличиваться на 1. Перечисления используются не только для введения новых типов. Часто их применяют просто для определения констант:

```
enum {MIN_VAL = -999, MID_VAL = 1, MAX_VAL = 999};
```

Упражнения

1. Создайте перечислимый тип цветов радуги.
2. Задайте перечислимый тип для 16 основных цветов так, чтобы результатом сложения двух значений этого типа был цвет, полученный от сложения соответствующих цветов.
3. Создайте перечислимый тип звуковой гаммы.

Структуры

В предыдущих главах мы широко использовали массивы. Они очень удобны для единообразной обработки однотипных данных. Отличительных особенностей у массива две: один и тот же тип всех его элементов и хранение элементов в соседних ячейках оперативной памяти.

Первая особенность часто бывает неоправданно ограничительной. Во многих задачах приходится объединять данные разных типов. Возьмем, например, список сотрудников какого-либо учреждения. В нем записан табельный номер каждого сотрудника — это число, фамилия и инициалы сотрудника — это строка символов, год рождения — число, должность — строка символов или значение перечислимого типа, стаж работы — число и т. д.

Для таких ситуаций в языке C предусмотрен обобщенный тип данных `struct`, называемый *структурой* (structure). Элементы структуры, называемые ее *членами* (members), хранятся в соседних ячейках памяти, но имеют разные типы. Вследствие этого нельзя обозначить все члены структуры одним именем и различать их по индексам, как в массиве. У членов структуры должны быть разные имена. Программист, определяя свою структуру, может написать ее метку и перечислить в фигурных скобках ее члены с указанием типа каждого члена.

Скажем, нам надо обработать ведомость сдачи экзамена студенческой группой. В ведомости записаны фамилии и инициалы студентов, номера зачетных книжек и оценки. Для описания оценок, предусмотренных в вузах, введем перечислимый тип `grade`:

```
typedef enum {neud, udovl, horosho, otlichno} grade;
```

Структура каждой строки этой ведомости может выглядеть так:

```
struct ved_row{
   char student_name[50];
   int record_book_number;
   grade mark;
} ivanov, petrov, sidorov;
```

Здесь сделано определение типа структуры `struct` с меткой `ved_row`. Такой тип с меткой записывается в дальнейшем двумя словами `struct ved_row`. У типа `struct ved_row` три члена: массив символов `student_name`, целочисленная переменная `record_book_number` и переменная `mark` перечислимого типа `grade`. Сразу же после определения типа `struct ved_row`, за закрывающей фигурной скобкой, определены переменные `ivanov`, `petrov`, `sidorov` этого типа.

Если мы не собираемся нигде больше ссылаться на структуру, то ее метку можно не писать. Структура останется безымянной:

```
struct {
   char student_name[50];
   int record_book_number;
   grade mark;
} ivanov, petrov, sidorov;
```

Так же, как и для перечислимых типов, оператором `typedef` можно дать имя типу структуры:

```
typedef struct {
   char student_name[50];
   int record_book_number;
   grade mark;
} ved_row_type;
```

Но на практике гораздо чаще используют не имя типа структуры, а метку типа. Определение переменных можно сделать позднее. Сначала можно определить только тип структуры и пометить его:

```
struct ved_row{
   char student_name[50];
   int record_book_number;
   grade mark;
};
```

Обратите внимание на то, что хотя переменных нет, но точка с запятой сохранилась! Огромное количество ошибок возникает из-за того, что забывают ставить эту точку с запятой.

Внимание!

Следите за тем, чтобы определение структуры заканчивалось точкой с запятой.

Затем можно определить переменные, как обычно, указывая их тип или метку и перечисляя их имена через запятую:

```
struct ved_row ivanov, petrov, sidorov;
```

Определения структур очень часто записываются в заголовочных файлах, а переменные типа структуры определяются в программе.

Типы членов структуры различаются между собой, поэтому со всей структурой целиком можно сделать только самые примитивные действия: передать в функцию как аргумент, возвратить из функции, скопировать в другую структуру того же типа: `ivanov = petrov`. Большинство же действий совершается с отдельными членами структуры. Поскольку у всех переменных одного типа члены называются одинаково, их приходится уточнять именем переменной-структуры. Для этого записывается имя переменной и через точку имя члена: `ivanov.mark`, `sidorov.mark`, `ivanov.record_book_number`. Эти *полные* имена — обычные переменные своих типов, с ними можно делать все, что предусмотрено их типом: брать элемент массива `sidirov.student_name[0]`, складывать целые числа: `ivanov.record_book_number + petrov.record_book_number`, хотя в данной структуре в этом сложении нет смысла, и т. д.

Член структуры может быть любого типа, кроме функции. В частности, им может быть снова структура. Например, мы можем детализировать имя, отчество и фамилию студента и использовать эту структуру типа `struct person` при определении структуры типа `struct ved_row`:

```
struct person{
   char name[50];
   char second_name[60];
   char surname[60];
};

struct ved_row{
   struct person student_name;
   int record_book_number;
   grade mark;
};
```

Тогда, чтобы узнать фамилию или имя студента, надо записать три идентификатора через точку: `ivanov.student_name.surname`, `petrov.student_name.name`.

Итак, можно определить структуру, членами которой будут массивы или структуры. С другой стороны, можно определить массив структур:

```
struct ved_row gr342[25], gr343[30], gr344[20];
```

Массив `gr342` содержит 25 структур типа `struct ved_row` — 25 строк ведомости 342-й учебной группы, массив `gr343` — 30 структур, массив `gr344` — 20 структур. При обращении к элементам этих массивов возникают такие имена:

```
gr342[10].mark,
```

```
gr343[1].student_name,
gr342[8].student_name.second_name,
gr344[5].student_name.name[0]
```

Каждый элемент массива — это структура, с которой можно делать все действия, допустимые для структур. Например, можно сделать присваивания:

```
ivanov = gr343[7];
petrov = gr344[18];
```

Комбинируя массивы и структуры в разных сочетаниях, можно построить сложные типы данных. Не стоит этим увлекаться без крайней необходимости, потому что программа, содержащая такие данные, сама становится сложной, плохо читаемой и плохо поддающейся отладке.

После определения переменных надо дать им какие-то значения. Это можно сделать обычным присваиванием:

```
gr342[0].student_name.name = "Иван";
gr342[0].student_name.second_name = "Петрович";
gr342[0].student_name.surname = "Сидоров";
gr342[0].student_name.record_book_number = 34201;
gr342[0].student_name.mark = otlichno;
```

Или начальной инициализацией при определении, подобно массиву:

```
struct ved_row ivanov =
    {{"Иван", "Петрович", "Сидоров'}, 34201, otlichno};
```

Со структурами допустимы операции взятия адреса и взятия значения по адресу. Поэтому можно сделать определение указателя на структуру:

```
struct ved_row *p_row = &ivanov;
```

Обращение к члену структуры через указатель должно выглядеть примерно так:

```
if ((*p_row).mark == ud) give_grant((*p_row).student_name, false);
```

Приоритет "точки" выше, чем приоритет "звездочки", поэтому поставлены скобки. Запись `*p_row.mark` привела бы к ошибке. Она означала бы, что `p_row.mark` — указатель и берется значение, на которое он указывает. В такой записи легко запутаться, поэтому в язык С введено еще одно, более краткое и более понятное обозначение в виде стрелки, построенной из дефиса и знака "больше":

```
if (p_row->mark == ud) give_grant(p_row->student_name, false);
```

Такое обозначение графически воплощает указатель на член структуры. Несмотря на введение нового обозначения, синтаксис операций остается

сложным. Приоритет инкремента и декремента ниже, чем у "звездочки", а значит еще ниже, чем у "точки" и "стрелки". Поэтому получаем следующие расшифовки некоторых операций с указателями на структуру:

- `++p_row->mark` — переход к следующему значению перечислимого типа переменной `mark`, а не переход к следующей структуре, т. е. `++(p_row->mark)`;
- `*p_row->student_name.name` — это символ `name[0]`, а не указатель `name` на строку символов;
- `*p_row++->student_name.name` — увеличение указателя `p_row` произойдет после вычисления выражения;
- `*++p_row->student_name.name` — недопустимая операция с константой `name`.

Интересно то, что членом структуры может быть указатель на нее:

```
struct list_node{
   char data[4096];
   struct list_node *next;
} ln1, *ln2;
```

Это определение кажется парадоксальным, т. к. член `next` указывает на еще не определенный тип структуры. На самом деле здесь определяется только адрес, точнее, ячейка, в которую потом будет занесен адрес структуры. Вот сделать членом структуры не указатель на нее, а переменную ее типа, действительно нельзя.

После сделанного выше определения мы получаем такие обозначения членов структур `ln1` и `ln2`:

- `ln1.data` — массив символов — член структуры `struct list_node`;
- `ln1.next` — указатель на структуру `struct list_node`;
- `*ln1.next` — структура `struct list_node`;
- `ln1.next->data` — член структуры `struct list_node`;
- `*ln1.next->data` — элемент `data[0]` члена структуры `struct list_node`;
- `ln1.next->next` — указатель на структуру `struct list_node`;
- `*ln1.next->next` — структура `struct list_node`;
- `ln2->data` — массив символов — член структуры `struct list_node`;
- `ln2->next` — указатель на структуру `struct list_node`;
- `*ln2->next` — структура `struct list_node`;
- `ln2->next->data` — член структуры `struct list_node`;

- `*ln2->next->data` — элемент `data[0]` члена структуры `struct list_node`;
- `ln2->next->next` — указатель на структуру `struct list_node`;
- `*ln2.next->next` — структура `struct list_node`.

Главное здесь — не запутаться в указателях и четко представлять себе, на что они указывают.

Структуры как аргументы и возвращаемые значения функций

Структуры часто используются в программировании как хранилища разнородной информации, которая, тем не менее, используется, как правило, вся целиком. Функции, составляющие программу, часто нуждаются в такой информации. Естественно предоставлять ее функциям через аргументы. Пусть, например, определена структура `circle` окружности с центром в точке с координатами `x` и `y` и радиусом `r`:

```
struct circle{
   double x, y, r;
};
```

Напишем функцию `in_circle()`, возвращающую 1, если заданная ее аргументами `x` и `y` точка лежит внутри окружности, и 0 в противном случае:

```
int in_circle(double x, double y, struct circle c){
   return (c.x - x) * (c.x - x) +
          (c.y - y) * (c.y - y) < c.r * c.r;
}
```

Использовать функцию `in_circle()` можно так:

```
struct circle c = {1.0, -1.0, 10.0);
int in = in_circle(0.0, -1.0, c);
```

Можно пойти дальше и определить структуру точки `point`:

```
struct point{
   double x, y;
};
```

Тогда у функции `in_circle1()` оба аргумента можно сделать структурами:

```
int in_circle1(struct point p, struct circle c){
   return (p.x - c.x) * (p.x - c.x) +
          (p.y - c.y) * (p.y - c.y) < c.r * c.r;
}
```

Для использования этой функции надо определить две структуры:

```
struct point p = {0.5, -1.5};
struct circle c = {1.0, -1.0, 10.0};
int in = in_circle1(p, c);
```

Очень часто информацию, хранящуюся в структуре, надо не только использовать, но и изменять внутри функции. Для этого функции передается указатель на структуру. Напишем функцию `move_circle()`, перемещающую центр окружности в точку `p`.

```
void move_circle(struct point p, struct circle *c){
   c->x = p.x;
   c->y = p.y;
}
```

Теперь в функцию `move_circle()` надо передавать адрес структуры:

```
struct point p = {0.5, -1.5};
struct circle c = {1.0, -1.0, 10.0};
move_circle(p, &c);
```

Структура может быть и результатом работы функции. Очень часто в функцию передаются члены структуры, и она создает экземпляр структуры.

```
struct circle create_circle(double x, double y, double r){
   struct circle tmp;
   tmp.x = x;
   tmp.y = y;
   tmp.r = r;
   return tmp;
}
```

Такую функцию можно использовать для определения и инициализации структур:

```
struct circle c1 = create_circle(0.0, 0.0, 1.0);
struct circle c2 = create_circle(1.0, -1.0, 10.0);
```

Наконец, функция может вернуть указатель на структуру. Именно так действует функция открытия файла `fopen()` — она возвращает указатель на структуру типа `FILE`. Мы можем переписать функцию `create_circle()` так, чтобы она возвращала указатель на созданную окружность:

```
struct circle *create_circle(double x, double y, double r){
   struct circle tmp;
```

```
   tmp.x = x;
   tmp.y = y;
   tmp.r = r;
   return &tmp;
}
```

Использование функции немного изменится:

```
struct circle *cp1 = create_circle(0.0, 0.0, 1.0);
struct circle *cp2 = create_circle(1.0, -1.0, 10.0);
```

Сведем все это вместе и приведем полный пример работы со структурой.

Листинг 7.1. Работа со структурой

```
#include <stdio.h>
#include <math.h>

struct point{
   double x, y;
};
struct circle{
   double x, y, r;
};

struct circle create_circle1(double x, double y, double r){
   struct circle tmp;
   tmp.x = x;
   tmp.y = y;
   tmp.r = fabs(r);
   return tmp;
}
struct circle *create_circle2(double x, double y, double r){
   struct circle tmp;
   tmp.x = x;
   tmp.y = y;
   tmp.r = fabs(r);
   return &tmp;
}
```

```
struct circle *create_circle3(point p, double r){
   struct circle tmp;
   tmp.x = p.x;
   tmp.y = p.y;
   tmp.r = fabs(r);
   return &tmp;
}
struct circle *create_circle4(circle c){
   struct circle tmp;
   tmp.x = c.x;
   tmp.y = c.y;
   tmp.r = c.r;
   return &tmp;
}

main(){
   struct point p = {1.0, -3.56};
   struct circle c = {1.2, 0.4, 10.0};

   struct circle  c1 = create_circle1(2.5, -1.2, 6.5);
   struct circle *c2 = create_circle2(1.5, -0.2, 8.0);
   struct circle *c3 = create_circle3(p, 2.5);
   struct circle *c4 = create_circle4(c);

   printf("x = %f, y = %f, r = %f\n", c1.x, c1.y, c1.r);
   printf("x = %f, y = %f, r = %f\n", c2->x, c2->y, c2->r);
   printf("x = %f, y = %f, r = %f\n", c3->x, c3->y, c3->r);
   printf("x = %f, y = %f, r = %f\n", c3->x, c3->y, c3->r);
}
```

Упражнения

4. Определите структуру, отражающую библиографическое описание книги.
5. Определите структуру, описывающую единицу товара в электронном магазине.
6. Опишите клиента электронного магазина.
7. Опишите корзину с товарами покупателя в электронном магазине.
8. Напишите функцию, подсчитывающую общую сумму стоимости всех товаров в корзине одного покупателя.

Объединения

Члены структуры занимают соседние ячейки памяти. В языке С есть средство поместить все члены в одну ячейку памяти. Такое средство называется *объединением* (union). Определение объединения, как и вообще весь синтаксис работы с объединением, внешне похоже на определение структуры. Определение выглядит следующим образом:

```
union common_field{
   unsigned char ch[4];
   int k;
   float f;
   char *p;
} un;
```

В это объединение входят члены, у которых на большинстве современных машин будет одинаковая длина — 4 байта. Все они помещаются в одну и ту же ячейку памяти, к которой мы можем обращаться, выбирая тот или иной член, как к массиву из четырех символов, как к целому числу, как к вещественному числу или как к адресу. Пусть содержимое этой ячейки равно 01000001 01000010 01000011 01000100. Тогда, поскольку младший адрес записан справа, а старший — слева, получаем следующие значения членов объединения:

```
un.ch == {'D', 'C', 'B', 'A'};
un.k == 1094861636;
un.f == 1.2141422e+01;
un.p == 1094861636;
```

Конечно, каждое такое объединение должно иметь какой-то смысл. Необдуманное наложение разных переменных на одну ячейку памяти приведет к трудно обнаруживаемым ошибкам.

Члены объединения не обязательно должны быть одной и той же длины. Размер ячейки определяется длиной самого длинного члена, все члены начинаются с начального адреса этой ячейки.

Содержимое общей ячейки можно изменить, используя любое ее поле, например:

```
un.ch[3] = '?';
un.k = 15;
un.p = &x;
```

Поэтому можно положить в ячейку значение одного, а взять значение другого типа. Инициализировать объединение при его определении можно

только значением, имеющим тип первого поля, в нашем примере это определение должно выглядеть так:

```
union common_field un1 = {'X', 'Y', 'Z', 'T'};
```

Как и у структур, можно определять массивы объединений, указатели на объединения и прочие комбинированные типы.

Объединение может быть членом какой-либо структуры. Это удобно, когда несколько типов различаются только в деталях. Пусть, например, мы описываем печатные издания для книжного магазина или библиотеки. У книги и журнальной статьи есть общие выходные данные: авторы, название, количество страниц, год публикации и есть различающиеся характеристики. Для книги надо указать издательство, для журнала его том и номер. Для общего описания и книги, и журнала можно определить структуру с вариантными членами, например, такого вида:

```
typedef {BOOK, MAGAZINE} printing_type;
struct printing{
   char *authors[3];
   char name[200];
   unsigned page_count;
   unsigned year_publ;
   printing_type type;
   union {
       char publ_house[64];
       int vol[2];
   } publ_place;
} pr1;
```

Для удобства работы с такой структурой в нее введен член `type`, помогающий определить, что описано данной конкретной структурой `pr1`: книга или журнал. Его можно использовать так:

```
if (pr1.type == BOOK) puts(pr1.publ_place.publ_house);
if (pr1.type == MAGAZINE)
   printf("v. %d, no. %d\n",
      pr1.publ_place.vol[0], pr1.publ_place.vol[1]);
```

Работа с такой структурой требует предельной внимательности. Вы должны отдавать себе полный отчет в том, какая информация хранится в данный момент в структуре с вариантными членами.

Упражнения

9. Определите объединение массива символов и целого числа. Выделите с помощью этого объединения старшую шестнадцатеричную цифру числа.
10. С помощью объединения целого типа и массива символов выведите на экран шестнадцатеричное представление целого числа.
11. Опишите одной структурой сведения о работающих жителях и пенсионерах. Для работающих людей укажите их должность, место работы и заработную плату, для пенсионеров — номер пенсионного удостоверения и размер пенсии.

Битовые поля

Как мы видели в предыдущих разделах этой главы, язык C позволяет хранить значения разных типов в соседних ячейках и даже в одной и той же ячейке. Создатели языка C пошли дальше и ввели в язык средство, называемое *битовыми полями* (bit-fields), которое позволяет разбить байты на отдельные совокупности битов и даже на отдельные биты, и обращаться к ним независимо друг от друга.

При определении битовых полей для каждого из них указываются тип `unsigned` и количество битов, составляющих поле. Каждое поле понимается компилятором как беззнаковое целое. Вся совокупность битовых полей определяется как структура. Например:

```
struct flags{
   unsigned sex: 1;
   unsigned citizen: 1;
   unsigned education: 4;
   unsigned status: 4;
} f1, f2;
```

Эта структура с меткой `flags` состоит из четырех полей. Поля `sex` и `citizen` занимают по одному биту и могут принимать только значения 0 и 1. Поля `education` и `status` занимают по 4 бита. У них могут быть значения от 0 до 15. Компилятор размещает поля в соседних битах, но выделяет для всей структуры целое число байтов, а иногда даже целое машинное слово, поэтому длина структуры `flags` будет больше 10 битов. Она может равняться двум или четырем байтам.

Для того чтобы как-то повлиять на распределение памяти, допускается определять безымянные поля. К таким полям нельзя обращаться, они только выравнивают поля по границам байтов. Кроме того, можно записать длину поля 0. Нулевая длина поля служит указанием компилятору на необ-

ходимость выравнивания полей на границу байта или машинного слова. С учетом этого предыдущее определение можно переписать следующим образом:

```
struct flags{
   unsigned sex: 1;
   unsigned citizen: 1;
   unsigned education: 4;
   unsigned : 2;
   unsigned status: 4;
   unsigned : 0;
} f1, f2;
```

Компилятор поместит первые три поля в один байт, четвертое поле в соседний байт и зарезервирует за всей структурой машинное слово.

Многие компиляторы не позволяют определять длинные поля битов. Они ограничивают общую длину всех полей битов, составляющих одну структуру, только одним машинным словом. У каждого поля битов тип целого без знака, поэтому его можно использовать в вычислениях как полноценное целое неотрицательное значение. Обращение к полю идет так же, как и к члену структуры:

```
f1.sex = 0;
f2.education = f1.education + 1;
if (f1.status == 4) get_sal();
```

Однако, у поля нет определенного адреса, поэтому к нему нельзя применить побитовую конъюнкцию, инкремент, декремент и нельзя определять массивы, состоящие из полей битов.

Поля битов недаром определяются как структуры. Дело в том, что можно определить структуру, часть членов которой — обычные переменные, а часть — поля битов. Например, в следующей структуре записывается владелец и номер телефона, а также его признаки: домашний, сотовый разных типов, мобильный:

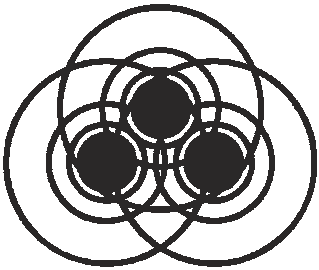
```
struct phone{
   char abonent[100];
   unsigned long phone_number;
   unsigned home: 1;
   unsigned sell: 2;
   unsigned mobile: 1;
};
```

Упражнения

12. Запишите байт признаков для анкеты, содержащий поля, отмечающие пол человека, его семейное положение, наличие телефона и водительских прав.
13. Вставьте байт признаков, определенный в предыдущем упражнении, в структуру, реализующую анкету.
14. Приведите пример структуры, содержащей и объединения, и поля битов.

Контрольные вопросы

1. Можно ли использовать значения перечислимых типов в арифметических операциях?
2. Могут ли быть перечислимые типы с отрицательными числовыми значениями?
3. Можно ли создать перечислимый тип с бесконечным диапазоном значений?
4. Что такое метка структуры?
5. Чем отличается метка структуры от типа структуры?
6. Почему определение структуры завершается точкой с запятой?
7. Могут ли функции быть членами структуры?
8. Могут ли структуры быть аргументами функций?
9. Могут ли указатели на структуру быть аргументами функций?
10. Может ли функция вернуть структуру в качестве результата своей работы?
11. Может ли функция вернуть указатель на структуру в качестве результата своей работы?
12. Можно ли сделать структуру членом объединения?
13. Какие типы данных допустимы для битовых полей?
14. Можно ли включить битовые поля в структуру как ее члены?



Глава 8

Препроцессор

На протяжении всей книги мы не раз обращались к препроцессору языка C. До сих пор мы использовали только его возможности по определению констант и включению в программу содержимого указанных ему файлов. Пришло время полностью рассмотреть работу препроцессора.

Препроцессор языка C — это небольшой текстовый редактор. Ему передается текст программы, написанной на языке C, он просматривает текст и первым делом удаляет из него все комментарии, оставляя на месте каждого комментария пробел. Затем он ищет предназначенные ему директивы и выполняет их. Преобразованный текст передается компилятору или сохраняется в файле. Как видите, комментарии не доходят до компилятора, он их не видит и ничего не знает о них.

Препроцессор обычно вызывается той же командой, что и компилятор, а его работа регулируется параметрами командной строки компилятора. Для того чтобы препроцессор системы программирования GCC просмотрел некоторый файл prog.c и записал результат своей работы в файл prog.i, не передавая его компилятору, надо вызвать компилятор с параметром `-E`, т. е. написать следующую командную строку:

```
gcc -E prog.c -o prog.i
```

Расширение имени файла i в системе программирования GCC означает, что файл не требует обработки препроцессором. При выполнении следующей команды для файла prog.i препроцессор уже не будет вызываться, а сразу же начнет работать компилятор, который создаст исполнимый файл с именем prog:

```
gcc prog.i -o prog
```

Каждая директива препроцессору записывается на отдельной строке. Перед директивой в этой строке могут быть только пробельные символы. Директива

начинается со знака номера #. Вот этот знак и отыскивает препроцессор, чтобы выполнить указания директивы.

Директивы препроцессору можно записать на любой строке файла и вне функций, и внутри них, если только в этом есть какой-то смысл. Старым версиям препроцессора требовалось писать знак # обязательно в первой колонке строки, а не во второй или далее, причем после него нельзя было оставлять пробелы. Сейчас эти ограничения сняты. Нужно понимать, что препроцессор просто обрабатывает текст и при наличии в нем директив выполняет соответствующие действия.

Мы уже встречались с директивами препроцессора #include и #define. Рассмотрим их теперь в полном объеме.

Включение файла директивой *#include*

Директива #include предназначена для включения в текст программы содержимого указанного файла. Это содержимое может быть любым, препроцессор не делает никаких проверок. Текст включаемого файла никак не преобразуется и ничем не отделяется от текста самой программы.

Имя файла в директиве #include можно записать в угловых скобках или в кавычках. В первом случае препроцессор будет искать файл только в заранее определенном месте, чаще всего это подкаталог usr/include каталога, в котором установлена система программирования. В этом подкаталоге обычно имеется множество вложенных подкаталогов, поэтому директива часто выглядит так:

```
#include <stdio.h>
#include <sys/fs/fs5/types.h>
#include <linux/stat.h>
```

Если имя файла записано в кавычках, то препроцессор, как правило, ищет его по указанному пути, начиная от корневого каталога или от текущего каталога в зависимости от того, написан ли полный путь или краткое имя файла.

```
#include "prog1.h"
#include "/home/khabib/cprogs/include/const.h"
#include "project12/include/const.h"
```

Подстановка *#define* и *#undef*

Мы использовали директиву #define для определения констант примерно так:

```
#define MAX_NUMBER 99999
#define Privet С добрым утром!
```

Увидев `#define`, препроцессор станет отыскивать все появления первого написанного в ней слова, `MAX_NUMBER`, `Privet`, во всем лежащем ниже директивы тексте файла. Найдя это слово, препроцессор просто подставит вместо него все следующие слова, написанные в директиве, вплоть до символа перевода строки исключительно. Если возникает необходимость записать длинную директиву на нескольких строках, то надо экранировать символ перевода строки, поставив в конце строки обратную наклонную черту:

```
#define LONG_TEXT Эта длинная директива \
продолжается на \
нескольких строках.
```

Препроцессор, подставляя строки, просто выбросит из них каждую пару символов, состоящую из обратной наклонной черты и символа перевода строки.

`#undef` отменяет подстановку указанного в ней слова. После данной директивы препроцессор перестанет делать подстановку строки "С добрым утром!" во всем оставшемся тексте:

```
#undef Privet
```

Действие директивы препроцессора `#define` не ограничивается подстановкой констант. С ее помощью можно определять *макросы*. Рассмотрим подробнее эту интересную возможность.

Макросы

Макрос, в отличие от простой константы, содержит параметры, использующиеся в его расшифровке. Получается выражение, очень похожее на функцию. Например:

```
#define sqr(x) x * x
```

Перед открывающей скобкой не должно быть пробела, иначе она будет понята препроцессором как начало значения константы `sqr`. Макрос затем можно записывать в программе как обычную функцию:

```
int k = 3 * sqr(2) - 4;
```

В отличие от функции, эта запись не дойдет до компилятора. Препроцессор преобразует ее в строку:

```
int k = 3 * 2 * 2 - 4;
```

По этой причине при записи макроса легко допустить ошибку. Запись:

```
z = a * sqr(x + y) - b;
```

будет преобразована в строку:

```
z = a * x + y * x + y - b;
```

Вряд ли это имелось в виду при создании макроса. Чтобы избежать таких ошибок, все члены макроса надо занести в скобки:

```
#define sqr(x) ((x) * (x))
```

Тогда препроцессор преобразует предыдущее выражение в строку:

```
z = a * ((x + y) * (x + y)) - b;
```

Несмотря на такие сложности, макросы очень широко применяются в программах, написанных на языке С. Большинство функций ввода-вывода, которые мы рассматривали, на самом деле макросы. Например:

```
#define getchar() getc(stdin)
#define putchar() putc(x, stdout)
```

Макросы обычно собираются в заголовочные файлы. Посмотрите содержимое любого файла с расширением h в вашей системе программирования С, и вы увидите в нем множество макросов.

Переменные препроцессора

В состав препроцессора входит несколько переменных, определяемых автоматически во время его работы. Их имена начинаются и заканчиваются двумя знаками подчеркивания, чтобы отличить их от любых переменных в программе. Состав этих переменных и их количество могут быть разными для разных препроцессоров. Вот их обычный состав:

- `__LINE__` — номер текущей строки обрабатываемого файла;
- `__FILE__` — имя обрабатываемого файла;
- `__DATE__` — дата компиляции;
- `__TIME__` — время компиляции;
- `__STDC__` — если файл соответствует стандарту языка С, то значение переменной равно 1, в противном случае какое-то другое значение.

Вот как можно использовать эти переменные при отладке программы:

```
FILE *f;
if ( (f = fopen("C:\\myfile.txt", "r+)) == NULL){
   fprintf(stderr, "Ошибка открытия файла %s отмечена в строке %d\n",
        __FILE__, __LINE__);
   fprintf(stderr, "Время компиляции %s  %s\n",
        __DATE__, __TIME__);
   return -1;
}
```

Различные компиляторы могут определять дополнительные переменные и константы. Например, во многих системах программирования по виду файла или по расширению его имени автоматически вызывается компилятор языка C или компилятор языка C++. В таких системах компилятор C++ определяет константу `__cplusplus`.

Условная компиляция *#ifdef*

Директива препроцессора `#ifdef`, записываемая в нескольких вариациях, позволяет передать компилятору те или иные участки программы в зависимости от выполнения некоторого условия. Строение директивы очень похоже на условный оператор. В первой строке встречается одно из трех слов `#if`, `#ifdef` или `#ifndef`, после которого записывается условие. Условие здесь — это какое-то константное выражение. На следующих строках записывается текст программы, который будет передан компилятору при выполнении условия, точнее говоря, если оно отлично от нуля.

Затем может идти необязательная часть директивы, начинающаяся со строки со словом `#else`, после которой записывается текст программы, передаваемый компилятору при невыполнении условия. Невыполнение условия означает, что оно равно нулю. Заканчивается директива словом `#endif`. Например, следующая директива передает компилятору оператор `a = 333`, если `X` не равно −10, или оператор `a = 555`, если `X` равно −10:

```
int a;
#if X + 10
a = 333;
#else
a = 555;
#endif
```

Следует учитывать, что константа `X` к этому времени уже должна быть определена.

Во второй форме директивы `#ifdef` записывается константа, предварительно определенная в директиве `#define`. При этом значение константы не имеет значения, важен сам факт ее определения. Например:

```
#ifdef i386
#define BUFSIZ 2048
#endif
#ifdef i586
#define BUFSIZ 4096
#endif
```

Где-то выше этих строк должно быть сделано определение:

```
#define i386
```

или:

```
#define i586
```

Смысл третьей формы `#ifndef` противоположен. Вот как определяется константа `EOF` в файле stdio.h:

```
#ifndef EOF
#define EOF (-1)
#endif
```

В некоторых системах программирования определяется и используется макрос `defined()`. Вот, например, определение константы `NULL`:

```
#if !defined(__cplusplus)
#define NULL ((void*)0)
#else
#define NULL (0)
#endif
```

Вот как можно включить в текст программы отладочную печать:

```
int value = x * f1(a + b, a - b) / f2(a % mod);
#ifdef DEBUG
fprintf(log, "%d: value = %d\n", __LINE__, value);
#endif
```

Для того чтобы отладочная информация записывалась в файл, на который указывает `log`, остается только написать строку:

```
#define DEBUG
```

Еще одно часто встречающееся применение условной компиляции связано с тем, что в файле, включаемом в программу директивой `#include`, может снова находиться директива `#include`. Она будет исполнена препроцессором. Этот процесс может продолжаться так, что один и тот же файл окажется включенным в программу несколько раз. Для того чтобы предотвратить это излишнее включение, начало и конец заголовочного файла оформляют следующим образом:

```
#if !defined(MYFILE_H)
#define MYFILE_H
/* Содержимое заголовочного файла */
#endif
```

Нумерация строк *#line*

Препроцессор всегда пересчитывает строки читаемого файла и заносит номер текущей строки в переменную `__LINE__`. Это значение можно изменить директивой `#line`. В ней просто записывается новый номер строки:

```
#line 20
```

Дополнительно этой же директивой можно изменить имя файла, хранящееся в переменной `__FILE__`:

```
#line 20 "newname.c"
```

Сообщение *#error*

В директиве `#error` после пробела записывается строка, которую препроцессор выводит в составе своего сообщения об ошибке. Например:

```
#if !defined(LINUX) && !defined(WINDOWS)
#error Следует определить либо LINUX, либо WINDOWS
#endif
```

Прагма *#pragma*

Директива `#pragma` определяет указания, специфичные для конкретных систем программирования языка C. Например, в некоторых системах следующим образом можно включить параметры командной строки компилятора:

```
#pragma option -a -k -zC_TEXT
```

После работы препроцессора компилятор будет вызван с этими параметрами. В директиву `#pragma` вкладывается самый разный смысл. Например, следующая директива выдаст предупреждение, если файл file.c был модифицирован позже данного файла:

```
#pragma GCC dependency "file1.c"
```

Это помогает следить за соответствием зависящих друг от друга файлов и вовремя компилировать их.

Данная директива выдаст сообщение об ошибке, если в тексте программы встретятся слова `gets` и `puts`:

```
#pragma GCC poison gets puts
```

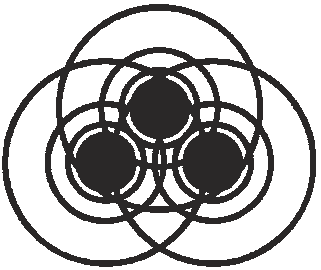
Это помогает предотвратить появление в программе нежелательных конструкций.

Упражнения

1. Определите макросы `max(x, y)` и `min(x, y)` для нахождения наибольшего и наименьшего из двух значений.
2. В файловых системах UNIX имена каталогов в полном имени файла разделяются наклонной чертой, в файловых системах MS Windows — обратной наклонной чертой. Определите с помощью препроцессора единый разделитель имен каталогов.
3. Укажите другие способы предотвращения множественного включения одного и того же файла директивой `#include`.

Контрольные вопросы

1. Что такое препроцессор?
2. Можно ли исключить препроцессор из процедур обработки исходного файла?
3. Можно ли записать результат работы препроцессора в файл и передать его компилятору позднее?
4. Может ли директива `#include` включить в программу не весь файл, а только его часть?
5. Что такое макрос?
6. Можно ли в макросе записать условные операторы, циклы, определить переменные?
7. Может ли имя макроса совпадать с именем какой-либо функции того же файла?
8. Что такое условная компиляция?
9. Можно ли использовать переменные препроцессора в программе на языке C?
10. Для чего нужна директива `#pragma`?



Глава 9

Расширения языка С в языке С++

Язык С создан в 1970 г. и с тех пор получил необычайно широкое распространение. На нем написаны целые операционные системы и тысячи приложений к ним. Все это время язык широко обсуждался, выявлялись его недостатки и неудобства, предлагались различные улучшения, которые были реализованы во многих компиляторах.

Через пятнадцать лет активного использования языка С Бьерн Страуструп (Bjarne Stroustrup) сделал решительный шаг — он создал язык С++. Как видно из названия этого языка, в котором к названию языка применена операция инкремента, язык С++ — это расширение языка С, продвижение на один шаг вперед. Главной целью расширения была реализация объектно-ориентированного программирования, но Страуструп учел опыт программирования на языке С и внес в него значительные дополнения, цель которых, как он сам говорит: "Сделать работу серьезных программистов более приятным занятием". В этой главе мы рассмотрим такие дополнения, а оставшуюся часть книги целиком посвятим объектно-ориентированным конструкциям языка С++.

Комментарии

В языке С для выделения комментария применяется наклонная черта и звездочка:

```
/* Это комментарий в языке С */
```

Почему для комментария выбрано такое странное сочетание символов? Дело в том, что надо было выбрать такой символ или комбинацию символов, которые никогда не встретятся в исходном тексте программы. Невозможно предположить, что кто-нибудь будет записывать операции деления и умножения подряд друг за другом.

В язык C++ введена еще одна форма комментария — он начинается с двух знаков наклонной черты (опять невозможное в обычных условиях сочетание символов) до символа перевода строки.

```
// Это комментарий в языке C++.
```

Комментарий в такой форме действует только до конца строки. Если надо пометить как комментарий несколько строк, как говорят, "закомментарить" их, то в начале каждой из них следует поставить подряд две наклонные черты.

Как и в языке C, комментарии убираются из текста программы препроцессором, до компилятора они не доходят, компилятор ничего не знает про комментарии.

Две наклонные черты некоторое время служили признаком того, что программа написана на языке C++. Но очень скоро такую форму комментария стали вводить в различные реализации языка C. Она оказалась удобной и сейчас встречается во многих других языках.

Строгая типизация

Язык C допускает большую свободу в использовании разных типов в одном выражении. Значения символьного типа `char` можно свободно употреблять в арифметических операциях, длинные числа могут быть обрезаны без всякого предупреждения. У вещественных чисел при преобразовании их к целому типу просто отбрасывается дробная часть. Это расширяет возможности программиста, но приводит к множеству ошибок, которые к тому же очень трудно выявить и исправить.

Язык C++ не рекомендует смешивать в одном выражении данные разных типов. Компилятор следит за их соответствием и сообщает о том, что он вынужден сделать преобразование, которое может привести к искажению результата. Для совместимости с языком C компилятор языка C++ ограничивается предупреждением в тех местах, где преобразование допускается компилятором языка C, но в новых конструкциях языка C++ компилятор осуществляет строгую проверку типов.

Логический тип

В языке C условие считается верным, если определяющее его выражение отлично от нуля. Это часто приводит к ошибкам при неверном составлении выражения, при округлении или просто по невнимательности. Для устранения таких ошибок в язык C++ введен логический тип `bool`. У него всего два значения: `true` (истина) и `false` (ложь). Мы можем определить переменные логического типа:

```
bool b1 = true, b2 = false;
```

И использовать их в условиях:

```
if (b1) x = 0;
else x = 1;
```

Результат сравнения в языке C++ — значение логического типа, поэтому можно написать:

```
b1 = x < y;
b2 = x == y;
```

Логические типы могут быть аргументами и возвращающими значениями функций:

```
bool f(int a, bool b){
   return a < 100 || b;
}
```

Для совместимости с языком C++ значения логического типа разрешается использовать в арифметических операциях, при этом `false` равно 0, а `true` — 1. При обратном преобразовании 0 преобразуется в `false`, а ненулевое значение — в `true`. Указатели тоже могут быть преобразованы в логические значения. Ненулевой указатель преобразуется в `true`, а `NULL` — в `false`.

Объявление переменных внутри блока

В языке C разрешается объявлять переменные в начале блока, сразу же после открывающей фигурной скобки. Язык C++ дает больше свободы — переменную можно определять непосредственно перед ее использованием и даже в заголовке цикла `for`. Например, можно написать такой код:

```
int m = 0;
void f(){
   for (int k = 0; k < 100; k++){
      m = 2 * k + 10;
      int m = 2; // Локальная переменная, действует в пределах блока.
      int i = 0; // Локальная переменная, обнуляется при каждой итерации.
      static int j = 0; // Статическая переменная, обнуляется один раз
                        // при первой итерации.
      for (int l = 0; l < k; l++)
         printf("i = %d, j = %d\n", i++, j++);
   }
   // i, k, l, локальная m, перестают существовать,
   // j существует, но невидима.
}
```

Переменные цикла `k` и `l` считаются локальными, они существуют только в пределах цикла и только на время выполнения цикла. В старых компиляторах языка C++ действовало другое правило: такие переменные действовали до конца блока, в котором создавался цикл. Это правило оказалось неудобным и было со временем изменено.

Как и комментарий, такая вольность в определении переменных оказалась очень удобной и постепенно перекочевала во многие реализации языка C.

Операция разрешения видимости ::

В примере предыдущего пункта определены две переменные с именем `m`. Локальная переменная `m` перекрывает глобальную переменную `m` с места своего определения до конца блока, в котором она определена. Иногда возникает необходимость обратиться к глобальной переменной, находясь в области действия локальной переменной. В языке C это невозможно, в языке C++ это можно сделать операцией разрешения видимости. Эта операция записывается двумя двоеточиями, написанными подряд, без пробелов. Двоеточия ставятся непосредственно перед глобальной переменной. Вот как можно обратиться к глобальной переменной в предыдущем примере:

```
int m = 0;
void f(){
   for (int k = 0; k < 100; k++){
      m = 2 * k + 10;
      int m = 2; // Локальная переменная, действует в пределах блока.
      m = 3 * k + 15; // Меняется локальная переменная.
      ::m = 5; // Меняется глобальная переменная.
   }
}
```

Подчеркнем, что операция разрешения видимости применима только к глобальным переменным, определенным на уровне 0. Перекрытую локальную переменную увидеть нельзя.

Определение констант

Старый стиль языка C предполагает определение констант директивой препроцессора `#define`, например:

```
#define MAX_VALUE 99999
```

Стандарт языка С ввел для определения констант служебное слово `const`:

```
const int MAX_VALUE = 99999;
```

Такое определение позволяет компилятору сделать проверку соответствия типов, тем самым предотвращая множество ошибок.

Язык С++, в отличие от языка С, предполагает строгую проверку всех типов, поэтому вторая форма определения констант становится в нем основной. При ее употреблении надо четко представлять себе, к чему применяется слово `const`. Поясним это на примерах:

```
const char *pc = "abcd"; // Указатель на строку-константу.
pc[2] = 'x';   // Ошибка.
pc = "klmnop"; // Правильно.
```

Предыдущее определение вводит указатель на неизменяемую строку. Сам указатель остается переменным, его можно перекинуть на другую строку:

```
char * const cp = "abcd"; // Константный указатель на строку.
cp[2] = 'x';   // Правильно.
cp = "klmnop"; // Ошибка.
```

Это определение вводит указатель-константу. Его нельзя изменить, он показывает на определенную ячейку памяти, но ее содержимое можно менять:

```
const char * const cpc = "abcd"; // Константный указатель
                                 // на строку-константу.
cpc[2] = 'x';   // Ошибка.
cpc = "klmnop"; // Ошибка.
```

Теперь нельзя менять ни указатель, ни содержимое ячейки, на которую он показывает.

Константы очень часто применяют для того, чтобы случайно не изменить аргумент функции. Например, прототип функции копирования строк выглядит так:

```
char *strcpy(char *p, const char *q);
```

Здесь строка, на которую показывает `q`, копируется в строку, на которую показывает указатель `p`. Строка `p` меняется, но строка `q` не должна меняться внутри функции `strcpy()`. Слово `const` во втором аргументе функции гарантирует это.

Ссылки

В языке C очень широко применяются указатели. Их запись не всегда удобна. Вспомним определение функции `swap()` (*см. гл. 5*):

```
void swap(int *a, int *b){
   int t = *a; *a = *b; *b = t;
}
```

Эта запись испещрена звездочками, ее трудно читать и понимать. Не совсем удобен и вызов этой функции.

```
void main(){
   int x = 5, y = 7;
   printf("До обмена: x = %d, y = %d\n", x, y);
   swap(&x, &y);
   printf("После обмена: x = %d, y = %d\n", x, y);
}
```

Здесь при вызове функции приходится записывать амперсанды.

В язык C++ наряду с указателями введены *ссылки* (references), которые отмечаются амперсандом. Ссылку можно считать синонимом переменной, на которую она ссылается. Применяя ссылки, функцию `swap()` можно записать так:

```
void swap(int &a, int &b){
   int t = a; a = b; b = t;
}
```

Здесь аргументы функции `a` и `b` — обычные переменные типа `int`. Знак амперсанда в заголовке функции показывает, что при вызове функции `a` и `b` станут ссылками на фактические аргументы. Вызов функции выглядит так:

```
void main(){
   int x = 5, y = 7;
   printf("До обмена: x = %d, y = %d\n", x, y);
   swap(x, y);
   printf("После обмена: x = %d, y = %d\n", x, y);
}
```

Теперь в функцию передаются обычные переменные `x` и `y`, а не их адреса. Имена `a` и `b` будут вторыми, но совершенно полноценными именами этих переменных и, следовательно, у них будут те же значения. Если же надо

предотвратить изменение содержимого аргумента функции, то можно сделать этот аргумент константой:

```
void f(const int &a);
```

Ссылкой может быть не только аргумент, но и результат выполнения функции:

```
int &f(int &a){
   return a;
}
```

Эта простая функция интересна тем, что ее можно использовать в такой парадоксальной форме:

```
f(x) = 1;
f(x)++;
```

Вызов функции стоит в левой части оператора присваивания! К нему применяется инкремент! Это оказалось возможным потому, что на самом деле в левой части будет стоять результат работы функции — ссылка на переменную `a`, т. е. синоним переменной `a`. После выполнения первого оператора присваивания содержимое переменной `a` будет равно 1, а затем еще увеличится на 1 после выполнения инкремента.

Ссылку можно определить и явно:

```
int x = 1;
int &y = x;
```

После этого определения ссылка `y` становится синонимом переменной `x`. Все операции, выполняемые с `y`, на самом деле выполняются с `x`. Можно написать:

```
y = 2;
y++;
```

При этом изменится значение переменной `x`, поэтому определение ссылки требует явной инициализации. Нельзя писать:

```
int &x; // Ошибка.
```

При работе со ссылками надо строго соблюдать соответствие типов. Если тип ссылки будет отличаться от типа переменной, на которую она ссылается, компилятор при приведении типов создаст временную переменную, которая и будет меняться. Например, можно сделать определение:

```
int x = 1;
char &y = x; // Возможное, но неверное определение.
```

В этом случае компилятором будет создана временная переменная `char t = (char)x`, и переменная `y` станет синонимом переменной `t`, а не переменной `x`.

Разумеется, нельзя определить ссылку на константу и на выражение. Следующие определения ошибочны:

```
int &a = 1;     // Ошибка.
int &b = x + y; // Ошибка.
```

Однако можно определить ссылку-константу:

```
const int &a = 1;
```

Компилятор может расшифровать эту запись так:

```
int t = 1;
const int &a = t;
```

Поскольку ссылка — это только синоним переменной, нельзя определить ссылку на ссылку, ссылку на битовое поле или массив ссылок. Нельзя определить указатель на ссылку, но можно определить ссылку на указатель, например, в такой форме:

```
void f(int *&x){
   x = 0;
}
```

Упражнение

1. Пересмотрите примеры *главы 4*, в которых рассмотрена работа с указателями, и скажите, какие из них могут работать со ссылками.

Прототипы функций

Язык C требовал записи в начале файла или в заголовочном файле прототипов тех функций, описание которых дано после их вызова. Язык C++ возводит это правило в абсолют — в начале файла или в заголовочном файле нужно обязательно написать прототипы всех функций или дать их определение. Напомним, что в прототипе важно указать тип возвращаемого значения функции, ее имя, и указать тип и порядок следования всех аргументов функции. Имена аргументов необязательны. Например:

```
void f1(int, double *);
void f2(); // Функция без аргументов.
```

В прототипах можно указать значения аргументов по умолчанию:

```
void f3(char *, int = 0, double = 1.0);
```

После этого функцию `f3()` можно вызывать с тремя аргументами, с двумя аргументами или с одним, первым, аргументом. Остальные аргументы получат значения по умолчанию. У первого аргумента функции `f3()` нет значения по умолчанию, поэтому его всегда надо записывать явно:

```
f3("abcd", 1, -4.25);
f3("abcd", 1);
f3("abcd");
```

Компилятор берет значения по умолчанию в том порядке, в котором они написаны, поэтому значения по умолчанию должны быть у всех последних аргументов. Следующий прототип неверен:

```
void f4(char * = "abc", int = 1, int); // Ошибка.
```

Здесь у последнего аргумента нет значения по умолчанию. Такую функцию надо всегда вызывать со всеми аргументами, что делает бессмысленными все остальные значения по умолчанию.

Функции с переменным числом аргументов

Вы, наверное, заметили интересную особенность функций ввода-вывода `printf()` и `scanf()` — у них переменное число аргументов. Язык С может реализовать такие функции, но не позволяет программисту самому определять их. Язык С++ разрешает это. В прототипе функции вместо типов и имен переменных аргументов ставится многоточие:

```
void f(int a ...);
```

При определении функции в заголовке тоже ставится многоточие, а внутри функции действуют библиотечные макросы `va_arg()`, `va_start()`, `va_end()`, работающие со структурой типа `va_list`. Макросы и структура определены в файле cstdarg.h.

Макрос `va_start()` просматривает количество и типы фактических аргументов и заполняет структуру типа `va_list` сведениями о них. В процессе выполнения функции эти сведения извлекаются макросом `va_arg()`, которому надо указать предполагаемый тип аргумента. Все это можно схематически записать следующим образом:

```
void f(int a …){
   va_list ap;
```

```
   va_start(ap, a);

   while(1){
      char *p = va_arg(ap, char *);
      if (p == 0) break;
      // Обработка аргумента p.
   }
   va_end(ap);
}
```

Встраиваемые функции

Язык C++ предлагает вместо директивы препроцессора `#define` применять константные переменные. Аналогично этому, вместо макросов предлагается записывать *встраиваемые* (inline) функции. Встраиваемые функции определяются так же, как и обычные функции, но помечаются словом `inline`. Например:

```
#define cube(x) ((x)*(x)*(x))
```

Этот макрос можно записать в виде встраиваемой функции:

```
inline int cube(int x){
   return x*x*x;
}
```

Кроме явной проверки типов, встраиваемая функция выгодна тем, что она компилируется один раз, а затем машинный код ставится во все места вызова функции.

К сожалению, встраиваемые функции трудно реализовать, поэтому все компиляторы значительно ограничивают их возможности: нельзя использовать условные операторы, циклы, операторы варианта. Если компилятор не может встроить функцию, то он генерирует обычную. Поэтому слово `inline` служит только пожеланием (hint) компилятору. В зависимости от своих возможностей компилятор может выполнить это пожелание, а может и не выполнить.

Явное приведение типов

В языке C часто применяется операция явного приведения типов, например:

```
double x = (double)123456789;
short sh = (short)x;
```

Язык C++ предлагает функциональную запись приведения типа:

```
double x = double(123456789);
short sh = short(x);
```

Но такая запись не совсем удобна, т. к. она предполагает, что имя типа записывается одним словом — нельзя вместо `(char *)x` написать `char *(x)`. В подобных случаях надо сделать определение вроде этого:

```
typedef char * chtype;
```

Потом следует писать `chtype(x)`. Такие ограничения приводят к тому, что в реальных программах функциональная запись используется реже старой записи, принятой в языке С.

Типы перечислений, структур, объединений

В языке С при определении переменных часто используются метки перечислений, структур, объединений, а не имена типов. При этом надо обязательно повторять служебные слова: `enum`, `struct`, `union`. Мы рассматривали примеры таких определений (*см. гл. 7*), вот некоторые из них:

```
enum traffic_light tl1, tl2;
struct ved_row ivanov, petrov, sidorov;
```

В языке С++ метки служат полноценными именами типов, которые они помечают, поэтому определение переменных можно делать без служебных слов `enum`, `struct`, `union`:

```
traffic_light tl1, tl2;
ved_row ivanov, petrov, sidorov;
```

Операции выделения памяти

В языке С есть несколько стандартных функций для динамического выделения памяти. Основная функция `malloc()` применяется так:

```
long *lp = (long *)malloc(sizeof(long));
```

После работы с динамической переменной память освобождается функцией `free()`:

```
free(lp);
```

Эти функции неудобны тем, что приходится постоянно думать о приведении типов и подсчитывать необходимый объем памяти. В язык С++ введены операции выделения и освобождения памяти.

Для выделения памяти служит операция, обозначаемая словом `new`. После этого служебного слова через пробел записывается тип переменной, под которую выделяется память. Значением операции будет указатель соответствующего типа на выделенную ячейку. При невозможности выделить память значением операции будет `NULL`.

Освобождение памяти выполняется операцией `delete`, в которой записывается указатель, ранее определенный операцией `new`. Записанную в начале этого пункта пару функций `malloc()`, `free()` можно заменить парой операторов:

```
long *lp = new long;
delete lp;
```

Как видите, такая запись гораздо проще и понятнее. В выделенную память сразу же можно записать начальное значение:

```
long *lp = new long(1L);
```

Так же просто выделяется память для массивов. После имени типа в квадратных скобках записывается длина массива:

```
char *s = new char[1024];
double *a = new double[100];
```

При освобождении памяти, выделенной для массива, надо в операции `delete` поставить квадратные скобки:

```
delete[] s;
delete[] a;
```

Если этого не сделать, то компилятор освободит только память, выделенную для первого элемента массива. Наконец, с помощью операции `new` можно направить определяемый указатель на уже выделенную память, например:

```
long *lp1 = new (lp)long;
```

После этого определения указатели `lp` и `lp1` будут содержать один и тот же адрес памяти.

Упражнение

2. Перепишите примеры *главы 4*, касающиеся выделения памяти, используя операции `new` и `delete`.

Перегрузка функций

В языке C для каждого типа аргумента нужно писать свою функцию. Например, абсолютное значение целого числа `x` типа `int` вычисляет функция `abs(x)`, а абсолютное значение вещественного числа типа `double` — функция `fabs(x)`. Это источник огромного числа ошибок, усугубляемых тем, что функции не посылают никаких сообщений о неправильном типе фактических аргументов, а просто выдают в качестве результата какое-то число.

Язык C++ позволяет написать и использовать в программе несколько функций с одним и тем же именем. Они должны различаться типами аргументов или их количеством. Например:

```
void mess(int);
void mess(char *);
```

Эти прототипы описывают функцию `mess()`, которую можно вызывать с целочисленным или строковым аргументом. Например:

```
mess(123);
mess("Ошибка определения переменной");
```

Говорят, что функция `mess()` *перегружена* (overload) с одного типа переменной на другой тип. Сами функции могут выглядеть так:

```
void mess(int k){
   fprintf(stderr, "Произошла ошибка № %d\n", k);
}
void mess(char *s){
   fprintf(stderr, "%s\n", s);
}
```

Перегрузка функций возможна потому, что компилятор меняет имя функции, добавляя к нему условные обозначения количества аргументов и их типов (name mangling). Например, имя `f` функций `f(int)`, `f(char)`, `f(char *)` может быть заменено именами `@f$qi`, `@f$qc`, `@f$qpc`. Поэтому для компилятора имя каждой функции остается уникальным.

В стандартной библиотеке языка C++ все функции, выполняющие одинаковую работу, перегружены. Например, есть только одна функция `abs(x)`, дающая абсолютное значение аргумента `x` любого типа.

Заметьте, что тип возвращаемого значения или модификаторы, такие как `const`, `inline`, не учитываются компилятором, поэтому нельзя перегрузить функцию, изменив возвращаемый ею тип.

Шаблоны функций

При перегрузке функций приходится писать множество функций с одинаковыми именами и разными типами. В некоторых случаях удается эту работу возложить на компилятор. Вы один раз пишете *шаблон* (template) функции, а компилятор генерирует нужный код. Напишем для примера шаблон функции `swap()`, уже несколько раз использовавшейся нами:

```
template <class T> void swap(T &a, T &b){
   T t; t = a; a = b; b = t;
}
```

Определение шаблона начинается со служебного слова `template`, после которого в угловых скобках записывается служебное слово `class` и условное обозначение типа аргументов функции. Это обозначение может быть любым, по традиции пишут заглавную букву `T`. Потом обычным образом записывается функция, в которой вместо конкретного типа ставится его условное обозначение `T`.

Компилятор просматривает вызовы функции, по виду фактических аргументов узнает их тип и генерирует код функции с конкретным значением типа. Вот как можно использовать написанный выше шаблон функции `swap()`:

```
void main(void){
   int k = 5, m = 7;
   double x = -3.4, y = 234.78;
   char *s1 = "Первая строка", *s2 = "Вторая строка";

   swap(k, m);
   swap(x, y);
   swap(s1, s2);
}
```

Если аргументов несколько и их типы различны, то в угловых скобках через запятую повторяется слово `class` и условные обозначения типа:

```
template <class T1, class T2> int f(T1 a, T2 b){ ... }
```

Шаблоны можно перегружать. Например, мы можем написать два шаблона функции, возвращающей наибольшее значение, — один для простых типов, другой для массивов:

```
template <class T> T get_max(T a, T b){
   return a > b ? a : b;
}
template <class T> T get_max(T a[], int n){
   T m = a[0];
   for (int k = 0; k < n; k++)
      if (a[k] > m) m = a[k];
   return m;
}
```

Эти шаблоны подходят для простых типов и массивов, но не годятся для строк. Для них можно написать отдельную функцию:

```
char *get_max(char *s1, char *s2){
   return strcmp(s1, s2) ? s1 : s2;
}
```

Вот как можно использовать эту функцию:

```
#include <string.h>

void main(){
   char s1[] = "Строка символов", s2[] = "Еще одна строка";
   int a = 4, b = 9;
   double x = -345.6509, y = 23.45078;
   double tmp[5] = {0.1, -4.05, 0.024, -8.904, 2.45};

   int  m   = get_max(a, b);
   double z = get_max(x, y);
   double t = get_max(tmp, 5);
   chat  *p = get_max(s1, s2);
}
```

Перегрузка функций и шаблонов очень удобна и часто используется в языке С++. Во всех производственных программах можно увидеть тому множество примеров.

Перегрузка операций

Не знаю, заметили ли вы, что в языках программирования одни и те же операции обрабатывают разные типы операндов и выполняют разную работу? Сложение `5 + 7` выполняется с целыми числами типа `int` в двоичной арифметике, а сложение `5.0 + 7.0` — в арифметике с плавающей точкой с типом `double`. Для компьютера это совершенно разные операции, но обе они обозначаются плюсом. Следовательно, компилятор перегружает операцию "плюс" на разные типы операндов. Такая перегрузка происходит во всех языках программирования, но программисту она чаще всего недоступна.

Язык С++ предоставляет возможность программисту перегружать почти все операции на свои типы данных. При этом операция рассматривается как функция, аргументами которой служат ее операнды. Например, сумму целых чисел можно записать как функцию:

```
int sum(int a, int b);
```

В языке С++ надо записывать стандартные имена таких функций, состоящие из слова `operator` и обозначения операции. Например:

```
int operator+(int a, int b);
double operator+(double a, double b);
```

Приведем полный пример. Некоторые языки программирования предлагают операцию сцепления строк, которая заключается в том, что сразу же после

одной строки, без всяких разделителей, записывается вторая строка. Например, сцепление двух строк:

```
"Первая строка" + ", вторая строка"
```

даст строку:

```
"Первая строка, вторая строка"
```

Определим операцию сцепления строк, перегрузив операцию "плюс" на указатели:

```
#include <string.h>
#define MAX_STRLEN 2048
char s[MAX_STRLEN];
char *operator+(const char *s1, const char *s2){
   if (strlen(s1) + strlen(s2) > MAX_STRLEN) return NULL;
   strcpy(s, s1);
   strcat(s, s2);
   return s;
}
```

Приведем более полный пример. В инженерных расчетах часто приходится выполнять действия с комплексными числами. Для языка C естественно определить структуру `Complex` с двумя членами: действительной и мнимой частью комплексного числа. Остается перегрузить обычные арифметические операции на эту структуру, и мы получим полноценный тип данных. Это сделано в листинге 9.1.

Листинг 9.1. Операции с комплексными числами

```
#include <stdio.h>
#include <math.h>

struct Complex{
   double re, im;
};

double cmod(Complex x){
   return sqrt(x.re * x.re + x.im * x.im);
}
Complex operator+(const Complex x, const Complex y){
   Complex z;
```

```
   z.re = x.re + y.re;
   z.im = x.im + y.im;
   return z;
}
Complex operator-(const Complex x, const Complex y){
   Complex z;
   z.re = x.re - y.re;
   z.im = x.im - y.im;
   return z;
}
Complex operator*(const Complex x, const Complex y){
   Complex z;
   z.re = x.re * y.re - x.im * y.im;
   z.im = x.im * y.re + x.re * y.im;
   return z;
}
Complex operator/(const Complex x, const Complex y){
   Complex z = {1e30, 1e30};
   double t = y.re * y.re + y.im * y.im;
   if (t > 1e-6){
      z.re = (x.re * y.re + x.im * y.im) / t;
      z.im = (x.im * y.re - x.re * y.im) / t;
   }
   return z;
}
int operator==(const Complex x, const Complex y){
   return (x.re == y.re && x.im == y.im) ? 1: 0;
}
int operator!=(const Complex x, const Complex y){
   return (x.re == y.re && x.im == y.im) ? 0: 1;
}
Complex &operator=(const Complex &x, const Complex &y){
   x.re = y.re;
   x.im = y.im;
   return x;
}
Complex &operator+=(const Complex &x, const Complex &y){
   x.re += y.re;
   x.im += y.im;
```

```
   return x;
}
Complex &operator-=(const Complex &x, const Complex &y){
   x.re -= y.re;
   x.im -= y.im;
   return x;
}
Complex &operator*=(const Complex &x, const Complex &y){
   double t = x.re;
   x.re = x.re * y.re - x.im * y.im;
   x.im = x.im * y.re + t * y.im;
   return x;
}
Complex &operator/=(const Complex &x, const Complex &y){
   Complex z = {1e30, 1e30};
   double t = y.re * y.re + y.im * y.im;
   if (t > 1e-6){
      double tmp = x.re;
      x.re = (x.re * y.re + x.im * y.im) / t;
      x.im = (x.im * y.re - tmp * y.im) / t;
   }else x = z;
   return x;
}
main(){
   Complex e = {1.0, 0.0}, i = {0.0, 1.0} a, b;
   a = e + i;
   b = e * i;
   a = a / b;
   b += a;
}
```

Обратите внимание на то, что при переопределении операций присваивания использованы ссылки на комплексные числа. Это сделано для того, чтобы можно было записывать подряд несколько присваиваний:

```
a = b = c;
```

Операции с одним операндом, такие как отрицание, инкремент и декремент, понимаются при перегрузке как функции с одним аргументом. Например, операцию отрицания для комплексных чисел можно определить так:

```
Complex &operator!(Complex &x){
```

```
    Complex t;
    t.re = !x.re;
    t.im = !x.im;
    return t;
}
```

Инкремент и декремент могут быть префиксными или постфиксными. Чтобы различить их, при определении постфиксной операции вводится второй аргумент типа `int`. Этот аргумент никогда не используется, он нужен только как пометка, отличающая постфиксную операцию от префиксной:

```
void operator++(Complex &x);
void operator++(Complex &x, int);
```

Поскольку ко всем типам обязательно применяется операция присваивания, компилятор C++ автоматически перегружает ее на каждый новый тип. Эта перегрузка заключается в простом копировании всех ячеек памяти. Если такая перегрузка вас не устраивает, то следует перегрузить присваивание явно.

Подводя итог, перечислим правила перегрузки операций.

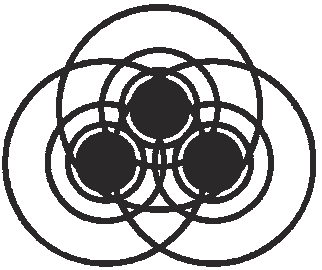
- Перегрузить можно только имеющиеся операции, нельзя ввести новую операцию.
- Для перегруженной операции можно использовать только тот же самый графический символ.
- Перегрузить операцию можно только на новый тип, нельзя, скажем, перегрузить сложение целых чисел.
- Нельзя изменить правило применения или приоритет операции. Например, нельзя сделать постфиксное отрицание или сделать бинарный инкремент.
- Нельзя перегружать условную операцию, операцию разрешения видимости и операцию взятия члена структуры ("точка").

Упражнения

3. Перегрузите операцию сложения для сложения комплексных и действительных чисел. Не забудьте про коммутативность сложения.
4. Перегрузите оставшиеся арифметические и логические операции для комплексных чисел.
5. Перегрузите для комплексных чисел операции `new` и `delete`.

Контрольные вопросы

1. Может ли комментарий в стиле C++ начинаться в середине строки?
2. Можно ли определить переменную внутри условного оператора?
3. Можно ли использовать переменную, определенную в заголовке цикла `for`, после окончания работы цикла?
4. В чем разница между указателем и ссылкой?
5. Можно ли определить ссылку на массив?
6. Может ли одна ссылка присвоить значение другой ссылке?
7. Какие операции можно делать с ссылками?
8. Можно ли определить массив ссылок?
9. Можно ли определить ссылку на структуру?
10. Может ли член структуры быть ссылкой?
11. Может ли функция вернуть ссылку?
12. Могут ли аргументы функции быть ссылками?
13. Какие ограничения накладываются на встраиваемые функции?
14. Может ли операция `new` выделить память для структуры?
15. Как выделить память под массив структур операцией `new`?
16. Можно ли перегрузить функцию, поменяв местами ее аргументы?
17. Можно ли перегрузить функцию, сменив тип возвращаемого ею значения?
18. Можно ли перегрузить стандартные функции, например, синус?
19. В каких случаях удобно записывать шаблоны функций?
20. Можно ли записать шаблон функции, аргументы которой — структуры?



Глава 10

Принципы ООП

В предыдущих главах мы вкратце рассмотрели парадигмы программирования и отметили, что в настоящее время господствует парадигма объектно-ориентированного программирования, сокращенно, ООП, которая представляет выполнение программы как взаимодействие *объектов*. Объекты содержат *поля*, предназначенные для хранения информации, и *методы* обработки информации, использующие эти поля в своей работе.

В начале работы программы создается один или несколько объектов. Объекты активны. Они выполняют свои методы обработки информации, сохраняя результаты обработки в своих полях, файлах, базах данных или в каких-то других хранилищах. По мере необходимости объекты обращаются к методам других объектов, передавая им нужные сведения. В процессе выполнения программы могут создаваться новые объекты и уничтожаться старые, ненужные объекты. Работа программы завершится, когда один из объектов выполнит метод завершения программы. Этот метод обычно сохраняет обработанную информацию в указанных ему хранилищах данных и удаляет все объекты, освобождая оперативную память и другие ресурсы, занятые объектами.

Представление программы набором взаимодействующих объектов позволяет смоделировать производственные процессы более точно, чем процедурное программирование, которое мы применяли до сих пор. В самом деле, в реальной жизни действуют живые или неживые объекты: люди, животные, роботы, станки, механизмы, приборы. В процессе работы они сохраняют, преобразуют и вырабатывают какую-то информацию, активно обмениваясь ей. Объем информации может быть слишком велик, чтобы его можно было обработать вручную, и тогда в дело вступают компьютеры. Компьютерная программа обработки и обмена информацией должна как можно точнее моделировать потоки информации, протекающие в реальных процессах. Тут-то

и помогает объектно-ориентированное программирование, предлагающее методику моделирования работы реальных объектов.

В начале разработки объектно-ориентированной программы сразу встает множество вопросов. Сколько объектов понадобится для правильной работы программы? Каким образом и в какое время создавать объекты? Как распределить работу между объектами? Как организовать взаимодействие объектов? В объектно-ориентированном программировании выработано несколько общепризнанных принципов, более или менее полно отвечающих на эти вопросы. Познакомимся с ними.

Принципы объектно-ориентированного программирования

Объектно-ориентированное программирование развивается уже более двадцати лет. За это время сформировалось несколько школ, каждая из которых предлагает свой набор приемов работы с объектами и по-своему излагает эти приемы. Бурные обсуждения и дискуссии, проходившие между представителями этих школ, позволили выработать несколько общепринятых принципов, признанных всеми школами и внедренных во все объектно-ориентированные языки программирования. Перечислим эти принципы.

Абстракция

Описывая поведение какого-либо объекта, например, автомобиля, мы строим его модель. Мы говорим: "Его вес около тонны, он вмещает пять человек, перемещается со скоростью двести километров в час, расходует десять литров бензина на сто километров пути". Модель, как правило, не может описать объект полностью, реальные объекты слишком сложны. Приходится отбирать только те характеристики объекта, которые важны для решения поставленной перед нами задачи. Для описания грузоперевозок важной характеристикой будет грузоподъемность автомобиля, а для описания автомобильных гонок она не существенна. Но для моделирования гонок обязательно надо смоделировать метод набора скорости данным автомобилем, а для грузоперевозок это не столь важно.

Для характеристики спортсмена обязательно надо указать его вес, рост, скорость реакции, спортивные достижения, а для ученого все эти качества несущественны, зато важны его квалификация, ученая степень, количество опубликованных научных работ.

Как говорят специалисты, мы должны *абстрагироваться* от некоторых конкретных деталей объекта, не обращать на них внимания и не описывать их. Конечно, это огрубляет модель, поэтому очень важно выбрать правильную степень абстракции. Слишком сильное огрубление, слишком высокая сте-

пень абстракции даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Можно охарактеризовать человека как "двуногое без перьев", но что это даст для его понимания? Слишком сильная конкретизация, другими словами, слишком низкая степень абстракции, сделает модель недопустимо сложной, перегруженной деталями, и потому непригодной для большинства реальных объектов.

Например, можно совершенно точно предсказать погоду на завтра в определенном месте, но современное состояние вычислительной техники таково, что расчеты по такой модели продлятся трое суток даже на самом мощном компьютере. Зачем нужна модель, дающая прогноз, опаздывающий на два дня? Ну, а точность модели, используемой синоптиками, мы все знаем сами. Зато расчеты по этой модели занимают всего несколько часов.

Итак, прежде всего нам надо выбрать уровень абстракции, необходимый для правильного описания реального информационного процесса. Затем следует выделить объекты, участвующие в этом процессе и установить связи между этими объектами. Как это сделать? Опишите процесс словами и проанализируйте получившиеся фразы. "Завод выпускает автомобили". Здесь два объекта — "Завод" и "Автомобиль". Производственно-технические характеристики завода составят набор полей объекта "Завод", а процесс выпуска автомобиля будет описан в виде набора методов объекта "Автомобиль".

Пример из другой области: "Преподаватель читает учебный курс". Полями объекта "Преподаватель" будут его фамилия, имя и отчество, научно-педагогический стаж, квалификация, ученая степень, выпущенные им учебники и методические пособия. Методами "Преподавателя" будут такие действия, как "читать", "писать", "повышать квалификацию", "проводить консультацию", "принимать зачет". Полями объекта "Учебный курс" будут его название, программа, количество часов, перечень учебных пособий. Будет ли объект "Учебный курс" обладать какими-то методами или в этом объекте будут только поля? Какие действия выполняет "Учебный курс"? По-видимому, единственным действием объекта "Учебный курс" будет предоставление своих полей другим объектам, значит, нужны методы доступа к полям объекта.

Таким образом, если в словесном описании процесса вам потребовалось сформулировать какое-то понятие, то оно и будет кандидатом на оформление его в виде объекта. Существительные, описывающие это понятие, будут полями объекта, а глаголы — методами будущего объекта.

В объектно-ориентированных языках модель информационного процесса записывается в виде одного или нескольких *классов* (classes). Каждый класс описывает свойства одного объекта. Класс можно считать проектом, слепком, чертежом, по которому затем будут создаваться конкретные объекты. При описании класса применяются знакомые нам конструкции программирования.

Поля, в которых объект будет хранить необходимую ему информацию, описываются массивами, переменными и константами. Количество переменных и их типы выбираются так, чтобы в наибольшей степени охарактеризовать объект. Они называются *полями класса* (class fields). Полями класса могут быть не только простые переменные, константы или массивы, но и другие объекты, и массивы объектов. Кроме полей в классе можно определить локальные переменные, хранящие промежуточные результаты работы методов класса.

Методы обработки информации, используемые объектом, описываются процедурами и функциями. Они называются *методами класса* (class methods). Сложные объекты могут содержать несколько десятков методов, а значит, несколько десятков процедур и функций. Методы класса активно используют поля класса, но кроме них могут создавать и свои локальные переменные, необходимые для работы метода.

Кроме полей и методов в классе можно описать и *вложенные классы* (nested classes), в которых также могут быть описаны вложенные классы. Таким образом, мы можем создать сложную матрешку вложенных классов.

Поля, методы и вложенные классы первого уровня называются *членами* класса (class members). Разные школы объектно-ориентированного программирования предлагают разные термины, мы используем терминологию, принятую в языке C++.

Рассмотрим набросок описания автомобиля:

```
class Automobile{

  int maxVelocity; // Поле, содержащее наибольшую скорость автомобиля.
  int speed;  // Поле, содержащее текущую скорость автомобиля.
  int weight; // Поле, содержащее вес автомобиля.
  int tonnage; // Поле, содержащее грузоподъемность автомобиля.

  // Прочие поля…

  void moveTo(int x, int y){

    // Метод, моделирующий перемещение
    // автомобиля в точку с координатами x и y.
    // Параметры метода x и y — не поля, а локальные переменные.

    int a = 1; // Локальная переменная — не поле.

    // Тело метода. Здесь описывается закон перемещения
    // автомобиля в точку (x, y).
```

```
  }

// Прочие методы…
};
```

После того как описание класса закончено, можно создавать конкретные объекты, как говорят, *экземпляры* (instances) описанного класса. При этом объявляются имена объектов, а именно записывается имя класса, и через пробел перечисляются экземпляры класса:

```
Automobile lada2115, fordScorpio, oka;
```

Поскольку имена полей, методов и вложенных классов у всех объектов одинаковы, т. к. они заданы в описании класса, их надо уточнять именем ссылки на объект:

```
lada2115.maxVelocity = 180;
lada2115.speed = 0;
lada2115.weight = 1250;
lada2115.tonnage = 850;

fordScorpio.maxVelocity = 220;
fordScorpio.speed = 0;
fordScorpio.weight = 1750;
fordScorpio.tonnage = 1250;

oka.maxVelocity = 350; // Почему бы и нет?
oka.speed = 0;
oka.weight = 650;
oka.tonnage = 450;

oka.moveTo(35, 120);
```

Как правило, поля только что созданного объекта должны сразу же получить определенные значения, образующие в совокупности *начальное состояние* объекта. Эти значения впоследствии меняются, образуя *текущее состояние* объекта. Конечно, можно давать значение каждому полю операторами присваивания, как сделано выше. Но это занимает много места, затрудняет проверку присваиваемых значений и ухудшает оформление программы.

В объектно-ориентированном программировании принято присваивание полями начальных значений записывать вместе с необходимыми проверками и другими нужными для создания объекта действиями в специальную

процедуру, называемую *конструктором* (constructor) объекта. Конструктор вызывается автоматически при создании объекта, при этом необязательно даже записывать его имя. По этой причине имя конструктора должно совпадать с именем класса. Для нашего примера с автомобилем можно написать конструктор, просто определяющий значения полей:

```
class Automobile{

  int maxVelocity; // Поле, содержащее наибольшую скорость автомобиля.
  int speed;  // Поле, содержащее текущую скорость автомобиля.
  int weight; // Поле, содержащее вес автомобиля.
  int tonnage; // Поле, содержащее грузоподъемность автомобиля.

  // Прочие поля.

  Automobile(int mv, int s, int w, int t){
    maxVelocity = mv;
    speed = s;
    weight = w;
    tonnage = t;
  }

  void moveTo(int x, int y){

    // Метод, моделирующий перемещение
    // автомобиля в точку с координатами x и y.
    // Параметры метода x и y — не поля, а локальные переменные.

    int a = 1; // Локальная переменная — не поле.

    // Тело метода. Здесь описывается закон перемещения
    // автомобиля в точку (x, y).
  }

// Прочие методы…
};
```

После введения в класс конструктора определение экземпляров класса можно записать так:

```
Automobile lada2115(180, 0, 1250, 850),
```

```
fordScorpio(220, 0, 1750, 1250),
oka(350, 0, 650, 450);
```

Согласитесь, что эта запись короче и нагляднее, чем вереница присваиваний, сделанных после определения объекта.

Поскольку современные языки программирования допускают перегрузку функций, в одном классе можно написать несколько конструкторов, отличающихся количеством и/или числом параметров. Например, в нашем классе `Automobile` можно записать конструктор без аргументов, просто обнуляющий все поля:

```
Automobile(){
  maxVelocity = 0;
  speed = 0;
  weight = 0;
  tonnage = 0;
}
```

Поскольку начальная скорость автомобиля, как правило, равна нулю, разумно записать еще один конструктор с тремя аргументами, положив скорость равной нулю и определяя конструктором остальные поля:

```
Automobile(int mv, int w, int t){
  maxVelocity = mv;
  speed = 0;
  weight = w;
  tonnage = t;
}
```

Запись класса с несколькими конструкторами будет выглядеть так:

```
class Automobile{

  int maxVelocity; // Поле, содержащее наибольшую скорость автомобиля.
  int speed;  // Поле, содержащее текущую скорость автомобиля.
  int weight; // Поле, содержащее вес автомобиля.
  int tonnage; // Поле, содержащее грузоподъемность автомобиля.

  // Прочие поля.

  Automobile(int mv, int s, int w, int t){
    maxVelocity = mv;
```

```
    speed = s;
    weight = w;
    tonnage = t;
  }

  Automobile(){
    maxVelocity = 0;
    speed = 0;
    weight = 0;
    tonnage = 0;
  }

  Automobile(int mv, int w, int t){
    maxVelocity = mv;
    speed = 0;
    weight = w;
    tonnage = t;
  }

  void moveTo(int x, int y){

    // Метод, моделирующий перемещение
    // автомобиля в точку с координатами x и y.
    // Параметры метода x и y — не поля, а локальные переменные.

    int a = 1; // Локальная переменная — не поле.

    // Тело метода. Здесь описывается закон перемещения
    // автомобиля в точку (x, y).
  }

// Прочие методы…
};
```

Теперь мы можем выбрать тот или иной способ определения экземпляров класса:

```
Automobile lada2115_1(180, 0, 1250, 850),
  lada2115_2(160, 1450, 950),
```

```
fordScorpio_001(220, 0, 1750, 1250),
fordScorpio_002(250, 1650, 1150),
oka1(350, 0, 650, 450),
oka2;
```

В последнем случае при определении объекта `oka2`, когда значения конструктору класса вообще не указаны, вызывается конструктор без параметров.

Конструктор класса — это особенный метод. Он выполняется еще до создания объекта, поэтому у него нет возвращаемого значения. Такое значение просто некому передавать, ведь методы объекта еще не работают. Поэтому при записи конструктора даже не пишется слово `void`. Тем не менее, конструктор может быть очень сложным. Он может содержать условные операторы, операторы цикла, может вызывать другие методы, определять объекты, динамически распределять оперативную память, устанавливать соединения с базами данных и сетевые соединения. Возможности конструктора определяются языком программирования. В следующей главе мы подробно разберем конструирование объектов в язык C++.

После того как определен конструктор класса, определяющий начальное состояние объекта, естественно определить *деструктор* (destructor) класса, подготавливающий завершение работы объекта. Он может освобождать память, разрывать соединения с базами данных и сетевыми устройствами, сбрасывать информацию в файлы или в базы данных.

В деструктор нет смысла передавать параметры, ведь они нигде не будут использоваться. Поэтому в каждом классе можно записать только один деструктор. Его имя, как и имя конструктора, совпадает с именем класса, но перед ним ставится тильда:

```
~Automobile(){
    // Завершающие действия.
}
```

Деструктор класса автоматически вызывается при завершении работы объекта. Программисту не надо специально обращаться к нему, но тем тщательнее его надо разрабатывать. Огромное число проблем, связанных с утечкой памяти, вызвано тем, что деструктор не освобождает полностью память, занятую объектом. Эта ошибка настолько распространена, что в современных языках программирования, таких как Java и C#, решили отказаться от деструкторов, возложив их функцию на специальную программу, называемую *сборщиком мусора* (garbage collector). Сборщик мусора запускается через определенные промежутки времени, просматривает доступную программе область оперативной памяти в поисках участков, на которые нет указателей, и освобождает эти участки.

Во многих ситуациях проектировщики строят несколько моделей информационного процесса с разной степенью детализации. Скажем, для конструирования пальто и шубы нужна менее точная модель контуров человеческого тела и его движений, а для конструирования фрака или вечернего платья — уже гораздо более точная. При этом более точная модель, с меньшей степенью абстракции, будет использовать уже имеющиеся методы менее точной модели.

Иерархия

Не кажется ли вам, что класс `Automobile` сильно перегружен? Действительно, в мире выпущены миллионы автомобилей разных марок и видов. Что между ними общего, кроме четырех колес? Да и колес может быть больше или меньше. Не лучше ли написать отдельные классы для легковых и грузовых автомобилей, для гоночных автомобилей и вездеходов? Как организовать все это множество классов? На этот вопрос объектно-ориентированное программирование отвечает так: надо организовать иерархию классов.

Иерархия объектов давно используется для их классификации. Особенно детально она проработана в биологии. Все мы знакомы с семействами, родами и видами. Мы можем сделать описание своих домашних животных (pet): кошек (cat), собак (dog), коров (cow) и прочих "домочадцев" следующим образом:

```
class Pet{
  // В этом классе описываем общие свойства всех домашних любимцев.

  Master person;  // Хозяин животного.
  int weight;     // Вес животного.
  int age;        // Возраст животного.
  Date eatTime[]; // Массив, содержащий время кормления.

  int eat(int food, int drink, Date time){ // Процесс кормления.
    // Начальные действия.

    if (time == eatTime[i]) person.getFood(food, drink);
    // Метод потребления пищи…
  }

  void voice(); // Звуки, издаваемые животным.

  // Прочее.
};
```

Затем создаем классы, описывающие более конкретные объекты, связывая их с общим классом:

```
class Cat: Pet{
  // В этом классе описываются свойства, присущие только кошкам:

  int mouseCatched; // Число пойманных мышей.

  void toMouse(); // Процесс ловли мышей.

  // Прочие свойства…
};

class Dog: Pet{
  // В этом классе описываются свойства собак:

  void preserve(); // Охранять…
};
```

Заметьте, что мы не повторяем общие свойства, описанные в классе `Pet`. Все его поля и методы наследуются автоматически. Мы можем определить объект класса `Dog` и использовать в нем все свойства класса `Pet` так, как будто они описаны в классе `Dog`:

```
Dog tuzik, sharik;
```

После этого определения можно будет написать:

```
tuzik.age = 3;
int p = sharik.eat(30, 10, 12);
```

Классификацию можно продолжить:

```
class Pointer: Dog{
  // Свойства породы "Пойнтер".
};
class Setter: Dog{
  // Свойства сеттеров.
};
```

Заметьте, что на каждом следующем уровне иерархии в класс добавляются новые свойства, но ни одно свойство не пропадает. Поэтому говорят, что класс `Dog` — *расширение* (extension) класса `Pet`. С другой стороны, количество объектов при этом уменьшается: собак меньше, чем всех домашних

животных. Поэтому часто говорят, что класс `Dog` — *подкласс* (subclass) класса `Pet`, а класс `Pet` — *суперкласс* или *надкласс* (superclass) класса `Dog`.

Часто используют генеалогическую терминологию: родительский класс, дочерний класс, класс-потомок, класс-предок, возникают племянники и внуки, вся беспокойная семейка вступает в отношения, достойные мексиканского сериала. В этой терминологии говорят о *наследовании* (inheritance) классов, в нашем примере класс `Dog` наследует класс `Pet`. В языке C++ принято суперкласс называть *базовым* классом (base class), а подкласс — *производным* классом (derived class). Мы не будем придерживаться определенной терминологии, выбирая в каждом случае наиболее понятный термин.

Мы еще не определили счастливого владельца нашего домашнего зоопарка. Опишем его в классе `Master`. Делаем набросок:

```
class Master{
  // Хозяин животного.

  string name; // Фамилия, имя.

  // Другие сведения…

  void getFood(int food, int drink); // Кормление.

  // Прочее…
};
```

Подобно тому, как в пределах одного класса допускается перегрузка методов, в подклассе мы можем *переопределить* (override) метод, записав заголовок метода суперкласса и изменив его тело. Приведем простейший пример:

```
class A{

  void f(){
    printf("Метод f() суперкласса A\n");
  }
};
class B: A{

  void f(){
    printf("Метод f() подкласса B\n");
  }
};
```

После определения объекта `a` класса `A` и обращения к его методу получим сообщение: "`Метод f() суперкласса A`".

```
A a;
a.f();
```

После определения объекта `b` класса `B` и обращения к его методу получим, естественно, сообщение: "`Метод f() подкласса B`".

```
B b;
b.f();
```

В примере с автомобилями мы можем описать грузовой автомобиль классом `Truck`, расширяющим класс `Automobile`:

```
class Truck extends Automobile{
  void moveTo(int x, int y){
     // Какие-то действия…
  }
  // Что-то еще…
};
```

Метод `moveTo()` класса `Truck` перекроет одноименный метод суперкласса `Automobile`. Определив экземпляр класса `Truck`, например:

```
Truck gazel;
```

и записав `gazel.moveTo(25, 150)`, мы обратимся к методу класса `Truck`. Произойдет *переопределение* (overriding) метода. Интересная ситуация получится, если определить указатели на объекты классов `A` и `B`. Мы можем реализовать следующий принцип объектно-ориентированного программирования — *полиморфизм* (polymorphism).

Полиморфизм

В последнем примере предыдущего раздела мы можем определить не объекты классов `A` и `B`, а указатели на них, выделяя динамически память для объектов:

```
void main(){
  A *a = new A;
  B *b = new B;
  a->f();
  b->f();
}
```

При этом никаких особенностей не возникает, методы работают точно так же и выдают те же сообщения. Но мы можем создать указатель типа `A`, показывающий на объект типа `B`:

```
A *a = new B;
```

Это допустимо — указатель `a`, типа `A`, конечно "не знает" те поля и методы, которые добавлены в класс `B`, но хорошо "знает" все, что есть в классе `A`. Что произойдет, если обратиться к методу `f()` с помощью указателя `a`?

```
void main(){
  A *a = new B;
  a->f();
}
```

Произойдет обращение к методу `A.f()` класса `A` или к методу `B.f()` класса `B`? Будет указатель `a` обращаться к методу класса или к методу объекта?

Разные языки программирования решают этот вопрос по-разному, но полиморфизм обязывает обращаться к методу объекта, а не класса, выполнять метод `f()` класса-наследника `B`. Этот принцип предоставляет очень интересные возможности. Мы можем изменять поведение объекта определенного класса, просто направляя указатель на те или иные подклассы этого класса. Разные объекты одного и того же класса будут вести себя по-разному. Например, один хозяин домашнего животного, выполняя метод `getFood()` класса `Master`, даст большой кусок мяса, другой хозяин бросит мозговую кость, третий нальет миску похлебки.

В следующей главе мы разберем подробнее реализацию полиморфизма в языке C++ и рассмотрим интересные примеры его применения.

Ответственность

В примере класса `Pet` и наследников этого класса хозяин и его домашние животные постоянно соприкасаются в жизни. Их взаимодействие выражается глаголами "гулять", "кормить", "охранять", "чистить", "ласкаться", "проситься" и прочими действиями. Для описания взаимодействия объектов применяется следующий принцип объектно-ориентированного программирования — обязанность или ответственность.

В нашем примере рассматривается только взаимодействие в процессе кормления, описываемое методом `eat()`. В этом методе животное обращается к хозяину, умоляя его применить метод `getFood()`.

В англоязычной литературе подобное обращение описывается словом message. Это понятие неудачно переведено на русский язык ни к чему не обязывающим словом *сообщение*. Лучше было бы использовать "послание",

"поручение" или даже "распоряжение". Но термин "сообщение" устоялся и нам придется его применять. Почему же не используется словосочетание "вызов метода", ведь говорят "вызов процедуры"? Потому что между этими понятиями есть, по крайней мере, три отличия:

- сообщение идет к конкретному объекту, знающему метод решения задачи, в примере этот объект — текущее значение переменной `person`. У каждого объекта свое текущее состояние, свои значения полей класса, и это может повлиять на выполнение метода;
- способ выполнения поручения, содержащегося в сообщении, зависит от объекта, которому оно послано. Как мы уже говорили, один хозяин поставит миску с Chappi, другой бросит кость, третий выгонит собаку на улицу. Это интересное свойство реализуется полиморфизмом;
- обращение к методу произойдет только на этапе выполнения программы, компилятор ничего не знает про метод. Это называется *поздним* или *динамическим связыванием* (dynamic binding) в противовес *раннему* или *статическому связыванию* (static binding), при котором процедура присоединяется к программе на этапе компоновки.

Итак, объект `sharik`, выполняя свой метод `eat()`, посылает сообщение объекту с именем `person` с просьбой выдать ему определенное количество еды и питья. Сообщение записано в строке `person.getFood(food, drink)`.

Этим сообщением заключается *контракт* (contract) между объектами `sharik` и `person`, суть которого в том, что объект `sharik` берет на себя *ответственность* (responsibility) задать правильные параметры в сообщении, а объект `person` возлагает на себя ответственность применить метод кормления `getFood()`, каким бы он ни был.

Для того чтобы правильно реализовать принцип ответственности, применяется еще один принцип объектно-ориентированного программирования — *модульность* (modularity).

Модульность и инкапсуляция

Этот принцип утверждает — каждый класс должен составлять отдельный модуль программы. Модуль должен предоставлять пользователям те и только те возможности, которые предписаны контрактом. Члены класса, к которым не планируется обращение извне, не должны быть видны, как говорят, они должны быть инкапсулированы.

В языке C++ инкапсуляция достигается вставкой модификатора `private:` перед описанием членов класса. Например:

```
private:
  int mouseCatched;
  string name;
  void preserve();
```

Эти члены классов становятся *закрытыми*, ими могут пользоваться только экземпляры того же самого класса, например, `tuzik` может дать поручение `sharik.preserve()`.

А если в классе `Master` мы напишем:

```
private:
  void getFood(int food, int drink);
```

в этом случае метод `getFood()` не будет найден объектами других классов, и несчастный `sharik` не сможет получить пищу.

В противоположность закрытости мы можем объявить некоторые члены класса *открытыми*, записав вместо слова `private` модификатор `public`, например:

```
public:
  void getFood(int food, int drink);
```

К таким членам может обратиться любой объект любого класса.

Принцип модульности предписывает открывать члены класса только в случае необходимости. Такой принцип часто исповедуется в жизни. Вспомните надпись на железнодорожном переезде: "Нормальное положение шлагбаума — закрытое".

Если же надо обратиться к закрытому полю класса, то рекомендуется включить в класс специальные *методы доступа* (access methods), отдельно для чтения этого поля (get-метод) и для записи значений в это поле (set-метод). Имена методов доступа рекомендуется начинать со слов get и set, добавляя к этим словам имя поля.

В нашем примере класса `Master` методы доступа к полю `name` в самом простом виде могут выглядеть так:

```
public:
string getName(){
    return name;
}
void setName(string newName){
    name = newName;
}
```

В реальных ситуациях доступ ограничивается разными проверками, особенно в set-методах, меняющих значения полей. Можно проверять тип вводимого значения, задавать диапазон значений, сравнивать со списком допустимых значений.

Кроме методов доступа рекомендуется создавать проверочные *is-методы*, возвращающие логическое значение `true` или `false`. Например, в класс `Master` можно включить метод, проверяющий, задано имя хозяина, или нет:

```
public:
bool isEmpty(){
  return name == null ? true : false;
}
```

Можно использовать этот метод для проверки при доступе к полю `name`, например:

```
if (master01.isEmpty())
   master01.setName("Иванов");
```

А можно записать метод `getName()` следующим образом:

```
public:
string getName(){
    return master01.isEmpty() ? "Нет имени" : name;
}
```

Итак, мы оставляем открытыми только методы, необходимые для взаимодействия объектов. При этом удобно спланировать классы так, чтобы зависимость между ними была наименьшей, как принято говорить в теории объектно-ориентированного программирования, было наименьшее *зацепление* (low coupling) между классами. Тогда структура программы сильно упрощается. Кроме того, такие классы удобно использовать как строительные блоки для построения других программ.

Напротив, члены одного и того же класса должны активно взаимодействовать друг с другом, как говорят, иметь тесную функциональную *связность* (high cohesion). Для этого в класс следует включать все методы, описывающие поведение моделируемого объекта, и только такие методы, ничего лишнего. Одно из правил достижения сильной функциональной связности, введенное Карлом Либерхером (Karl J. Lieberherr), получило название *закон Деметра*. Этот закон гласит: "В методе `m()` класса *A* следует использовать только методы класса *A*, методы классов, к которым принадлежат аргументы метода `m()`, и методы классов, экземпляры которых создаются внутри метода `m()`".

Объекты, построенные по этим правилам, подобны кораблям, снабженным всем необходимым. Они уходят в автономное плавание, готовые выполнить любое поручение, на которое рассчитана их конструкция. Будут ли закрытые члены класса доступны его наследникам? Если в классе `Pet` написано:

```
private:
  Master person;
```

можно ли использовать поле `person` в классе `Dog`? Другими словами, можно ли записать конструкцию `sharik.person`? Разумеется, нет. Ведь в противном случае каждый интересующийся закрытыми полями класса *A* может расширить его классом *B* и просмотреть закрытые поля класса *A* через экземпляры класса *B*.

Когда надо разрешить доступ наследникам класса, но нежелательно открывать его всему миру (world), тогда используется *защищенный* (protected) доступ, отмечаемый модификатором `protected`, например, объект `sharik` может обратиться к полю `person` родительского класса `Pet`, если в классе `Pet` это поле описано так:

```
protected:
  Master person;
```

Таков основной набор принципов объектно-ориентированного программирования, и из этого общего схематического описания видно, что язык C++ позволяет легко их воплощать. Вы уже поняли, как записать класс, его поля и методы, как инкапсулировать члены класса, как сделать расширение класса и какими принципами следует при этом пользоваться. В следующей главе мы разберем подробнее правила записи классов на языке C++ и рассмотрим дополнительные их возможности.

Но, говоря о принципах объектно-ориентированного программирования, я не могу удержаться от того, чтобы не напомнить основной принцип всякого программирования.

Принцип KISS

Самый основной, базовый и самый великий принцип программирования на любом языке — принцип KISS — не нуждается в переводе. Он расшифровывается так: "Keep It Simple, Stupid"!

В самом деле, чем проще написана программа, тем легче ее отладить, тем лучше она будет работать. Кроме того, создатели языка проверяют его работу стандартными конструкциями, которые входят потом в руководства по языку. Эти-то проверенные конструкции и следует использовать в своих программах. Вычурные и запутанные программы показывают глубину мышления и остроумие их создателей, но очень неудобны для сопровождения и модификации и больше подвержены ошибкам.

Вообще говоря, сложность программы не может быть меньше сложности задачи, которую она решает. Надо только постараться, чтобы сложность программы не превысила сложность задачи. Стандартную задачу надо решать стандартными методами, используя стандартные конструкции языка. Нестандартную задачу тоже лучше решить стандартными методами и только если это не получается, вводить сложные конструкции.

Каждый язык программирования устанавливает свой стиль. Стиль языка C — это лаконичность, указатели и широкое употребление большого числа коротких функций. Стиль языка C++ — это ссылки, классы и шаблоны. Стиль всякого объектно-ориентированного языка программирования — использование стандартных библиотек классов.

При изучении языка программирования важно усвоить его стиль и следовать этому стилю в своих программах. Удивительно, но правильный стиль программирования очень помогает в написании правильных эффективных программ.

Наследование или вложение

В примере с домашними животными мы сделали объект `person` класса `Master` — владелец животного — полем класса `Pet`. Многие объектно-ориентированные языки программирования позволяют определить класс `Master` прямо внутри класса `Pet`, сделав класс `Master` вложенным классом. Это выглядит следующим образом:

```
class Pet{
  // В этом классе описываем общие свойства всех домашних любимцев.

  class Master{
    // Хозяин животного.

    string name; // Фамилия, имя.

    // Другие сведения…

    void getFood(int food, int drink); // Кормление.

    // Прочее…
  };

  int weight;     // Вес животного.
  int age;        // Возраст животного.
  Date eatTime[]; // Массив, содержащий время кормления.

  int eat(int food, int drink, Date time){ // Процесс кормления.
    // Начальные действия.

    if (time == eatTime[i]) person.getFood(food, drink);
```

```
    // Метод потребления пищи…
  }

  void voice(); // Звуки, издаваемые животным.

  // Прочее.
};
```

Вложение класса удобно тем, что методы внешнего класса могут напрямую обращаться к полям и методам вложенного в него класса. Но ведь того же самого можно было добиться по-другому. Может, следовало расширить класс `Master`, сделав класс `Pet` его наследником?

Итак, появляются две возможности сделать поля и методы общими для нескольких классов: наследование и вложение. Можно сделать иерархию наследования или иерархию вложения. Образно говоря, можно вырастить генеалогическое дерево классов-наследников, а можно вырезать матрешку вложенных классов. Что предпочесть?

Теория объектно-ориентированного программирования советует, прежде всего, выяснить, в каком отношении находятся объекты классов `Master` и `Pet` — в отношении "класс `Master` является экземпляром класса `Pet`" или в отношении "класс `Master` является частью класса `Pet`". Скажем, "собака является животным", или "собака является частью животного"? Другой пример: "мотор является автомобилем" или "мотор является частью автомобиля"? Ясно, что собака — животное, и в этой ситуации надо выбрать наследование, но мотор — часть автомобиля и здесь надо выбрать вложение.

Отношение "класс *A* является экземпляром класса *B*" по-английски записывается как "a class *A* is a class *B*", поэтому называется отношением "is-a". Отношение же "класс *A* является частью класса *B*" по-английски "a class *A* has a class *B*", и такое отношение называется отношением "has-a".

Отношение "is-a" — это отношение "обобщение-детализация", отношение большей или меньшей абстракции, и ему соответствует наследование классов. Отношение "has-a" — это отношение "целое-часть", и ему соответствует вложение классов.

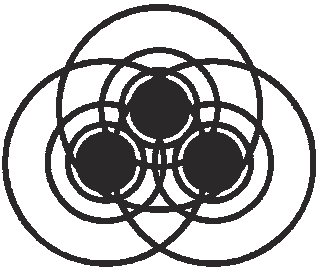
Вернемся к нашим животным и их хозяевам и постараемся ответить на вопрос: "класс `Master` является экземпляром класса `Pet`?" или "класс `Master` является частью класса `Pet`?" Ясно, что не верно ни то, ни другое. Классы `Master` и `Pet` не связаны ни тем, ни другим образом. Поэтому мы и сделали объект класса `Master` полем класса `Pet`.

Упражнения

1. Запишите классы, описывающие процесс сдачи экзамена. Какие объекты участвуют в этом процессе? Как они взаимодействуют?
2. Опишите процесс работы библиотеки в виде набора классов.
3. Создайте классы, описывающие работу деканата.
4. Опишите информационную систему своего факультета.

Контрольные вопросы

1. Что такое абстрагирование и как оно используется в моделировании?
2. По каким соображениям выбирается та или иная степень абстрагирования?
3. Какими последствиями грозит неправильный выбор степени абстракции?
4. Чем следует руководствоваться при определении числа объектов в создаваемой модели?
5. Каким образом следует оформить объект? Как решить, какие поля и методы в него включать?
6. Как организовать взаимодействие объектов? Какие рекомендации дает теория ООП?
7. Что такое контракт объекта? Каковы средства его реализации?
8. Что такое класс в объектно-ориентированном программировании?
9. Как записываются поля и методы класса?
10. Надо ли указывать поля класса в аргументах методов этого же класса?
11. Что такое инкапсуляция? Для чего она применяется?
12. Как реализуется инкапсуляция при записи класса?
13. Может ли к закрытым полям и методам класса обратиться другой объект того же класса?
14. Что такое наследование класса?
15. Может ли метод класса-наследника обратиться к закрытым полям и методам своего предка?
16. Что такое полиморфизм?
17. Как полиморфизм реализуется в объектно-ориентированных языках?
18. Следует ли всерьез принимать принцип KISS?



Глава 11

Реализация ООП в языке C++

Главное, ради чего создавался язык C++, — это классы и все, что связано с ними в объектно-ориентированном программировании. Язык C++ начинался как "Язык C с классами". Под таким названием в 1983 г. появилось первое сообщение Бьярна Страуструпа о новом языке. Классы были введены в язык C++ как естественное развитие структур `struct` языка C. Структуры состоят из членов различных типов, которые, естественно, становятся полями класса. Остается добавить описание методов, конструкторов и деструктора и класс готов.

Ранее мы определили структуры `point` и `circle` для работы с графическими объектами на плоскости (*см. гл. 7*). Там были определены функции `create_circle()`, придающие значения членам структуры `circle`, а также функции `in_circle()` и `move_circle()`, работающие с этой структурой. В листинге 7.1 была записана программа, использующая эти структуры.

Структуру `circle` легко дописать, сделав ее классом. Функции `create_circle()` после изменения их имен и аргументов станут конструкторами класса, а функции `in_circle()` и `move_circle()` — методами класса. Такое дополнение структуры `circle` сделано в листинге 11.1.

Листинг 11.1. Класс на основе структуры

```
#include <stdio.h>
#include <math.h>

struct point{
   double x, y;
};
```

```
struct circle{

   double x, y, r;

   circle(double ax, double ay, double ar){
      x = ax;
      y = ay;
      r = fabs(ar);
   }
   circle(point p, double ar){
      x = p.x;
      y = p.y;
      r = fabs(ar);
   }
   circle(const circle& c){
      x = c.x;
      y = c.y;
      r = c.r;
   }

   int in_circle(double ax, double ay){
      return (x - ax) * (x - ax) +
          (y - ay) * (y - ay) < r * r;
   }
   int in_circle(point p){
      return (p.x - x) * (p.x - x) +
          (p.y - y) * (p.y - y) < r * r;
   }
   void move_circle(point p){
      x = p.x;
      y = p.y;
   }
};

main(){
   point p = {1.0, -3.56};

   circle c1 = circle(2.5, -1.2, 6.5);
   circle c2 = circle(1.5, -0.2, 8.0);
```

```
   circle c3 = circle(p, 2.5);
   circle c4 = circle(c1);

   printf("x = %f, y = %f, r = %f\n", c1.x, c1.y, c1.r);
   printf("x = %f, y = %f, r = %f\n", c2.x, c2.y, c2.r);
   printf("x = %f, y = %f, r = %f\n", c3.x, c3.y, c3.r);
   printf("x = %f, y = %f, r = %f\n", c4.x, c4.y, c4.r);

   printf("%d\n", c1.in_circle(0.02, -1.44));
   printf("%d\n", c1.in_circle(p));
   c4.move_circle(p);
   printf("x = %f, y = %f, r = %f\n", c4.x, c4.y, c4.r);
}
```

Сравнивая листинг 11.1 с листингом 7.1, вы убедитесь, что программа стала проще за счет уменьшения количества аргументов у методов — теперь в них не надо перечислять члены структуры; и за счет упорядочения имен конструкторов они должны совпадать с именем класса.

Обратите внимание на конструктор:

```
circle(const circle& c);
```

Его параметр — ссылка на класс `circle`. Этот *конструктор копирования* работает во многих ситуациях: в перегруженной операции присваивания, при копировании объекта, передаваемого в функцию по значению, при формировании возвращаемого значения функции. Ввиду его важности компилятор сам создает конструктор копирования по умолчанию, если его нет в классе. Такой конструктор просто копирует все нестатические поля. В классе `circle` конструктор копирования именно этим и занимается, поэтому его можно было не записывать, тем более что здесь легко допустить неочевидную ошибку — если параметр будет не ссылкой, а объектом класса, то компилятор начнет копировать и параметр, создав бесконечную рекурсию:

```
circle(const circle c); // Ошибка!
```

Обратите внимание еще и на то, что в языке C++ структура — это полноценный тип данных, и слово `struct` при определении переменных и записи аргументов функций писать необязательно.

Язык C++ допускает еще одно сокращение. При определении экземпляров класса можно опускать имя конструктора. С учетом этого сокращения функцию `main()` листинга 11.1 можно переписать так:

```
main(){
   point p = {1.0, -3.56};
```

```
   circle c1(2.5, -1.2, 6.5);
   circle c2(1.5, -0.2, 8.0);
   circle c3(p, 2.5);
   circle c4(c1);

   printf("x = %f, y = %f, r = %f\n", c1.x, c1.y, c1.r);
   printf("x = %f, y = %f, r = %f\n", c2.x, c2.y, c2.r);
   printf("x = %f, y = %f, r = %f\n", c3.x, c3.y, c3.r);
   printf("x = %f, y = %f, r = %f\n", c4.x, c4.y, c4.r);

   printf("%d\n", c1.in_circle(0.02, -1.44));
   printf("%d\n", c1.in_circle(p));
   c4.move_circle(p);
   printf("x = %f, y = %f, r = %f\n", c4.x, c4.y, c4.r);
}
```

В листинге 11.1 все конструкторы и методы класса определены прямо внутри класса. Гораздо чаще внутри класса записываются только их прототипы, а определение выполняется вне класса. Это делается не только для наглядности, но и потому, что компилятор методы, определенные внутри класса, считает inline-функциями и пытается встроить их код в каждое обращение к таким методам.

При определении конструкторов и методов вне класса надо каким-то образом указать класс, к которому они относятся. Это делается с помощью операции разрешения видимости, записываемой двумя двоеточиями (*см. гл. 9*). В листинге 11.2 приведен этот способ определения конструкторов и методов класса.

Листинг 11.2. Определение конструкторов и методов вне класса

```
#include <stdio.h>
#include <math.h>

struct point{
   double x, y;
};

struct circle{

   double x, y, r;
```

```
   circle(double ax, double ay, double ar);
   circle(point p, double ar);
   circle(const circle& c);

   int in_circle(double ax, double ay);
   int in_circle(point p);
   void move_circle(struct point p);
};

circle::circle(double ax, double ay, double ar){
   x = ax;
   y = ay;
   r = fabs(ar);
}
circle::circle(point p, double ar){
   x = p.x;
   y = p.y;
   r = fabs(ar);
}
circle::circle(const circle& c){
   x = c.x;
   y = c.y;
   r = c.r;
}
int circle::in_circle(double ax, double ay){
   return (x - ax) * (x - ax) +
          (y - ay) * (y - ay) < r * r;
}
int circle::in_circle(point p){
   return (p.x - x) * (p.x - x) +
          (p.y - y) * (p.y - y) < r * r;
}
void circle::move_circle(struct point p){
   x = p.x;
   y = p.y;
}

main(){
```

```
   point p = {1.0, -3.56};

   circle c1(2.5, -1.2, 6.5);
   circle c2(1.5, -0.2, 8.0);
   circle c3(p, 2.5);
   circle c4(c1);

   printf("x = %f, y = %f, r = %f\n", c1.x, c1.y, c1.r);
   printf("x = %f, y = %f, r = %f\n", c2.x, c2.y, c2.r);
   printf("x = %f, y = %f, r = %f\n", c3.x, c3.y, c3.r);
   printf("x = %f, y = %f, r = %f\n", c4.x, c4.y, c4.r);

   printf("%d\n", c1.in_circle(0.02, -1.44));
   printf("%d\n", c1.in_circle(p));
   c4.move_circle(p);
   printf("x = %f, y = %f, r = %f\n", c4.x, c4.y, c4.r);
}
```

Как видно из листинга 11.2, операция разрешения видимости записывается в заголовке конструктора или метода и действует сразу на все поля класса, записываемые внутри этого конструктора или метода. Применять эту операцию к каждому полю уже не надо.

Итак, с помощью структур можно сделать полноценное описание класса. При этом все члены класса по умолчанию считаются открытыми, `public`. Закрытые члены класса, описанного как структура, надо помечать отдельно, как сделано в листинге 11.3.

Листинг 11.3. Инкапсуляция в структуре

```
#include <stdio.h>
#include <math.h>

struct point{
   double x, y;
};

struct circle{

private:
```

```
   double x, y, r;

public:
   circle(double ax, double ay, double ar);
   circle(point p, double ar);
   circle(const circle& c);

   double getX(){ return x; }
   void setX(double ax){ x = ax; }

   double getY(){ return y; }
   void setY(double ay){ y = ay; }

   double getR(){ return r; }
   void setR(double ar){ r = fabs(ar); }

   void pr();

   int in_circle(double ax, double ay);
   int in_circle(point p);
   void move_circle(struct point p);
};

circle::circle(double ax, double ay, double ar){
   x = ax;
   y = ay;
   r = fabs(ar);
}
circle::circle(point p, double ar){
   x = p.x;
   y = p.y;
   r = fabs(ar);
}
circle::circle(const circle& c){
   x = c.x;
   y = c.y;
   r = c.r;
}
```

```
int circle::in_circle(double ax, double ay){
   return (x - ax) * (x - ax) +
          (y - ay) * (y - ay) < r * r;
}
int circle::in_circle(point p){
   return (p.x - x) * (p.x - x) +
          (p.y - y) * (p.y - y) < r * r;
}
void circle::move_circle(struct point p){
   x = p.x;
   y = p.y;
}
void circle::pr(){
   printf("x = %f, y = %f, r = %f\n", x, y, r);
}

main(){
   point p = {1.0, -3.56};

   circle c1(2.5, -1.2, 6.5);
   circle c2(1.5, -0.2, 8.0);
   circle c3(p, 2.5);
   circle c4(c1);

   c1.pr();
   c2.pr();
   c3.pr();
   c4.pr();

   printf("%d\n", c1.in_circle(0.02, -1.44));
   printf("%d\n", c1.in_circle(p));
   c4.move_circle(p);
   c4.pr();
}
```

В листинге 11.3 по всем правилам объектно-ориентированного программирования, изложенным в предыдущей главе, поля класса `x`, `y` и `r` инкапсулированы. Для доступа к ним написаны методы доступа `getX()`, `setX()` и т. д.

Кроме того, для удобства вывода полей на консоль написан метод `pr()`, иначе в функции `printf()` пришлось бы записывать метод доступа к каждому полю.

Описание класса

Как вы только что видели, полноценное описание класса в языке C++ можно сделать с помощью структур. Описание класса можно сделать и с помощью объединения `union`. Тем не менее, в язык C++ для описания класса специально введено ключевое слово `class`, которое можно написать вместо слова `struct` при описании класса. Единственное отличие, которое вносит слово `class`, заключается в том, что все члены такого класса по умолчанию считаются закрытыми, `private`. Другими словами, следующие два описания класса `A` совершенно эквивалентны.

```
struct A{
private:
   double x;
public:
   double getX(){ return x; }
   void setX(double ax){ x = ax; }
};
class A{
   double x;
public:
   double getX(){ return x; }
   void setX(double ax){ x = ax; }
};
```

Сохраняется даже точка с запятой после закрывающей фигурной скобки, потому, что здесь можно определить экземпляры класса.

Итак, описание класса начинается со слова `class`, `struct` или `union`, после которого записывается имя класса. Далее записывается тело класса. Тело класса, в котором в любом порядке перечисляются конструкторы, деструктор, поля, методы, вложенные классы, заключается в фигурные скобки.

Упражнение

1. Перепишите листинги 11.1—11.3, записывая классы со словом `class`. Проследите за правильным доступом к членам класса.

Указатель на себя *this*

Очень часто методу класса нужно сослаться на объект, которому принадлежит метод. В дальнейшем мы часто будем сталкиваться с такой необходимостью. Уже в листинге 11.5 вы можете увидеть применение такой ссылки. Для реализации этой потребности в каждый нестатический метод неявно передается указатель на объект, который выполняет этот метод. Имя этого указателя `this`. Метод всегда может использовать указатель `this` для уточнения объекта, даже если в этом нет необходимости. Например, методы доступа к полю `x` класса `A` предыдущего раздела можно записать так:

```
double getX(){ return this->x; }
void setX(double x){ this->x = x; }
```

Употребление указателя `this` подобно тому, как мы говорим про себя "я", а не "Иван Петрович" или "Марья Ивановна".

Описание полей класса

При описании поля указывается его тип, затем, через пробел, имя и, может быть, начальное значение после знака равенства, которое можно записать константным выражением. Все это делается точно так же, как мы не раз делали в предыдущих главах.

Описание поля может начинаться с одного или нескольких необязательных модификаторов `static`, `const`, `mutable`, `volatile`. Статическое поле, т. е. поле, объявленное с модификатором `static`, едино для всех экземпляров класса. Ячейка оперативной памяти выделяется для него при первой загрузке класса, до создания экземпляров класса. Поэтому им можно пользоваться, даже не создавая ни одного объекта класса. Изменение статического поля одним объектом класса влечет изменение его значения во всех других объектах.

Константное поле, помеченное модификатором `const`, изменять нельзя, компилятор будет за этим следить и выдавать сообщение при каждой попытке изменения константы. Поля, объявленные `mutable`, можно изменять даже тогда, когда весь экземпляр класса объявлен константой, т. е. со словом `const`.

Модификатор `volatile` означает, что поле может быть неожиданно изменено каким-то внешним процессом, и мы не можем рассчитывать на то, что у него есть какое-то определенное значение. Компилятор не будет пытаться оптимизировать работу с таким полем.

Упражнение

2. Подумайте, не требуют ли поля класса `circle` из листингов 11.1—11.3 модификаторов, и вставьте их при необходимости.

Описание методов класса

При описании метода указывается тип возвращаемого им значения или слово `void`, затем, через пробел, имя метода, потом, в скобках, список параметров. После этого, в фигурных скобках, расписывается выполняемый метод. Описание метода может начинаться с модификаторов `static`, `virtual`. О модификаторе `virtual` мы поговорим позднее, а модификатор `static` превращает метод в *статический*.

Статический метод предназначен для работы со статическими полями. В нем нет указателя `this`, поэтому его можно использовать без создания объектов, записывая при обращении к статическому методу имя класса, а не имя объекта. Например:

```
class A{
   static int x;
public:
   static int getX(){ return x; }
};
main(){
   int n = A::getX();
}
```

Отсутствие указателя `this` ограничивает возможности статического метода: он не может обращаться к нестатическим полям и методам своего класса напрямую, не создавая его экземпляров. Компилятор заметит такое нарушение контракта метода и выдаст сообщение об ошибке.

В списке параметров метода через запятую перечисляются тип и имя каждого параметра. Перед типом какого-либо параметра может стоять модификатор `const`. Такой параметр нельзя изменять внутри метода. Список параметров может отсутствовать, но скобки сохраняются. После закрывающей круглой скобки может стоять модификатор `const`, например:

```
class A{
public:
   int zero() const{
      return 0;
   }
};
```

В этом случае слово `const` означает, что это *константный* метод. Константные методы могут применяться объектами, созданными с модификатором `const`, например:

```
main(){
```

```
   const A a;
   int n = a.zero();
}
```

Обращаться к обычным, не константным, методам таким объектам запрещено.

Контракт константного метода предусматривает, что он не может быть статическим. Он не может изменять нестатические поля класса, кроме полей `mutable`, и может обращаться только к константным методам класса. Компилятор следит за выполнением контракта и сообщает о его нарушении.

После круглой скобки, закрывающей список параметров нестатического метода, можно записать слово `volatile`. Аналогично константному методу, этот модификатор показывает, что к этому методу могут обращаться объекты, созданные с модификатором `volatile`. Обращаться к обычным методам таким объектам запрещено.

Упражнение

3. Подумайте, не требуют ли методы класса `circle` из листингов 11.1—11.3 модификаторов и вставьте их при необходимости.

Пример класса

В листинге 11.4 показано, как можно оформить в виде класса метод деления пополам для нахождения корня нелинейного уравнения, записанный нами ранее в листингах 3.2 и 5.3.

Листинг 11.4. Нахождение корня нелинейного уравнения методом бисекции

```
#include <stdio.h>
#include <math.h>
      class Bisection{

         const static double EPS = 1e-8; // Константа.

         double a, // Закрытые поля класса.
                b,
                root;
```

```
public:

   double getRoot(){return root;}  // Метод доступа.

   double f(const double x){

      return x*x*x - 3*x*x + 3; // Если хотите,
          // напишите что-нибудь другое.
   }

   Bisection(double a, double b){
      this->a = a; this->b = b;
   }

   void bisect(){ // Параметров нет — метод
                  // работает с полями класса.
     double y = 0.0; // Локальная переменная — не поле.
     do{
        root = 0.5 *(a + b);
        y = f(root);
        if (fabs(y) < EPS) break;

        if (f(a) * y < 0.0) b = root;
        else a = root;

     }while (fabs(b-a) >= EPS);
   }
};

main(){
   Bisection b(0.0, 1.5);
```

```
        b.bisect();
        printf("x = %f, f() = %f\n",
          b.getRoot(), // Обращаемся к корню через метод доступа.
          b.f(b.getRoot()));
    }
```

В описании метода `f()` сохранен старый, процедурный стиль: метод получает аргумент, обрабатывает его и возвращает результат. Описание метода `bisect()` выполнено в духе объектно-ориентированного программирования: метод активен, он сам обращается к полям экземпляра `b` и сам заносит результат в нужное поле. Метод `bisect()` — это внутренний механизм класса `Bisection`, поэтому он закрыт, `private`.

Упражнение

4. Перепишите метод `f()` из листинга 11.4 в объектно-ориентированном стиле.

Класс комплексных чисел

В подавляющем большинстве случаев классы создаются для того, чтобы ввести новый тип данных со своим набором операций. Например, в науке и технике очень часто используются комплексные числа. Было бы удобно работать с ними так, как это принято в математике, применяя обычные знаки арифметических операций. Для этого надо создать класс комплексных чисел и перегрузить на него обычные операции. В листинге 11.5 приведен пример такого класса. Сразу же надо сказать, что класс `complex` есть в стандартной библиотеке классов языка C++ и в своих программах лучше пользоваться стандартным классом.

Листинг 11.5. Класс комплексных чисел

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

    class Complex{

        double re, im;
```

```
public:

   Complex(double a = 0.0, double b = 0.0){
      re = a; im = b;
   }
   double getRe() const{ return re; }
   double getIm() const{ return im; }
   double mod() const{ return sqrt(re * re + im * im); }
   double arg() const{ return atan2(im, re); }

   int operator==(const Complex &z){
      return re == z.re && im == z.im;
   }
   int operator==(const double &a){
      return re == a && im == 0.0;
   }
   Complex &operator+=(const Complex &z){
      re += z.re;
      im += z.im;
      return *this;
   }
   Complex &operator+=(const double &a){
      re += a;
      return *this;
   }
   Complex &operator-=(const Complex &z){
      re -= z.re;
      im -= z.im;
      return *this;
   }
```

```
Complex &operator-=(const double &a){
   re -= a;
   return *this;
}
Complex &operator*=(const Complex &z){
   double t = re;
   re = re * z.re - im * z.im;
   im = t  * z.im + im * z.re;
   return *this;
}
Complex &operator*=(const double &a){
   re *= a;
   im *= a;
   return *this;
}
Complex &operator/=(const Complex &z){
   double m2 = z.re * z.re + z.im * z.im;
   if (m2 < 1e-8){
      fprintf(stderr, "Деление на нуль\n");
      exit(-1);
   }
   double t = re;
   re = (re * z.re + im * z.im)/ m2;
   im = (im * z.re - t  * z.im)/ m2;
   return *this;
}
Complex &operator/=(const double &a){
   if (fabs(a) < 1e-8){
      fprintf(stderr, "Деление на нуль\n");
```

```
            exit(-1);
        }
        re /= a;
        im /= a;
        return *this;
    }
    Complex operator+(const Complex &z){
        Complex t;
        t.re = re + z.re;
        t.im = im + z.im;
        return t;
    }
    Complex operator+(const double &a){
        Complex t;
        t.re = re + a;
        t.im = im;
        return t;
    }

    Complex operator-(const Complex &z){
        Complex t;
        t.re = re - z.re;
        t.im = im - z.im;
        return t;
    }
    Complex operator-(const double &a){
        Complex t;
        t.re = re - a;
        t.im = im;
```

```
        return t;
    }
    Complex operator*(const Complex &z){
        Complex t;
        t.re = re * z.re - im * z.im;
        t.im = re * z.im + im *z.re;
        return t;
    }
    Complex operator*(const double a){
        Complex t;
        t.re = re * a;
        t.im = im * a;
        return t;
    }
};

main(){
    Complex z1(2.0, -3.5), z2(1.5), z3;
    z3 = z1 + z2;
    z2 += -4.58;
    z1 /= z2 *= z1;
}
```

В классе `Complex` описан только один конструктор, но за счет того, что его параметрам даны начальные значения, его можно использовать с двумя параметрами, с одним параметром и без параметров, что и сделано в функции `main()`.

После конструктора, методов доступа `get()`, `set()` и методов `mod()`, `arg()`, возвращающих модуль и аргумент комплексного числа, перегружена операция сравнения `==` и четыре арифметические операции: `+`, `-`, `*`, `/`. Операция присваивания не перегружена, по умолчанию она будет реализована простым копированием. Зато перегружены составные арифметические операции присваивания: `+=`, `-=`, `*=`, `/=`. В объектно-ориентированном программи-

ровании составные операции присваивания гораздо эффективнее обычных арифметических операций, т. к. они не создают новые объекты, а только изменяют существующие. Создание объектов требует много ресурсов и не следует умножать число экземпляров класса без необходимости.

Обратите внимание на то, что перегрузка операции, оформленная как член класса, записывается только с одним параметром-операндом. Первым операндом служит объект самого класса. На использование операции это не влияет, она по-прежнему записывается с двумя операндами. Просто запись `z1 + z2` интерпретируется как `z1.operator+(z2)`.

Можно перегрузить операцию как отдельную функцию, не входящую в состав класса, например:

```
  Complex &operator+=(Complex &z1, const Complex &z2){

     z1.re += z2.re;

     z1.im += z2.im;

     return z1;

  }

  Complex operator+(const Complex &z1, const Complex &z2){

Complex t;
     t.re = z1.re + z2.re;

     t.im = z1.im + z2.im;

     return t;

  }
```

В этом случае, как обычно, запись `z1 + z2` интерпретируется как `operator+(z1, z2)`.

Упражнение

5. Перепишите переопределения операций из листинга 11.5 в виде отдельных функций.

Класс с деструктором

До сих пор мы записывали простейшие конструкторы, просто определяющие поля класса, и создавали объекты, которым не требовался деструктор. Приведем пример класса, в котором деструктор необходим.

Строки символов в языке C представлены массивом символов, завершающимся нуль-символом. Такое представление не позволяет проводить опера-

ции со строкой: получать длину строки, отыскивать в ней подстроку или отдельный символ и т. д. Оформим строку в виде класса, снабженного операцией сцепления и методами работы с ней. Опять-таки напоминаю, что это только учебный пример. В состав стандартной библиотеки классов языка C++ входит класс `string`, который разумно использовать в своих программах. *Подробную информацию см. в гл. 15.*

Листинг 11.6. Класс строк символов

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

      class String{

         char *str;

      public:

         int length;

         String(int len = 0){
            if (len < 0){
              fprintf(stderr, "Отрицательная длина\n");
              exit(-1);
            }
            str = new char[len + 1];
            if (str == NULL){
              str = "";
              length = 0;
            }else
              length = len;
         }
```

```
String(char *s){
   length = strlen(s);
   str = new char[length + 1];
   if (str == NULL){
     str = "";
     length = 0;
   }else
     strcpy(str, s);
}
~String(){
   delete[] str;
}

String &operator=(const String &s){
   str = s.str;
   length = s.length;
   return *this;
}
String &operator=(const char *s){
   str = s;
   length = strlen(s);
   return *this;
}
String &operator+=(const String &s){
   length += s.length;
   char *t = new char[length + 1];
   strcpy(t, str);
   strcat(t, s.str);
   str = t;
```

```
        return *this;
    }
    String &operator+=(const char *s){
        length += strlen(s);
        char *t = new char[length + 1];
        strcpy(t, str);
        strcat(t, s);
        delete[] str;
        str = t;
        return *this;
    }
    String operator+(const String &s){
        String tmp;
        tmp.length = length + s.length;
        char *t = new char[tmp.length + 1];
        strcpy(t, str);
        strcat(t, s.str);
        tmp.str = t;
        return tmp;
    }
    String operator+(const char *s){
        String tmp;
        tmp.length = length + strlen(s);
        char *t = new char[tmp.length + 1];
        strcpy(t, str);
        strcat(t, s);
        tmp.str = t;
        return tmp;
    }
```

```
   int charAt(const int pos) const{
      if (pos >= 0 && pos < length)
         return str[pos];
      return -1;
   }
   int indexOf(const char ch){
       int k = 0;
       for (; k <= length; k++)
          if (str[k] == ch) break;
       return k < length ? k : -1;
   }
};

main(){
   String s1(20), s2("Строка символов.");
   s1 = s2;
   s1 += s2;
   printf("%c\n", s2.charAt(6));
}
```

Конструктор класса `String` из листинга 11.6 выделяет память для хранения массива символов. Поэтому понадобился деструктор класса, освобождающий эту область памяти.

Упражнение

6. Дополните класс `String` из листинга 11.6 методом `substring()` поиска заданной подстроки. Какие параметры надо передать этому методу?

Наследование классов

В предыдущей главе мы познакомились с важнейшим принципом объектно-ориентированного программирования — принципом наследования — по-

зволяющим классифицировать объекты, создавать более или менее детальные модели информационных процессов и легко переходить от одной модели к другой.

Посмотрим, как этот принцип реализован в языке C++ на шутливом примере классификации домашних животных. Чтобы не перегружать изложение деталями, в классе `Pet`, введенном нами в предыдущей главе, оставлено только два поля: `name` — кличка животного, `person` — его хозяин и только один метод `voice()` — звуки, издаваемые им. От класса `Pet` унаследованы классы `Cat`, `Dog` и `Cow`. Они переопределяют метод `voice()` и вводят свои методы, характерные для кошек, собак и коров.

Листинг 11.7. Классификация животных

```
#include <stdio.h>

class Pet{

   String name;
   Master person;

public:

   Pet(String pName, String mName):name(pName),person(mName){}
   Pet(){ name = "Nameless"; }

   String getName(){ return name; }
   void setName(String s){ name = s; }

   void voice(){
      printf("? ? ?\n");
   }
};

class Cat: public Pet{
public:

   Cat(String pName, String mName):Pet(pName, mName){}
   Cat():Pet(){}
```

```
   int mouseCatched;

   void toMouse();

   void voice(){
      printf("Мяу! Мяу!\n");
   }
};

class Dog: public Pet{
public:

   Dog(String pName, String mName):Pet(pName, mName){}
   Dog():Pet(){}

   void preserve();

   void voice(){
      printf("Гав! Гав!\n");
   }
};

class Cow: public Pet{
public:

   Cow(String pName, String mName):Pet(pName, mName){}
   Cow():Pet(){}

   void giveMilk();

   void voice(){
      printf("My-y-y!\n");
   }
};

main(){
   Pet pet;
   pet.voice();
```

```
   Cat murka("Мурка", "Иванова");
   murka.voice();
   murka.Pet::voice();

   Dog tuzik("Тузик", "Петров");
   tuzik.voice();
   tuzik.Pet::voice();

   Cow nochka("Ночка", "Сидорова");
   nochka.voice();
   nochka.Pet::voice();
}
```

Инициализация полей классов

Поля `name` и `person` класса `Pet` сами являются объектами классов `String` и `Master` и для своей инициализации требуют вызова конструкторов своих классов. Для этого в конструкторе класса `Pet` применена *инициализация полей класса*, заключающаяся в том, что в заголовке конструктора после списка аргументов и двоеточия перечисляются поля (а не их классы!), требующие инициализации, и в скобках указываются их значения. Эти значения будут переданы соответствующим по типу и количеству аргументов конструкторам соответствующих классов.

```
Pet(String pName, String mName):name(pName),person(mName){}
```

Как видите, в конструкторе класса `Pet` кроме инициализации полей больше ничего делать не надо, поэтому тело конструктора пусто. Конструкторы полей класса выполняют свою работу до конструктора самого класса, в том порядке, в котором поля записаны в классе.

Для конструкторов по умолчанию, т. е. конструкторов без аргументов или конструкторов, у всех аргументов которых есть значения по умолчанию, инициализация полей классов не нужна. Инициализацию можно провести для любых полей класса, необязательно, чтобы они были объектами. Например:

```
class A{

   int a;
   int b;
   double x;

public:

    A(int a1 = 0, int a2 = 1, double x1 = 0.1):a(a1), b(a2), x(x1){}
};
```

Дело в том, что компилятор для каждого поля простого типа неявно создает класс, содержащий это поле. Автоматически созданный конструктор этого неявного класса дает полю значение. Этот-то конструктор и инициализирует поле.

Для полей-констант это единственный способ инициализации, поскольку потом их значения нельзя изменить.

Конструкторы, деструкторы и наследование

Обратите внимание на то, что конструкторы не наследуются — их приходится заново определять в подклассах. Тем не менее, при создании объекта подкласса для его правильного функционирования сначала надо создать объект суперкласса. Для этого автоматически выполняются конструкторы всех предков, начиная с самого далекого. При этом вызываются конструкторы по умолчанию. Если такового у какого-то предка нет, то конструктор по умолчанию создается автоматически. Такой конструктор просто выделяет память для всех полей и обнуляет ее.

Если же надо вызвать не конструктор по умолчанию, а какой-нибудь другой конструктор предка, то об этом должен позаботиться программист. Для этого в синтаксис конструктора введена возможность вызова конструктора суперкласса. Этот вызов записывается так же, как и инициализация полей класса. Как видно из листинга 11.7, конструктор суперкласса записывается после параметров конструктора подкласса и двоеточия, перед открывающей фигурной скобкой. Конструкторы классов `Cat`, `Dog` и `Cow` только это и делают, их тело пусто.

```
Cat(String pName, String mName):Pet(pNname, mName){}
```

Вызов конструктора суперкласса можно совместить с инициализацией полей самого класса, перечислив все через запятую в любом порядке.

Деструкторы полей класса и полей классов-предков вызываются в порядке, обратном порядку вызова конструкторов. Очень важно, чтобы были вызваны все деструкторы и освобождены все ресурсы, занимаемые объектом. Для этого деструктор самого далекого предка следует сделать *виртуальным*, например:

```
virtual ~String(){
   delete[] str;
}
```

Это обеспечит выполнение всей цепочки деструкторов даже для виртуальных методов.

Обращение к переопределенному методу суперкласса

Несмотря на то, что метод `voice()` класса `Pet` переопределен в подклассах, он не потерян — объекты подклассов могут обратиться к нему, используя операцию разрешения области видимости, как сделано в функции `main()`:

```
murka.Pet::voice();
```

Конечно, у объекта должен быть доступ к методу.

Доступ к членам суперкласса

Обращение к методу `voice()` суперкласса `Pet` объектами подклассов `Cat`, `Dog` и `Cow` допустимо потому, что он объявлен с уровнем доступа `public`. Закрытые (`private`) поля `name` и `person` класса `Pet` доступны наследникам, как и всем остальным классам, только через методы доступа, которые, как правило, объявляются открытыми, `public`.

Это правило может распространяться дальше по иерархии. Класс `Pointer`, наследник класса `Dog`, тоже может обратиться к методу `voice()` своего предка `Pet`:

```
class Pointer: Dog{
  // Свойства породы "Пойнтер".
};
main(){
   Pointer sharik;
   sharik.Pet::voice();
}
```

Впрочем, откомпилировав этот код и попытавшись выполнить его, вы получите сообщение об ошибке. Дело вот в чем: открытые и защищенные поля и методы класса `Pet` оказались закрытыми в подклассе `Dog` и, следовательно, недоступны в его подклассе `Pointer`. Произошло это потому, что при определении класса `Dog` мы не указали в заголовке уровень доступа к суперклассу `Pet`:

```
class Dog: Pet{ . . . };
```

По умолчанию суперкласс оказался закрытым, его открытые и защищенные члены при этом закрываются, предыдущая запись эквивалентна следующей записи:

```
class Dog: private Pet{ . . . };
```

Если мы хотим сделать суперкласс открытым в подклассе, оставив прежний уровень доступа к открытым и защищенным членам суперкласса, то надо написать:

```
class Dog: public Pet{ . . . };
```

Если же написать заголовок класса следующим образом, то открытые члены суперкласса станут защищенными в подклассе:

```
class Dog: protected Pet{ . . . };
```

Защищенные и закрытые члены суперкласса останутся такими же в подклассе. Для суперклассов-структур по умолчанию принято открытое наследование. Запись:

```
struct A{ . . . }
class B: A{ . . . }
```

эквивалентна записи:

```
class B: public A{ . . . }
```

Реализация полиморфизма

Переопределение методов приводит к интересным результатам. В классе `Pet` мы описали метод `voice()` и переопределили его в подклассах `Cat`, `Dog` и `Cow`. Объекты каждого из этих классов, разумеется, используют свой вариант метода `voice()`. То же самое делают указатели на эти классы. Интересная ситуация возникнет, если определить указатель типа суперкласса и направить его на объект типа подкласса, например, вот так:

```
Pet *p = new Dog("Шарик", "Иванов");
p->voice();
```

В таком определении нет ничего недопустимого — указатель типа `Pet` "знает" все унаследованные члены класса `Dog` и может с ними работать. Конечно, он "не видит" добавленные члены, например, метод `preserve()`, но это может и не потребоваться. Нельзя сделать наоборот, потому что этому указателю нужны адреса всех членов класса `Dog`, а у него в распоряжении только члены класса `Pet`:

```
Dog *d = new Pet();  // Нельзя!
```

Какой метод `voice()` будет выполняться? Метод класса `Pet` или метод класса `Dog`? Ответ четко определен: будет выполняться метод `voice()` класса `Pet`. Таким образом, в языке C++ полиморфизм, по умолчанию не действует. Для того чтобы включить механизм полиморфизма, надо метод `voice()` объявить в классе `Pet` виртуальным, написав в его заголовке слово `virtual`. Это сделано в листинге 11.8.

Листинг 11.8. Пример полиморфного метода

```
#include <stdio.h>

class Pet{

   String name;
   Master person;

public:

   Pet(String pName, String mName):name(pName),person(mName){}
   Pet(){ name = "Nameless"; }

   String getName(){ return name; }
   void setName(String s){ name = s; }

protected:

   virtual void voice(){
      printf("? ? ?\n");
   }
};

class Cat: Pet{
public:

   Cat(String pName, String mName):Pet(pNname, mName){}
   Cat():Pet(){}

   int mouseCatched;

   void toMouse();

   void voice(){
      printf("Мяу! Мяу!\n");
   }
};

class Dog: Pet{
public:

   Dog(String pName, String mName):Pet(pNname, mName){}
   Dog():Pet(){}
```

```
   void preserve();
   void voice(){
      printf("Гав! Гав!\n");
   }
};

class Cow: Pet{
public:

   Cow(String pName, String mName):Pet(pNname, mName){}
   Cow():Pet(){}

   void giveMilk();

   void voice(){
      printf("Му-у-у!\n");
   }
};

main(){
     Pet *singer[3];
     singer[0] = new Dog;
     singer[1] = new Cat;
     singer[2] = new Cow;
     for (int i = 0; i < 3; i++)
        singer[i]->voice();
  }
}
```

Откомпилировав и выполнив эту программу, вы увидите, что животные "поют" своими голосами!

Все дело здесь в определении массива `singer[]`. Хотя массив указателей `singer[]` имеет тип `Pet`, каждый его элемент указывает на объект своего типа `Dog`, `Cat`, `Cow`. При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки! Так в языке C++ реализуется полиморфизм.

Абстрактные классы

При описании класса `Pet` мы не можем задать в методе `voice()` никакого полезного алгоритма, поскольку у всех животных совершенно разные голоса.

Мы вышли из этого положения, выводя на консоль вопросительные знаки, но согласитесь, что это не лучший выход из положения.

В таких случаях лучше записать только заголовок виртуального метода и поставить вместо тела метода и фигурных скобок знак равенства, ноль и точку с запятой:

```
virtual void voice() = 0;
```

Такой метод называется *чисто виртуальным методом* (pure virtual method). Класс, в котором есть хотя бы один чисто виртуальный метод, называется *абстрактным* (abstract) классом. Создать экземпляры абстрактного класса, а тем более использовать их, не удастся, поскольку чисто виртуальный метод не определен. Тем не менее, можно определить указатель или ссылку на абстрактный класс.

Как же использовать абстрактные классы? Только порождая от них подклассы, в которых определены чистые виртуальные методы абстрактного класса. Зачем же нужны абстрактные классы? Не лучше ли сразу написать нужные классы с полностью определенными методами, а не наследовать их от абстрактного класса? Для ответа снова обратимся к листингу 11.8.

Хотя элементы массива `singer[]` ссылаются на подклассы `Dog`, `Cat`, `Cow`, но все-таки это переменные типа `Pet` и ссылаться они могут только на поля и методы, описанные в суперклассе `Pet`. Дополнительные поля подкласса для них недоступны. Попробуйте обратиться, например, к методу `preserve()` класса `Dog`, написав `singer[0].preserve()`. Компилятор "скажет", что он не может это реализовать.

Поэтому метод, который реализуется в нескольких подклассах, приходится выносить в суперкласс, а если там его нельзя реализовать, то объявлять чистым виртуальным. Таким образом, абстрактные классы группируются на вершине иерархии классов.

В абстрактных классах записывается проект информационной системы. Содержащиеся в них чисто виртуальные методы показывают, *что* делает система, какова ее функциональность, ничего не говоря о том, *как* это выполняется. Например, мы можем сказать, что все классы нашей иерархии способны осуществлять сортировку, написав в абстрактном классе чисто виртуальный метод:

```
virtual void sort() = 0;
```

Сортировка будет реализована в подклассах, причем каждый подкласс может использовать свой метод сортировки в зависимости от типа сортируемой информации.

Абстрактные классы полностью воплощают фундаментальный принцип объектно-ориентированного программирования — принцип абстракции. Многие объектно-ориентированные языки программирования, например,

Java и C#, вводят специальный тип — интерфейс, — в котором содержатся только константы и чисто виртуальные методы.

Кстати, можно задать пустую реализацию метода, просто поставив пару фигурных скобок, ничего не написав между ними, например:

```
void voice(){}
```

Получится полноценный метод. Но это искусственное решение, запутывающее структуру класса.

Множественное наследование

До сих пор мы получали подклассы только от одного суперкласса, каждый класс `B` или `C` происходил от одного класса `A`, как показано на рис. 11.1, *а*. Но часто возникает необходимость породить класс `D` от двух классов `B` и `C`, как показано на рис. 11.1, *б*. Это называется *множественным наследованием* (multiple inheritance).

Множественное наследование очень легко реализуется на языке C++. Достаточно написать в заголовке класса несколько его суперклассов через запятую:

```
class D: B, C{ . . . };
```

При этом можно независимо задать открытый, защищенный или закрытый доступ к каждому суперклассу `B` или `C`:

```
class D: public B, protected C{ . . . };
```

Множественное наследование вносит в программирование дополнительные трудности. В классах `B` и `C` могут оказаться одноименные методы, например:

```
#include <stdio.h>

class B{
protected:
   void f(){
      printf("Метод класса B\n");
   }
};
class C{
protected:
   void f(){
      printf("Метод класса C\n");
   }
};
```

Классу D может понадобиться обращение к этому методу:

```
class D: public B, public C{ . . . };
main(){
   D d;
   d.f(); // Ошибка!
}
```

Какой метод f() будет выполняться? Унаследованный от класса B или класса C? Это непонятно, такое обращение к методу f() ошибочно. Необходимо уточнение имени метода именем класса, которому метод принадлежит. Например:

```
main(){
   D d;
   d.B::f();
   d.C::f();
}
```

В некоторых случаях разрешить неоднозначность труднее. Например, у нас есть общий класс автомобилей Automobile, от которого можно породить класс грузовиков Truck и класс легковых автомобилей Car. Но вот надо описать пикап Pickup. Этот класс должен наследовать свойства и грузовых, и легковых автомобилей. Такая схема показана на рис. 11.1, *в*, где классы B и C сами порождены от одного класса A. Это так называемое "ромбовидное" наследование.

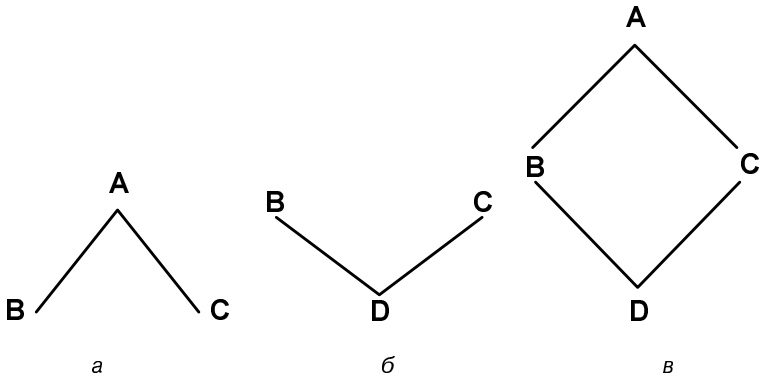


Рис. 11.1. Разные варианты наследования

Ромбовидное наследование иногда приводит к трудностям. В самом деле, пусть в классе A определен метод f(), к которому мы обращаемся из какого-то метода класса D. Можем мы быть уверены, что метод f() выполняет то, что написано в классе A, т. е. это метод A.f()? Может, он переопределен

в классах `B` и `C`? Если так, то каким вариантом мы пользуемся: `B.f()` или `C.f()`? Конечно, можно определить экземпляры классов и обращаться к методам этих экземпляров, но это совсем другой разговор.

Неоднозначность возникает из-за того, что суперклассы несколько раз включаются в подкласс, как показано на рис. 11.2, *а*, где объект класса `A` два раза появляется в классе `D`. В подклассе появляется несколько одноименных полей и методов. Кроме того, конструктор подкласса будет несколько раз вызывать конструктор суперкласса.

В разных языках программирования неоднозначность устраняется поразному, главным образом, уточнением имени метода `f()`, как было сделано выше, но это всегда требует дополнительных усилий. Вокруг множественного наследования много споров, есть его ярые приверженцы и столь же ярые противники. Во многих объектно-ориентированных языках, таких как Java или C#, множественное наследование вообще запрещено. Не будем встревать в эти споры, наше дело — наилучшим образом использовать средства языка для решения своих задач.

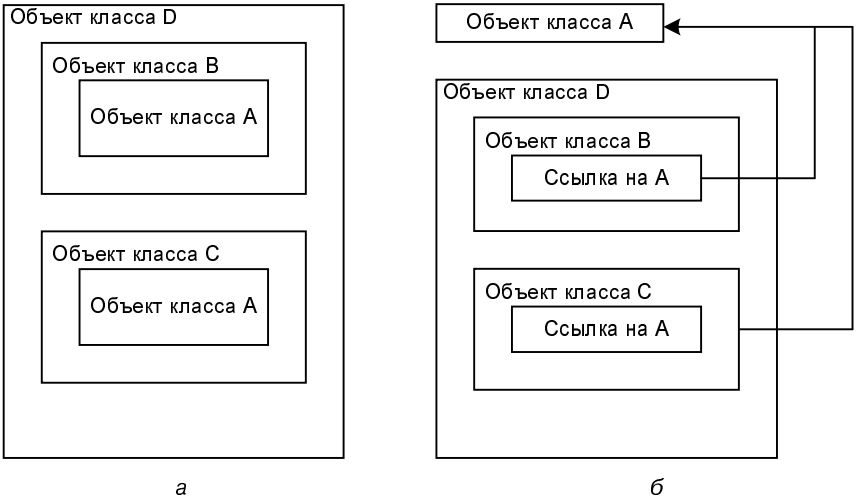


Рис. 11.2. Виртуальные базовые классы

В языке C++, кроме уточнения имен, есть еще один способ устранения неоднозначности множественного наследования. Мы можем объявить суперкласс виртуальным:

```
class A{
protected:
   void f(){
      printf("Метод класса A\n");
```

```
   }
};
class B: virtual public A{ . . . };
class C: virtual public A{ . . . };
class D: public B, public C{ . . .};
```

Класс `A` становится *виртуальным базовым классом* (virtual base class). После этого в подкласс включается не сам класс `A`, а только ссылка на него, как показано на рис. 11.2, *б*. Все ссылки направлены на один экземпляр суперкласса, сохраняя однозначность его полей и методов. После этого становится ясным, что в следующей функции идет обращение к методу `f()` класса `A`:

```
main(){
   D d;
   d.f(); // Правильно!
}
```

Друзья класса

Закрытые, `private`, члены класса недоступны методам других классов и отдельно записанным функциям. Защищенные, `protected`, члены класса доступны другим классам, только если это унаследованные классы. Такое правило соответствует принципу инкапсуляции и в большинстве случаев полезно. Но бывают ситуации, в которых удобнее разрешить доступ к закрытым полям, чем пользоваться методами доступа.

Для того чтобы разрешить отдельным функциям, членам класса или целому классу доступ к закрытым и защищенным членам класса `A`, надо объявить эти функции, члены или класс дружественными классу `A`. Объявление делается в любом месте класса `A` с помощью пометки `friend`, как показано в следующем примере:

```
class X{
   int f();
};

void func(X& x, Y& y){
   x.k = x.f2() + y.n;
}

class Y{
```

```
   friend void func(X&, Y&);
   int n;
   void f1(A& a){
       a.k = a.f2();
   }
};
class A{
   friend class X;
   friend void func(X&, Y&);
   friend void Y::f1();
   int k;
   int f2();
};
```

После такого объявления все методы класса `X`, функция `func()` и метод `f1()` класса `Y` получают прямой доступ к закрытым членам `k` и `f2()` класса `A`. Кроме того, функция `func()` получает доступ к закрытым и защищенным членам класса `Y`.

Методы дружественного класса `X`, дружественная функция `func()` и дружественный метод `f1()` все-таки не становятся полноценными членами класса `A`. Они не получают указатель `this` и не могут напрямую пользоваться членами класса `A`. Нельзя написать:

```
void func(){
   k = f2() + n;
}
```

поскольку внутри функции `func()` неизвестно, каким объектам принадлежат члены `k`, `f2()` и `n`. Объекты должны быть переданы функции `func()` через ее аргументы.

В отношении друзей действуют еще три правила:

1. Дружественность не взаимна. Из того, что класс `X` дружествен классу `A`, вовсе не следует, что класс `A` дружествен классу `X` и может пользоваться его закрытыми и защищенными членами, например, методом `f()`.
2. Дружественность не наследуется. Наследники класса `X` не получают доступ к классу `A`.
3. Дружественность не передается по наследству. Наследники класса `A` не считают класс `X` своим другом, так же как и функцию `func()` и метод `f1()` класса `Y`.

Пространства имен

Большие проекты, содержащие десятки и сотни классов и отдельных функций, как правило, реализуются группой программистов. Каждый программист создает свой набор классов, называя их по своему усмотрению. После сборки проекта может оказаться, что двум классам, решающим совершенно разные задачи, даны одинаковые имена. Изменение имени во всех местах, где оно используется, может занять довольно много времени.

Очень часто возникает необходимость сгруппировать классы по области их применения или по другим признакам, не включающим наследование или вложенность.

В этих случаях язык C++ рекомендует ввести *пространство имен* (namespace). Все имена, которые надо включить в одно пространство, записываются внутри именованного блока

```
namespace MyNames{
    // Классы, отдельные функции, глобальные переменные.
   const int MAXLEN = 9999;
   void func();
   class A;
}
```

В блоке обычно только перечисляются прототипы функций и имена. Их определение дается в другом месте. Очень часто блоки `namespace` записываются в заголовочных файлах.

Блок `namespace` с одним и тем же именем можно записать несколько раз, каждый блок добавляет в пространство имен новые имена. Можно написать далее:

```
namespace MyNames{
   double f(double);
}
```

В этом случае в пространстве имен `MyNames` будет четыре имени.

Уточнение имени

Вне своего пространства имена уточняются с помощью операции разрешения видимости:

```
if (k < MyNames::MAXLEN) f1(a[k]);
MyNames::func();
MyNames::A a = new MyNames::A();
```

Пространство имен можно сравнить с городом. Все улицы города должны носить разные имена. Но в разных городах названия улиц могут совпадать. В каждом городе есть Садовая улица, Шоссейная улица, Центральная улица. Поэтому, говоря о разных городах, мы уточняем название улицы, добавляя город, в котором она расположена. Если же разговор идет об одном городе, в таком уточнении нет нужды, ясно, о какой улице идет речь.

Директива *using namespace*

При частом использовании уточненных имен текст программы утяжеляется, теряет свою наглядность и становится слишком длинным. В таких случаях можно применить директиву `using namespace`, указав в ней имя пространства имен. Так, предыдущий фрагмент можно переписать следующим образом:

```
using namespace MyNames;
if (k < MAXLEN) f1(a[k]);
func();
A a = new A();
```

Увидев директиву `using namespace`, компилятор будет во всех следующих строках отыскивать встреченные имена в текущем пространстве и в пространстве имен `MyNames`. Разумеется, имена в этих пространствах должны быть различны. Совпадающие имена придется уточнять именем пространства имен.

Вложение пространств имен

Директиву `using namespace` можно записать и внутри блока, определяющего пространство имен:

```
namespace MyNewNames{
   using namespace MyNames;
   void f2();
}
```

Тем самым пространство имен `MyNames` вкладывается в пространство имен `MyNewNames`. Имена из пространства имен `MyNames` теперь лежат в пространстве имен `MyNewNames` и можно написать:

```
if (k < MyNewNames::MAXLEN) f1(a[k]);
MyNewNames::func();
MyNewNames::A a = new MyNewNames::A();
```

Такие имена будут сначала отыскиваться в пространстве имен `MyNewNames`, затем в пространствах имен, указанных в директивах `using namespace`.

Объявление *using*

Если же, наоборот, вы не хотите делать уточнение только для некоторых имен, то можете воспользоваться объявлением `using`, указав в нем полное имя. В дальнейшем это имя можно использовать без уточнения.

```
using MyNames::MAXLEN;
using MyNames::A;
if (k < MAXLEN) f1(a[k]);
MyNames::func();
A a = new A();
```

Для имен из стандартной библиотеки классов языка C++ выделено пространство имен, названное `std`. В этой книге мы часто будем использовать данные имена и применять для сокращения записи директиву `using namespace std`.

Неименованное пространство имен

Можно определить и неименованное пространство имен:

```
namespace{
   double fmod(double, int);
   const double FM = 2.4523;
}
```

На самом деле компилятор даст какое-то уникальное имя этому пространству. Поскольку к именам из этого пространства как-то надо обращаться, компилятор тут же вставит директиву `using namespace`. В результате мы получим что-то вроде:

```
namespace XXX{
   double fmod(double, int);
   const double FM = 2.4523;
}
using namespace XXX;
```

Вследствие этого именами `fmod` и `FM` из неименованного пространства имен можно будет пользоваться в пределах файла, в котором все это написано. Итак, неименованное пространство имен ограничивает видимость имени файлом.

Псевдонимы пространства имен

Для уже введенного имени пространства имен можно создать псевдоним:

```
namespace mnn = MyNewNames;
```

Псевдоним можно использовать для сокращения записи — вместо `MyNewNames::MAXLEN` писать `mnn::MAXLEN`. Кроме того, псевдоним можно использовать так же, как мы обычно используем константы — при необходимости сменить истинное имя пространства имен, например, при обновлении библиотеки классов, нам достаточно сменить это имя только в одном месте.

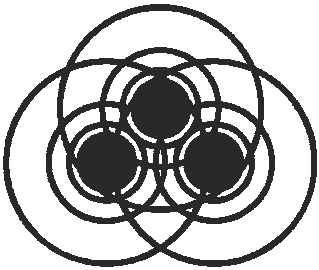
Упражнения

7. Запишите на языке C++ классы, описывающие процесс сдачи экзамена. Какие объекты участвуют в этом процессе? Как они взаимодействуют? Каковы методы сдачи экзамена?
8. Опишите процесс работы библиотеки в виде набора классов на языке C++.
9. Создайте на языке C++ классы, описывающие работу деканата.
10. Опишите на языке C++ информационную систему своего факультета.

Контрольные вопросы

1. Как реализуется абстрагирование в языке C++?
2. Равносильны ли определения класса служебными словами `class`, `struct` и `union`?
3. Как реализуется инкапсуляция в языке C++?
4. Может ли один экземпляр класса обратиться к закрытому члену другого экземпляра того же класса?
5. Может ли в классе быть несколько конструкторов?
6. Может ли в классе быть несколько деструкторов?
7. Как в языке C++ реализуется полиморфизм?
8. Что такое виртуальный метод класса?
9. Сохраняется ли виртуальность при переопределениях метода в классах-наследниках?
10. Может ли статический метод быть полиморфным?
11. Какие способы защиты параметра метода от его случайного изменения внутри метода вы можете предложить?

12. Что такое константный метод класса?
13. Может ли метод класса-наследника обратиться к закрытым полям и методам своего предка?
14. Можно ли открытый метод класса-предка переопределить закрытым методом в классе-наследнике?
15. Можно ли закрытый метод класса-предка переопределить открытым методом в классе-наследнике?
16. Можно ли запретить наследование какого-либо класса, сделать класс ненаследуемым?
17. Какой класс называется абстрактным?
18. Какие опасности несет множественное наследование? Как их избежать?
19. Что такое виртуальный базовый класс?
20. Что такое "друзья" класса?



Глава 12

Обработка исключительных ситуаций

Исключительные ситуации (exceptions) могут возникнуть во время выполнения (runtime) программы, прервав ее обычный ход. К ним относится деление на ноль, отсутствие загружаемого файла, отрицательный или вышедший за верхний предел индекс массива, невозможность выделения оперативной памяти или переполнение выделенной памяти и масса других неприятностей, которые могут случиться в самый неподходящий момент. Конечно, можно предусмотреть большинство таких ситуаций и застраховаться от них как-нибудь так:

```
if (something == wrong){
   // Предпринимаем аварийные действия
}else{
   // Обычный ход действий
}
```

Но это не выход. Посмотрите на любую штатную производственную программу, написанную на языке C или Pascal, и увидите, что она на две трети состоит из таких проверок. Кроме того, обработка исключительной ситуации происходит в том же месте, где она возникла, а это не всегда удобно. Во многих случаях хотелось бы сосредоточить всю обработку в одном месте, чтобы, например, предпринять одни и те же действия в ответ на разные ситуации.

В объектно-ориентированных языках программирования принят следующий подход. При возникновении исключительной ситуации исполняющая система создает объект определенного класса, соответствующего возникшей ситуации, содержащий сведения о том, что, где и когда произошло. Этот

объект передается на обработку программе, в которой возникло исключение. Если программа не обрабатывает исключение, то объект возвращается обработчику по умолчанию исполняющей системы. Обработчик, как правило, поступает очень просто: выводит на консоль сообщение о происшедшем исключении и прекращает выполнение программы.

Приведем пример. В программе листинга 12.1 может произойти деление на ноль, если запустить ее с аргументом `0`. В программе нет никаких средств обработки такой исключительной ситуации.

Листинг 12.1. Программа без обработки исключений

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[]){
    int n = atoi(argv[1]);
    printf("10 / n = %d\n",(10 / n));
    printf("После деления.\n");
}
```

Запишем программу в файл zerodiv.cc, откомпилируем в исполнимый файл zerodiv и запустим три раза.

```
$ zerodiv 5
```

Первый раз аргумент `argv[1]` равен `5` и программа выводит результат:

```
10 / n = 2

После деления.
```

Второй раз аргумент равен `0`:

```
$ zerodiv 0
```

Вместо результата мы получаем сообщение исполняющей системы, говорящее о том, что в функции `main()` произошла исключительная ситуация при выполнении операции с плавающей точкой:

```
Floating point exception
```

Третий раз запустим программу вообще без аргумента:

```
$ zerodiv
```

В массиве `argv[]` нет элементов, его длина равна нулю, а мы пытаемся обратиться к элементу `argv[1]`. Получаем сообщение о нарушении защиты памяти:

```
Segmentation fault
```

Блоки перехвата исключения

Мы можем перехватить и обработать исключение в программе. При описании обработки применяется бейсбольная терминология. Говорят, что исполняющая система или программа *выбрасывает* (throw) объект-исключение. Этот объект *пролетает* через всю программу, появившись сначала в том методе, где произошло исключение, а программа в одном или нескольких местах *пытается* (try) его *перехватить* (catch) и обработать. Обработку можно сделать полностью в одном месте, а можно обработать исключение в одном месте, выбросить снова, перехватить в другом месте и обрабатывать дальше.

Мы уже много раз в этой книге сталкивались с необходимостью обрабатывать различные исключительные ситуации, но не делали этого, потому что не хотели отвлекаться от основных конструкций языка. Не вводите это в привычку! Хорошо написанные объектно-ориентированные программы обязательно должны обрабатывать все возникающие в них исключительные ситуации.

Для того чтобы попытаться перехватить объект-исключение, выброшенное оператором `throw`, надо весь код программы, в котором может возникнуть исключительная ситуация, охватить оператором `try{}catch(){}`. Каждый блок `catch(){}` перехватывает исключение только одного типа — того, который указан в его аргументе, но можно написать несколько блоков `catch(){}` для перехвата нескольких типов исключений.

Например, мы знаем, что в программе листинга 12.1 могут возникнуть исключения двух типов. Напишем классы исключительных ситуаций и блоки их обработки, как это сделано в листинге 12.2.

Листинг 12.2. Программа с блоками обработки исключений

```
#include <stdio.h>
#include <stdlib.h>

class MyException{
protected:
   int ind;
   char msg[256];
public:
   MyException(){ ind = -1; }
   const char* what(){
      sprintf(msg, "MyException: %d\n", ind);
      return msg;
```

```
   }
};

class ZeroDivide: public MyException{
public:
   ZeroDivide(): MyException(){}
   ZeroDivide(int n){
      ind = n;
   }
   const char* what(){
      sprintf(msg, "Zero divide: %d\n", ind);
      return msg;
   }
};

class IndexOutOfRange: public MyException{
   int ind;
   char msg[256];
public:
   IndexOutOfRange(): MyException(){}
   IndexOutOfRange(int n){
      ind = n;
   }
   const char* what(){
      sprintf(msg, "Array index out of range: %d\n", ind);
      return msg;
   }
};

main(int argc, char *argv[]){
    try{
       if (argc < 2) throw IndexOutOfRange(argc);
       int n = atoi(argv[1]);
       printf("После ввода числа.\n");

       if (n == 0) throw ZeroDivide(n);
       printf("10 / n = %d\n",(10 / n));
       printf("После деления.\n");
```

```
    }catch(ZeroDivide& ae){
       printf("Деление на нуль: %s\n", ae.what());
    }catch(IndexOutOfRange& arre){
       printf("Выход за пределы массива: %s\n",  arre.what());
    }catch(MyException& e){
       printf("Исключение общего вида: %s\n", e.what());
    }
    printf("Окончание программы.\n");
}
```

В программу вставлены блок `try{}` и три блока перехвата `catch(){}` для трех типов исключений, описанных в классах `ZeroDivide` и `IndexOutOfRange`. Эти классы унаследованы от класса `MyException`. Обработка исключения заключается просто в выводе сообщения методом `what()` соответствующего класса-исключения.

Кроме блоков перехвата в листинге 12.2 после каждого действия делается трассировочная печать, чтобы можно было проследить за порядком выполнения программы. Программа запущена три раза: с аргументом `5`, с аргументом `0` и вообще без аргумента.

После первого запуска, при обычном ходе программы, выводятся все сообщения. После второго запуска, приводящего к делению на ноль, управление сразу же после попытки деления, еще до присваивания, передается в соответствующий блок `catch(ZeroDivide& ae){}`, потом выполняется то, что написано в блоке `finally{}`. После третьего запуска управление при попытке выполнения функции `atoi()` передается в другой блок `catch(IndexOutOfRange& arre){}`.

При возникновении исключения какого-либо другого типа, относящегося к классу `MyException`, оно будет перехвачено и направлено в блок `catch(MyException& e){}`. Интересно, что пустой блок `catch(){}`, в котором между фигурными скобками нет ничего, даже пробела, тоже считается обработкой исключения, при возникновении исключительной ситуации выполнение программы не прекратится, а продолжится за блоком `try{}catch(){}`.

Конечно, в такой простой программе, как в листинге 12.2, где применяются только сообщения, выдаваемые функцией `what()`, необязательно создавать объекты классов `ZeroDivide` и `IndexOutOfRange`. Достаточно сгенерировать эти сообщения оператором `throw`, как показано в листинге 12.3. Но в более сложной обстановке могут потребоваться подробные сведения о возникшей ситуации. Эти сведения будет хранить класс-исключение и предоставлять их обработчику исключительной ситуации. Так сделано в стандартной библиотеке шаблонов, с которой мы ознакомимся позднее.

Кроме того, классификация исключений позволяет перехватывать более или менее общие ситуации и обрабатывать их одинаково или различно.

Листинг 12.3. Программа с сообщениями об исключениях

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[]){
    try{
       if (argc < 2) throw "Выход за границу массива.";
       int n = atoi(argv[1]);
       printf("После ввода числа.\n");

       if (n == 0) throw "Деление на нуль.";
       printf("10 / n = %d\n",(10 / n));
       printf("После деления.\n");

    }catch(char *e){
       printf("%s\n", e);
    }
    printf("Окончание программы.\n");
}
```

Немного выше было сказано, что выброшенное исключение пролетает через всю программу. Что это означает? Изменим программу листинга 12.3, вынеся деление в отдельную функцию `f()`. Получим листинг 12.4.

Листинг 12.4. Выбрасывание исключения из функции

```
#include <stdio.h>
#include <stdlib.h>

void f(int n) throw(char *){
    if (n == 0) throw "Деление на нуль.";
    printf(" 10 / n = %d\n", (10 / n));
}
main(int argc, char *argv[]){
    try{
```

```
        if (argc < 2) throw "Выход за границу массива.";
        int n = atoi(argv[1]);
        printf("После ввода числа.\n");

        f(n);
        printf("После деления.\n");

    }catch(char *e){
        printf("%s\n", e);
    }
    printf("Окончание программы.\n");
}
```

Откомпилировав и запустив программу листинга 12.4, убедимся, что вывод программы не изменился. Исключения, возникающие в функции `f()`, пролетают через эту функцию, вылетают в функцию `main()`, там перехватываются и обрабатываются.

Часть *throw* заголовка функции

То обстоятельство, что функция не обрабатывает возникающее в ней исключение, а *выбрасывает* (throw) его, следует отмечать в заголовке функции служебным словом `throw` и указанием класса исключения:

```
void f(int n) throw(char *){
   if (n == 0) throw "Деление на нуль.";
   printf(" 10 / n = %d\n" + (10 / n));
}
```

Если метод или конструктор выбрасывает несколько исключений, то их типы надо перечислить через запятую в скобках после слова `throw`. Заголовок функции `main()` листинга 12.1, если бы исключения, которые он выбрасывает, не обрабатывались, следовало бы написать так:

```
void main(int argc, char** argv)
         throw(ZeroDivide, IndexOutOfRange){
  // Содержимое метода
}
```

Если тип исключения не указан в заголовке функции, то при его возникновении выполняется стандартная функция `unexpected()`, описанная в заголовочном файле exception. Она просто завершает программу, вызывая для этого стандартную функцию `terminate()`.

Программист может изменить стандартное действие функции `unexpected()`. Для этого надо написать функцию-обработчик и вызвать стандартную функцию `set_unexpected()`, передав ей в качестве параметра указатель на эту функцию-обработчик.

Перенесем теперь обработку деления на ноль в функцию `f()` и добавим трассировочную печать, как это сделано в листинге 12.5.

Листинг 12.5. Обработка исключения внутри функции

```
#include <stdio.h>
#include <stdlib.h>

class MyException{
protected:
   int ind;
   char msg[256];
public:
   MyException(){ ind = -1; }
   const char* what(){
      sprintf(msg, "MyException: %d\n", ind);
      return msg;
   }
};

class ZeroDivide: public MyException{
public:
   ZeroDivide(): MyException(){}
   ZeroDivide(int n){
      ind = n;
   }
   const char* what(){
      sprintf(msg, "Zero divide: %d\n", ind);
      return msg;
   }
};

class IndexOutOfRange: public MyException{
public:
   IndexOutOfRange(): MyException(){}
```

```
   IndexOutOfRange(int n){
      ind = n;
   }
   const char* what(){
      sprintf(msg, "Array index out of range: %d\n", ind);
      return msg;
   }
};

void f(int n){
   try{
       if (n == 0) throw ZeroDivide(n);
       if (n == 0) throw "Деление на нуль.";
       printf(" 10 / n = %d\n", (10 / n));
       printf("После выполнения функции.\n");
   }catch(ZeroDivide& ae){
       printf("Деление на нуль: %s\n", ae.what());
    }
}

main(int argc, char** argv){
   try{
       if (argc < 2) throw IndexOutOfRange(argc);
       int n = atoi(argv[1]);
       printf("После ввода числа.\n");
       f(n);
       printf("После выполнения функции.\n");
    }catch(IndexOutOfRange& arre){
       printf("Выход за пределы массива: %s\n",  arre.what());
    }catch(MyException& e){
       printf("Исключение общего вида: %s\n", e.what());
    }
}
```

Внимательно проследите за передачей управления и заметьте, что исключение класса `ZeroDivide` уже не выбрасывается в функцию `main()`.

Оператор `try{}catch(){}` внутри функции `f()` можно рассматривать как вложенный в оператор обработки исключений в методе `main()`.

Оператор *throw*

Этот оператор очень прост: после слова `throw` через пробел записывается объект класса-исключения, переменная или константа, например, строка. Очень часто объект создается прямо в операторе `throw`, например, можно написать:

```
throw IndexOutOfRange(n);
```

Оператор `throw` можно записать в любом месте программы, в том числе и внутри блока `catch(){}`. Оператор `throw` немедленно выбрасывает записанный в нем объект-исключение и дальше обработка этого исключения идет в блоках `catch(){}`, как мы видели выше. Исключение, выброшенное в блоке `catch(){}`, будет обработано в вызывающей функции или в объемлющем блоке `try{}catch(){}`.

Внутри блока `catch(){}` можно записать оператор `throw` без параметра, тогда исключение выбрасывается повторно, что позволяет его обрабатывать по частям в нескольких местах программы.

Итак, исключение, не обработанное или только частично обработанное в той функции, где оно возникло, выбрасывается в ту функцию, которая вызвала функцию-источник исключения. Если исключение не обработано полностью в этой вызывающей функции, то оно выбрасывается дальше.

Если исключение пролетело без обработки всю "матрешку" вызовов функций, то оно, в конце концов, обрабатывается стандартной функцией `terminate()`, описанной в заголовочном файле exception. Эта функция просто завершает выполнение программы, вызывая для этого стандартную функцию `abort()`. Если вам надо изменить такое стандартное действие функции `terminate()`, то напишите функцию, выполняющую нужные вам действия, и передайте указатель на нее в стандартную функцию `set_terminate()`.

Порядок обработки исключений

Итак, каждый блок `catch(){}` перехватывает один определенный тип исключений. Если надо одинаково обработать несколько типов исключений, то можно воспользоваться тем, что классы-исключения образуют иерархию. Изменим еще раз листинг 12.2, получив листинг 12.6.

Листинг 12.6. Порядок обработки исключений

```
#include <stdio.h>
#include <stdlib.h>

class MyException{
protected:
```

```
   int ind;
   char msg[256];
public:
   MyException(){ ind = -1; }
   const char* what(){
      sprintf(msg, "MyException: %d\n", ind);
      return msg;
   }
};

class ZeroDivide: public MyException{
public:
   ZeroDivide(): MyException(){}
   ZeroDivide(int n){
      ind = n;
   }
   const char* what(){
      sprintf(msg, "Zero divide: %d\n", ind);
      return msg;
   }
};

class IndexOutOfRange: public MyException{
   int ind;
   char msg[256];
public:
   IndexOutOfRange(): MyException(){}
   IndexOutOfRange(int n){
      ind = n;
   }
   const char* what(){
      sprintf(msg, "Array index out of range: %d\n", ind);
      return msg;
   }
};

main(int argc, char *argv[]){
    try{
```

```
        if (argc < 2) throw IndexOutOfRange(argc);
        int n = atoi(argv[1]);
        printf("После ввода числа.\n");

        if (n == 0) throw ZeroDivide(n);
        printf("10 / n = %d\n",(10 / n));
        printf("После деления.\n");

    }catch(MyException& e){
        printf("Исключение общего вида: %s\n", e.what());
    }catch(ZeroDivide& ae){
        printf("Деление на нуль: %s\n", ae.what());
    }catch(IndexOutOfRange& arre){
        printf("Выход за пределы массива: %s\n",  arre.what());
    }
    printf("Окончание программы.\n");
}
```

В листинге 12.6 блоки `catch(){}` переставлены местами. Блок, обрабатывающий исключение класса `exception`, записан первым. Это приводит к тому, что два других блока `catch(){}` не работают, они никогда не получают управление. Впрочем, большинство компиляторов замечают такую ошибку и выдают предупреждение об этом.

Дело здесь в том, что классы `ZeroDivide` и `IndexOutOfRange` унаследованы от класса `MyException` и относятся к его типу. Исполняющая система в поисках обработчика просматривает блоки `catch(){}` в порядке их написания, пока не найдет блок подходящего типа. На этом поиск заканчивается и управление передается найденному блоку. Таким образом, перемещаясь по иерархии классов-исключений, мы можем обрабатывать сразу более или менее крупные совокупности исключений.

Создание собственных исключений

Примеры, приведенные в этой главе, мало убедительны. Мы не избавились от проверки данных в условных операторах. Более того, исключительные ситуации генерируются искусственно оператором `throw`. Но даже в таких простых примерах мы получили два преимущества. Это возможность сосредоточить все сведения, относящиеся к исключению, в специальных классах-исключениях и возможность вынести обработку исключительной ситуации в блоки `catch(){}`.

Все достоинства и мощь объектно-ориентированной обработки исключений проявляются в сложных проектах, содержащих десятки классов с множеством методов. Разрабатывая такой проект, проектировщик сразу же продумывает иерархию классов-обработчиков, охватывающую все или большинство исключительных ситуаций. Во все методы, в которых могут возникнуть исключения, вставляются операторы `throw`, генерирующие объекты-исключения. Впоследствии при обращении к этим методам используются блоки `try{}catch(){}`.

Что же надо предусмотреть, проектируя иерархию классов-исключений? Прежде всего, нужно четко определить ситуации, в которых будет возникать ваше собственное исключение, и подумать, не будет ли его перехват невольно перехватывать также и другие, неучтенные исключения. Потом надо выбрать суперкласс создаваемого класса-исключения. Им может быть класс `exception` из стандартной библиотеки шаблонов или свой собственный. После этого можно написать класс-исключение, который, как правило, содержит, кроме всего прочего, два конструктора и переопределение метода `what()`.

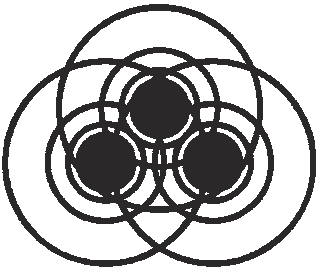
Упражнения

1. Внесите обработку исключений в класс `Complex`, определенный в предыдущей главе.
2. Внесите обработку исключений в класс `String`, определенный в предыдущей главе.
3. Пусть метод `int cipher(char c)` получает одну цифру в виде символа и возвращает значение этой цифры в виде целого числа. Какие исключительные ситуации могут возникнуть при неправильной записи параметра метода? Создайте классы этих исключений и реализуйте метод `cipher()`, выбрасывающий объекты этих классов.
4. Опишите информационную систему своего факультета.

Контрольные вопросы

1. Что такое исключительная ситуация?
2. Можно ли предусмотреть заранее все исключительные ситуации, возникающие в программе?
3. Какой способ обработки исключительных ситуаций предлагает объектно-ориентированное программирование?
4. Какие конструкции предлагает язык C++ для обработки исключительных ситуаций?

5. Имеет ли значение порядок записи блоков `catch(){}`?
6. Может ли исключительная ситуация возникнуть в блоке `catch(){}`?
7. Можно ли в блок `catch(){}` вложить блок `try{}catch(){}` для обработки возникшей там исключительной ситуации?
8. Можно ли вкладывать блоки `try{}catch(){}` один в другой?
9. Что должен содержать класс-исключение?
10. Где создаются объекты классов-исключений?
11. Что делает исполняющая система C++, если исключительная ситуация не обрабатывается в программе?
12. Можно ли изменить эти стандартные действия исполняющей системы C++?
13. Что делает исполняющая система C++, если метод не обрабатывает исключительную ситуацию и не декларирует это в своем заголовке?
14. Можно ли изменить эти стандартные действия исполняющей системы C++?
15. Можно ли обработать одну и ту же исключительную ситуацию несколько раз?



Глава 13

Дополнительные конструкции языка C++

Синтаксические конструкции языка C++, разобранные нами в двух предыдущих главах, полностью реализуют важнейшие принципы объектно-ориентированного программирования. Используя только эти конструкции, можно создавать полноценные объектно-ориентированные программы, выполняющие самую сложную обработку информации. Но это еще не все, что есть в языке C++. В этой главе мы посмотрим, какие дополнительные возможности предоставляются для построения программ.

Шаблоны классов

Мы уже познакомились с шаблонами функций — мощной конструкцией, позволяющей возложить на компилятор рутинную работу по созданию целого множества схожих функций. Аналогичная конструкция — *шаблон класса* (class template) — позволяет сделать то же самое для создания классов.

Класс `String` (*см. гл. 11*) работает только с массивом символов. Ничто не мешает написать такой же класс, работающий с массивом чисел. Но числа различных типов, `short`, `int`, `float`, `double`, требуют различного описания массивов, хотя методы работы с ними одинаковы. Было бы утомительно описывать все заново для каждого типа чисел. Гораздо удобнее написать шаблон класса, по которому компилятор будет создавать конкретные классы для каждого типа.

В листинге 13.1 показано, как это можно написать на языке C++. Шаблон класса `Vector` подходит для любых типов, в которых определены операции присваивания, взятия элемента массива, выделения и освобождения памяти

и определены математические функции взятия абсолютной величины и извлечения квадратного корня из числа.

Описание шаблона класса подобно описанию шаблона функции (*см. гл. 9*). Оно начинается со слова `template`, после которого в угловых скобках через запятую перечисляются параметры шаблона. Параметр может быть типом, предваряемым словом `class`, или константой, перед которой указывается тип константы. Тип не обязательно должен быть классом, слово `class` просто говорит о том, что это тип, а не константа. Многие компиляторы позволяют писать более понятное слово `typename` вместо слова `class`. При описании шаблона можно задать начальные значения параметрам:

```
template<class T = int, int = 100> class A{};
```

В этом случае при создании экземпляра шаблона соответствующие параметры можно не указывать. Параметры определяются при создании экземпляра шаблона. Для параметра-типа задается конкретный тип, для параметра-константы — определенное значение. Экземпляр шаблона — это обычный класс с обычными свойствами, ничем не отличающийся от любого другого класса.

Листинг 13.1. Шаблон класса

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

template <class T, int dim = 100> class Vector{

   T *v;

public:
   Vector();
   Vector(const Vector<T, dim>&);

   ~Vector(){
      delete[] v;
   }

   int getDim(){ return dim; }

   T & operator[](int index);

   Vector<T, dim>& operator+(Vector<T, dim>&);
```

```
   Vector<T, dim>& Vector<T, dim>::operator+(T& t);
   Vector & operator*(T&);

   T norm1();
   T norm2();
   T normInfinity(){

      T norm = fabs(v[0]);
      for (int i = 1; i < dim; i++)
         if (norm < fabs(v[i])) norm = fabs(v[i]);

      return norm;
   }
};

template <class T, int d>
Vector<T, d>::Vector(){

   v = new T[d];
}

template <class T, int d>
Vector<T, d>::Vector(const Vector<T, d>& vect){

   v = new T[vect.d];
   for (int k = 0; k < vect.d; k ++) v[k] = vect[k];
}

template <class T, int d>
T & Vector<T, d>::operator[](int index){

   if (index >= 0 && index < d) return v[index];
   else{
      fprintf(stderr, "Неверный индекс.\n");
      exit(-1);
   }
}
```

```
template <class T, int d>
Vector<T, d>& Vector<T, d>::operator+(Vector<T, d>& vect){

   if (dim != vect.d){
      fprintf(stderr, "Размерности не совпадают.\n");
      exit(-1);
   }
   for (int k = 0; k < d; k++) v[k] += vect.v[k];

   return *this;
}

template <class T, int d>
Vector<T, d>& Vector<T, d>::operator+(T& t){

   for (int k = 0; k < dim; k++) v[k] *= t;

   return *this;
}

template <class T, int d>
T Vector<T, d>::norm1(){

   T norm = 0;
   for (int i = 0; i < d; i++) norm += fabs(v[i]);

   return norm;
}

template <class T, int d>
T Vector<T, d>::norm2(){

   T norm = 0;
   for (int i = 0; i < d; i++) norm += v[i] * v[i];

   return sqrt(norm);
}
```

```
main(){

   Vector<int, 10> v1;
   Vector<int, 1000> v2;
   Vector<double> v3;

   for (int k = 0; k < 10; k++) v1[k] = k * k;
   for (int k = 0; k < 1000; k++) v2[k] = k * k + 1;
   for (int k = 0; k < v3.getDim(); k++) v3[k] = 1 /(k + 1);

   double d = v3.norm2();
   printf("%d\n", v2.norm1());
}
```

Все методы шаблона класса автоматически становятся шаблонами функций. Поэтому в их описании не обязательно записывать заголовок `template<>`, это подразумевается. Причем конкретные методы создаются по шаблону только во время первого обращения к ним, поэтому ошибки соответствия типов обнаруживаются только в этот момент, а не во время компиляции.

В листинге 13.1 определение конструкторов и некоторых методов вынесено за пределы класса, чтобы показать, что они реализуются как шаблоны функций. Обратите внимание на то, что в заголовках и параметрах методов после имени класса повторяется список параметров шаблона в угловых скобках, например:

```
Vector<T, d>::norm2(){
```

Это необходимо потому, что список параметров — неотъемлемая часть имени шаблона класса и список параметров в угловых скобках должен всегда сопровождать имя шаблона класса. Не существует класса `Vector`, а существуют реализации `Vector<int, 50>`, `Vector<double, 1000>` шаблона класса. Параметры в скобках не надо записывать только в имени конструктора и деструктора, поскольку это имя конструктора или деструктора, а не класса.

Деструктор класса и метод `normInfinity()` определены внутри класса. Их определение не требует шаблона `template<>`, потому что параметры шаблона автоматически передаются в метод, определенный внутри класса. Учтите, что шаблоны функций-членов класса не могут быть виртуальными.

Упражнения

1. Напишите шаблон класса `Vector` с одним параметром `<class T>`. Как изменятся конструкторы шаблона? Как определять экземпляры классов по такому шаблону?

2. Дополните шаблон класса `Vector` из листинга 13.1 методом вычисления скалярного произведения двух векторов. Какие параметры надо передать этому методу?

Друзья шаблона

В шаблоне класса можно объявить друзьями функции или отдельные методы классов. Они не обязаны быть шаблонами функций. Простые функции будут дружественными каждому классу, созданному по этому шаблону. Но если в числе их параметров встретится шаблон класса, то для каждого класса, созданного по этому шаблону, будет создана своя дружественная функция.

Например, у функции, вычисляющей скалярное произведение векторов, должен быть доступ к закрытым полям `v` и `dim` класса `Vector`, поэтому ее разумно объявить дружественной классу `Vector`. Можно написать функцию, возвращающую скалярное произведение типа `double`, поскольку это самый широкий числовой тип:

```
double scalProd(Vector<T, d>& v1, Vector<T, d>& v2){

   double s = 0.0;
   for (int k = 0; k < d; k++) s += v1[k] * v2[k];
   return s;
}
```

Такая функция будет дружественной любому классу, созданному по шаблону. Но можно написать шаблон функций, возвращающих тип элементов вектора, и объявить этот шаблон другом класса `Vector`:

```
template <class T, int d = 100> class Vector{

   friend T scalProd(const Vector<T, d>& v1, const Vector<T, d>& v2);

   // Прочее.
};

template<class T, int d>
T scalProd(const Vector<T, d>& v1, const Vector<T, d>& v2){

   T s = 0.0;
   for (int k = 0; k < d; k++) s += v1[k] * v2[k];
   return s;
}
```

В этом случае для каждого класса, полученного по шаблону, будет создана своя функция `scalProd()`, возвращающая значение конкретного типа.

Друзьями шаблона класса могут быть другие классы и шаблоны классов. Для них действуют те же правила, что и для друзей-функций. В следующей главе вы увидите, каким образом друзья облегчают программирование.

Члены класса — реализации шаблонов

Реализации шаблонов классов — это обычные классы, они могут быть полями других классов или других шаблонов. Пусть шаблон `A<T>` выглядит так:

```
template<class T> class A{
   T t;
public:
   A(T &t){
      this.t = t;
   }
};
```

Его реализацию можно сделать полем шаблона класса `B<T>`:

```
template<class T> class B{
   A<T> a;
public:
   B(A<T> &t){
        a = t;
   }
};
```

или полем простого класса `C`:

```
class C{
   A<T> a;
public:
   C(A<T> &t){
        a = t;
   }
};
```

Определить экземпляры классов `B` и `C` можно так:

```
main(){
```

```
   A<int> a;
   B<int> b(a);
   C c(a);
}
```

Членами класса могут быть и шаблоны классов. Они описываются прямо внутри класса, и их описание подобно описанию вложенных классов.

Наследование шаблонов

Шаблоны классов могут полноценно участвовать в иерархии классов. С одной стороны, шаблоны могут быть наследниками как обычных классов, так и шаблонов классов. С другой стороны, они могут служить базовыми классами для обычных классов и для шаблонов классов.

Наиболее естественно наследование шаблона класса от шаблона с теми же параметрами по такой схеме:

```
template<class T> class A{
   T t;
public:
   A(T &t){
      this.t = t;
   }
};

template<class T> class B: public A<T>{
   int n;
public:
   B(T &t, int n): A<T>(t){
      this.n = n;
   }
};
```

У унаследованного шаблона класса могут быть и дополнительные параметры, кроме тех, что заданы в родительском шаблоне. Если у всех реализаций шаблона должен быть один общий тип, то удобно наследовать шаблон от обычного класса. Схема наследования такова:

```
class A{
   int n;
   String *s;
```

```
public:
   A(int n, String *s){
      this.n = n; this.s = s;
   }
};

template<class T> class B: public A{
   T t;
public:
   B(T &t, int n, String *s): A(n, s){
      this.t = t;
   }
};
```

Очень часто унаследованный шаблон конкретизирует указатель типа `void *`, определенный в родительском классе:

```
class A{
   void *t;
   String *s;
public:
   A(void *t, String *s){
      this.t = t; this.s = s;
   }
};

template<class T> class B: public A{
public:
   B(T *t, String *s): A(t, s){}
};
```

Наконец, можно унаследовать обычный класс от шаблона класса:

```
template<class T> class A{
   T t;
public:
   A(T &t){
     this.t = t;
   }
};
```

```
class B: public A<T>{
   int n;
public:
   B(T &t, int n): A<T>(t){
      this.n = n;
   }
};
```

В этом случае унаследованный класс `B<T>` будет шаблоном класса с теми же параметрами, что и родительский класс.

Упражнение

3. Расширьте шаблон класса `Vector<T, int>` шаблоном `IndVector<T, int>`, допускающим отрицательные индексы элементов.

Специализированные версии шаблона

Мы уже сталкивались с ситуацией, в которой кроме шаблона функции `get_max()` пришлось написать отдельный вариант функции `get_max()` для строк (*см. гл. 9*). Такая же ситуация может встретиться при использовании шаблона класса. В этой ситуации можно конкретизировать шаблон, создав его специализированные версии. Например, нам может понадобиться вектор, содержащий строки класса `String`, но операции, определенные в шаблоне `Vector<T, int>`, не могут работать со строками, их надо переписать.

При описании специализированной версии шаблона параметры в угловых скобках записываются после имени класса, перед фигурной скобкой, открывающей тело класса. Затем записывается тело класса, содержащее все, что должно быть в специализированной версии:

```
template<> class Vector<String, 100>{
   // Тело класса.
};
```

Это полная специализация, на что указывает пустое содержимое угловых скобок `template<>`. Мы получили не шаблон, а обычный класс. Поэтому обозначение шаблона с пустым списком параметров `template<>` можно опустить:

```
class Vector<String, 100>{
   // Тело класса.
};
```

Но при использовании современных компиляторов это делать не рекомендуется, поскольку такую запись легко спутать с определением экземпляра шаблона. Можно сделать частичную специализацию, определив только некоторые параметры:

```
template<int n = 100> class Vector<String, int>{
   // Тело класса.
};
```

При этом получается шаблон класса с меньшим числом параметров.

Упражнение

4. Реализуйте частичную и полную специализацию шаблона `Vector<T, int>` для класса строк `String`, определенного в *гл. 11*.

Информация о типе на этапе выполнения

Типы данных существуют только для компилятора. Компилятор преобразует их в директивы загрузчику для выделения участков оперативной памяти нужного размера. Тем не менее, язык C++ позволяет сохранить информацию о типах и предоставлять ее. Для этого в язык введена операция `typeid()`. Как видите, она оформлена с аргументом подобно функции. Ее аргументом служит выражение или объект, тип которого надо определить, а результатом — создание объекта класса `type_info`, константная ссылка на который возвращается операцией `typeid()`.

```
Vector<int, 100> v1;
const type_info& rtti = typeid(v1);
const char *typeName = rtti.name();
```

Стандартный класс `type_info` описан в заголовочном файле `<typeinfo>`, который надо включать в программы, использующие класс `type_info`.

Разумеется, получать таким образом имя типа статических данных бессмысленно — двумя строками выше это имя указано явно. Информация о типе на этапе выполнения (RTTI, Run-Time Type Information) полезна в условиях полиморфизма, когда выполняются методы объекта, а не класса. Но и в этом случае полезнее не имя класса, а сравнение его с известным классом. Для этого в классе `type_info` перегружены операции `==` и `!=`.

Еще одно замечание. Методы RTTI хорошо работают только для *полиморфных типов* — классов, у которых есть хотя бы один виртуальный метод. Допустим, у нас есть метод `work(Pet *)` использования полезных свойств домашних животных. Полиморфный класс `Pet` мы определили (*см. гл. 10*) и

описали на языке C++ в листинге 11.8. У каждого из наследников класса `Pet` — классов `Cat`, `Dog` и `Cow` — есть специфические методы `toMouse()`, `preserve()`, `giveMilk()`. Метод `work(Pet *)` должен их использовать, но он не знает, указатель на объект какого именно класса ему передан.

Есть много способов узнать класс объекта в этом случае, в первую очередь, это правильное планирование и использование виртуальных функций. Посмотрим, как эту задачу можно решить средствами RTTI:

```
#include <typeinfo.h>

void work(Pet *p){
   if (typeid(*p) == typeid(Cat)){

      Cat *cat = (Cat *)p;
      cat.toMouse();

   }else if (typeid(*p) == typeid(Dog)){

      Dog *dog = (Dog *)p;
      dog.preserve();

   }else if (typeid(*p) == typeid(Cow)){

      Cow *cow = (Cow *)p;
      cow.giveMilk();
   }
}
```

Если бы класс `Pet` не был полиморфным, все сравнения оказались бы ложны. Обычная операция явного преобразования типа вида `(Cow *)p`, примененная в этом примере, ненадежна. Она не проверяет соответствие типов во время выполнения. Она не работает, если базовый класс объявлен виртуальным.

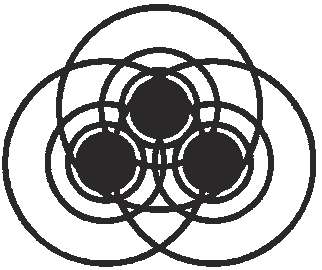
Поскольку подобные преобразования типов приходится делать довольно часто, в систему RTTI введена операция `dynamic_cast<T>(ptr)`. Ее можно применять к указателям или ссылкам `ptr` на полиморфный тип. Операция `dynamic_cast` пытается привести этот тип к типу `T`. При удачном приведении возвращается тип `T`. Обычно тип `T` — это указатель или ссылка на объект. При неудаче возвращается `NULL`, если `T` — указатель, или выбрасывается

исключение типа `Bad_cast`, если `T` — ссылка. С использованием операции `dynamic_cast` предыдущий пример будет выглядеть так:

```
#include <typeinfo.h>

void work(Pet *p){
   if (typeid(*p) == typeid(Cat)){

      Cat *cat = dynamic_cast<Cat *>(p);
      if (cat) cat.toMouse();

   }else if (typeid(*p) == typeid(Dog)){

      Dog *dog = dynamic_cast<Dog *>(p);
      if (dog) dog.preserve();

   }else if (typeid(*p) == typeid(Cow)){

      Cow *cow = dynamic_cast<Cow *>(p);
      if (cow) cow.giveMilk();
   }
}
```

Контрольные вопросы

1. Что такое шаблон класса?
2. Будут ли методы шаблона класса шаблонами?
3. Должны ли друзья класса быть шаблонами функций?
4. Могут ли поля обычного класса быть шаблонами класса?
5. Могут ли поля шаблона класса быть шаблонами класса?
6. Могут ли параметры шаблона быть не обычными классами, а шаблонами классов?
7. Может ли наследник обычного класса быть шаблоном класса?
8. Может ли наследник шаблона класса быть обычным классом?
9. В каких случаях необходимо динамическое определение типа во время выполнения?
10. Почему вместо обычного приведения типа в систему RTTI введена операция `dynamic_cast`?



Глава 14

Динамические структуры данных

До сих пор мы объединяли данные в массивы или в структуры. Это структурирование данных характерно тем, что элементы массива или члены структуры хранятся в соседних ячейках оперативной памяти, что значительно сокращает время поиска данных и ускоряет их обработку. Особенно быстро обрабатывается массив, поскольку его элементы однотипны, у них одинаковая длина. Чтобы перейти к следующему элементу массива, не нужно тратить время на поиск, достаточно переместиться на фиксированное число байтов, равное длине элемента массива.

Эта особенность массивов делает их очень удобными для однообразной обработки данных, когда надо быстро их просмотреть, для того чтобы найти сумму, наибольшее, наименьшее или среднее значение, отсортировать элементы. Но уже для поиска нужных сведений в большом объеме информации массивы неудобны. Для этого лучше подходят бинарные деревья поиска.

Массивы совершенно неудобны для пополнения или удаления информации. Их длина фиксирована. При добавлении элемента в массив придется менять его длину, а для этого придется создавать новый массив. После каждого удаления элемента из массива надо перемещать его элементы, чтобы они снова лежали в соседних ячейках оперативной памяти.

Для решения задач, в которых количество данных заранее неизвестно, задач, в которых данные часто добавляются и удаляются, в задачах поиска надо искать другие структуры данных, другие способы их хранения. В этой главе мы рассмотрим различные структуры данных и способы их хранения в оперативной памяти.

Стек

Начнем с не совсем обычной структуры данных, называемой *стеком* (stack). Стек реализует порядок работы с данными подобно магазину винтовки —

первым выстрелит патрон, положенный в магазин последним, — или подобно железнодорожному тупику — первым из тупика выйдет вагон, загнанный туда последним. Такой порядок обработки называется LIFO (Last In — First Out).

Стек похож на стопку бумаги, лежащей на столе. В каждый момент времени мы можем видеть или взять только верхний лист бумаги, тот, который мы положили на стол последним. Чтобы добраться до других листов, нам надо снять все вышележащие листы. Новые листы бумаги мы кладем сверху. Конечно, можно их положить вниз или между другими листами, но это неудобно.

Таким образом, работа со стеком состоит всего из двух основных операций:

- `push()` — положить на стек новый элемент;
- `pop()` — снять со стека верхний элемент.

Для удобства работы нужны дополнительные операции:

- `read()` — прочитать верхний элемент, не снимая его со стека;
- `empty()` — проверить, не пуст ли стек.

Можно еще хранить количество элементов стека в поле `size`, увеличивая значение этого поля на единицу при каждой вставке и уменьшая при удалении. Можно вместо этого определять количество элементов методом `size()`, подсчитывающим количество элементов стека при каждом обращении к методу. Но это не характерно для стека. Как правило, количество элементов в стеке не подсчитывается. Отслеживается только переполнение и опустошение стека.

Написанный выше минимальный набор операций превращает стек в *абстрактный* тип данных. Мы описали операции, но еще не решили, как их выполнять. Мы знаем, *что* должен делать стек, но не знаем, *как* он будет это делать.

Реализация стека может быть самой разной. Сделаем две реализации: на основе массива и на основе указателей. Они представлены в листингах 14.1—14.3. Во всех трех листингах сделана реализация шаблоном класса, ведь методы работы стека не зависят от типа хранящихся в нем элементов.

Листинг 14.1. Стек на основе массива

```
#include <stdio.h>
#include <stdlib.h>

      const unsigned int STACK_SIZE = 1024;


      template <class T> class Stack{
```

```
    int size; // Наибольшее количество элементов стека.
    int top;  // Индекс верхнего элемента.
    T *v;     // Динамический массив элементов стека.

  public:

    Stack(){
      v = new T[STACK_SIZE];
      if (!v){
        fprintf(stderr, "Нехватка памяти.\n");
        exit(-1);
      }
      size = STACK_SIZE;
      top = 0;
    }

    Stack(const int n){

      v = new T[n];
      if (!v){
        fprintf(stderr, "Нехватка памяти.\n");
        exit(-1);
      }
      size = n;
      top = 0;
    }

    ~Stack(){

      delete[] v;
    }
```

```
    bool empty(){
       return top < 0;
    }

    void push(const T& t){

       if (top == size){
         fprintf(stderr, "Переполнение стека.\n");
         return;
       }
       v[++top] = t;
    }

    T pop(){

       if (empty()){
         fprintf(stderr, "Стек пуст.\n");
         return 0;
       }
       return v[top--];
    }

    T read(){

       if (empty()){
         fprintf(stderr, "Стек пуст.\n");
         return 0;
       }
       return v[top];
    }
};
```

```
main(){

    Stack<String> st1;
    Stack<double> st2(100);
    Stack<int> st3;

    st1.push("Первый элемент.");
    st1.push("Второй элемент.");

    printf("%s\n", st1.pop());
    printf("%s\n", st1.read());

    st2.push(-27.05);

    for (int k = 0; k < 10; k++) st3.push(10 * k);

    for (int k = 0; k < 10; k++)
          printf("%d\n", st3.pop());
}
```

В начале работы стек пуст и индекс верхнего элемента `top` равен нулю. Первое применение метода `push()` заполняет элемент `v[0]`, после чего индекс `top` увеличивается на единицу. При следующем применении метода `push()` заполняется элемент `v[1]` и т. д. На рис. 14.1 показан стек с несколькими элементами.

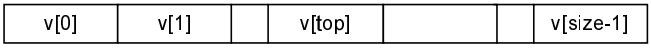


Рис. 14.1. Стек на основе массива

В каждый момент времени доступен только верхний элемент стека `v[top]`. При снятии элемента со стека методом `pop()` сначала возвращается элемент `v[top]`, после чего индекс `top` уменьшается на единицу.

Стек, описанный в листинге 14.1, хранит данные, размер каждого из которых может быть значительным. Во многих случаях данные уже расположены в оперативной памяти, и надо лишь организовать их в виде стека.

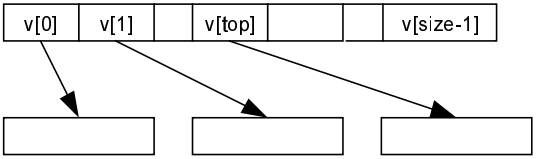


Рис. 14.2. Стек указателей на основе массива

В таком случае нет смысла копировать их в стек, достаточно хранить в стеке только указатели на данные, как показано на рис. 14.2. Этот способ организации стека особенно удобен в тех случаях, когда надо создать несколько структур на одном и том же множестве данных. Такая организация стека сделана в листинге 14.2.

Листинг 14.2. Стек указателей на основе массива

```
#include <stdio.h>
#include <stdlib.h>

    const unsigned int STACK_SIZE = 1024;

    template <class T> class Stack{

       int size; // Наибольшее количество элементов стека.
       int top;  // Индекс верхнего элемента.
       T **v;   // Динамический массив указателей на данные.

    public:

       Stack(){
          v = new T*[STACK_SIZE];
          if (!v){
            fprintf(stderr, "Нехватка памяти.\n");
            exit(-1);
          }
```

```
    size = STACK_SIZE;
    top = 0;
}

Stack(const int n){

    v = new T*[n];
    if (!v){
      fprintf(stderr, "Нехватка памяти.\n");
      exit(-1);
    }
    size = n;
    top = 0;
}

~Stack(){

    delete[] v;
}

bool empty(){
    return top < 0;
}

void push(T *t){

    if (top == size){
      fprintf(stderr, "Переполнение стека.\n");
      return;
```

```
        }
        v[++top] = t;
    }

    T *pop(){

        if (empty()){
          fprintf(stderr, "Стек пуст.\n");
          return 0;
        }
        return v[top--];
    }

    T *read(){

        if (empty()){
          fprintf(stderr, "Стек пуст.\n");
          return 0;
        }
        return v[top];
    }
};


main(){

    Stack<String> st1;
    Stack<double> st2(100);
    Stack<int> st3;

    st1.push("Первый элемент.");
    st1.push("Второй элемент.");
```

```
    printf("%s\n", st1.pop());
    printf("%s\n", st1.read());

    double x = -27.05;

    st2.push(&x);

    int a[] = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55};

    for (int k = 0; k < 10; k++) st3.push(a + k);

    for (int k = 0; k < 10; k++)
          printf("%d\n", *st3.pop());
}
```

Реализация стека массивом страдает недостатком, обычным для массива, — опасностью его переполнения. От этого недостатка свободна реализация стека указателями, представленная в листинге 14.3.

Каждый элемент стека представлен шаблоном класса `Elem`, содержащим указатель `v` на данные и указатель `next` на нижележащий элемент стека. Элементы стека произвольно располагаются в памяти, они связаны только указателями `next`, как показано на рис. 14.3.

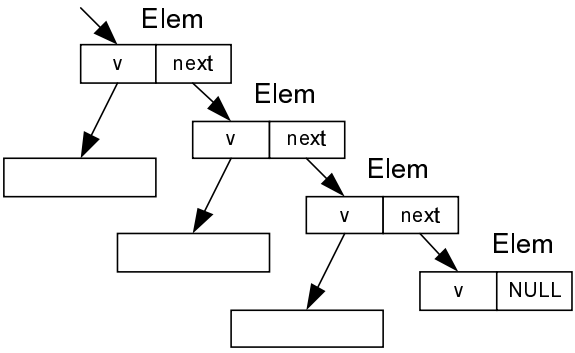


Рис. 14.3. Стек, основанный на указателях

Оперативная память для элементов стека выделяется динамически, по мере заполнения стека. Этим занимается метод `push()`. Схема работы метода показана на рис. 14.4.

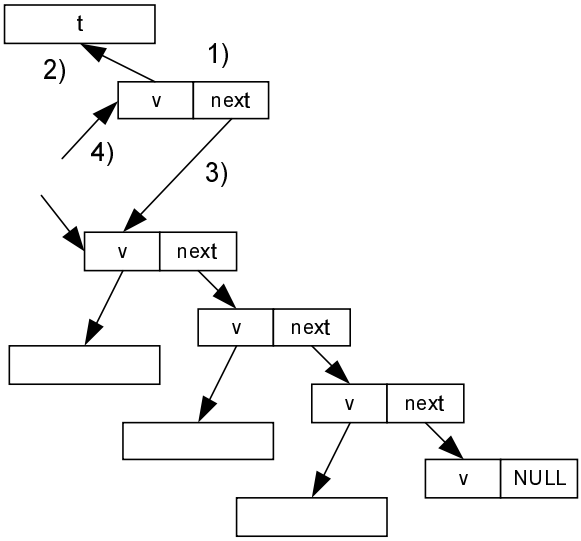


Рис. 14.4. Схема метода `push()`

Метод `pop()`, снимая элемент со стека, освобождает память, выделенную для него.

Листинг 14.3. Стек, основанный на указателях

```
#include <stdio.h>
#include <stdlib.h>

template <class T> class Stack;

template<class T> class Elem{

   friend class Stack<T>;

   T *v;
   Elem<T> *next;
public:
   Elem<T>(){
```

```
      v = 0;
      next = 0;
   }
   Elem<T>(T *t){
      v = t;
      next = 0;
   }
   Elem<T>(T *t, Elem *n){
      v = t;
      next = n;
   }
};

template <class T> class Stack{
   Elem<T> *top;  // Указатель на верхний элемент.

public:

   Stack(){

      top = new Elem<T>;
      if (!top){
        fprintf(stderr, "Нехватка памяти.\n");
        exit(-1);
      }
   }

   ~Stack(){

      Elem<T> *t = top;
      delete top;

      while (t){
         Elem<T> *tmp = t->next;
         delete t;
         t = tmp;
      }
   }
```

```
   bool empty(){
      return top == 0;
   }

   void push(T *t){

      Elem<T> *pe = new Elem<T>(t, top);
      top = pe;
   }

   T *pop(){

      if (empty()){
        fprintf(stderr, "Стек пуст.\n");
        return 0;
      }
      T *t = top->v;
      Elem<T> *tmp = top;
      top = top->next;
      delete tmp;
      return t;
   }

   T *read(){

      if (empty()){
        fprintf(stderr, "Стек пуст.\n");
        return 0;
      }
      return top->v;
   }
};

main(){

   Stack<String> st1;
   Stack<double> st2;
   Stack<int> st3;
```

```
   st1.push("Первый элемент.");
   st1.push("Второй элемент.");

   printf("%s\n", *st1.pop());
   printf("%s\n", *st1.read());

   double x = -27.05;
   st2.push(&x);

   int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   for (int k = 0; k < 10; k++) st3.push(a + k);

   for (int k = 0; k < 10; k++)
         printf("%d\n", *st3.pop());
}
```

Использование стека не зависит от его реализации. Функция `calc()`, показанная в листинге 14.4, реализует простейший стековый калькулятор. Он одинаково работает с любым стеком.

Листинг 14.4. Простейший стековый калькулятор

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void calc(){

   Stack<double> c;
   double a, b;
   char ch[80];

   do{
      gets(ch);
      switch(ch[0]){
         case '+': a = *c.pop(); b = *c.pop(); a+=b; c.push(&a); break;
         case '-': a = *c.pop(); b = *c.pop(); a-=b; c.push(&a); break;
         case '*': a = *c.pop(); b = *c.pop(); a*=b; c.push(&a); break;
```

```
        case '/': a = *c.pop(); b = *c.pop();
            if (abs(b) > 1e-8){ a/=b; c.push(&a);} break;
        default: a = atof(ch); c.push(&a);
      }
   }while(ch[0] != 'q');
}
```

Упражнения

1. Организуйте с помощью стека проверку парности скобок в арифметическом выражении.
2. Напишите функцию `reverse()`, меняющую порядок символов в заданной строке на обратный, с помощью стека.
3. Напишите функцию `palindrome()`, проверяющую, является ли заданная строка палиндромом.

Дек

Дек (deque — double ended queue) — это "стек с двумя концами". В дек можно добавлять и удалять элементы с обоих его концов. Для этого в деке определяются методы `pushBack()`, `popBack()`, `pushFront()`, `popFront()`.

Дек можно реализовать массивом. В такой реализации заполнение массива надо начинать с середины, как показано на рис. 14.5. Методы `pushBack()` и `popBack()` работают с элементом массива `v[back]`, а методы `pushFront()` и `popFront()` — с элементом `v[front]`.

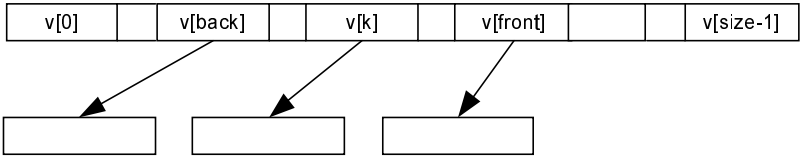


Рис. 14.5. Дек, реализованный массивом

При этом надо тщательно следить за тем, чтобы индексы `front` и `back` не выходили за границы массива и чтобы индекс `front` был не меньше индекса `back`.

Реализацию дека можно сделать и указателями. Для этого каждый элемент дека надо снабдить двумя указателями `next` и `prec`, направленными на следующий и на предыдущий элемент дека. На концы дека тоже направлены два указателя `front` и `back`. Методы `pushBack()` и `popBack()` работают с указателями `front` и `next`, а методы `pushFront()` и `popFront()` — с указате-

лями `back` и `prec`. Получаются как бы два стека, направленные в разные стороны. Схема такой реализации показана на рис. 14.6.

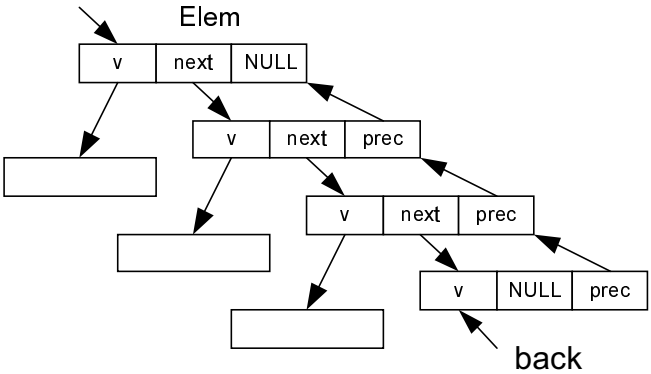


Рис. 14.6. Дек, реализованный указателями

Упражнения

4. Сделайте реализацию дека массивом, взяв за основу листинг 14.2.
5. Сделайте реализацию дека указателями, взяв за основу листинг 14.3.
6. Напишите логическую функцию `isPalindrome()`, проверяющую, является ли заданная строка палиндромом. В палиндроме символы, равноотстоящие от его концов, совпадают. Его можно одинаково читать слева направо или справа налево. Классический пример палиндрома — строка Фета: "А роза упала на лапу Азора".

Очередь

Очередь (queue) — хорошо всем известная структура данных. Каждому из нас не раз приходилось говорить: "Кто последний? Я за вами". Элементы добавляются в конец очереди, а удаляются из начала очереди. Такая структура часто описывается выражением: "Первым пришел — первым ушел" (FIFO, First In — First Out). Метод, ставящий элемент в очередь, обычно называется `enqueue()`, а метод, удаляющий элемент из очереди, называется `dequeue()`.

Переходя к реализации очереди, можно сказать, что очередь — это дек, у которого элементы не добавляются в конец и не удаляются из начала. Поэтому можно унаследовать класс `Queue` от класса `Deque`, реализовав метод `enqueue()` вызовом метода `pushBack()`, а метод `dequeue()` — вызовом метода `popFront()`.

```
template <class T> class Queue: Deque<T>{
public:
   void enqueue(T *t){
       pushBack(t);
   }
   T *dequeue(){
       return popFront();
   }
};
```

Наследование сделано закрытым, чтобы работа с очередью велась только методами класса `Queue`, а не класса `Deque`.

Упражнения

7. Напишите реализацию очереди массивом.
8. Напишите реализацию очереди указателями.

Кольцо

Реализация очереди массивом страдает существенным недостатком. После ряда добавлений и удалений элементы очереди смещаются к концу массива. Через некоторое время начало очереди упрется в конец массива. Поступит сигнал о переполнении очереди, хотя очередь может быть невелика, первые элементы массива могут быть свободны, они давно удалены из очереди. Положение можно исправить, перенеся индекс начала очереди в начало массива:

```
if (front == size - 1 && back > 0) front = 0;
else{
   fprintf("Переполнение очереди.\n");
   return;
}
```

Таким образом, очередь замыкается в *кольцо* (ring). При этом надо следить за тем, чтобы очередь не "съела" сама себя — индекс начала кольца `front` не оказался больше индекса `back` его конца.

Как правило, при реализации очереди массивом организуется именно кольцо, а не простая очередь. Термин "кольцо" в таком случае даже не упоминается, сразу предполагается, что индекс начала очереди вычисляется по модулю размера массива:

```
front = (front++) % size;
```

Упражнение

9. Напишите реализацию кольца массивом.

Линейный список

Линейный список (list) — это структура данных, предназначенная для быстрой вставки элементов в определенное место. Такая структура необходима, например, для пополнения упорядоченной информации. Основные операции со списком — это вставить элемент в список, удалить элемент из списка, просмотреть список в поисках какого-либо элемента.

Массив совершенно не подходит для реализации линейного списка. Для вставки элемента в середину массива его придется "раздвинуть", переместив элементы ближе к концу массива, что займет много времени. Кроме того, после нескольких вставок массив переполнится. Удалять элементы из массива тоже неудобно. В массиве не может быть "дырок". Его элементы придется перемещать, чтобы скрыть освободившееся после удаления место.

Итак, для реализации линейного списка удобнее указатели. Мы реализуем список в том виде, который показан на рис. 14.3, только указатель на начало списка назовем `first`, а не `top`.

Основная операция — вставка элемента в список — выполняется методом `insert()`. Его работа показана на рис 14.7.

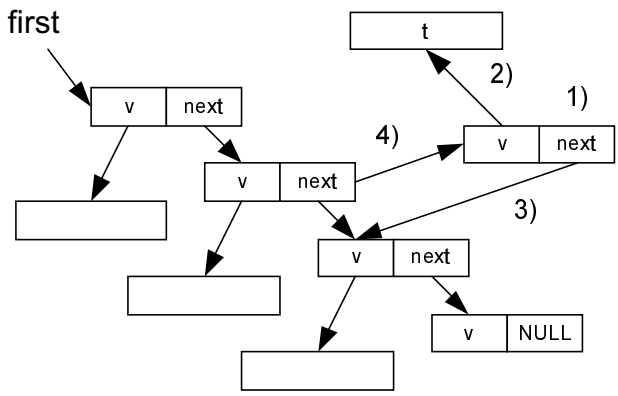


Рис. 14.7. Вставка элемента в линейный список

Методу передается указатель на данные и указатель на элемент, за которым вставляется новый элемент. В начале работы метода выделяется память под новый элемент списка и определяется указатель `v` на данные. Указатель на следующий элемент `next` нового элемента становится равным указателю

next текущего элемента. После этого указатель next меняется. Он указывает на новый элемент, тем самым включая его в список.

Удаление элемента из списка выполняется еще легче. Особенно легко удалить элемент, следующий за текущим элементом. Для этого достаточно "перекинуть" указатель next текущего элемента "через" следующий за ним элемент. Это делает метод removeNext(), которому передается указатель на текущий элемент.

Сложнее удалить текущий элемент, ведь указатель надо "перекидывать" через него, а предыдущий элемент неизвестен. В методе remove() сделан фокус, заключающийся в том, что текущий элемент не удаляется, а его содержимое заменяется содержимым следующего за ним элемента. После этого удаляется следующий элемент, содержащий ту же самую информацию.

Листинг 14.5. Линейный список

```
#include <stdio.h>
#include <stdlib.h>

template<class T> class List;

template<class T> class Elem{
   friend class List<T>;
   T *v;
   Elem<T> *next;
public:
   Elem<T>(){
      v = 0;
      next = 0;
   }
   Elem<T>(T *t){
      v = t;
      next = 0;
   }
   Elem<T>(T *t, Elem *n){
      v = t;
      next = n;
   }
};

template <class T> class List{
```

```
   Elem<T> *first;  // Указатель на начало списка.

public:

   List<T>(){

      first = 0;
   }

   ~List<T>(){

      if (empty()) return;

      Elem<T> *t = first;

      while (t){
         Elem<T> *tmp = t->next;
         delete t;
         t = tmp;
      }
   }

   Elem<T> *getFirst(){

       return first;
   }

   Elem<T> *getNext(Elem<T> *e){

       return e->next;
   }

   T *getData(Elem<T> *e){

       return e->v;
   }

   bool empty(){
```

```
   return first == 0;
}

void add(T *t){

   if (empty()){
      Elem<T> *pe = new Elem<T>(t);
      first = pe;
   }else{
      Elem<T> *pe = new Elem<T>(t, first);
      if (!pe){
        fprintf(stderr, "Нехватка памяти.\n");
        exit(-1);
      }
      first = pe;
   }
}

void insert(T *t, Elem<T> *e){
   Elem<T> *tmp = new Elem<T>(t);
   if (!tmp){
     fprintf(stderr, "Нехватка памяти.\n");
     exit(-1);
   }
   tmp->next = e->next;
   e->next = tmp;
}

void removeNext(Elem<T> *e){

   if (empty()) return;

   Elem<T> *tmp = e->next;
   e->next = e->next->next;
   delete tmp;
}

void remove(Elem<T> *e){
```

```
      if (empty()) return;

      Elem<T> *tmp = e->next;
      e->v = e->next->v;
      e->next = e->next->next;
      delete tmp;
   }
};

main(){

   List<String> lst1;
   List<double> lst2;
   List<int> lst3;

   lst1.add("Первый элемент.");
   lst1.add("Второй элемент.");
   double x = -27.05;
   lst2.add(&x);

   int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   for (int k = 0; k < 10; k++) lst3.add(a + k);

   for (Elem<int> *e = lst3.getFirst(); e != NULL; e = lst3.getNext(e))
         printf("%d\n", *lst3.getData(e));
}
```

Двунаправленный список

Линейный список можно просматривать только в одном направлении: от начала к концу. От текущего элемента можно перейти только к следующему элементу, но никак нельзя вернуться к предыдущему элементу и пройти в обратном направлении. Поэтому список чаще делают двунаправленным, добавляя к каждому элементу еще один указатель, показывающий на предыдущий элемент.

Такую схему, показанную на рис 14.6, мы уже использовали при реализации дека. Поэтому можно реализовать двунаправленный список, расширив класс `Deque` методами вставки и удаления. Сделайте это в качестве упражнения.

Упражнение

10. Реализуйте двунаправленный список.

Бинарное дерево

Список удобен для вставки и удаления элементов. Он решает задачу сохранения упорядоченности в отсортированном списке, но не решает задачу поиска, о которой мы говорили (*см. гл. 4*), поскольку допускает только линейный поиск. Для бинарного поиска нужна другая структура данных. Чаще всего для этого применяются *деревья* (tree).

Дерево состоит из *корня* (root), в котором есть указатели на *узлы* (nodes) дерева. Каждый узел является корнем своего *поддерева* (subtree). В каждом узле есть указатели на узлы следующего уровня. В конце этой рекурсии расположены *листья* (leafs) дерева, содержащие пустые указатели. Говорят, что корень дерева расположен на нулевом *уровне*. Корни его поддеревьев находятся на первом уровне и т. д. Наибольший уровень называется *высотой* дерева.

Если в каждом узле дерева есть только два указателя, пустые или нет, то дерево называется *бинарным* (binary tree). Структура бинарного дерева показана на рис 14.8. В компьютерных науках принято рисовать дерево перевернутым, поэтому на рисунке его корень находится вверху, а листья — внизу.

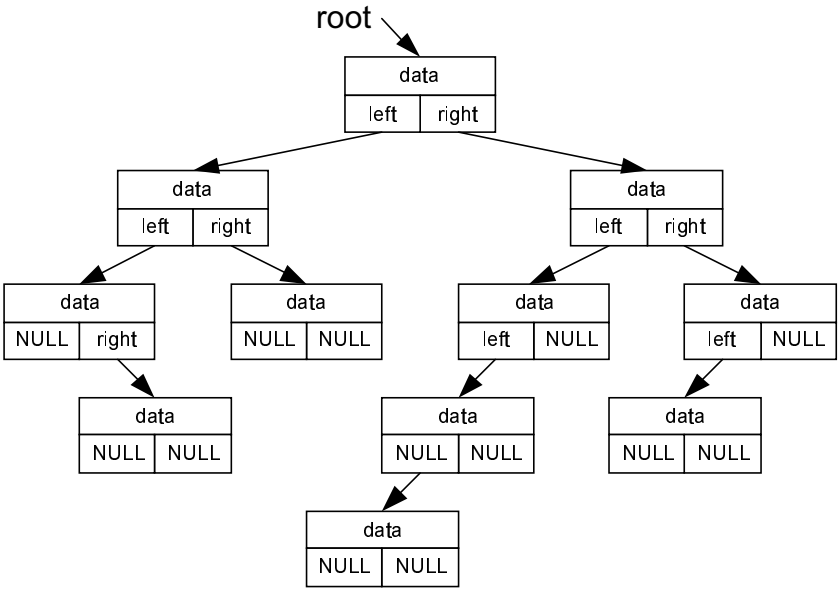


Рис. 14.8. Бинарное дерево

Для быстрого поиска информацию в дереве удобно расположить так, чтобы на каждом уровне в левом поддереве все значения были меньше, чем в правом поддереве. Такое расположение данных показано на рис. 14.9.

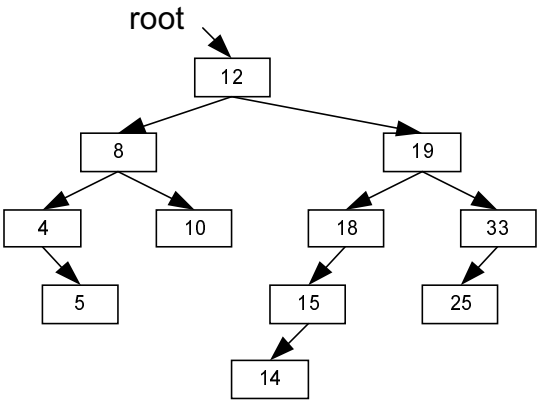


Рис. 14.9. Бинарное дерево поиска

Бинарное дерево, хранящее упорядоченную таким образом информацию, называется *бинарным деревом поиска*. Оно удобно и для сортировки. Просмотрите бинарное дерево поиска слева направо, не учитывая уровни элементов, и вы увидите, что элементы содержат возрастающие значения.

В листинге 14.6 мы построим бинарное дерево поиска. Для этого создадим класс `Tree`. Его узлы и листья будут объектами класса `Node`. Чтобы обеспечить прямой доступ к элементам узла, класс `Node` объявлен дружественным классу `Tree`. Класс `Node` содержит указатель `data` на данные и указатели `left` и `right` на левое и правое поддерево. Конструктор класса просто обнуляет указатели.

Поскольку определение дерева рекурсивно, для его построения написана рекурсивная функция `insertNode()`. Она сравнивает значение узла со значением нового элемента и переходит к левому или правому поддереву. Рекурсия заканчивается, когда указатель в выбранном узле пуст. Этот указатель направляется на новый элемент, включая его тем самым в дерево. Тот же алгоритм применен в методе `search()` для поиска элемента по его ключу `data`.

Рекурсия применяется и деструктором. Он обходит все узлы и листья дерева, удаляя узел, когда он становится листом.

Листинг 14.6. Бинарное дерево поиска

```
#include <stdio.h>
#include <stdlib.h>

template<class T> class Tree;

template<class T> class Node{
   friend class Tree<T>;
   T *data;
   Node<T> *left, *right;
public:
   Node<T>(){
      data = 0;
      left = 0;
      right = 0;
   }
   Node<T>(T *t){
      data = t;
      left = 0;
      right = 0;
   }
};

template <class T> class Tree{

   Node<T> *root;  // Указатель на корень дерева.

public:

   Tree<T>(){

      root = 0;
   }

   ~Tree<T>(){

      deleteNodes(root);
   }
```

```
void deleteNodes(Node<T> *n){

   if (!n) return;

   delete n;
   deleteNodes(n->left);
   deleteNodes(n->right);
}

Node<T> *getRoot(){

   return root;
}

Node<T> *getLeft(Node<T> *n){

   return n->left;
}

Node<T> *getRight(Node<T> *n){

   return n->right;
}

T *getData(Node<T> *n){

   return n->data;
}

void add(T *t){

   Node<T> *n = new Node<T>(t);
   if (!n){
     fprintf(stderr, "Нехватка памяти.\n");
     exit(-1);
   }
   insertNode(root, n);
}
```

```
   void insertNode(Node<T> *node, Node<T> *elem){

       if (node == 0) node = elem;
       else
         if (node->data < elem->data)
           insertNode(node->left, elem);
         else if (node->data > elem->data)
           insertNode(node->right, elem);
   }

   Node<T> *search(T *t){

      return searchNode(root, t);
   }

   Node<T> *searchNode(Node<T> *node, T t){

       if (node->data == t) return node;
       else
         if (node->data < t)
           searchNode(node->left, t);
         else
           searchNode(node->right, t);
   }
};

main(){

   Tree<String> t1;
   Tree<double> t2;
   Tree<int> t3;

   t1.add("First element.");
   t1.add("Second element.");
   double x = -27.05;
   t2.add(&x);

   int a[] = {5, 7, 4, 8, 2, 10, 1, 11, 3, 12};
```

```
    for (int k = 0; k < 10; k++) t3.add(a + k);

    for (Node<int> *n = t3.getRoot(); n != NULL; n = t3.getRight(n))
          printf("%d\n", *t3.getData(n));
}
```

Записанный в листинге 14.6 алгоритм построения бинарного дерева поиска сильно зависит от порядка поступления исходных данных. Если исходные данные уже упорядочены, то бинарное дерево выродится в линейный список и бинарный поиск превратится в линейный поиск. Поскольку невозможно заранее предугадать порядок поступления элементов, сначала строят такое дерево, какое получится, а затем преобразуют его в *сбалансированное* дерево.

Бинарное дерево называется сбалансированным, если число уровней каждого его левого поддерева отличается от числа уровней соответствующего правого поддерева не более чем на единицу. Перед балансировкой дерево надо осмотреть и определить, какие его поддеревья не сбалансированы. Затем перестроить каждое несбалансированное дерево, сделав его корнем другой элемент.

Например, балансировку бинарного дерева, изображенного на рис. 14.9, можно улучшить, сделав корнем правого поддерева не узел, содержащий число 19, а узел с числом 18.

Существуют детально проработанные алгоритмы балансировки деревьев. Есть алгоритмы вставки и удаления элементов из дерева, сохраняющие балансировку. Их рассмотрение выходит за рамки нашей книги.

Обход бинарного дерева

Есть еще одна важная задача, кроме построения дерева, вставки и удаления элементов, — задача обхода дерева с целью выполнить какую-то операцию со всеми элементами дерева. Такой операцией может быть поиск какого-нибудь элемента с заданными свойствами.

Обход бинарного дерева начинается с его корня (К). Затем можно перейти к его левому (Л) или правому (П) поддереву. После прохода по всем уровням дерева мы попадаем на его лист, из которого надо вернуться к узлам дерева, чтобы продолжить обход. При этом мы оказываемся в произвольном узле три раза. Первый раз, когда спускаемся от корня к узлу, второй раз, когда поднимаемся к узлу от левого поддерева, и третий раз, когда поднимаемся к узлу от правого поддерева. Это показано на рис. 14.10.

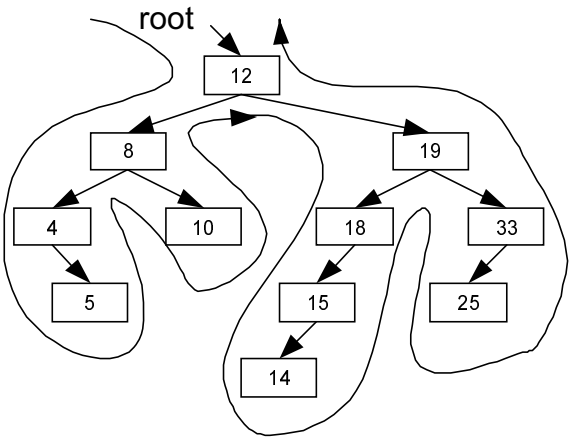


Рис. 14.10. Обход бинарного дерева

Мы можем обработать содержимое узла дерева при первом появлении в нем. Обозначим такую обработку КЛП. Мы можем обработать содержимое при втором появлении в узле, ЛКП, или при третьем появлении, ЛПК. Наконец, мы можем обходить дерево не слева направо, спускаясь каждый раз к левому поддереву, как показано на рис. 14.10, а справа налево. Получаем способы обхода КПЛ, ПКЛ, ПЛК.

Последние три способа обхода бинарного дерева совершенно симметричны первым трем способам, они не вносят ничего нового. Отметим еще, что упорядоченный обход бинарного дерева поиска в порядке возрастания его элементов дает способ ЛКП, а в убывающем порядке — способ ПКЛ.

Рекурсивный метод обхода дерева по способу ЛКП `inOrder()` записывается очень просто:

```
void inOrder(Node *t){
   if (t != NULL){
      inOrder(t->left);
      process(t->data);
      inOrder(t->right);
   }
}
```

Метод `preOrder()` обхода бинарного дерева по способу КЛП отличается только порядком обращения к методам:

```
void preOrder(Node *t){
```

```
   if (t != NULL){
      process(t->data);
      preOrder(t->left);
      preOrder(t->right);
   }
}
```

Аналогично записывается метод `postOrder()` обхода бинарного дерева в порядке ЛПК:

```
void postOrder(Node *t){
   if (t != NULL){
      postOrder(t->left);
      postOrder(t->right);
      process(t->data);
   }
}
```

Упражнение

11. Занесите в класс `Tree` листинга 14.6 методы обхода бинарного дерева.

B-дерево

Бинарные деревья просты и очень хороши для хранения относительно небольшого количества данных. Если же количество информации возрастает настолько, что она уже не умещается целиком в оперативной памяти, то при каждом переходе к поддереву приходится обращаться к диску за новой порцией данных. Это сильно замедляет обработку элементов дерева, поскольку обращение к диску происходит в тысячи раз медленнее, чем обращение к ячейке оперативной памяти.

В то же время с увеличением объема бинарного дерева сильно возрастает количество его уровней, увеличивается высота дерева. Это увеличивает количество поддеревьев и еще больше замедляет обработку элементов дерева.

Очевидный выход заключается в том, чтобы увеличить количество поддеревьев в каждом узле дерева, т. е. перейти от бинарных деревьев к тернарным и вообще к n-арным деревьям, в которых у каждого поддерева может быть до n поддеревьев. Это уменьшит высоту дерева, но зато увеличит количество поддеревьев, находящихся на одном уровне. Общий выигрыш окажется ничтожным.

В структуре *B-дерева* (Би-дерево) применен комбинированный подход. Узлы B-дерева, называемые *страницами* (pages), не содержат данных. Они содержат

только указатели, называемые *индексами* (indexes). Число индексов меняется от страницы к странице. Наибольшее число индексов называется *порядком* B-дерева. Для лучшей балансировки введено условие на количество индексов — для всякой узловой страницы оно должно быть не меньше половины порядка. Новая страница создается только после того, как число индексов, направленных на данные в листе B-дерева, достигнет его порядка.

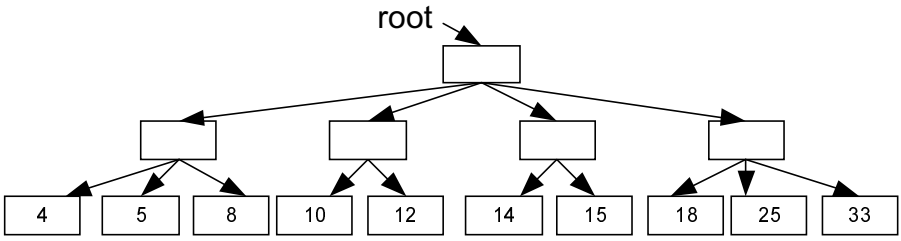


Рис. 14.11. B-дерево порядка 4

На рис. 14.11 показано B-дерево порядка 4, содержащее целые числа. При его построении сначала определяются индексы страницы, на которую показывает указатель `root`. Они направляются на поступающие числа. Пусть это будут числа 10, 12, 14, 18. При поступлении пятого числа, пусть им будет число 8, страница разбивается на две страницы. Индекс числа 10 перенаправляется на одну из новых страниц. На ней сразу же создаются индексы чисел 10, 12 и индекс нового числа 8. На второй странице остаются индексы чисел 14 и 18. Это состояние B-дерева показано на рис. 14.12.

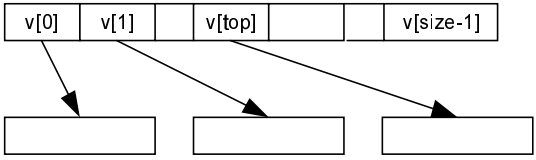


Рис. 14.12. Разбиение страницы B-дерева

При поступлении следующего числа 15 его индекс просто записывается на вторую страницу. Теперь она содержит индексы чисел 14, 15, 18. После поступления числа 25 вторая страница полностью заполнена, она содержит числа 14, 15, 18, 25.

Пусть далее поступает число 33. Вторая страница разбивается на две страницы, присоединяемые к корневой странице. Индекс числа 14 перебрасывается на одну из новых страниц, индекс числа 18 — на другую и процесс построения B-дерева продолжается.

Упражнения

12. Опишите процесс внесения в B-дерево, изображенного на рис. 14.11, числа 7.
13. Опишите процесс добавления к B-дереву рис. 14.12 чисел 20 и 23.
14. Проследите за тем, как изменится B-дерево рис. 14.12 при последовательном удалении чисел 12, 14, 10. Помните, что на каждой узловой странице должно быть не менее двух индексов.

Контейнеры и итераторы

Рассмотренные в этой главе структуры данных: стек, дек, очередь, списки, деревья, предназначены для хранения данных любых типов. Поэтому они носят общее название *контейнеров* (containers). Конечно, все типы охватить невозможно, у каждого контейнера есть свои ограничения, но создатели контейнеров стремятся сделать так, чтобы в них хранилось как можно больше типов данных.

Вторая особенность контейнеров заключается в том, что они не ограничивают количество хранящихся в них элементов. Емкость контейнера зависит только от наличия доступной оперативной памяти.

Контейнер — это не только хранилище, но и методы обработки данных. Именно набор методов, а не способ реализации, определяет тот или иной контейнер. Например, стек можно реализовать и массивом, и линейным списком, и даже деревом. Просто методы стека должны давать доступ только к одному, верхнему элементу.

У каждого контейнера, даже у стека, должны быть средства обхода контейнера, просмотра его элементов. Как мы видели, это легко сделать для линейного списка, но не так просто для дерева, особенно для B-дерева. Методы обхода контейнера сильно зависят от его реализации. Вначале для каждой структуры данных были разработаны свои методы просмотра элементов, со своими названиями и аргументами. Это противоречило основному принципу контейнера — дать общее средство работы с данными разных типов.

В конце концов, объектно-ориентированное программирование выработало единый подход к решению задачи обхода контейнера и просмотра его элементов. Методы обхода контейнера описали в отдельном абстрактном классе. Как правило, этот класс называется `iterator`. Этот класс обязательно содержит метод `next()` или перегруженную операцию `n++` перехода к следующему элементу контейнера, а также метод `hasMore()`, позволяющий отследить окончание просмотра контейнера или просто перегруженную операцию ==, позволяющую сравнить указатель на текущий элемент с нулевым значением, означающим конец контейнера.

Конечно, класс `iterator` может содержать и другие методы, общие для всех контейнеров, скажем, метод сортировки контейнера. Итератор может выдавать первый и последний элемент контейнера, сравнивать два элемента контейнера и выполнять другие операции.

Каждый контейнер реализует методы класса `iterator`, создает объект этого класса и предоставляет указатель на этот объект. Таким образом, если у нас есть экземпляр `v` контейнера `Vector`, то мы можем просто написать:

```
Vector<int>::iterator it = v.iterator;
while (it.hasMore())
   f(it.next());
```

Или, если в итераторе перегружены операции:

```
Vector<int>::iterator it = v.iterator;
while (it != NULL){
   f(it);
   it++;
}
```

В этих примерах мы предположили, что итератор `it` сразу же установлен на начало контейнера, хотя это не видно из его определения. Кроме того, мы предположили, что у итератора есть пустое значение `NULL`. Это верно не всегда. Очень часто контейнер предоставляет не просто итератор, а итераторы, направленные на его начало и за его конец методами `begin()` и `end()`. С их использованием предыдущий цикл будет выглядеть понятнее:

```
Vector<int>::iterator it = v.begin();
while (it != v.end(){
   f(it);
   it++;
}
```

В таких итераторах нет понятия пустого значения.

Для удобства просмотра контейнера в обратном направлении в нем часто создается *обратный итератор* (reverse iterator) — объект класса `reverse_iterator`. Его начальное значение устанавливается на последний элемент контейнера методом `rbegin()`, а конечное значение — перед первым элементом контейнера методом `rend()`. Увеличение обратного итератора операцией инкремента означает переход к предыдущему элементу контейнера.

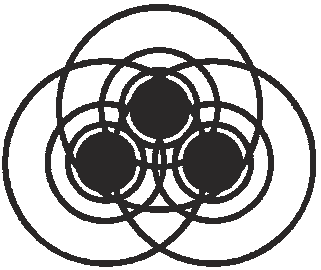
Многие разработчики контейнеров идут дальше и создают *двунаправленный* (bidirectional) итератор, перемещающийся в обоих направлениях. В нем реализованы обе операции: и инкремент, и декремент.

Наконец, иногда создается *итератор прямого доступа* (random-access iterator), в котором, как в обычных указателях, реализована вся адресная арифметика.

Использование итераторов оказалось удобным и полезным. Их стали применять для ввода и вывода элементов из контейнеров, поиска определенного элемента в контейнере и для других целей. На основе контейнеров и итераторов возникла целая *стандартная библиотека шаблонов* STL (Standard Template Library), входящая в стандарт языка C++.

Контрольные вопросы

1. Что называется структурой данных?
2. Можно ли создать структуру данных на диске?
3. Каковы преимущества и недостатки массивов для хранения информации?
4. Чем вызвана необходимость создания динамических структур данных?
5. Приведите примеры применения стека, дека, очереди.
6. Нужно ли знать количество элементов стека?
7. В каких задачах удобен линейный список?
8. Для решения каких задач строятся бинарные деревья?
9. В чем преимущества B-деревьев перед бинарными деревьями?



Глава 15

Работа со строками

Очень большое место в обработке информации занимает работа с текстами. В самом простом виде "плоский" текст[1] записывается просто как последовательность символов. Каждый символ занимает один, два, три или четыре байта в зависимости от выбранной кодировки. Независимо от размера ячейки памяти, выделяемой для каждого символа, можно считать, что в программу, работающую с "плоским" текстом, поступает последовательность символов.

Во всяком языке программирования должны быть средства работы с текстом. Как мы уже видели, язык C очень близко подходит к исходной структуре "плоского" текста. В нем текст записывается как массив символов типа `char`, завершающийся нуль-символом. Каждый такой массив в языке C называется *строкой* (string) символов, несмотря на то, что в нем может храниться множество текстовых строк.

Функции языка C работают непосредственно с отдельными символами. Стандарт языка предоставляет библиотеку функций, обрабатывающих строки, описанную в заголовочном файле string.h. Это небольшая библиотека, входящие в нее функции выполняют только самые необходимые операции со строками. Язык C++ как наследник и расширение языка C позволяет использовать все эти средства. Они описаны в заголовочном файле cstring (пишется без расширения h!), но можно использовать и заголовочный файл string.h.

В объектно-ориентированных языках программирования естественно оформить строку символов в виде класса, включив в него методы обработки строк символов. Сама строка символов, ее исходный текст записывается как поле класса, обычно закрытое, `private`.

---

[1] Текст без форматирующей информации.

Создатели языка C++ не могли обойти вниманием работу с текстами и включили все необходимые для этого средства в стандартную библиотеку шаблонов STL. Они описаны в заголовочном файле string, который надо включать во все программы, использующие эти средства. Обратите внимание на то, что у заголовочных файлов библиотеки STL нет расширения h.

Напомним еще, что все имена библиотеки STL принадлежат пространству имен `std`, поэтому их надо предварять префиксом `std::` или записывать в начале файла директиву (*см. гл. 11*):

```
using namespace std;
```

Типы отдельных символов

Средства работы со строками библиотеки STL могут использовать обычные для языка C/C++ типы `char`, `unsigned char` и `signed char`. Это типы символов, записанных в однобайтовой кодировке со знаком или без знака. Для работы с типом `char` можно применять функции из стандартных библиотек языка C, описанных в заголовочных файлах cstring, cstdlib и cctype. Можно записывать старые названия заголовочных файлов: string.h, stdlib.h и ctype.h.

В связи с повсеместным распространением кодировки Unicode в языке C++ появился "широкий" тип `wchar_t` (wide char type). Тип `wchar_t` — это не обязательно Unicode, он может хранить и другую кодировку. Данные типа `wchar_t` могут занимать два или более байта оперативной памяти. Для записи двух- и четырехбайтовых констант этого типа в языке C++ предусмотрены шестнадцатеричные обозначения вида `\u2C3A` и `\U0034FFDD`. Функции, работающие с типом `wchar_t`, описаны в заголовочном файле cwtype. Его старое имя wtype.h.

Разработчики библиотеки STL подстраховались на случай появления каких-нибудь новых кодировок и унифицировали работу с символами. Они определили шаблон *свойств символов* `char_traits<T>`. Вот его прототип:

```
template<class Ch> struct char_traits{};
```

В библиотеке STL есть несколько специализаций этого шаблона. Тип `char` входит в специализацию `char_traits<char>` шаблона под именем `char_type`. Точнее говоря, в классе `char_traits<char>` есть запись:

```
typedef  char  char_type;
```

Введен и тип `int_type` для численного кода символов:

```
typedef  int  int_type;
```

В шаблоне описаны методы преобразования кода в символ и обратно:

```
static char_type to_char_type(const int_type&);
static int_type to_int_type(const char_type&);
```

Кроме того, специализация шаблона класса `char_traits<char>` перегружает операции сравнения `==`, `!=`, `<`, `>`, `<=`, `>=` на символы типа `char_type`, упорядочивая, таким образом, набор символов. Это может быть алфавитный порядок, а может быть и какой-нибудь другой. Все зависит от реализации операций сравнения символов.

Тип `wchar_t` входит в специализацию `char_traits<wchar_t>` шаблона под тем же именем `char_type`. В классе `char_traits<wchar_t>` есть запись:

```
typedef  wchar_t  char_type;
```

Таким же образом вводится тип `int_type`, написаны такие же методы преобразования и так же перегружаются операции сравнения. В итоге все методы работы со строками всех классов библиотеки STL рассматривают отдельный символ независимо от его кодировки как экземпляр шаблона `char_traits<T>` и используют методы шаблона `char_traits<T>` для работы с ним.

Шаблон `char_traits<T>` описывает методы сравнения символов, что полезно для сортировки, метод `find()` поиска заданного символа в строке, методы копирования `copy()` и замены `move()` символов и массивов символов. Кроме того, в шаблоне описан метод `compare()` сравнения двух массивов символов, аналог функции `strcmp()` из стандартной библиотеки функций языка C.

Шаблон класса *basic_string*

Поскольку методы обработки строк зависят от реализации шаблона `char_traits<T>`, для работы с ними создан шаблон класса `basic_string<Ch, Tr, A>`. Вот его заголовок:

```
template<class Ch,
   class Tr = char_traits<Ch>,
    class A = allocator<Ch> >
class basic_string{};
```

Как видно из приведенного заголовка, у шаблона три параметра. Параметр `Ch` участвует в реализации шаблона `char_traits<Ch>`, параметр `Tr` — это тип отдельного символа строки, по умолчанию равный `char_traits<Ch>`. Третий параметр шаблона `A` — это распределитель памяти (allocator) — еще одно обобщение стандартной библиотеки шаблонов. Распределитель памяти — это шаблон класса, в котором собраны методы выделения и освобождения памяти для типа, указанного параметром шаблона:

```
template<class T> class allocator{};
```

В шаблоне класса `basic_string<Ch, Tr, A>` в качестве распределителя памяти по умолчанию предлагается стандартная реализация шаблона `allocator<Ch>`.

Она описана в заголовочном файле memory. Обратите внимание на синтаксис заголовка шаблона. Конструкция `template` закрывается двумя угловыми скобками:

```
... class A = allocator<Ch> >
```

Для того чтобы компилятор не принял их за операцию `>>`, между скобками надо оставлять пробел.

Итак, при реализации шаблона класса `basic_string<Ch, Tr, A>` надо, как минимум, задать тип символа. В стандартной библиотеке шаблонов так сделано для двух наиболее часто используемых реализаций с типами символов `char` и `wchar_t`. Эти реализации получили имена `string` и `wstring`. Имена введены следующими операторами `typedef`, записанными в заголовочном файле string:

```
typedef  basic_string<char>  string;
typedef  basic_string<wchar_t>  wstring;
```

В шаблоне класса `basic_string<Ch, Tr, A>` перегружены операции сравнения строк `==`, `!=`, `<`, `>`, `<=`, `>=`, операции присваивания `=` и `+=`, операция взятия отдельного символа `[]`. Кроме того, перегруженная операция `+` сцепляет две строки, давая в результате строку, содержащую все символы первой строки, за которыми без всяких разделителей идут все символы второй строки.

Шаблон класса `basic_string<Ch, Tr, A>` описывает множество методов, значительно облегчающих обработку текста. Мы познакомимся с ними в следующих разделах этой главы.

Классы *string* и *wstring*

Как видно из предыдущего раздела, если надо стандартно работать с текстом, записанным какой-либо однобайтовой кодировкой с типом символов `char`, то можно просто занести текст в объект класса `string` и пользоваться методами этого класса. Если же текст записан в Unicode или другой кодировке символов типа `wchar_t`, то удобно пользоваться объектом класса `wstring`.

Мы будем рассматривать класс `string`, но все написанное ниже относится и к классу `wstring`. Точнее говоря, все написанное ниже принадлежит шаблону `basic_string<Ch, Tr, A>`.

Как создать строку

Перед работой со строкой ее следует создать. Это можно сделать разными способами. Самый простой способ создать строку — это создать объект типа `string` с помощью перегруженной операции присваивания:

```
string s1 = "Это строка.";
```

Если константа длинная, то можно записать ее в нескольких строках текстового редактора, связывая их операцией сцепления, иначе в строку попадут лишние пробелы и символы перевода строки:

```
string s2 = "Это длинная строка, " +
    "записанная в двух строках исходного текста";
```

Самый правильный способ создать объект с точки зрения объектно-ориентированного программирования — это воспользоваться его конструктором. Класс `string` предоставляет шесть конструкторов:

- `string()` — создается объект с пустой строкой;
- `string(const char *s)` — объект создается из строки `s` символов, записанной в стиле языка C;
- `string(size_type n, char c)` — создается строка длины `n`, в которой `n` раз повторяется символ `c`;
- `string(const string *str, size_type pos = 0, size_type n = len)` — из одного объекта `str` создается другой, содержащий подстроку строки `str`, начинающуюся с индекса `pos` и содержащую `n` символов или остаток строки `str`, если `n` велико;
- `string(const char *s, size_type n)` — из строки символов в стиле языка C `s` выделяется `n` начальных символов. Если `n` превышает длину строки `s`, то будут копироваться следующие байты памяти;
- `template<class Iter> string(Iter begin, Iter end)` — строка создается из элементов некоторой коллекции, итератор `Iter` которой служит параметром шаблона.

В листинге 15.1 приведены примеры создания строк.

Листинг 15.1. Создание строк

```
#include<string>

using namespace std;

main(){
   string s1; // Пустая строка.
   string s2 = "Строка символов.";
   string s3 = s2; // Копия строки.
   string s4("Еще одна строка.");
   string s5(50, '?'); // 50 вопросительных знаков.
   string s6(s4, 4, 4); // Слово "одна".
   string("Третья строка.", 5); // слово "Треть".
```

```
   string(v.begin(), v.end()); // Копия некой коллекции v.

   char s[] = "Строка в стиле языка C.";
   string s7(s);
   string s8(s + 15, 4); // Слово "язык".
   String s9(s + 15, s + 19); // Тоже слово "язык".
}
```

Сцепление строк

Со строками можно производить операцию *сцепления строк* (concatenation), обозначаемую знаком "+" (плюс). Эта операция создает новую строку, просто составленную из состыкованных первой и второй строк, как показано в начале этого раздела. Ее можно применять и к константам, и к переменным, и даже к отдельным символам. Например:

```
string attention = "Внимание: ";
string s = attention + "неизвестный символ";
s = '&';
```

Вторая операция — присваивание `+=` — применяется к переменной, стоящей в левой части, добавляя к ней строку или символ, стоящие в правой части операции присваивания:

```
attention += s;
s += '&';
```

Действия со строками

В шаблоне класса `basic_string<Ch, Tr, A>`, а следовательно, и в классе `string`, есть множество методов для работы со строками. Посмотрим, что они позволяют делать с текстом, хранящимся в строке.

Как узнать длину строки

Для того чтобы узнать длину строки, т. е. количество символов в ней, надо обратиться к методу `size()`:

```
string s = "Write once, run anywhere.";
size_type len = s.size();
```

У метода `size()` есть синоним — метод `length()`. Он определен следующим образом:

```
size_type length() const{
   return size();
}
```

Метод `capacity()` возвращает объем выделенной для строки оперативной памяти. Как правило, он совпадает с длиной строки. Логический метод `empty()` возвращает значение `true`, если строка пуста. Он определен очень просто:

```
bool empty() const{
   return size() == 0;
}
```

Как выбрать символы из строки

Выбрать символ с индексом `ind` (индекс первого символа строки равен нулю) можно с помощью перегруженной операции `s[ind]` или методом `at(ind)`. Например, после определения переменная `ch` будет иметь значение `t`:

```
string s = "Write once, run anywhere.";
char ch = s[3];
```

Операция `[]` не отслеживает выход индекса за границы массива. Метод `at(ind)` более надежен. Если индекс `ind` отрицателен или не меньше чем длина строки, то при применении метода `at(ind)` возникает исключительная ситуация `out_of_range`. Индекс и метод `at()` можно использовать и для изменения отдельных символов строки, например:

```
s[3] = 'A';
```

Записать все символы строки в массив символов можно методом `data()`, например:

```
const char *arr = s.data();
```

Метод `data()` не записывает в конец массива нуль-символ. Чтобы получить строку в стиле языка C, надо воспользоваться методом `c_str()`, например:

```
const char *cstr = s.c_str();
```

Если же надо скопировать в массив только часть строки, то можно воспользоваться методом `copy()`. Вот его прототип:

```
size_type copy(Ch *arr, size_type len, size_type pos = 0) const;
```

Он копирует `len` символов в массив `arr`, начиная с символа, стоящего в позиции `ind`. Например:

```
string s = "Write once, run anywhere.";
char arr[20];

s.copy(arr, 4 6);
```

Как просмотреть строку

Библиотека STL относит строки к коллекциям. Как всякая коллекция, строка создает итераторы для обхода своих элементов. Экземпляры итераторов создаются и предоставляются следующими методами:

```
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;
reverse_iterator rbegin();
reverse_iterator rend();
const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;
```

Как видно из этого списка, у строк есть простые и константные итераторы, а также итераторы, обходящие коллекцию в обратном порядке. Имена обратных итераторов начинаются с буквы "r". Методы `begin()` и `rend()` устанавливают итератор на начало строки, а методы `end()` и `rbegin()` — на конец строки. С использованием итератора можно очень просто просмотреть строку:

```
string s("Просмотрим эту строку.");

iterator it = s.begin();

while (it != s.end()){
   Ch ch = *it;
   it++;
}
```

Следующий цикл просматривает строку от конца к началу:

```
reverse_iterator rit = s.rbegin();

while (rit != s.rend()){
   Ch ch = *rit;
   rit++;
}
```

Как выбрать подстроку

Метод `substr(int begin, int n)` выделяет подстроку длиной `n` символов, начинающуюся с символа с индексом `begin`. Метод `substr(int begin)` вы-

деляет подстроку, простирающуюся от индекса begin включительно до конца строки. Например:

```
string s = "Write once, run anywhere.";
string sub1 = s.substr(6, 4);
string sub2 = s.substr(16);
```

В результате получим в строке sub1 значение "once", а в sub2 — значение "anywhere".

Как сравнить строки

Основные методы сравнения строк, которые предлагает шаблон класса basic_string<Ch, Tr, A> и, следовательно, класс string, — это пять методов compare().

Метод s.compare(const string &str) возвращает целое число типа int, отрицательное, если данная строка s лексикографически расположена раньше строки str, равное нулю, если строки совпадают, и положительное в противном случае. При этом более длинная строка считается больше строки, которая короче ее и содержит те же начальные символы.

```
string s1("Эта строка A меньше.");
string s2("Эта строка B больше.");
string s3("Эта строка");
int cmp1 = s1.compare(s2); // Отрицательно.
int cmp2 = s1.compare(s3); // Положительно.
```

Метод s.compare(const char *arr) сравнивает таким же образом данную строку s с массивом символов arr, записанным в стиле языка C.

```
int cmp3 = s2.compare("Эта строка C еще больше"); // Отрицательно.
```

Метод s.compare(size_type ind, size_type n, const string &str) сравнивает только n символов, начиная с позиции ind строки s. Строка str сравнивается целиком. Например:

```
int cmp4 = s1.compare(0, 10, s3); // Равно нулю.
```

Четвертый метод сравнения:

```
s.compare(size_type ind1, size_type n1, const string &str,
   size_type ind2, size_type n2)
```

Не только в первой строке, но и в строке str выбирает для сравнения только n2 символов, начиная с символа, находящегося в позиции ind2.

```
int cmp5 = s1.compare(13, 6, s3, 13, 6); // Положительно.
```

Наконец, пятый метод сравнения работает с массивом символов `arr`, записанным в стиле языка:

```
s.compare(size_type ind1, size_type n1, const char *arr,
   size_type n2);
```

В этом методе не задается начальная позиция в массиве `arr`, поскольку указатель типа `char *` всегда можно направить на любую позицию массива, например, вот так:

```
char ch[] = "Еще одна строка.";
int cmp5 = s1.compare(4, 6, ch + 9, 6); // Равно нулю.
```

Перегруженные операции `==`, `!=`, `<`, `>`, `<=`, `>=` делают более грубое сравнение строк, поскольку в результате их применения получается истина (`true`) или ложь (`false`), в зависимости от лексикографического расположения строк. Их реализация основана на методах `compare()`. Они применяются, главным образом, в условных операторах вида:

```
if (s1 == s2) …
if (name != "Иванов") …
if (s1 < s2) …
```

Как найти символ или подстроку в строке

Для поиска отдельных символов и целых строк в заданной строке применяется целое множество методов: `find()`, `rfind()`, `find_first_of()`, `find_last_of()`, `find_first_not_of()`, `find_last_not_of()`. Под каждым из этих шести имен скрывается четыре метода с аргументами типа строки, массива символов, подмассива символов и отдельного символа. Все методы возвращают индекс первого совпавшего символа или очень большое число, если поиск завершился неудачно. Поиск всегда ведется с учетом регистра букв.

Рассмотрим прототипы методов на примере `find()`:

- `size_type find(const basic_string &str, size_type ind = 0) const;` — отыскивает подстроку `str` в данной строке, начиная с индекса `ind`, возвращает индекс начала первого появления подстроки `str` в данной строке;
- `size_type find(const Ch *p, size_type ind = 0) const;` — отыскивает подстроку, на которую направлен указатель `p`, в данной строке, начиная с индекса `ind`, возвращает индекс начала первого появления подстроки `p` в данной строке;
- `size_type find(const Ch *p, size_type ind, size_type n) const;` — отыскивает подстроку, заданную указателем `p`, в данной строке, начиная

с индекса `ind` и заканчивая индексом `n`, возвращает индекс начала первого появления подстроки `p` в данной строке;

- `size_type find(Ch ch, size_type ind = 0) const;` — отыскивает символ `ch` в данной строке, начиная с индекса `ind`, возвращает индекс первого появления символа `ch` в данной строке.

Вот примеры поиска методами `find()`:

```
string s = "Строка, в которой мы ищем подстроку.";
string sub = "рок";
size_type ind1 = s.find(sub); // ind1 == 2
size_type ind2 = s.find(sub, ind1 + 1); // ind2 == 31
size_type ind3 = s.find("кот"); // ind3 == 10
size_type ind4 = s.find('о'); // ind4 == 3
```

Методы `rfind()` аналогичны методам `find()`, но ведут поиск в обратном направлении, от конца текста к его началу. Они, следовательно, отыскивают последнее появление подстроки в строке. Значение по умолчанию аргумента `ind` в этих методах — очень большое число, заведомо большее длины текста. Продолжим наш пример:

```
size_type ind5 = s.rfind(sub); // ind5 == 31
size_type ind6 = s.rfind(sub, ind5 - 1); // ind6 == 2
```

Методы `find_first_of()` аналогичны методам `find()`, но отыскивают первое появление любого символа подстроки в строке. Например:

```
size_type ind7 = s.find_first_of("кот"); // ind7 == 1
```

Методы `find_last_of()` делают то же самое, ведя поиск от конца строки к ее началу. В том же примере получаем:

```
size_type ind8 = s.find_last_of("кот"); // ind8 == 33
```

Методы `find_first_not_of()` аналогичны методам `find_first_of()`, но отыскивают в строке первое появление любого символа, не входящего в подстроку. Например:

```
size_type ind9 = s.find_first_not_of("конкурс"); // ind9 == 1
```

Наконец, методы `find_last_not_of()` ведут в строке поиск отсутствующих в подстроке символов в обратном направлении. В том же примере получаем:

```
size_type ind10 = s.find_last_not_of("конкурс"); // ind10 == 30
```

Как заменить подстроку

Шаблон класса basic_string<Ch, Tr, A>, а, следовательно, класс string, предлагают добрый десяток методов замены replace(). В них указывается та часть строки, которую надо заменить, и заменяющий ее символ или набор символов. Заменяемая часть строки выделяется индексом первого ее символа и количеством символов или итераторами, указывающими на начало и конец заменяемой подстроки. Заменяющий набор символов задается строкой, массивом символов или отдельным символом.

Все методы меняют строку и возвращают ссылку на нее. Длина строки после замены может уменьшиться или увеличиться. Можно заменить десяток символов одним символом, а можно заменить один символ строкой из сотни символов. В частности, есть методы erase(), просто удаляющие подстроку.

Мы не будем выписывать прототипы методов replace(), а просто приведем примеры замен подстрок и отдельных символов:

```
string s1("Строка, подлежащая изменению.");
string s2("Заменяющая строка");
s1.replace(0, 6, s2); // "Заменяющая строка, подлежащая изменению."
s1.replace(19, 5, "удал"); // "Строка, подлежащая удалению."
s1.replace(11, 13, s2, 2, 13); // "Строка, подменяющая строению."
```

Как заменить символ

Заменить один символ строки с известным индексом ind можно простым присваиванием, например:

```
s[ind] = 'C';
```

Кроме того, для замены символа можно воспользоваться итераторами, например:

```
*(s.begin() + ind) = 'C';
```

Разумеется, для замены символа можно применить и методы replace().

Как добавить подстроку

Самый простой способ добавить подстроку к данной строке — воспользоваться перегруженной операцией присваивания:

```
string s = "Строка, к которой ";
s += "добавим эту строку.";
```

Кроме этой операции в библиотеке STL есть еще шесть методов `append()`, добавляющих подстроку, массив символов или отдельный символ в конец строки. Все методы возвращают ссылку на измененную строку. Надо заметить, что при перегрузке операции += применялись методы `append()`.

Два метода просто добавляют к данной строке строку или массив символов, ссылка или указатель на которые записаны как аргумент метода:

```
string s1 = "К этой строке добавим ";
string s2 = "эту ";
s1.append(s2); // "К этой строке добавим эту "
s1.append("и эту строку"); // "К этой строке добавим эту и эту строку"
```

Третий метод добавляет только один символ, но зато столько раз, сколько указано его первым аргументом:

```
s1.append(3, '!'); // "К этой строке добавим эту и эту строку!!!"
```

Следующие два метода добавляют не всю строку или массив, указанные в их аргументах, а только подстроку или подмассив:

```
s2.append(s1, 14, 8); // "эту добавим "
s2.append("и эту строку", 5); // "эту добавим и эту"
```

Наконец, шестой метод добавляет к строке часть коллекции, отмеченную начальным и конечным значением итератора. Например, чтобы добавить к строке целую коллекцию `coll`, можно обратиться к методу `append()` следующим образом:

```
s2.append(coll.begin(), coll.end());
```

Как вставить подстроку

Восемь методов `insert()` предназначены для вставки строки, массива символов или отдельного символа в указанное место данной строки. Место вставки указывается первым параметром почти каждого из методов `ind`. Это индекс элемента строки, перед которым будет сделана вставка. Он должен быть неотрицательным и меньше длины строки, иначе возникнет исключительная ситуация `out_of_range`.

При вставке строка раздвигается, ее длина увеличивается. Индексы символов, идущих за вставленными символами, тоже увеличиваются на число вставленных символов. Методы возвращают ссылку на ту же преобразованную строку. Типичный метод `insert()` с прототипом вставляет строку `str` в данную строку перед ее символом с индексом `ind`:

```
basic_string &insert(size_type ind, const basic_string &str)
```

Например:

```
string s1("Нотеста");
string s2("та про");
s1.insert(2, s2);
```

Аналогичный метод вставляет массив символов. Например:

```
string s = "Это большая строка";
s.insert(4, "не");
```

После выполнения операторов значением переменной `s` будет строка `"Это небольшая строка"`. Еще один пример:

```
s.insert(s.size() - 1, "добавление");
```

Данный метод будет работать так же как метод:

```
s.append("добавление");
```

Третий метод с прототипом:

```
basic_string &insert(size_type ind, size_type n, Ch ch);
```

`n` раз вставляет в указанную позицию `ind` данной строки символ `ch`. Например, после выполнения следующих операторов мы получим строку `"a << b"`:

```
string s("a  b");
s.insert(2, 2, '<');
```

Следующие два метода вставляют в указанное место данной строки не всю строку или массив символов, а только их часть.

```
basic_string &insert(size_type ind1, size_type n1,
     const basic_string &str, size_type ind2, size_type n2);

basic_string &insert(size_type ind1, size_type n1,
     const basic_string &str, size_type n2);
```

Остальные три метода вставки используют итераторы для указания места, в которое вставляется символ или часть коллекции.

Как удалить подстроку

Два метода `erase()`удаляют подстроку из данной строки. В первом методе с прототипом подстрока задается начальной позицией и длиной:

```
basic_string &erase(size_type ind = 0, size_type len = npos);
```

Метод возвращает указатель на измененную строку. Например:

```
s.erase(10, 3);
```

Во втором методе подстрока задается итераторами ее начала включительно и конца исключительно:

```
iterator erase(iterator begin, iterator end);
```

Метод возвращает итератор, направленный на следующий за удаленными элемент `end`. Как показывают значения аргументов по умолчанию, метод `erase()` без аргументов удаляет все содержимое строки.

Как удалить символ

Третий метод удаления удаляет из данной строки один элемент и возвращает итератор на следующий элемент:

```
iterator erase(iterator pos);
```

Как переставить строки

Метод `swap(string &s1, string &s2)` меняет местами две строки, просто меняя ссылки на них. Этот метод удобен, например, для перестановки строк во время их сортировки.

Ввод и вывод строк

Отдельная функция, не относящаяся ни к какому классу, вводит последовательность символов, заканчивающуюся символом окончания строки `eol`, из входного потока `in` в строку `s`:

```
basic_istream &getline(basic_istream &in, basic_string &s, Ch eol);
```

Если символ окончания строки не указан, то им будет служить символ `\n`. Символ окончания строки не записывается в данную строку `s`. Например, для ввода строки с клавиатуры достаточно написать:

```
string s;
getline(cin, s);
```

Кроме того, как обычно в коллекциях, для строк операции >> и << перегружены на ввод и вывод текстов с использованием потоков. Об этом мы поговорим в следующей главе.

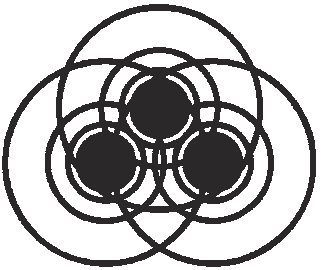
Упражнения

1. Напишите метод подсчета количества строк в заданном тексте.
2. Напишите метод подсчета количества слов в заданном тексте.

3. Напишите метод, отыскивающий в строке заданное слово без учета регистра букв.
4. Напишите метод, меняющий порядок символов в тексте на обратный.
5. Напишите логический метод, определяющий, является ли строка палиндромом.
6. Напишите метод, переводящий все буквы текста в верхний регистр.
7. Напишите метод, записывающий заданное ему число в виде строки цифр, знаков плюс или минус, и десятичной точки.

Контрольные вопросы

1. Почему в библиотеке STL символ не простой тип, а объект шаблона класса `char_traits<T>`?
2. Почему в библиотеке STL текст представлен объектом, а не массивом символов?
3. Каковы преимущества и недостатки массивов для хранения символов?
4. Различают ли методы обработки строки регистр входящих в нее букв?



Глава 16

Потоки ввода-вывода

Для того чтобы отвлечься от особенностей конкретных устройств ввода-вывода, в программировании употребляется понятие *потока* (stream). Считается, что в программу идет *входной поток* (input stream) символов Unicode или просто байтов, воспринимаемый в программе функциями `get()`, `read()` или `scanf()`. Из программы функциями `put()`, `write()` или `printf()` выводится *выходной поток* (output stream) символов Unicode или байтов. При этом неважно, куда направлен поток: на консоль, на принтер, в файл или в сеть, функции `put()` и `printf()` ничего об этом не знают.

Можно представить себе поток как трубу, по которой в одном направлении последовательно "текут" символы или просто байты, один за другим. Функции `read()`, `scanf()`, `write()`, `printf()` взаимодействуют с одним концом трубы, другой конец соединяется с источником или приемником данных.

Объектно-ориентированное программирование развивает и обобщает представление о потоке ввода-вывода. С этой точки зрения поток — это объект некоторого класса, обладающего методами чтения из потока и записи в поток. Впрочем, для удобства работы входной и выходной потоки обычно разделяют и оформляют в виде разных классов.

Поскольку обмен данными с внешним носителем информации происходит очень медленно, в оперативной памяти, как правило, выделяется область промежуточного хранения данных, называемая *буфером* (buffer) ввода-вывода. Поток данных сначала накапливается в буфере, а затем поступает на обработку. Буфер ввода-вывода можно сравнить с почтовым ящиком, в котором письма накапливаются перед отправкой.

Содержимое буфера ввода-вывода поступает на обработку, как правило, только после того, как он заполнится. После обработки буфер очищается, как говорят, сбрасывается, и может снова заполняться. Иногда надо обработать

содержимое буфера еще до его заполнения. Для таких случаев разработан метод `flush()`, принудительно сбрасывающий буфер. В частности, метод `flush()` применяется при закрытии потока, чтобы последняя порция данных не "застряла" в буфере.

Операции ввода-вывода могут пользоваться буфером, а могут действовать напрямую. В зависимости от этого выделяются классы буферизованного и небуферизованного доступа к внешним устройствам.

Кроме того, можно написать классы, методы которых выполняют некоторые предварительные преобразования потока данных, например, разбивку его на строки или преобразование байтовых кодировок символов в Unicode.

Так возникает более или менее развитая совокупность классов, занятых обслуживанием ввода-вывода. В стандартной библиотеке шаблонов STL языка C++ есть целая библиотека шаблонов классов-потоков. Она описана в заголовочных файлах ios, iostream, istream, ostream, fstream, sstream.

Во главе иерархии шаблонов стоит класс `ios_base`, от которого наследует шаблон буферизованного класса `basic_ios<Ch, Tr>`, описанный в заголовочном файле ios. Его прототип выглядит так:

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_ios;
```

`basic_ios<Ch, Tr>` описывает общие свойства потоков, связанные с форматированием, буферизацией, национальными особенностями. Буфер ввода-вывода в библиотеке STL — это объект класса `streambuf`. В классе `basic_ios<Ch, Tr>` содержится указатель на него. Через этот указатель осуществляется работа с буфером.

Шаблон `basic_ios<Ch, Tr>` сделан виртуальным базовым классом для прочих классов-потоков. От него наследуют шаблоны классов ввода `basic_istream<Ch, Tr>` и вывода `basic_ostream<Ch, Tr>`. Вот их прототипы:

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_istream: virtual basic_ios<Ch, Tr>;

template<class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream: virtual basic_ios<Ch, Tr>;
```

Поскольку при вводе и выводе данных наиболее часто применяются типы `char` и `wchar_t`, для потоков ввода-вывода этих типов придуманы специальные имена. Вот их определения:

```
typedef  basic_istream<char>  istream;
typedef  basic_istream<wchar_t>  wistream;
```

```
typedef  basic_ostream<char>  ostream;
typedef  basic_ostream<wchar_t>  wostream;
```

Итак, для ввода-вывода байтовых потоков типа `char` используются классы `istream` и `ostream`, для ввода-вывода символов в кодировке Unicode предназначены классы `wistream` и `wostream`.

Написав в программе следующую директиву препроцессора, мы сразу получаем несколько стандартных потоков:

```
#include <iostream>
```

Они определены в заголовочном файле iostream:

```
istream cin;
ostream cout, cerr, clog;
```

Эти потоки называются *стандартным вводом* (`cin`), *стандартным выводом* (`cout`), *стандартным небуферизованным выводом сообщений* (`cerr`) и *стандартным выводом сообщений* (`clog`). Стандартные потоки могут быть соединены с разными конкретными устройствами ввода-вывода. Как правило, стандартный ввод `cin` связан с клавиатурой, а стандартный вывод `cout` — с экраном дисплея.

Потоки `cerr` и `clog` предназначены для вывода системных сообщений программы: трассировки, сообщений об ошибках или о выполнении каких-то этапов программы. Такие сведения обычно заносятся в специальные журнальные файлы, а не выводятся на консоль. Тем не менее, потоки `cerr` и `clog` в начале работы связаны с экраном дисплея консоли. Потом их можно перенаправить на другие устройства вывода.

Кроме стандартных байтовых потоков определены стандартные расширенные потоки для работы с Unicode, UTF-8 и другими символьными кодировками:

```
wistream wcin;
wostream wcout, wcerr, wclog;
```

Очень часто ввод-вывод связан с файлами. Поэтому в библиотеке STL есть специальные шаблоны классов `basic_ifstream<Ch, Tr>` и `basic_ofstream<Ch, Tr>`, предназначенные для работы с файлами. Эти шаблоны унаследованы от шаблонов классов `basic_istream<Ch, Tr>` и `basic_ostream<Ch, Tr>` соответственно.

Понятие потока оказалось настолько удобным и облегчающим программирование ввода-вывода, что в C++ созданы потоки, направляющие символы

или байты не на внешнее устройство, а в строку типа basic_string. Это делают шаблоны классов basic_istringstream<Ch, Tr> и basic_ostringstream<Ch, Tr>, унаследованные от шаблонов классов basic_istream<Ch, Tr> и basic_ostream<Ch, Tr> соответственно.

Файлы и строки можно просматривать в любом направлении, начиная с любого байта. В них можно изменить произвольный байт, не меняя предыдущие. Эти особенности отражены в обобщенном шаблоне класса прямого доступа к потоку basic_iostream<Ch, Tr>. Он унаследован от шаблонов basic_istream<Ch, Tr> и basic_ostream<Ch, Tr>, следовательно, содержит методы и того и другого потока. От обобщенного шаблона прямого доступа наследуют шаблоны прямого доступа к файлу basic_fstream<Ch, Tr> и строке basic_stringstream<Ch, Tr>.

Для выполнения ввода-вывода классы-потоки библиотеки STL не только применяют обычные методы read(), write(), get(), put(), но и перегружают для этого операции << и >> так, что их запись напоминает перемещение данных в поток или из потока.

Консольный ввод-вывод

В этом разделе мы часто будем использовать объект cin класса istream и объект cout класса ostream без их определений. Напомним, что для этого надо записать в программе следующие строки:

```
#include <iostream>


using namespace std;
```

До сих пор для ввода-вывода, связанного с консолью, мы использовали функции из стандартной библиотеки языка C, главным образом, функции scanf(), gets(), printf(), puts(). Их можно применять и в программах на языке C++, но в языке C++ есть свои, мощные и удобные средства консольного ввода-вывода.

Классы istream и ostream перегружают операцию сдвига вправо (>>) для ввода и операцию сдвига влево (<<) для вывода всех стандартных типов языка C++. Их употребление очень просто. Слева от знака операции ставится объект класса-потока, входной для операции >> и выходной для операции <<. Справа от знака операции записывается переменная, в которую записываются вводимые данные операцией сдвига >> или выражение, результат которого выводится операцией сдвига <<. Создается зрительное впечатление сдвига данных в поток или из потока. Посмотрите, как это сделано в листинге 16.1.

Листинг 16.1. Простейший ввод и вывод

```
#include<iostream>
using namespace std;

main(){
   char name[80];
   cout << "Введите ваше имя: ";
   cin >> name;
   cout << "Добрый день, " << name << "!\n";
}
```

В результате операций >> и << получается ссылка на поток, поэтому каждую из них можно записывать цепочкой, одну за другой. Операции выполняются слева направо, и в левой части каждой следующей операции оказывается поток как результат предыдущей операции. Операции >> и << удобны еще и тем, что, применяя их, не надо задумываться над типами данных. В следующем примере смешаны строковый, целый и символьный типы, и в каждом случае выполняется операция, перегруженная на соответствующий тип:

```
int a = 5, b = 7;
cout << "sum = " << a + b << '\n';
```

Особенности ввода

Операция ввода >> перегружена так, что она читает с консоли только непробельные символы. Дело происходит так. Поскольку ввод буферизуется, вводимые с клавиатуры символы не поступают сразу в программу, а накапливаются в буфере. Это позволяет редактировать вводимые данные, возвращаясь клавишей забоя или клавишей перемещения курсора.

Операция >> выполняется после нажатия клавиши <Enter>. Только после этого читается накопленная в буфере строка введенных символов. При этом начальные пробельные символы пропускаются, и чтение начинается с первого непробельного символа. Если этот символ не соответствует типу переменной, участвующей в операции, то чтение прекращается и операция возвращает нулевое значение. Если же символ соответствует типу переменной, то чтение продолжается до конца буфера или до того момента, когда встретится недопустимый символ.

В листинге 16.2 приведен пример программы, суммирующей числа, введенные с клавиатуры.

Листинг 16.2. Суммирование чисел

```
#include<iostream>
using namespace std;

main(){
   double sum = 0.0, x;
   cout << "Введите несколько чисел: \n";

   while (cin >> x) sum += x;

   cout << "Сумма = " << sum << '\n';
}
```

В условии цикла `while()` использовано свойство операции `>>` выдавать нулевое значение в случае ввода неверного символа. Это приведет к прекращению цикла. Пусть с клавиатуры введена строка:

```
-40 75.27 18,05 2e-5
```

Она будет прочитана до запятой, которая недопустима в записи чисел типа `double`. Будут прочитаны числа −40, 75.27 и 18, после чего цикл прекратит работу. Дробная часть третьего числа 18.05 уже не будет обработана операцией ввода. Если каждый раз вводить отдельные символы, как в следующем примере, то все пробелы, табуляции и символы перевода строки будут пропущены, поскольку ввод каждый раз будет начинаться с непробельного символа:

```
char ch;
while (cin >> ch) cout << ch;
```

Введем строку с пробелами:

```
a  b  c  d
```

После нажатия клавиши <Enter> получим строку без пробелов:

```
Abcd
```

Если такое поведение операции `>>` вас не устраивает, то используйте вместо нее один из методов `get()` или `getline()` класса `basic_istream`. Аргументом этих методов служит символ или указатель на строку символов типа `char *`. Эти методы считают каждый символ просто символом, а не цифрой числа или разделителем. С использованием метода `get()` предыдущий пример посимвольного копирования стандартного ввода в стандартный вывод будет выглядеть так:

```
char ch;
while (cin.get(ch)) cout.put(ch);
```

В этом случае все пробельные символы будут считаны и перейдут в выходной поток. Во многих случаях предпочтительнее ввести в программу "сырую" строку символов и проанализировать ее в самой программе, разбив на слова, числа и прочие лексические единицы. Для этого удобно использовать метод:

```
istream &get(char *buf, int n, char term);
```

Его аргументом служит указатель на строку символов в стиле языка C `buf`. У него есть еще два аргумента — целое число `n`, на единицу большее количества вводимых символов, и символ, завершающий ввод, `term`, которым по умолчанию сделан символ `\n`. Например:

```
char a[100];
cin.get(a, 100, '.');
```

Из входного потока в массив `a[]` будет прочитано максимум 99 символов. Чтение закончится, если встретится точка или закончится входной поток. Если встретилась точка, то она уже не попадет в массив `a[]`, а останется во входном потоке и может быть прочитана следующим методом `get()`. Метод `get()` не записывает в массив `a[]` символ окончания строки `\n`, хотя и оставляет место. Это делает метод `getline()`. С его использованием предыдущий пример можно записать так:

```
char a[100];
cin.getline(a, 100, '.');
```

Массив `a[]` будет заполняться до тех пор, пока не будет введено 99 символов или встретится точка или закончится поток. После этого в массив будет записан символ `\n`. Еще одно отличие метода `getline()` от метода `get()` заключается в том, что метод `getline()` удалит точку из входного потока и следующее применение метода `get()` или метода `getline()` начнет чтение уже после точки. Поясним это на примере. Пусть с клавиатуры введена строка:

```
Что случилось? Когда?
```

После выполнения операторов:

```
char a[80], ch;
cin.get(a, 80, '?').get(ch);
cout << a << ch;
```

получим на экране дисплея строку:

```
Что случилось?
```

А после выполнения операторов:

```
char a[80], ch;
cin.getline(a, 80, '?').get(ch);
cout << a << ch;
```

получим две строки:

```
Что случилось
К
```

Заметили, как в предыдущих примерах методы `get()` и `getline()` вызываются один за другим? Такая запись возможна потому, что они возвращают ссылку на поток, из которого вызваны. Этот поток участвует в следующем вызове метода.

Еще один метод чтения берет из входного потока максимум `n` символов, а не `n - 1`, как методы `get()` и `getline()`:

```
istream &read(char *buf, int n);
```

Он более удобен для ввода данных из файла. Иногда, вводя символы из входного потока, нужно знать, какой символ идет следующим. Например, мы хотим ввести только одно предложение, которое может закончиться точкой, вопросительным знаком или восклицательным знаком. Нам надо предусмотреть появление этих символов. Это позволяет сделать метод `peek()`. Он просматривает следующий символ, но не забирает его из входного потока. Ввод можно оформить так:

```
char a[800], ch;
int k = 0;
while ((ch = cin.peek()) != '.' && ch != '?' && ch != '!')
   cin.get(a[k++]);
a[k] = '\0';
```

Метод `peek()` на самом деле читает следующий символ, но затем возвращает его во входной поток. Более общий метод возвращает во входной поток символ `ch`, записанный в его аргументе. Этот символ будет прочитан первым при следующем чтении входного потока:

```
istream &putback(char ch);
```

Перегрузка операций >> и <<

Операции сдвига `>>` и `<<` легко перегрузить на другие типы данных. Например, перегрузка операции сдвига влево `<<` для класса `Complex` из листинга 11.5:

```
ostream &operator<<(ostream &os, Complex &z){
   return os << z.re << " + i " << z.im;
}
```

После нее можно выводить комплексные числа следующим образом:

```
Complex a(2.5, -3.7);
cout << a;
```

Для операций сдвига >> и <<, выполняемых с шаблонами классов, надо перегружать шаблоны соответствующих функций. Например, для линейного списка из листинга 14.5 это может выглядеть так:

```
template<class T> ostream &operator<<(ostream &os, Elem<T> &ob){
   return os << ob.v <<'\n';
}
template<class T> ostream &operator<<(ostream &os, Elem<T> *pob){
   return os << pob->v <<'\n';
}
template<class T> istream &operator>>(ostream &is, Elem<T> &ob){
   cout << "\nВведите следующий элемент: "';
   return is >> ob.v;
}
```

Форматирование вывода

Операция сдвига влево << удобна для вывода значений разных типов, но она не делает никакого форматирования выводимых данных. Она выводит значения, просто преобразуя их в строку символов по стандартным правилам. В частности, действительные числа выводятся в таком же виде, в каком их выводит функция `printf()` по спецификации `%g`. Что делать, если надо изменить стандартные правила форматирования данных при выводе? Есть три способа форматирования:

- вызвать соответствующий метод класса `basic_ios`;
- установить соответствующие флаги с помощью метода `flags()` класса `basic_ios`;
- применить манипуляторы.

Методы класса *basic_ios*

Первый способ использует три метода класса `basic_ios`. Метод `width(int n)` задает ширину поля для вывода очередного значения. Например:

```
char ch = '?';
cout.width(10);
cout << ch;
cout << ch;
```

Если заданная ширина поля окажется меньше количества выводимых символов, то операция << расширит его до нужного значения. Если заданная ширина поля окажется больше количества выводимых символов, то слева останутся пробелы. Особенность этого способа в том, что метод `width()` действует только на одно выполнение операции, после чего ширина поля обнуляется. В приведенном выше примере символ `ch` займет второй раз уже не 10, а только одну позицию на консоли.

Второй метод `fill(char ch)` определяет символ, заполняющий свободные позиции, остающиеся при большой ширине поля, установленной методом `width()`. По умолчанию это пробел. Например:

```
cout.fill('0');
cout.width(10);
cout << 5555;
```

Третий метод `position(int n)` задает число значащих цифр при выводе чисел. По умолчанию это шесть цифр.

Флаги

Более точное форматирование можно сделать с помощью флагов, определенных в классе `ios_base`, как значения определенного там же типа `fmtflags`. Флаги устанавливаются методом `flags()` или меняются методом `setf()` того же класса. Названия флагов таковы:

- `skipws` — пропускать пробельные символы при вводе (по умолчанию);
- `left` — выравнивать значения по левому краю при выводе;
- `right` — выравнивать значения по правому краю при выводе (по умолчанию);
- `internal` — знак числа выводить слева, само число справа;
- `dec` — выводить числа в десятичной системе (по умолчанию);
- `oct` — выводить числа в восьмеричной системе;
- `hex` — выводить числа в шестнадцатеричной системе;
- `showbase` — вывод шестнадцатеричного числа предварять символами `0x`, вывод восьмеричного числа предварять нулем;
- `showpoint` — выводить десятичную точку для вещественного числа, даже если дробная часть числа отсутствует;
- `uppercase` — выводить заглавные, а не строчные шестнадцатеричные буквы и экспоненту;
- `showpos` — выводить знак "+";

- `scientific` — выводить числа с плавающей точкой;
- `fixed` — выводить числа с фиксированной точкой;
- `unitbuf` — очищать буфер после каждого ввода или вывода;
- `stdio` — очищать поток после каждого ввода или вывода;
- `backfield` — относится к системе счисления (`dec`, `oct`, `hex`);
- `adjustfield` — относится к выравниванию (`left`, `right`, `internal`);
- `floatfield` — относится к вещественным числам (`scientific`, `fixed`).

Метод `flags()` без аргументов показывает установленные флаги.

Аргументом метода `flags(fmtflags)` может быть один из перечисленных выше флагов или объединение флагов вертикальной чертой — операцией дизъюнкции. Например:

```
ios_base::fmtflags old_flags
   cout.flags(ios_base::left | ios_base::fixed | ios_base::hex);
```

Метод `flags()` возвращает совокупность старых флагов, в предыдущем примере мы запоминаем их в переменной `old_flags`. Со временем можно будет восстановить старые флаги:

```
cout.flags(old_flags);
```

Или изменить только отдельные флаги:

```
cout.flags(old_flags | ios_base::uppercase);
```

Метод `setf(fmtflags)` действует так, как в приведенном только что примере. Он меняет только флаги, указанные в его аргументе, не меняя других. Он определен через метод `flags(fmtflags)` следующим образом:

```
fmtflags setf(fmtflags fl){
   return flags(flags() | fl);
}
```

У второго метода `setf(fmtflags fl1, fmtflags fl2)` два аргумента. Первый аргумент `fl1` — флаг или набор флагов, как в методе `flags(fmtflags)`, а второй аргумент `fl2` показывает, к какой области относится первый аргумент. Значениями второго аргумента служат последние три из перечисленных выше флагов.

Метод `setf(fmtflags fl1, fmtflags fl2)` сначала сбрасывает флаги, относящиеся ко второму аргументу `fl2`, а затем устанавливает один из них, указанный в первом аргументе `fl1`. Зачем это нужно? Приглядевшись внима-

тельнее, вы увидите, что флаги, относящиеся ко второму аргументу, взаимно исключают друг друга. В каждой из этих групп можно установить только один флаг. Метод `setf(fmtflags fl1, fmtflags fl2)` безошибочно делает эту работу.

Манипуляторы

С помощью флагов можно точно установить параметры форматирования, но использование их затруднительно, т. к. надо тщательно записать аргумент метода `flags()` или метода `setf()`. Манипуляторами пользоваться легче и нагляднее. Они вставляются прямо в операцию сдвига влево `<<`, изменяя поток "на ходу". Например, мы можем задать вывод чисел в шестнадцатеричной форме, вставив в поток манипулятор `hex`:

```
cout << hex << a << b << 345 << endl;
```

В конце строки тоже стоит манипулятор `endl`, сбрасывающий буфер и добавляющий в конец потока символ перевода строки. Вот список манипуляторов:

- `skipws` — пропускать пробельные символы;
- `noskipws` — сброс флага `skipws`;
- `showbase` — вывод шестнадцатеричного числа предварять символами `0x`, вывод восьмеричного числа предварять нулем;
- `noshowbase` — сброс флага `showbase`;
- `showpoint` — выводить десятичную точку, даже если дробная часть числа отсутствует;
- `noshowpoint` — сброс флага `showpoint`;
- `endl` — добавить символ `\n` и очистить буфер;
- `ends` — добавить к потоку нуль-символ;
- `flush` — очистить буфер;
- `scientific` — выводить числа с плавающей точкой;
- `fixed` — выводить числа с фиксированной точкой;
- `showpos` — выводить знак "+";
- `noshowpos` — сброс флага `showpos`;
- `uppercase` — выводить заглавные шестнадцатеричные буквы и экспоненту;
- `nouppercase` — сброс флага `uppercase`;
- `internal` — знак числа выводить слева, само число справа;
- `boolalpha` — символьное представление `true` и `false`;
- `noboolalpha` — сброс флага `boolalpha`;
- `dec` — выводить числа в десятичной системе счисления;

- `oct` — выводить числа в восьмеричной системе счисления;
- `hex` — выводить числа в шестнадцатеричной системе счисления;
- `setbase(int)` — установить систему счисления, записав ее основание 8, 10, 16 в аргументе;
- `setfill(int)` — установить символ, заполняющий свободные позиции;
- `setprecision(int)` — установить количество значащих цифр при выводе чисел;
- `setiosflags(long)` — установить указанные флаги;
- `resetiosflags(long)` — сбросить указанные флаги;
- `setw(int)` — установить ширину поля аналогично методу `width()`.

Состояние потока

В классе `basic_ios` есть логические методы, позволяющие отследить текущее состояние потока. Метод `good()` возвращает `true`, если предыдущие операции ввода-вывода завершились успешно и можно попытаться выполнить следующую операцию. Методы `fail()` и `bad()` сигнализируют о том, что поток испорчен и операции ввода-вывода невозможны. Если метод `fail()` дает `true`, но метод `bad()` дает `false`, то предыдущие операции завершились успешно. Если же и метод `bad()` вернул `true`, то в этом нельзя быть уверенным.

Метод `eof()` (end of file) дает `true` при достижении конца ввода. Метод `rdstate()`, в отличие от предыдущих методов, дает не логическое значение `true` или `false`, а целое число, содержащее совокупное состояние потока. Дело в том, что поток хранит свое состояние в виде совокупности битовых флагов типа `iostate`, определенного в классе `ios_base`. Отдельные биты отслеживаются методами `good()`, `fail()`, `bad()`, `eof()`, а метод `rdstate()` возвращает всю совокупность битов. В состояниях `fail()` и `bad()` операции ввода-вывода возвращают нулевой указатель. Этим можно воспользоваться для проверок вида:

```
if (!cout) cerr << "Ошибка вывода.";
```

Файловый ввод-вывод

Почти всякая программа в процессе своей работы обращается к файлам для чтения или записи информации. Очень много программ читает из файлов исходную информацию и заносит результаты своей работы в файлы. Многие программы используют файлы для хранения промежуточной информации. Поэтому в библиотеке STL разработаны шаблоны `basic_ifstream<Ch, Tr>` и `basic_ofstream<Ch, Tr>` классов-потоков, оптимизированных для работы с файлами. Они унаследованы от шаблонов

basic_istream<Ch, Tr> и basic_ostream<Ch, Tr>. Вот заголовки этих шаблонов, записанные в заголовочном файле fstream:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ifstream: public basic_istream<Ch, Tr>;


template<class Ch, class Tr = char_traits<Ch> >
class basic_ofstream: public basic_ostream<Ch, Tr>;
```

Поскольку файлы в большинстве современных операционных систем понимаются как последовательность байтов, для файлового ввода-вывода чаще всего создаются байтовые потоки с элементами типа char. Для них определены имена ifstream и ofstream. Вот как выглядят эти определения:

```
typedef  basic_ifstream<char>  ifstream;
typedef  basic_ofstream<char>  ofstream;
```

Потоки ifstream и ofstream особенно удобны для бинарных файлов, хранящих сжатые архивы, изображения, звук, машинные коды исполнимых файлов. Но очень много файлов содержат тексты, составленные из символов. Несмотря на то, что символы могут храниться в кодировке Unicode, эти тексты чаще всего записаны в байтовых кодировках. Поэтому и для текстовых файлов можно использовать байтовые потоки. Если же тексты хранятся в Unicode или какой-нибудь другой не байтовой кодировке, то можно воспользоваться реализациями шаблонов классов ввода-вывода, сделанными на основе типа wchar_t. Для них определены имена wifstream и wofstream. Вот как выглядят эти определения:

```
typedef  basic_ifstream<wchar_t>  wifstream;
typedef  basic_ofstream<wchar_t>  wofstream;
```

Создание, открытие и закрытие файловых потоков

В конструкторах файловых потоков задается имя файла указателем на строку символов. Конструкторы не только создают объект, но и отыскивают файл и открывают его. Например:

```
ifstream f1("file1.txt");
ofstream f2("file2.txt");
```

По умолчанию входные потоки типа ifstream открываются на чтение, они не изменяют файл. Выходные потоки типа ofstream открываются на запись, обнуляя файл, если он уже существует.

Внимание!

Содержимое файла, открытого на запись конструктором с одним аргументом, стирается!

Если надо изменить режим открытия файла по умолчанию, то можно воспользоваться конструктором с двумя аргументами. Второй аргумент этого конструктора указывает, для чего открывается файл. Этот аргумент принимает значение одной из следующих констант типа `openmode`, определенных в классе `ios_base`:

- `in` — открыть поток для чтения;
- `out` — открыть поток для записи и обнулить, если он существует;
- `app` — открыть для добавления в конец потока;
- `ate` — открыть поток и перемотать в конец;
- `tranc` — обнулить файл, если он существует;
- `binary` — считать файл двоичным.

Последние два режима можно сочетать с предыдущими, записывая их в конструкторе через вертикальную черту — операцию дизъюнкции, например:

```
ifstream f1("file1.txt", ios_base::in | ios_base::binary);
```

После создания потока и работы с ним поток нужно закрыть методом `close()`. Потом его можно заново открыть в том же или другом режиме методом `open()`, имеющим те же аргументы, что и конструктор файлового потока. В следующем примере файл, созданный ранее для записи, закрывается и открывается для чтения:

```
f2.close();
f2.open("file2.txt", ios_base::in);
```

Деструктор потока закрывает файл перед уничтожением потока.

Перемещение по потоку

После открытия потока в нем, а точнее, в связанном с ним буфере, устанавливается текущая позиция. В режимах `in`, `out` это начало потока, нулевой байт файла. В режимах `app`, `ate` начальная текущая позиция — конец файла. Чтение и запись всегда начинаются с текущей позиции потока. После чтения и записи текущая позиция увеличивается на прочитанное или записанное число символов.

Текущую позицию всегда можно изменить, переместившись к концу или к началу потока на некоторое число символов. Для файла, открытого для чтения,

это делает метод `seekg()`, перемещение по файлу, открытому для записи, выполняется методом `seekp()`. Вот их прототипы:

```
basic_istream &seekg(off_type n, ios_base::seekdir d = ios_base::beg);
basic_ostream &seekp(off_type n, ios_base::seekdir d = ios_base::beg);
```

Тому и другому методу указывается количество символов `n`, на которые надо переместиться по потоку, и направление перемещения `d` — к концу файла или к его началу. Направление задается одной из следующих констант типа `seekdir`, определенных в классе `ios_base`:

- `beg` — перемещение от начала файла к его концу, смещение не должно быть отрицательным;
- `cur` — перемещение от текущей позиции вперед или назад;
- `end` — перемещение от конца файла к его началу, смещение не должно быть положительным.

Например, после выполнения следующих операторов чтение бинарного файла prog.dat пойдет с 20-ой позиции от конца файла:

```
ifstream ifs("prog.dat", ios_base::binary);
ifs.seekg(-20, ios_base::end);
```

Еще один пример. После выполнения следующих операторов пятым символом файла prog.cc будет звездочка, а шестым — наклонная черта:

```
ofstream ofs("prog.cc");
ofs.seekp(5);
ofs << '/';
ofs.seekp(-2, ios_base::cur);
ofs << '*';
```

Чтение из файла и запись в файл

Сразу после выполнения конструктора можно читать файл, открытый для чтения, или записывать в файл, открытый для записи. Это можно делать перегруженными операциями сдвига `>>` и `<<`, как и в любом потоке, или методами `get()`, `getline()`, `read()`, `put()`, `write()`. В листинге 16.3 приведен полный цикл работы с файлом.

Листинг 16.3. Работа с файлом

```
#include<iostream>
#include<fstream>
```

```
#include<stdlib.h>
using namespace std;

main(){
   char fname[256];

   cout << "Введите имя входного файла.\n";
   cin >> fname;

   ifstream ifs(fname);
   if (!ifs){
      cerr << "Входной файл невозможно открыть.\n";
      return;
   }

   cout << "Введите имя выходного файла.\n";
   cin >> fname;

   ofstream ofs(fname);
   if (!ofs){
      cerr << "Выходной файл невозможно создать.\n";
      return;
   }

   char ch;
   while (ifs && ofs){
     ifs.get(ch); ch = toupper(ch); ofs.put(ch);
   }

   ifs.close();
   ofs.close();

   cout << "Введите вашу подпись.\n";

   ofs.open(fname, ios_base::app);
   cin >> fname;
   ofs << fname;
   ofs.close();
}
```

Узнать текущую позицию в потоке, открытом для чтения, можно методом `tellg()`, а в потоке, открытом для записи, — методом `tellp()`.

Прямой доступ к файлу

Поскольку файлы хранятся на диске — устройстве с прямым доступом — их можно просматривать не только с начала, но и с любого байта. По файлам можно перемещаться в прямом и обратном направлении. Информацию можно записать в любое место файла, а не только в его конец.

Как говорилось в начале этой главы, особенности прямого доступа к потоку отражены в шаблоне класса `basic_iostream<Ch, Tr>`. Он наследует все свойства входных и выходных потоков и обладает методами чтения из потока и записи в поток. Кроме всего прочего, он наследует и оба буфера — один для введенных из потока символов, другой для символов, выводимых в поток. Буферы синхронизируются так, что перемещение по одному из них вызывает такое же перемещение во втором буфере.

От шаблона `basic_iostream<Ch, Tr>` непосредственно унаследован шаблон класса прямого доступа к файлу `basic_fstream<Ch, Tr>`. Как принято в библиотеке STL, две реализации этого шаблона, сделанные для типов `char` и `wchar_t`, получили имена `fstream` и `wfstream` соответственно. Метафора потока не полностью отражает возможности работы с файлом. Тем не менее, классы `fstream` и `wfstream` относятся к потокам по факту своего наследства. Не у всех компиляторов для потоков прямого доступа типа `fstream` или типа `wfstream` установлен режим открытия по умолчанию. При создании или открытии потока режим надо задать явно, например, можно сделать такое определение объекта:

```
fstream fs1("file1.txt", ios_base::in | ios_base::out);
```

После открытия можно переместиться по файлу, используя для этого метод `seekg()` или метод `seekp()`. Оба метода унаследованы от своих предков и работают одинаково.

После того как текущая позиция установлена в нужное место файла, можно записывать или читать файл перегруженными операциями сдвига `<<` и `>>` или унаследованными методами. Разрешается чередовать чтение и запись, не закрывая и повторно открывая файл, но некоторые системы файлового ввода-вывода при этом могут дать сбой. Специалисты рекомендуют перед сменой чтения и записи выполнить метод изменения текущей позиции `seekg()` или метод `seekp()`. Если при этом позиция не должна измениться, то можно задать нулевое смещение.

Строковые потоки

В начале главы уже говорилось о том, что библиотека STL содержит средства организации потока, связанного со строкой, — объектами типа basic_string<Ch, Tr, A>. Эти средства собраны в шаблоне входного потока basic_istringstream<Ch, Tr>, в шаблоне выходного потока basic_ostringstream<Ch, Tr> и в шаблоне прямого доступа basic_stringstream<Ch, Tr>. Все определения собраны в заголовочный файл sstream, который надо подключать к программе, использующей строковые потоки.

Поскольку для строк чаще всего используются две реализации шаблона basic_string<Ch, Tr, A> — классы string и wstring, основанные на типах char и wchar_t соответственно — то и для строковых потоков сделаны такие же реализации. Этим реализациям даны специальные названия, перечисленные ниже:

```
typedef  basic_istringstream<char>  istringstream;
typedef  basic_ostringstream<char>  ostringstream;
typedef  basic_stringstream<char>  stringstream;

typedef  basic_istringstream<wchar_t>  wistringstream;
typedef  basic_ostringstream<wchar_t>  wostringstream;
typedef  basic_stringstream<wchar_t>  wstringstream;
```

Итак, мы можем читать строку типа string, так же как и файл, операцией сдвига вправо >> или методами get(), getline(), read() класса istringstream, записывать символы в строку операцией сдвига влево << или методами put(), write() класса ostringstream или применять те же операции и методы класса stringstream для чтения и записи.

Такой подход к строкам кажется странным, ведь они находятся не на внешнем устройстве, а в оперативной памяти. Вспомните, однако, что даже при вводе-выводе, связанном с файлом, мы фактически работаем не с ним, а с буферами, размещенными в оперативной памяти. Все действия выполняются не с файлом, а с областью памяти и только после заполнения буфера данные, хранящиеся в нем, переносятся на диск и записываются в файл.

Какую пользу можно извлечь из того, что после определения строки:

```
string s("строка символов.");
```

мы не пишем:

```
s[0] = 'C';
```

а определяем поток и пишем в него:

```
stringstream oss(s, ios_base::in | ios_base::out);
oss << 'C';
```

Польза в том, что операции вывода в поток перегружены на множество типов, и нам, например, не надо думать о том, как преобразовать число, чтобы получить строку его цифр. Мы просто пишем два оператора и получаем символьную запись:

```
stringstream oss;
oss << -25.67e-9;
```

Символьный поток динамически расширяется, чтобы вместить всю записанную в него информацию. Обратно, если в строке записаны числа в символьном виде, то также легко можно преобразовать их в числовой тип. Вот как, например, можно их просуммировать:

```
int n, sum = 0;
while (oss >> n) sum += n;
```

Чтобы получить содержимое символьного потока в виде строки класса `string`, не надо считывать в цикле каждый символ. Эта операция выполняется за один раз специально разработанным методом `str()`. Продолжая пример с числом, нужно написать всего один оператор, чтобы получить строку с символьным представлением числа:

```
string s1 = oss.str();
```

Классический пример применения символьного потока — это разбиение строки на отдельные слова. В листинге 16.4 используется свойство операции сдвига вправо `>>` пропускать пробельные символы.

Листинг 16.4. Разбиение строки на отдельные слова

```
#include<iostream>
#include<sstream>
#include<stdlib.h>

using namespace std;

main(){
```

```
   string word, s("Строка, которая будет разбита на слова.");

   istringstream iss(s);

   while (iss >> word)
      cout << word << endl;
}
```

Используя средства форматирования, имеющиеся во входном потоке, можно легко провести более сложный синтаксический разбор строки.

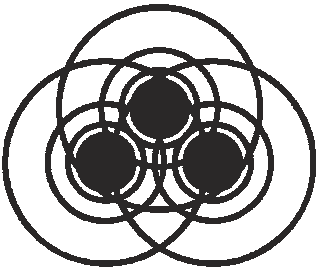
Упражнения

1. Напишите программу, выводящую на экран дисплея символы по одному на строке.
2. Напишите программу, выводящую на экран дисплея слова по одному на строке.
3. Напишите операторы, вводящие числа, набранные на клавиатуре, в переменные разных числовых типов.
4. Напишите программу, выводящую на экран символы, введенные с клавиатуры, в обратном порядке.
5. Напишите программу, подсчитывающую во время чтения число символов, слов и строк во входном потоке.
6. Напишите программу, читающую входной поток и переводящую все буквы в верхний регистр.
7. Напишите программу, копирующую один файл в другой, убирая при этом лишние пробельные символы.
8. Напишите программу, копирующую последовательно несколько файлов в один файл, отделяя в нем содержимое одного файла от другого пустой строкой.

Контрольные вопросы

1. Какие преимущества дает представление ввода-вывода в виде потока?
2. Что такое буфер ввода-вывода?
3. Как происходит работа с буфером?
4. Буферизуется ли поток символов, вводимых с клавиатуры?

5. Каковы особенности ввода данных с клавиатуры?
6. В чем разница между стандартным выводом `cout` и стандартным выводом сообщений `cerr` или `clog`?
7. Почему в выходной поток можно выводить данные разных типов, не заботясь об их преобразовании к одному типу?
8. Какое свойство перегруженных операций сдвига позволяет записывать их цепочкой?
9. Можно ли организовать поток, выводящий на экран дисплея графические изображения?
10. Как организовать поток символов для их печати на принтере?
11. Можно ли организовать поток данных на устройство проигрывания звуков?
12. В каких случаях удобно использовать поток, связанный со строкой?



Глава 17

Классы-контейнеры

Ранее мы уже познакомились с динамическими структурами данных и построили некоторые из них (*см. гл. 14*). Они предназначены для хранения произвольного, не фиксированного заранее количества элементов и обработки их по определенным для каждой структуры алгоритмам. Эти структуры настолько важны для обработки информации и настолько часто применяются, что их просто необходимо было включить в стандартную библиотеку шаблонов STL языка C++. В этой главе мы познакомимся с контейнерами библиотеки STL.

Контейнеры библиотеки STL (векторы, стеки, списки) специализированы для выполнения определенных действий: вставок, удалений, поиска и сравнений. Для каждой такой операции можно подобрать наиболее подходящий контейнер. В контейнерах можно хранить объекты произвольного типа, но классы этих объектов должны иметь конструктор по умолчанию и конструктор копирования. В этих классах должны быть перегружены операции присваивания `=` и сравнения, хотя бы на равенство `==` и на неравенство `!=`. В некоторых случаях требуется перегрузка операции `<`.

Шаблон класса *vector*

В классе, созданном по шаблону `vector<T, A>`, описанному в заголовочном файле vector, может храниться любое количество элементов типа `T`. Элементы получают индексы 0, 1, 2, К каждому элементу вектора можно непосредственно обратиться по индексу, как и к элементу массива.

Второй параметр шаблона `A` — это распределитель памяти, по умолчанию равный стандартному распределителю для класса `T`, а именно `allocator<T>`. Этот распределитель памяти использует операции `new` и `delete`, обычные для типа `T`.

Кроме количества элементов, называемого *размером* (size) вектора, есть еще размер буфера оперативной памяти, выделенного для вектора — *емкость* (capacity) вектора. Емкость зависит от выбранного распределителя памяти, который автоматически выделяет следующую порцию ячеек памяти, когда при увеличении вектора его размер приближается к текущей емкости. При необходимости можно изменить емкость, установив новый размер вектора `n` методом `resize()`. Вот его прототип:

```
void resize(size_type n, T t = T());
```

Как видно из этого прототипа, новые элементы вектора будут заполнены объектом `t`. Второй аргумент метода `resize()` можно не записывать, тогда объект `t` будет создан конструктором по умолчанию класса `T`. Мы не будем рассматривать распределение памяти для элементов вектора, считая, что этим занимается распределитель по умолчанию.

Как создать вектор

В шаблоне класса `vector<T>` пять конструкторов:

- `vector()` — конструктор по умолчанию, создает пустой объект нулевого размера;
- `vector(const vector &v)` — конструктор копирования;
- `vector(int n)` — создает объект указанного размера `n`, заполненный копиями объекта типа `T`, созданного конструктором по умолчанию класса `T`;
- `vector(int n, T &t)` — создает объект указанного размера `n`, заполненный копиями объекта `t`;
- `vector(cont it1, cont it2)` — вектор создается копированием некоторого диапазона элементов другого контейнера `cont`, выделяемого итераторами `it1` и `it2`.

Вот, например, определение вектора `v1` из 1000 целых чисел, заполненного нулями:

```
vector<int> v1(1000);
```

Следующий вектор `v2` заполняется десятью семерками:

```
vector<int> v2(10, 7);
```

Следующая строка создает вектор `v3` из 100 комплексных чисел, созданных конструктором `Complex()`, т. е. чисел с нулевой действительной и мнимой частью:

```
vector<Complex> v3(100);
```

Вектор `v4` состоит из пятидесяти комплексных чисел `z`:

```
Complex z(1.5, -3.47);
vector<Complex> v4(50, z);
```

Если в классе `T` нет конструктора по умолчанию, то создавать вектор конструктором `vector(int n)` нельзя. После создания вектора можно обращаться к его элементам, как к элементам массива:

```
v1[0] = -5; v1[1] = 22;
v1[2] = v2[0] + 3 * v1[1] - 10;
```

Операция `[]` не отслеживает попытки обратиться к несуществующему элементу вектора, например, к элементу `v1[-1]` или к элементу `v1[1000]`. За этим должен следить сам программист.

Как добавить элемент в вектор

Вектор в библиотеке STL сконструирован как универсальный шаблон, по которому можно строить линейные структуры данных: стеки, очереди, кольца. Поэтому в нем есть метод добавления элементов в конец вектора и метод вставки нового элемента в указанное место вектора. Эти методы автоматически расширяют вектор, увеличивая его размер. Добавление элемента к концу вектора выполняет метод:

```
void push_back(const T &t);
```

Вот как с его помощью можно постепенно наращивать вектор:

```
vector<string> v;
string word;
string lastWord = "Конец";
while (cin >> word && word != lastWord)
   v.push_back(word);
```

Следующий метод вставляет новый элемент `t` перед позицией, на которую направлен итератор `pos`:

```
iterator insert(iterator pos, const T &t)
```

Например, вставку перед первым элементом вектора `v1` можно сделать оператором:

```
v1.insert(v1.begin(), 555);
```

Следующий метод вставляет `n` новых элементов с одним и тем же значением `t` перед позицией, на которую направлен итератор `pos`:

```
iterator insert(iterator pos, size_type n, T &t)
```

Метод, вставляющий элементы из другого вектора в диапазоне итераторов `first`, `last` перед позицией, на которую направлен итератор `pos`:

```
iterator insert(iterator pos, initer first, initer last)
```

После вставки новых элементов в вектор индексы всех следующих за ними элементов увеличиваются на количество вставленных элементов. Это требует некоторого времени. Быстрее всего происходит добавление элементов в конец вектора, поэтому для этой операции создан особый метод `push_back()`. Если порядок элементов в векторе не имеет значения, то лучше пользоваться этим методом.

Как заменить элемент

Изменить элемент вектора с индексом `ind` можно простым присваиванием с использованием операции `[]` или метода `at()`, например:

```
v1[ind] = 777;
v1.at(ind) = 777;
```

Кроме того, итераторы перегружают операцию обращения к элементу по адресу, поэтому для изменения элемента вектора можно записать оператор вида:

```
*(v1.begin + ind) = 777;
```

Как узнать размер вектора

Количество элементов в векторе всегда можно узнать методом `size()`. Метод `capacity()` возвращает емкость вектора. Логический метод `empty()` возвращает `true`, если в векторе нет ни одного элемента. Он устроен очень просто:

```
bool empty() const{
   return size() == 0;
}
```

Как обратиться к элементу вектора

Кроме того, что в векторе перегружена операция `[]` и к его элементам можно обращаться, как к элементам массива, существует еще метод `at(size_type ind)`, возвращающий ссылку на элемент вектора с индексом `ind`, например:

```
v1.at(2) = 33;
int a = v1.at(2);
```

Удобство метода `at()` заключается в том, что он, в отличие от операции `v1[ind]`, проверяет попадание индекса `ind` в диапазон индексов вектора и выбрасывает исключение класса `out_of_range`, если индекс `ind` отрицателен или превысил число `v1.size() - 1`.

Методы `front()` и `back()` дают первый и последний элементы вектора, т. е. `v1[0]` и `v1[v1.size() - 1]`.

Вектор, как и всякий контейнер, определяет итератор. Два метода возвращают его конкретные значения. Метод `begin()` возвращает итератор, направленный на первый элемент вектора, метод `end()` — итератор, который находится за концом вектора. Таким образом, мы можем определить указатель на начало вектора:

```
vector<int>::iterator p1 = v1.begin();
```

Или, проще:

```
int *p1 = v1.begin();
```

Вместо того чтобы писать:

```
int k1 = v1.[0], k2 = v1[1];
v[1] = 55;
```

мы можем написать:

```
int k1 = *p1, k2 = *p1++;
*p1 = 55;
```

В отличие от этого, указатель `*p2` не направлен ни на один элемент, он находится за концом вектора:

```
int *p2 = v1.end();
```

Последний элемент вектора можно получить только после перемещения указателя, например, так:

```
*p2--;
```

Итераторами, как всякими указателями, удобно пользоваться для прохода по вектору. Например, все элементы вектора `v1` можно просмотреть в следующем цикле:

```
for (int *p = v1.begin(); p != v1.end(); p++)
   cout << *p << endl;
```

Для удобства просмотра вектора в обратном направлении в нем создается *обратный итератор* (reverse iterator) — объект класса `reverse_iterator`. Его начальное значение устанавливается на последний элемент вектора методом `rbegin()`, а конечное значение — перед первым элементом вектора методом `rend()`. Увеличение обратного итератора операцией инкремента означает переход к предыдущему элементу вектора. Например, обход и вывод всех элементов вектора `v1` в обратном направлении можно сделать так:

```
for (int *p = v1.rbegin(); p != v1.rend(); p++)
   cout << *p << endl;
```

Векторы обычно реализуются с двунаправленными итераторами, в которых перегружены операции инкремента и декремента.

Как удалить элементы вектора

Метод `pop_back()` удаляет последний элемент вектора, уменьшая размер вектора на единицу. Он не возвращает никакого значения.

Метод *`iterator erase(iterator pos)`* удаляет элемент, на который направлен итератор `pos`. Размер вектора уменьшается на единицу. Индексы элементов, следующих за удаленным элементом, тоже уменьшаются на единицу. Например, первый элемент вектора `v1` можно удалить оператором:

```
v1.erase(v1.begin());
```

Последний элемент — оператором:

```
v1.erase(v1.end() - 1);
```

Метод *`iterator erase(iterator pos1, iterator pos2)`* удаляет элементы вектора от `pos1` включительно до `pos2` исключительно, уменьшая размер вектора на число удаленных элементов. Вектор "сдвигается" и индексы элементов, следующих за удаленными элементами, уменьшаются. Например, чтобы оставить в векторе `v1` только первый и последний элементы, достаточно написать оператор:

```
erase(v1.begin() + 1, v1.end() - 1);
```

Метод `clear()` удаляет все элементы вектора, обнуляя его размер. Например, на вектор `v1` следующие операторы действуют одинаково:

```
v1.clear();
```

```
v1.erase(v1.begin(), v1.end());
```

Шаблон класса *stack*

Ранее мы сделали несколько реализаций стека (*см. гл. 14*), основанных на массивах и указателях. Работа со стеком не зависела от реализации, методы добавления элемента на стек `push()` и удаления элемента со стека `pop()` работали совершенно одинаково везде. Библиотека STL тоже отвлекается от физической реализации стека, но она идет дальше, позволяя выбрать контейнер для хранения элементов стека. Точнее говоря, стек определяется в заголовочном файле stack как шаблон класса следующего вида:

```
template<class T, class C = deque<T> > class std::stack{
protected:
   C cont;
public:
```

```
   // Прочие члены шаблона класса.
};
```

Таким образом, стек зависит не только от типа `T` хранящихся в нем элементов, но и от контейнера `cont`, тип `C` которого указан вторым параметром шаблона. По умолчанию для хранения элементов стека выбирается дек. Стек предоставляет только те методы контейнера, которые нужны для его правильной работы. В стеке один конструктор. Вот его определение:

```
stack(const C &a = C()): cont(a){}
```

Как видите, конструктором стека по умолчанию служит конструктор по умолчанию того контейнера, в котором хранятся элементы стека. В конструктор можно передать экземпляр контейнера, созданного другим конструктором контейнера. Методы стека просто обращаются к методам соответствующего контейнера. Вот эти определения:

```
void push(const T &t){
   cont.push_back(t);
}
void pop(){
   cont.pop_back();
}
T &top(){
   return c.back();
}
bool empty() const{
   return cont.empty();
}
size_type size() const{
   return cont.size();
}
```

Итак, стек в библиотеке STL — это просто оболочка над каким-нибудь контейнером, предоставляющая те методы контейнера, которые характерны для стека. Листинг 17.1 показывает, как можно использовать стек для проверки парности скобок или каких-нибудь других символов в арифметическом выражении.

Листинг 17.1. Проверка парности скобок

```
#include <iostream>
#include <stack>
#include <string>
```

```
bool checkParity(string expression,
                        string open, string close){
    stack stk;
    stringstream st(stk);
    string symb;

    while (st >> symb){
      if (symb == open)  stk.push(open);
      if (symb == close) stk.pop();
    }
    return stk.empty();
}

main(){
    cout <<
       checkParity("a - (b - (c - a) / (b + c) - 2)", "(", ")" );
  }
}
```

Шаблон класса *deque*

Дек в библиотеке STL определен как полноценный шаблон класса `deque<T, A>`, подобно шаблону вектора `vector<T, A>`. Мы договорились не рассматривать распределение памяти, поэтому будем считать, что у второго параметра шаблона взято значение по умолчанию. Описание дека сделано в заголовочном файле deque. В шаблоне класса `deque<T>` пять конструкторов:

- `deque()` — конструктор по умолчанию, создает пустой объект нулевого размера;
- `deque(const deque &d)` — конструктор копирования;
- `deque(int n)` — создает объект указанного размера `n`, заполненный копиями объекта типа `T`, созданного конструктором по умолчанию класса `T`;
- `deque(int n, T &t)` — создает объект указанного размера `n`, заполненный копиями объекта `t`;
- `deque(cont it1, cont it2)` — дек создается копированием некоторого диапазона элементов другого контейнера `cont`, выделяемого итераторами `it1` и `it2`.

Как видите, создание дека очень похоже на создание вектора. Вот, например, определение дека `d1` из 100 символов "#":

```
deque<char> d1(100, '#');
```

Следующий дек d2 заполняется пустыми строками:

```
deque<string> d2(10);
```

Если в классе T нет конструктора по умолчанию, то создавать дек конструктором deque(int n) нельзя.

Как добавить элемент в дек

Как следует из определения дека, добавлять элементы можно только к его началу или концу. Добавление элемента к концу дека выполняет точно такой же метод, как и в векторе:

```
void push_back(const T &t);
```

Добавление элемента к началу дека производится следующим методом:

```
void push_front(const T &t);
```

Хотя это и не характерно для дека, но в класс deque<T> все-таки введены методы вставки. Дело в том, что класс deque<T> считается базовым для построения других контейнеров. Мы уже видели, как на основе дека строится стек. Далее мы посмотрим, как на его основе создаются очереди.

```
iterator insert(iterator pos, const T &t)
```

Этот метод вставляет новый элемент t перед позицией, на которую направлен итератор pos. Например, вставку перед первым элементом дека d1 можно сделать оператором:

```
d1.insert(d1.begin(), 'A');
```

Метод, который вставляет n новых элементов с одним и тем же значением t перед позицией, на которую направлен итератор pos:

```
iterator insert(iterator pos, size_type n, T &t)
```

Следующий метод вставляет элементы из другого дека или вектора в диапазоне итераторов first, last перед позицией, на которую направлен итератор pos:

```
iterator insert(iterator pos, initer first, initer last)
```

Как удалить элементы из дека

Метод pop_back() удаляет последний элемент дека, уменьшая его размер на единицу. Он не возвращает никакого значения. Метод pop_front() удаляет первый элемент дека, уменьшая его размер на единицу. Он тоже не возвращает никакого значения. Класс deque<T> позволяет удалять внутренние элементы, хотя это не характерно для дека. Метод, позволяющий удалить элемент, на который направлен итератор pos:

```
iterator erase(iterator pos);
```

Размер дека уменьшается на единицу. Например, так же как и в векторе, первый элемент дека `d1` можно удалить оператором:

```
d1.erase(d1.begin());
```

Последний элемент — оператором:

```
d1.erase(d1.end() - 1);
```

Метод, позволяющий удалить элементы дека от `pos1` включительно до `pos2` исключительно, уменьшая размер дека на число удаленных элементов:

```
iterator erase(iterator pos1, iterator pos2);
```

Метод `clear()` удаляет все элементы дека, обнуляя его размер.

Как узнать размер дека

Количество элементов в деке всегда можно узнать методом `size()`. Логический метод `empty()` возвращает `true`, если в деке нет ни одного элемента.

Прочие методы дека

Несмотря на свое название, шаблон класса `deque<T>` универсален. Его элементы проиндексированы. В нем перегружена операция `[]`, позволяющая обратиться к любому его элементу по индексу, точно так же, как и в векторе. Более того, в нем есть все методы шаблона класса `vector<T>`, да еще и все методы шаблона класса `list<T>`. Поэтому с деком класса `deque<T>` можно обращаться как с вектором или со списком. Тем не менее, создатели класса `deque<T>` предупреждают, что в нем оптимизированы только операции с концами дека, а обращение к другим элементам может быть не эффективно.

Шаблон класса *queue*

Очередь класса `queue<T, C>`, подобно стеку, реализована, как оболочка над каким-нибудь контейнером типа `C`. Контейнер сделан защищенным полем класса. Сам шаблон класса схематически выглядит так:

```
template<class T, class C = deque<T> > class std::queue{
protected:
   C cont;
public:
   queue(const C &a = C()): cont(a){}

   void push(const T &t){ cont.push_back(t); }
```

```
   void pop(){ cont.pop_front(); }

   bool empty() const{ return cont.empty(); }
   size_type size() const{ return cont.size(); }

   T &front(){ return cont.front(); }
   T &back(){ return cont.back(); }
};
```

Как видно из этого описания, метод `push()` добавляет элемент к концу контейнера `cont`, а метод `pop()` удаляет элемент из начала контейнера `cont`, связанного с очередью. По умолчанию в качестве контейнера `cont` берется дек класса `deque<T>`.

Шаблон класса *priority_queue*

Шаблон класса `priority_queue<T, C, Cmp>` реализует структуру данных, называемую *очередью с приоритетом*. Элементы очереди с приоритетом отсортированы так, что первым в очереди всегда стоит наибольший элемент. Поэтому метод `front()` всегда возвращает наибольший элемент, а метод `pop()` удаляет его из очереди с приоритетом.

Порядок сортировки можно изменить. В общем случае сравнение элементов очереди выполняется методами специального класса-сравнителя. Класс-сравнитель указывается как параметр `Cmp` шаблона. В самом простом случае сравнением занимается перегруженная операция "меньше". Вот как выглядит схематически шаблон очереди с приоритетом:

```
template<class T,
   class C = vector<T>,
   class Cmp = less<typename C::value_type>  >
class std::priority_queue{
protected:
   C cont;
   Cmp cmp;
public:
   priority_queue(const Cmp & a1 = Cmp(),
       const C &a2 = C()): cont(a2), cmp(a1){}

   void push(const T &t);
```

```
    void pop();

    bool empty() const{ return cont.empty(); }
    size_type size() const{ return cont.size(); }

    T &top(){ return cont.front(); }
};
```

Как видно из этого описания, очередь с приоритетом хранит элементы в одном из своих защищенных полей — контейнере `cont`. По умолчанию в качестве контейнера `cont` берется вектор класса `vector<T>`.

Объект класса, методы которого сравнивают элементы для сортировки, — это второе защищенное поле `cmp`. По умолчанию для сравнения просто применяется операция `<`, перегруженная в классе `T`.

Метод `push()` добавляет элемент в контейнер `cont`, сохраняя его упорядоченность, а метод `pop()` удаляет наибольший элемент упорядоченного контейнера `cont`, обычно находящийся в начале контейнера. Метод `top()` возвращает этот элемент.

Кроме того, методом `size()` можно узнать количество элементов очереди, а методом `empty()` проверить наличие элементов в ней.

Шаблон класса *list*

Линейный список класса `list<T, A>` оптимизирован для вставок и удалений элементов. В классе `list<T, A>` нет обращения к элементу по индексу операцией `[]` потому, что такая операция выполняется медленно. Метода `at()` нет по той же причине. Для доступа к элементу предоставляются итераторы. В шаблоне класса `list<T>` пять конструкторов:

- `list()` — конструктор по умолчанию, создает пустой объект нулевого размера;
- `list(const list &d)` — конструктор копирования;
- `list(int n)` — создает объект указанного размера `n`, заполненный копиями объекта типа `T`, созданного конструктором по умолчанию класса `T`;
- `list(int n, T &t)` — создает объект указанного размера `n`, заполненный копиями объекта `t`;
- `list(cont it1, cont it2)` — список создается копированием некоторого диапазона элементов другого контейнера `cont`, выделяемого итераторами `it1` и `it2`.

Как видите, создание списка очень похоже на создание вектора и дека. Например, следующий список `lst1` будет содержать 10 символов "?":

```
list<char> lst1(10, '?');
```

Как добавить элементы в список

Вставлять элементы можно в любое место списка. В нем есть методы добавления элементов в начало и конец списка, а также метод вставки нового элемента в указанное место списка. Добавление элемента в конец списка выполняет метод:

```
void push_back(const T &t);
```

Добавление элемента в начало списка производится методом:

```
void push_front(const T &t);
```

Метод, позволяющий вставить новый элемент `t` перед позицией, на которую направлен итератор `pos`:

```
iterator insert(iterator pos, const T &t);
```

Например, вставку перед первым элементом списка `lst1` можно сделать следующим оператором:

```
lst1.insert(lst1.begin(), 'A');
```

Следующий метод вставляет `n` новых элементов с одним и тем же значением `t` перед позицией, на которую направлен итератор `pos`:

```
iterator insert(iterator pos, size_type n, T &t);
```

Метод, позволяющий вставить элементы из другого списка, вектора или дека в диапазоне итераторов `first`, `last` перед позицией, на которую направлен итератор `pos`:

```
iterator insert(iterator pos, initer first, initer last);
```

Как удалить элементы из списка

Метод `pop_back()` удаляет последний элемент списка. Он не возвращает никакого значения. Метод `pop_front()` удаляет первый элемент списка. Он тоже не возвращает никакого значения. Метод, позволяющий удалить элемент, на который направлен итератор `pos`:

```
iterator erase(iterator pos);
```

Например, первый и последний элементы списка `lst1` можно удалить операторами:

```
lst1.erase(lst1.begin());
lst1.erase(lst1.end() - 1);
```

Следующий метод удаляет элементы списка от `pos1` включительно до `pos2` исключительно:

```
iterator erase(iterator pos1, iterator pos2);
```

Удалить все копии подряд идущих одинаковых элементов списка, кроме одного, можно методом `unique()`. Пусть, например, список `lst1` состоит из элементов: 11, 22, 5, 5, 33, 5, 44, 5, 5, 66. После выполнения оператора `lst1.unique()` в списке останутся элементы: 11, 22, 5, 33, 5, 44, 5, 66.

Чтобы удалить из списка все дубликаты, его надо отсортировать методом `sort()`, а потом применить метод `unique()`. Метод `clear()` удаляет все элементы списка.

Как заменить элемент

Итераторы перегружают операцию обращения к элементу по адресу, поэтому для изменения элемента списка можно записать оператор вида:

```
*(lst1.begin + ind) = 'N';
```

Изменение элементов не характерно для списка.

Как узнать размер списка

Количество элементов в списке всегда можно узнать методом `size()`. Логический метод `empty()` возвращает `true`, если в списке нет ни одного элемента.

Как обратиться к элементу списка

Методы `front()` и `back()` дают первый и последний элементы списка. Список, как и всякий контейнер, определяет итератор. Итератор списка очень похож на итератор вектора. Два метода возвращают его конкретные значения. Метод `begin()` возвращает итератор, направленный на первый элемент списка, метод `end()` — итератор, который находится за концом списка. Таким образом, мы можем определить указатель на начало списка:

```
vector<int>::iterator p1 = lst1.begin();
```

Или, проще:

```
int *p1 = lst1.begin();
```

Можем написать:

```
char ch1 = *p1, ch2 = *p1++;
*p1 = 'B';
```

В отличие от этого, указатель `*p2` не направлен ни на один элемент, он находится за концом списка:

```
int *p2 = lst1.end();
```

Последний элемент списка можно получить только после перемещения указателя, например, если итератор реализует операцию декремента:

```
*p2--;
```

Итераторами, как всякими указателями, удобно пользоваться для прохода по списку. Например, все элементы списка `lst1` можно просмотреть в следующем цикле:

```
for (char *p = lst1.begin(); p != lst1.end(); p++)
   cout << *p << endl;
```

Для удобства просмотра списка в обратном направлении в нем создается обратный итератор — объект класса `reverse_iterator`. Его начальное значение устанавливается на последний элемент списка методом `rbegin()`, а конечное значение — перед первым элементом списка методом `rend()`. Увеличение обратного итератора операцией инкремента означает переход к предыдущему элементу списка. Например, обход и вывод всех элементов списка `lst1` в обратном направлении можно сделать так:

```
for (char *p = lst1.rbegin(); p != lst1.rend(); p++)
   cout << *p << endl;
```

Сортировка списка

В библиотеке STL сортировка элементов контейнера, как правило, выполняется отдельными глобальными функциями. Их аргументами служат итераторы, выделяющие диапазон элементов контейнера, подлежащий сортировке. Этот подход мы рассмотрим в следующей главе. Пока же отметим, что для сортировки требуются итераторы с произвольным доступом к элементам, в которых перегружены обе операции инкремента и декремента.

Списки предназначены для последовательного доступа и редко реализуют такие итераторы. Поэтому у них имеются свои методы сортировки. Следующий метод сортирует список в порядке возрастания элементов:

```
void sort();
```

Шаблон метода сортирует список, сравнивая его элементы методами класса `Cmp`:

```
template<class Cmp> void sort(Cmp cmp);
```

Слияние списков

Методы, позволяющие добавлять к данному отсортированному списку все элементы другого отсортированного списка `lst`, сохраняя упорядоченность списка:

```
void merge(list &lst);
template<class Cmp> void merge(list &lst, Cmp cmp);
```

Список `lst` не копируется, а перемещается. После слияния список `lst` станет пустым. Эти методы можно применять к несортированным спискам, но тогда никакая упорядоченность не гарантируется.

Ассоциативный массив *map*

Обычный массив состоит из элементов, снабженных целыми неотрицательными индексами. Например, в массиве `char a[] = {'G', 'W', 'M', 'Z'}` четыре элемента: `a[0] = 'G'`, `a[1] = 'W'`, `a[2] = 'M'`, `a[3] = 'Z'`.

Можно сказать, что массив состоит из пар "индекс — значение". В нашем примере это пары "0 — G", "1 — W", "2 — M", "3 — Z". Возникает вопрос: почему индексы обязаны быть только целыми неотрицательными числами? Почему им не быть словами или действительными числами? Ведь многие задачи приводят к тому, чтобы хранить пары "ключ — значение", в которых ключ вообще не является числом.

Из таких пар "Фамилия И.О. — номер" состоит, например, телефонный справочник. Еще один пример — анкета. Ее можно представить как совокупность пар: "Фамилия — *Иванов*", "Имя — *Петр*", "Отчество — *Сидорович*", "Год рождения — *1975*" и т. д. Графы анкеты — ключи, в анкету мы вписываем их конкретные значения. Подобных примеров можно привести множество. Информационные технологии должны дать способ хранения и обработки таких пар.

Вернувшись к индексной записи пар "ключ — значение", например, anketa["Фамилия"] = "*Иванов*", anketa["Имя"] = "*Петр*", anketa["Отчество"] = "*Сидорович*", мы получим *ассоциативный массив*. Другие названия этой структуры данных — *словарь* (dictionary) или *отображение* (map), поскольку мы получили однозначное отображение множества ключей на множество значений. Ключи уникальны, они не могут повторяться.

Библиотека STL сначала определяет класс пары "ключ — значение" — шаблон `pair<Key, T>`, описанный в заголовочном файле utility. Ассоциативный массив состоит из объектов этого класса. Например, телефонная книга состоит из объектов класса `pair<string, long>`, анкета — из объектов типа `pair<string, string>`. Создать пару можно конструктором вида:

```
pair<string, long> abonent("Иванов П. С.", 2345670);
```

По умолчанию пара заполняется ключом и значением, полученными конструкторами по умолчанию соответствующих классов. Ключ пары типа `pair<Key, T>` — это поле `first`, а значение — поле `second`. Поэтому к ключу пары `abonent` можно обратиться, как к полю `abonent.first`, а к значению — как к полю `abonent.second`.

Далее, пары в ассоциативном массиве хранятся в порядке возрастания ключей, подобно тому, как в обычном массиве элементы хранятся в порядке возрастания индексов. Поэтому в классе ключей `Key` должен быть какой-то способ упорядочения. В самом простом случае порядок возрастания определяется перегруженной операцией `<`.

В результате оказывается, что шаблон ассоциативного массива `map` зависит от четырех параметров: класса ключей пары `Key`, способа его упорядочения `Cmp`, класса значений пары `T` и распределителя памяти `A`. Поэтому его прототип выглядит следующим образом:

```
template<class Key, class T,
   class Cmp = less<Key>,
   class A = allocator<pair> > class map;
```

У последних двух параметров шаблона есть значения по умолчанию, поэтому при его реализации достаточно указать два параметра — классы ключей и значений.

Как создать ассоциативный массив

В шаблоне класса `map<Key, T>` три конструктора:

- `map()` — конструктор по умолчанию, создает пустой объект;
- `map(const map &m)` — конструктор копирования;
- `map(cont it1, cont it2)` — ассоциативный массив создается копированием некоторого диапазона элементов другого контейнера `cont`, выделяемого итераторами `it1` и `it2`.

Как заполнить ассоциативный массив

В ассоциативном массиве перегружены операции взятия элемента и присваивания, поэтому обычный способ занесения в него элементов похож на определение элементов массива:

```
map<string, long> phoneBook;
phoneBook["Иванов"]  = 2233445;
phoneBook["Петров"]  = 3455430;
phoneBook["Сидоров"] = 9874561;
```

Другой способ — определить пары и воспользоваться методом `insert()`:

```
pair<string, long> abonent1("Иванов",  2233445);
pair<string, long> abonent2("Петров",  3455430);
pair<string, long> abonent3("Сидоров", 9874561);
phoneBook.insert(abonent1);
phoneBook.insert(abonent2);
phoneBook.insert(abonent3);
```

Метод `insert()` возвращает пару типа `pair<iterator, bool>`, в которой значение `second` равно `true`, если пара успешно занесена в массив, а итератор (ключ пары `first`) направлен на него. Это можно использовать, например, так:

```
pair<map<string, long>::iterator, bool>
   p = phoneBook.insert(abonent1);

if (p.second) cout << "Абонент занесен в массив.\n";
else cout << "Абонент уже был в массиве.\n";

map<string, long>::iterator it = p.first; // И т. д.
```

Как узнать размер ассоциативного массива

Количество элементов в массиве можно узнать методом `size()`. Логический метод `empty()` возвращает `true`, если в массиве нет ни одного элемента.

Как получить значение по ключу

Проще всего получить значение элемента ассоциативного массива с помощью индекса:

```
long numb = phoneBook["Иванов"];
string name("Петров");
cout << name << ": " << phoneBook[name];
```

Если элемента с таким ключом в массиве не было, то он будет создан конструктором по умолчанию, имеющимся в классе значений пары. В данном примере в телефонной книге появится элемент с нулевым значением. Ключ и значение пары можно получить итератором:

```
map<string, long>::iterator it = phoneBook.begin();
string name = it->first;
long numb = it->second;
```

Как найти элемент ассоциативного массива

Метод `find(const Key &k)` находит элемент с ключом `k` и возвращает итератор, направленный на него. Если такого элемента нет в массиве, то возвращается итератор, направленный за конец массива, `end()`. Например:

```
map<string, long>::iterator it = phoneBook.find("Иванов");
if (it != phoneBook.end()) cout << "Элемент найден.\n";
else cout << "Такого элемента нет.\n";
```

Как получить все значения массива

Обойти ассоциативный массив и просмотреть все его элементы можно с помощью итераторов. Как обычно, массив предоставляет прямой и обратный итератор методами `begin()`, `end()`, `rbegin()`, `rend()`. Например, просмотреть нашу телефонную книгу можно таким циклом:

```
for (map<string, long>::iterator it = phoneBook.begin();it != phone-
Book.end(); it++)

   cout << it->first <<": " << it-> second << endl;
```

Как удалить элементы

Метод `erase(key)` удаляет пару с ключом `key`. Он возвращает число удаленных элементов. Например:

```
int n = phoneBook.erase("Сидоров");
if (n) cout << "Элемент удален.\n";
else cout << "Такого элемента не было.\n";
```

Метод `erase(iterator)` удаляет пару, на которую направлен итератор. Он не возвращает никакого значения. Например:

```
phoneBook.erase(phoneBook.begin());
```

Метод удаления `erase(it1, it2)` удаляет диапазон элементов, на начало которого направлен итератор `it1`, а на элемент за концом диапазона — итератор `it2`. Например, очистить весь массив можно оператором:

```
phoneBook.erase(phoneBook.begin(),phoneBook.end());
```

Для этого можно использовать специальный метод `clear()`.

Приведем полный пример работы с ассоциативным массивом. В листинге 17.2 показано, как можно использовать класс `map<Key, T>` для создания телефонного справочника.

Листинг 17.2. Телефонный справочник

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

main(int argc, char *argv[]){

    map<string, string> yp;
    yp["John"]   = "123-45-67";
    yp["Lennon"] = "567-34-12";
    yp["Bill"]   = "342-65-87";
    yp["Gates"]  = "423-83-49";
    yp["Batman"] = "532-25-08";

    string name(argv[1]);
    map<string, string>::iterator it = yp.find(name);

    if (it != yp.end())
       cout << name << "'s phone = " << yp[name] << endl;
    else
       cout << "Sorry, no such name" << endl;
}
```

Шаблон класса *multimap*

При обсуждении в предыдущем разделе ассоциативного массива мы говорили, что в нем должна соблюдаться уникальность ключей, так же как в обычном массиве не существует двух различных элементов с одинаковым индексом. Тем не менее, в задачах обработки информации может встретиться необходимость хранения нескольких элементов с одним и тем же ключом. Например, у одного абонента может быть несколько телефонных номеров, в анкете можно указать несколько адресов, учебных заведений или детей. Для решения таких задач в библиотеке STL есть шаблон класса `multimap<Key, T>`.

Шаблон `multimap<Key, T>` очень похож на шаблон ассоциативного массива. Он содержит все методы шаблона `map<Key, T>`, переработанные с учетом многозначности ключей. Например, метод `find(key)` возвращает итератор, направленный на первый из элементов с ключом `key`. Число элементов с ключом `key` можно получить методом `count(key)`.

В шаблоне `multimap<Key, T>` индексация теряет смысл, поэтому операция `[]` в нем отсутствует. Обращение к элементам выполняется итераторами.

Вставка элементов производится методами `insert()`. Можно вставить сколько угодно элементов с одним и тем же ключом, поэтому метод `insert()` возвращает не пару, а просто итератор, направленный на вставленный элемент.

Метод `lower_bound(key)` вернет итератор, направленный на первый элемент с ключом `key`, а метод `upper_bound(key)` вернет итератор, указывающий на следующий за последним элементом с ключом `key`. Поэтому все значения с заданным ключом можно просмотреть в таком цикле:

```
multimap<string, string>::iterator it1 =
   anketa.lower_bound("Имя");
multimap<string, string>::iterator it2 =
   anketa.upper_bound("Имя");

cout << "Имена: " << endl;

for (multimap<string, string>::iterator p = it1; p != it2; p++)
   cout << p->second << endl;
```

Находить итераторы, направленные на начало и конец по отдельности, как сделано выше, неудобно. В шаблоне `multimap<Key, T>` есть метод `equal_range(key)`, возвращающий пару типа `pair<iterator, iterator>`. Ключ этой пары `first` содержит итератор, направленный на первый элемент с ключом `key`, а значение этой пары `second` содержит итератор, направленный на элемент, следующий за последним элементом с ключом `key`. С использованием этого метода предыдущий пример можно переписать так:

```
typedef  multimap<string, string>::iterator  Mit;
pair<Mit, Mit> pit = anketa.equal_range("Имя");

cout << "Имена: " << endl;

for (Mit p = pit.first; p != pit.second; p++)
   cout << p->second << endl;
```

Шаблон класса *set*

Шаблон `set<Key, Cmp, A>` реализует структуру данных, называемую *множеством* (set). Множество — это совокупность неповторяющихся значений. Каждый элемент встречается в нем только один раз. Для множества не характерна упорядоченность, важно только присутствие или отсутствие элемента.

В библиотеке STL множество строится на основе ассоциативного массива. Используется уникальность ключей и множество понимается как ассоциа-

тивный массив, в котором значения просто совпадают с ключами. Каждый элемент понимается как пара "ключ — ключ".

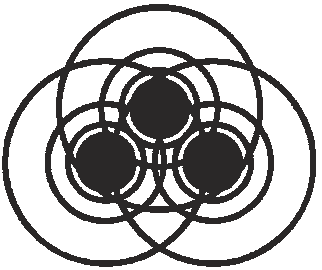
Такой подход позволяет перенести на множества класса `set<Key>` все методы работы с ассоциативным массивом класса `map<Key, Key>` за исключением индексации. Во множестве, как и в векторе, стеке, деке, списке, хранятся только элементы, и в операции `[]` нет никакого смысла.

Упражнения

1. Напишите метод, разбивающий заданный текст на отдельные слова, и сохраняющий их в контейнере. Какой контейнер наиболее удобен для этого?
2. Используя стек класса `stack`, напишите метод, проверяющий парность скобок в арифметическом выражении.
3. Напишите метод, подсчитывающий количество повторений каждого символа в заданном тексте, т. е. создайте частотный словарь символов.
4. Напишите метод, подсчитывающий количество повторений каждого слова в заданном тексте, т. е. создайте частотный словарь слов.

Контрольные вопросы

1. Может ли контейнер содержать элементы разных типов?
2. Можно ли заполнить контейнер элементами одного типа, а потом очистить его и заполнить элементами другого типа?
3. Есть ли ограничения на количество элементов контейнера?
4. Моделируя карточную игру, какой контейнер вы выбрали бы для хранения колоды карт?
5. Какой контейнер лучше всего подходит для хранения шахматных фигур?
6. Что такое итератор?
7. Обязан ли итератор перегружать обе операции: инкремент и декремент?
8. Почему в список класса `list` включены методы сортировки, а в вектор класса `vector` — нет?
9. На основе каких контейнеров можно организовать стек класса `stack`?
10. Какие контейнеры подходят для организации очереди класса `queue`?
11. Что такое ассоциативный массив?
12. Как организовать сортировку контейнера?
13. Как осуществить поиск элемента в контейнере?
14. Какими методами ускоряется поиск в отсортированном контейнере?



Глава 18

Утилиты стандартной библиотеки

Стандартная библиотека шаблонов STL стремится вынести операции, общие для всех контейнеров, за пределы контейнера. Такой подход мы уже видели в итераторах. Для обхода контейнера создается экземпляр итератора, и мы перемещаемся по контейнеру перегруженной операцией ++. Для каждого контейнера операция ++ перегружена по-своему, но нас это не интересует, мы пользуемся ей одинаково во всех контейнерах.

Кроме обхода и просмотра элементов во всех контейнерах требуется сортировка, поиск, вставка и удаление элементов, подсчет числа элементов. Для этого в библиотеке STL есть глобальные, не относящиеся ни к одному классу, функции-утилиты. Их аргументами служат итераторы, направленные на элементы какого-то контейнера. Функции одинаково работают с этими итераторами, не интересуясь, какому контейнеру они принадлежат.

Интересно, что функции-утилиты применимы и к обычным массивам, надо только вместо итераторов подставлять в них указатели на элементы массива. Функции-утилиты находятся в пространстве имен `std` и описаны в заголовочном файле algorithm.

Посмотрим, какие операции выполняют утилиты стандартной библиотеки шаблонов STL.

Сортировка

Функции сортировки, а точнее, шаблоны функций, работают с контейнерами с помощью итераторов. Их аргументами служит итератор на начало множества элементов контейнера, которое надо отсортировать, и итератор,

направленный на элемент, следующий за концом этого множества. Например, для того чтобы отсортировать все элементы контейнера, надо получить его итераторы методами `begin()` и `end()`.

Функциям сортировки нужны итераторы с произвольным доступом, такие какие предоставляет вектор класса `vector<T>`. Заметим, что в списке класса `list<T>` нет такого итератора, поэтому шаблон класса `list<T>` использует для сортировки свои методы `sort()`.

Тип итератора передается как параметр шаблона функции сортировки. Прототип шаблона простейшей функции сортировки выглядит так:

```
template<class It> void sort(It first, It last);
```

Она используется очень просто:

```
vector<string> v(100);
// Как-то заполняем вектор строками.
sort(v.begin(), v.end());
```

Вот пример сортировки массива:

```
int a[] = {8, 5, 6, 2, 4};
sort(a, a + 5);
```

Вторая функция основана на более медленном, но более надежном алгоритме:

```
template<class It> void stable_sort(It first, It last)
```

В частности, она оставляет одинаковые элементы контейнера в их прежнем порядке. Иногда нужно работать только с первыми элементами отсортированного контейнера.

Следующая функция перемещает `middle - first` меньших элементов контейнера, находящихся в диапазоне от `first` включительно до `last` исключительно, в начало этого диапазона, и сортирует только их:

```
template<class It> void partial_sort(It first, It middle, It last);
```

После завершения работы функции отсортированы только элементы, находящиеся в диапазоне от `first` включительно до middle исключительно. Элементы, оставшиеся в диапазоне (`middle`, `last`), по величине оказываются больше всех элементов, перенесенных в начальный диапазон (`first`, `middle`), но они не отсортированы. Например:

```
int a[] = {8, 5, 6, 2, 3, 10, 1, 7, 0};
partial_sort(a, a + 4, a + 9);
```

Массив после выполнения функции примет вид:

```
a[] = {0, 1, 2, 3, 8, 5, 6, 10, 7};
```

Все перечисленные выше функции сортируют контейнер в порядке возрастания его элементов, используя для сравнения перегруженную операцию <. Точнее говоря, они применяют объект шаблона класса-сравнителя less<T>, определенного в заголовочном файле functional. Вот это определение:

```
template<class T> struct less: public binary_function<T, T, bool>{


   bool operator()(const T &x, const T &y) const{
      return x < y;
   }
}
```

Мы подробно рассмотрим такие определения объектов-функций ниже. Пока же отметим, что шаблон класса less<T> унаследован от общего шаблона класса binary_function<T, T, bool> — шаблона объекта-функции с двумя аргументами типа T, возвращающей логическое значение типа bool. Функции или объекты-функции, возвращающие значение типа bool, называются в библиотеке STL *предикатами* (predicates).

Наследование означает, что класс less<T> относится к предикатам с двумя аргументами. Класс объектов-функций перегружает операцию обращения к функции. Перегрузка операции () в шаблоне less<T> сделана так, что получившаяся в результате перегрузки функция возвращает true или false, в зависимости от результата сравнения элементов x и y операцией <. В заголовочном файле functional определено еще несколько предикатов, предназначенных для сравнения. Вот их полный список:

- equal_to — сравнивает элементы на равенство, используя операцию ==;
- not_equal_to — сравнивает элементы на неравенство, используя операцию !=;
- greater — сравнивает элементы по величине, используя операцию >;
- less — сравнивает элементы по величине, используя операцию <;
- greater_equal — сравнивает элементы по величине, используя операцию >=;
- less_equal — сравнивает элементы по величине, используя операцию <=;
- logical_and — выполняет конъюнкцию, используя операцию &&;
- logical_or — выполняет дизъюнкцию, используя операцию ||;
- logical_not — выполняет отрицание, используя операцию !.

Эти и другие предикаты используются следующими функциями сортировки. Они сравнивают элементы каким-нибудь предикатом `Cmp`, заданным как параметр шаблона. Вот их прототипы:

```
template<class It, class Cmp>
void sort(It first, It last, Cmp cmp);

template<class It, class Cmp>
void stable_sort(It first, It last, Cmp cmp);

template<class It, class Cmp>
void partial_sort(It first, It middle, It last, Cmp cmp);
```

Например, сортировку в порядке убывания величины элементов массива можно сделать следующим образом:

```
int a[] = {8, 5, 6, 2, 4};
greater<int> g;

sort(a, a + 5, g);
```

Можно написать свой предикат для сравнения и использовать его для сортировки:

```
template<class T> bool gt(const T &a, const T &b){
   return a > b;
}

sort(a, a + 5, gt<int>);
```

Можно написать предикат не в виде функции, а в виде объекта-функции, о чем речь пойдет немного позднее. В листинге 18.1 приведен пример сортировки вектора.

Листинг 18.1. Сортировка вектора

```
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>

using namespace std;

template<class T> bool gt(const T &a, const T &b){
   return a > b;
}
```

```
main(){
   vector<string> v(10);
   char a[20];

   for (int k = 0; k < 10; k++){

      sprintf(a, "%d", k * k - 3 * k + 1);
      v[k] = a;
      cout << v[k] << "  ";
   }
   cout << endl;

   sort(v.begin(), v.end(), gt<string>);

   for (int k = 0; k < 10; k++)
     cout << v[k] << "  ";
}
```

Поиск

Вторая широко распространенная операция — поиск заданных элементов контейнера — выполняется одной из функций-утилит поиска. Шаблоны функций поиска, как и функции сортировки, получают итераторы на начало (включительно) и конец (исключительно) диапазона поиска. Кроме того, им передается образец для поиска. Функции возвращают итератор, направленный на найденный элемент. Самая простая функция с прототипом находит первое появление значения `sample` в диапазоне элементов от `first` включительно до `last` исключительно:

```
template<class It, class T>
It find(
    It first, It last,
    const T &sample);
```

Во время поиска она сравнивает элементы контейнера с элементом `sample` операцией ==. Поиск в массиве можно сделать так:

```
int a[] = {8, 5, 6, 2, 5, 10, 5, 7, 0};
int n = *find(a, a + 9, 5);
```

В результате `n` будет равным 5, т. е. элемент `a[1]`. Поиск в векторе выполняется аналогично:

```
vector<int> v;
```

```
for (int k = 0; k < 10; k++)
   v[k] = k * k -3 * k + 1;
vector<int>::iterator it = find(v.begin(), v.end(), 2);
if (it != v.end()) cout << *it;
else cout << "Элемент не найден.";
```

Вторая функция поиска, `find_if()`, применяет для поиска и сравнения предикат — объект класса `Pred`. Ее прототип:

```
template<class It, class Pred>
It find_if(
    It first, It last,
    Pred pred);
```

Следующая пара функций завершает поиск, как только в диапазоне от `first1` до `last1` одного контейнера будет найден один из элементов, взятых из диапазона от `first2` до `last2` другого или того же самого контейнера:

```
template<class It1, class It2>
It1 find_first_of(
    It1 first1, It1 last1,
    It2 first2, It2 last2);

template<class It1, class It2, class Pred>
It1 find_first_of(
    It1 first1, It1 last1,
    It2 first2, It2 last2,
    Pred pred);
```

Например, чтобы отыскать в массиве `int a[] = {8, 5, 6, 2, 5, 10, 5, 7, 0}` одно из чисел — элементов массива `int b[] = {2, 6, 5, 7}`, достаточно воспользоваться функцией:

```
int n = *find(a, a + 9, b, b + 4);
```

Получим `n`, равное 6, т. е. элемент `a[2]`. Еще две функции отыскивают совпадения. Они находят в заданном диапазоне первую встреченную пару рядом стоящих одинаковых элементов, возвращая итератор, направленный на первый из двух одинаковых элементов:

```
template<class It>
It adjacent_find(
```

```
   It first, It last);
template<class It, class Pred>
It adjacent_find(
    It first, It last,
    Pred pred);
```

Все предыдущие функции отыскивали только один элемент. Следующие функции отыскивают подпоследовательность, заданную итераторами, в заданном диапазоне элементов контейнера. Они возвращают итератор, направленный на первый элемент найденной подпоследовательности. При неудаче возвращается итератор, направленный на `last1`, т. е. за конец диапазона:

```
template<class It1, class It2>
It1 search(
    It1 first1, It1 last1,
    It2 first2, It2 last2);

template<class It1, class It2, class Pred>
It1 search(
    It1 first1, It1 last1,
    It2 first2, It2 last2,
    Pred pred);
```

Например, чтобы отыскать в массиве `int a[] = {8, 5, 6, 2, 5, 10, 5, 7, 0}` указатель на подпоследовательность `int b[] = {6, 2, 5}`, достаточно воспользоваться функцией:

```
int *p = find(a, a + 9, b, b + 3);
```

Получим `p` равное `a + 2`.

Следующие функции отыскивают не первое, а последнее появление подпоследовательности элементов и возвращают итератор, направленный на начало найденной подпоследовательности, или итератор `last1`, если подпоследовательность не найдена:

```
template<class It1, class It2>
It1 find_end(
    It1 first1, It1 last1,
    It2 first2, It2 last2);

template<class It1, class It2, class Pred>
It1 find_end(
    It1 first1, It1 last1,
```

```
    It2 first2, It2 last2,
    Pred pred);
```

Наконец, последняя пара функций поиска определяет, нет ли в диапазоне от `first` до `last` по меньшей мере `n` подряд идущих элементов `sample`:

```
template<class It, class Size, class T>
It search_n(
    It first, It last,
    Size n, const T &sample);

template<class It, class Size, class T, class Pred>
It search_n(
    It first, It last,
    Size n, const T &sample,
    Pred pred);
```

Бинарный поиск

В отсортированном контейнере можно организовать бинарный поиск. Это значительно ускорит поиск в контейнерах с большим числом элементов. Следующие функции просто показывают, есть ли в контейнере элемент `sample`, возвращая `true` или `false`. Вторую функцию необходимо применять в том случае, когда сортировка была сделана с применением предиката `pred`:

```
template<class It, class T>
bool binary_search(
    It first, It last,
    const T &sample);

template<class It, class T, class Pred>
bool binary_search(
    It first, It last,
    const T &sample,
    Pred pred);
```

Одинаковые элементы в отсортированном контейнере всегда идут подряд. Поэтому следующие функции бинарного поиска возвращают не просто итератор, направленный на найденный элемент, а итератор, направленный на первый (`lower_bound`) или последний (`upper_bound`) из них. Прототипы этих функций выглядят так:

```
template<class It, class T>
It lower_bound(
```

```
    It first, It last,
    const T &sample);

template<class It, class Pred>
It lower_bound(
    It first, It last,
    const T &sample,
    Pred pred);

template<class It, class T>
It upper_bound(
    It first, It last,
    const T &sample);

template<class It, class Pred>
It upper_bound(
    It first, It last,
    const T &sample,
    Pred pred);
```

Функции с тремя аргументами следует применять в том случае, когда элементы контейнера при сортировке сравнивались предикатом `pred`. Еще две функции возвращают итераторы, направленные на начало и за конец диапазона повторений найденного элемента в виде пары итераторов типа `pair<It, It>`:

```
template<class It, class T>
pair<It, It> equal_range(
    It first, It last,
    const T &sample);

template<class It, class Pred>
pair<It, It> equal_range(
    It first, It last,
    const T &sample,
    Pred pred);
```

В листинге 18.2 показано применение этих функций поиска.

Листинг 18.2. Бинарный поиск в векторе

```
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>

using namespace std;

template<class T> bool gt(const T &a, const T &b){
   return a > b;
}

main(){
   vector<int> v(10);

   for (int k = 0; k < 10; k++){

      v[k] = k * k - 3 * k + 1;
      cout << v[k] << "  ";
   }
   cout << endl;

   sort(v.begin(), v.end(), gt<int>);

   typedef vector<int>::iterator Iter;

   if (binary_search(v.begin(), v.end(), 5))
        pair<Iter, Iter> p = equal_range(v.begin(), v.end(), 5);

   for (Iter k = p.first; k != p.second; k++)
     cout << *k << "  ";
   cout << endl;
}
```

Подсчет числа элементов

Многие контейнеры допускают повторение одинаковых элементов. Часто появляется необходимость не только найти элемент, но и узнать, сколько раз он встречается в контейнере. Библиотека STL предлагает для этого функции-утилиты `count()`. Они подсчитывают число элементов с одним и

тем же значением, расположенных в диапазоне, заданном итераторами `first` и `last`. Первая функция с прототипом определяет число элементов со значением `sample`:

```
template<class It, class T>
int count(
    It first, It last,
    const T &sample);
```

Вторая функция сложнее, она определяет число элементов, удовлетворяющих предикату `pred`:

```
template<class It, class Pred>
int count(
    It first, It last,
    Pred pred);
```

Применение этих функций ничем не отличается от применения функций поиска.

Копирование

Следующие функции копируют элементы контейнера, расположенные в диапазоне от `first` включительно до `last` исключительно, в другой или тот же самый контейнер на место, указанное итератором `result`. Они определены с помощью итераторов и выглядят очень просто:

```
template<class From, class To>
To copy(From first, From last, To result){
   while (first != last) *result++ = *first++;
   return result;
}

template<class From, class To>
To copy_backward(From first, From last, To result){
   while (first != last) *--result = *--last;
   return result;
}
```

Копирование с помощью этих функций выполняется следующим образом:

```
main(){
   vector<string> v1(100);
   // Вектор как-то заполняется.
```

```
   vector<string> v2(100);

   copy(v1.begin(), v1.end(), v2.begin());
}
```

Вторую функцию удобно применять в тех случаях, когда итератор `result` направлен на элемент, лежащий в диапазоне (`first`, `last`). Это позволит избежать искажения элементов в процессе копирования.

Замена

Функция `replace()` пробегает диапазон элементов контейнера от итератора `first` до итератора `last` и заменяет каждый встреченный элемент со значением `oldValue`, давая ему новое значение `newValue`:

```
template<class It, class T>
void replace(It first, It last, const T &oldValue, const T &newValue){
   while (first != last){
     if (*first == oldValue) *first = newValue;
     first++;
   }
}
```

Пример ее использования:

```
main(){
   vector<string> v(100);
   // Вектор как-то заполняется.
   replace(v.begin(), v.end(), "Иванова", "Сидорова");
}
```

Вторая функция, `replace()`, использует для поиска заменяемого элемента предикат класса `Pred`. Она выглядит не сложнее предыдущей функции:

```
template<class It, class Pred, class T>
void replace_if(It first, It last, Pred p, const T &newValue){
   while (first != last){
     if (p(*first)) *first = newValue;
     first++;
   }
}
```

Удаление

Функция `remove()` удаляет из заданного диапазона (`first`, `last`) контейнера элементы, указанные своим значением:

```
template<class It, class T>
It remove(It first, It last, const T &value);
```

Для второй функции `remove_if()` надо задать не удаляемый элемент, а предикат, определяющий его:

```
template<class It, class Pred>
It remove_if(It first, It last, Pred p);
```

Обычная реализация этих функций перемещает удаленные элементы в начало контейнера и возвращает итератор, направленный на конец последовательности удаленных элементов. Поэтому размер контейнера не меняется, но его начальные, удаленные, элементы недоступны.

Объекты-функции

Функции-утилиты, рассмотренные в предыдущих разделах этой главы, активно используют предикаты. Мы задавали предикаты в виде логической функции, шаблона функции или как объект класса, описанного в заголовочном файле functional. Подобные классы, называемые по-разному: "объекты-функции", "функторы", "функциональные классы", мы рассмотрим в этом разделе.

Объект-функция — это класс, в котором перегружена операция вызова функции `operator()`. Операцию `operator()`, перегруженную без аргументов, называют *генератором*, перегруженную с одним аргументом — *унарной функцией*, а перегруженную с двумя аргументами — *бинарной функцией*. Таким образом, простейший унарный объект-функция выглядит следующим образом:

```
template<class T> class A{
public:

   void operator()(T &t)(){
      cout << "Простейший объект-функция выводит значение " << t << endl;
   }
};
```

Применять его можно так:

```
main(){
   A<int> a;

   for (int k = 0; k < 10; k++) a(k);
}
```

Для сохранения единства обозначений объекты-функции удобнее наследовать от базовых шаблонов классов, находящихся в пространстве имен `std` и описанных в заголовочном файле functional. Вот как они выглядят:

```
template<class Arg, class Res> struct unary_function{
   typedef Arg argument_type;
   typedef Res result_type;
};

template<class Arg1, class Arg2, class Res> struct binary_function{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Res result_type;
};
```

Как видно из этих описаний, операция вызова функции `operator()` может возвращать значение произвольного типа `Res`, а также значение `void`. В функциях сортировки и поиска применяются предикаты — функции типа `bool`, но в других утилитах могут использоваться функции других типов. С применением наследования простейший объект-функция будет выглядеть так:

```
template<class T> class A: public unary_function<T, void>{
public:

   void operator()(T &t){
      cout << "Простейший объект-функция выводит значение " << t << endl;
   }
};
```

В библиотеке STL есть не только предикаты, но и арифметические объекты-функции `plus`, `minus`, `multiplies`, `divides`, `modulus`, `negate`. Например, объект-функция `plus` определен так:

```
template<class T>
struct plus: public binary_function<T, T, T>{
```

```
   T operator()(const T &x, const T &y) const{
      return x + y;
   }
};
```

Для чего нужно создавать целый класс, используемый как функция? Почему нельзя обойтись простыми функциями или шаблонами функций? Бывают ситуации, в которых нужно дать функции какие-нибудь предварительные значения. Для их хранения удобно использовать поля класса. Продолжим предыдущий пример, внеся в класс `A` поле `a` и конструктор:

```
template<class T> class A: public unary_function<T, void>{
   T a;
public:

   A(T &a){
      this.a = a;
   }

   void operator()(T &t){
      if (t < a)
         cout << "Данное значение " << t << " меньше поля " << a << endl;
      else
         cout << "Значение " << t << " не меньше поля " << a << endl;
   }
};
```

При применении такого объекта-функции нужно задавать параметр конструктора:

```
main(){
   A<int> a(5);

   for (int k = 0; k < 10; k++) a(k);
}
```

Алгоритмы

Приведенные в предыдущих разделах этой главы функции далеко не исчерпывают библиотеку STL. В ней есть еще много функций, реализующих самые разные алгоритмы. Заканчивая обзор библиотеки стандартных шаблонов, приведем еще несколько примеров.

Утилита `for_each()` моделирует цикл. Она пробегает диапазон элементов контейнера, определенный итераторами `first` и `last`, и выполняет унарную функцию `f` с каждым элементом диапазона. Вот прототип этой утилиты:

```
template<class It, class Func>
Func for_each(It first, It last, Func f);
```

Большинство циклов в предыдущих примерах можно было организовать с помощью утилиты `for_each()`.

Утилиты `min()` и `max()` сравнивают два заданных элемента и возвращают меньший или больший из них. Сравнение выполняется перегруженной операцией "меньше", "больше" или предикатом какого-нибудь класса `Cmp`. Вот прототипы утилиты `min()`:

```
template<class T> const &T min(const T &a, const T &b);

template<class T, class Cmp>
const &T min(const T &a, const T &b, Cmp cmp);
```

Прототипы утилиты `max()` выглядят точно так же. Следующие утилиты выбирают наименьший или наибольший элемент из заданного диапазона. Прототипы функции `min_element()` выглядят так:

```
template<class It> It min_element(It first, It last);

template<class T, class Cmp>
const &T min_element(It first, It last, Cmp cmp);
```

Прототипы функции `max_element()` выглядят точно так же.

На этом мы закончим обзор библиотеки STL, но в заключение надо сказать, что в ней есть еще много утилит. Они генерируют элементы контейнера, преобразуют их, меняют их порядок, устраняют дублирование и делают множество других полезных действий. Их количество увеличивается с каждой новой версией библиотеки стандартных шаблонов. Посмотрите документацию к вашему компилятору, и очень вероятно, что вы найдете в библиотеке STL готовое решение вашей задачи.

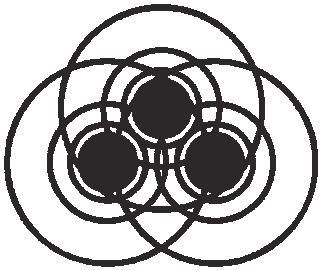
Упражнения

1. Напишите функцию, сортирующую заданную последовательность целых чисел по остатку от деления их на число `n`: сначала числа, делящиеся на `n`, потом числа, дающие 1 в остатке, потом числа, дающие в остатке 2, и т. д.
2. Определите число элементов каждой группы, полученной в упражнении 1.

3. Организуйте сортировку последовательности слов по числу символов в них.
4. Отсортируйте контейнер, содержащий строки символов, по числу пробелов, содержащихся в строке.
5. Отсортируйте контейнер, содержащий строки символов, по числу гласных букв, содержащихся в строке.
6. Напишите функцию, отыскивающую в заданном контейнере, содержащем строки символов, все строки, в которых есть заданное слово.

Контрольные вопросы

1. В каком порядке сортирует элементы контейнера функция `sort()`?
2. В каком порядке сортирует элементы контейнера функция `stable_sort()`?
3. В каком порядке сортирует элементы контейнера функция `partial_sort()`?
4. Как задать сортировку элементов контейнера в убывающем порядке?
5. Как задать нужный вам порядок сортировки в функциях сортировки?
6. Почему в классе `list<T>` есть свои методы сортировки, а в классе `vector<T>` их нет?
7. Что возвращает функция `find()` при успешном результате поиска?
8. Что возвращает функция `find()`, если элемент не найден?
9. Как определить число одинаковых элементов, найденных какой-либо функцией поиска?
10. Какое преимущество у объектов-функций по сравнению с обычными функциями?
11. В каких случаях удобнее задавать обычный цикл, а в каких удобнее пользоваться функцией `for_each()`?



Глава 19

Жизненный цикл программы

Мы уже изучили достаточно много, чтобы понять, что невозможно один раз написать программу, откомпилировать ее и забыть о сделанной работе. Во-первых, практически невозможно написать программу без ошибок. Большинство из них, к счастью, обнаруживается компилятором. Современные компиляторы не только находят синтаксические ошибки, но и указывают место их обнаружения. Вы возвращаетесь к исходному тексту программы и исправляете его. Чаще всего это только несколько небольших поправок, но иногда приходится переделывать большие куски программы.

Во-вторых, после того как исправлены ошибки, выявленные во время компиляции, и программа, наконец, заработала, часто обнаруживается, что она делает не то, что предполагалось. Скажем, программа вычисляет вес человека, а он будет равен тонне или, напротив, сотне граммов. Расстояние до Луны окажется равным паре метров, а расстояние до ближайшего магазина — сотне километров. В таких ошибках виноват неправильный алгоритм решения задачи. Для устранения дефекта приходится возвращаться к тексту программы и изменять заложенный в нее алгоритм.

В-третьих, после того как программа полностью отлажена и дает верные результаты, выясняется, что требования к ней изменились. Посмотрев готовую программу, заказчик, как правило, хочет расширить ее возможности, улучшить интерфейс с пользователем, переделать что-то еще. Для этого снова придется обращаться к исходному тексту программы.

В-четвертых, жизнь не стоит на месте, и уже после некоторого времени эксплуатации программы приходится изменять ее, чтобы подогнать под изменившиеся условия. Например, модифицировать программу под новую операционную систему. Для этого снова приходится пересматривать исходный текст программы.

Наконец, жизнь меняется настолько, что программа окончательно устаревает и ее приходится заменять новой программой.

Чтобы программа была способна выдержать все эти испытания, в нее надо внести дополнительные возможности, превращающие простую программу в *программный продукт*. Программный продукт не создается одновременно. Он проходит целый жизненный цикл, состоящий из нескольких этапов: проектирования, разработки, отладки, сопровождения, модификации. Рассмотрим эти этапы, но сначала посмотрим, чем программа отличается от программного продукта.

Программный продукт

Каждый программист хочет, чтобы его программу использовали как можно больше людей на самых разных машинах. Для этого он должен сделать программу удобной для использования. Конечно, программу надо тщательно протестировать и отладить. Она обязательно должна давать верные результаты. Необходимо предусмотреть и обработать все исключительные ситуации, чтобы программа не давала неожиданные сообщения об ошибках. Пользователю надо дать удобный и привычный интерфейс, чтобы он мог сразу же начать работу с программой, а не ломать голову, думая как сделать то или иное действие. В программе следует предусмотреть как можно более широкие возможности: взаимодействие с большим количеством устройств ввода-вывода, способность работать в разной операционной среде. Программу надо снабдить оперативной справкой (Help) и документацией, объясняющей ее особенности и условия эксплуатации. По крайней мере, к программе надо приложить файл README с краткими пояснениями по ее установке и использованию. Наконец, программу надо сделать легко устанавливаемой (install) на компьютер и так же легко и полностью удаляемой из него.

Программа, выполняющая эти требования, становится *программным продуктом*, который можно распространять под какой-нибудь лицензией и длительное время сопровождать, выпуская модификации и новые версии продукта. Все файлы, составляющие программный продукт, обычно упаковываются в один или несколько архивных файлов, называемых *дистрибутивом*, и в таком виде передаются потребителю.

Программный продукт может стать компонентом *программного комплекса* — набора тесно связанных программ, вместе решающих сложную задачу. Самый крупный программный комплекс на вашей машине — это операционная система. Она состоит из сотен компонентов.

Требования к компонентам программного комплекса еще выше, чем требования к отдельному программному продукту. Компоненты надо согласовать между собой так, чтобы они работали как единое целое. Для этого выраба-

тываются стандартные интерфейсы между компонентами, и разработчик программного продукта ограничивается рамками этих интерфейсов.

Кроме того, многие компоненты программного комплекса работают одновременно, что ограничивает ресурсы, выделяемые каждому из них. Разработчик компонента вынужден учитывать эти ограничения, что еще больше осложняет его задачу. Программист, участвующий в разработке программного продукта или компонента программного комплекса, должен уметь работать в команде, руководствоваться документами, регламентирующими разработку, строго выполнять все требования, предъявляемые к разработке.

Определение требований к продукту

Программный продукт предназначен для решения поставленной перед ним задачи или даже нескольких задач по обработке информации. Чем точнее сформулирована задача, тем точнее будет решать ее программный продукт. Поэтому разработку программного продукта следует начинать с четкого и как можно более полного определения требований к нему.

Дело осложняется тем, что заказчик программного продукта часто не может полностью сформулировать то, что он хочет получить. Его требования ограничиваются общими фразами вроде: "Сделайте мне систему, такую же, как CoolProduct, но рассчитанную на наши правила товарооборота и отчетности". Получив же такую систему, заказчик чаще всего восклицает: "Но я совсем не это имел в виду!" Поэтому требования к программному продукту вырабатываются в результате долгих обсуждений и уточнений. Зачастую они изменяются уже в ходе разработки, хотя этого надо всячески избегать.

Требования, которым должен удовлетворять продукт, можно отнести к нескольким категориям. Это требования к функциональности продукта, к надежности его функционирования, условия эксплуатации продукта, технические средства, необходимые для работы продукта. Рассмотрим подробнее каждое из этих требований.

Требования к функционированию продукта

Самое главное требование к программному продукту заключается в том, чтобы он решал поставленную задачу в максимальном объеме и с максимально возможной точностью. Потребителя продукта редко интересует алгоритм решения задачи. Ему просто нужно получить результаты как можно точнее и быстрее. Более того, потребитель хочет, чтобы управление программой было простым и привычным для него. Ему надо сразу сесть и начать работу с программой, у него нет времени на обучение и нет желания долго вникать в особенности продукта и изучать новые команды для работы с ним.

Современные программные продукты взаимодействуют с пользователем с помощью графического интерфейса. Обычный сеанс работы с продуктом выглядит следующим образом. При запуске продукта на экране дисплея открывается окно, содержащее различные графические компоненты (controls, widgets): поля ввода исходных и промежуточных данных, кнопки выбора, раскрывающиеся списки, из которых можно выбрать нужные пункты, командные кнопки подтверждения или отмены действий и другие элементы управления программой. Пользователь вводит исходные данные или выбирает их из предложенного списка и запускает механизм решения задачи.

В процессе решения у него запрашиваются дополнительные сведения, на экран выводятся промежуточные результаты, в строке состояния появляются различные сообщения системы. При этом часто открываются дополнительные окна со своим набором управляющих графических элементов или с сообщениями. Если решение задачи занимает много времени, то на экран выводится индикатор прогресса в виде так называемого "градусника" или "песочных часов".

Наконец, результаты работы программного продукта выводятся на экран дисплея, и предлагается записать их файл в том или ином формате, или вывести на печать.

На этапе определения требований подробно описывается графический интерфейс пользователя: задается необходимое количество окон, состав управляющих элементов в каждом из них, расположение элементов в окне. Описывается состав каждого управляющего элемента и действия, которые программный продукт должен выполнить при выборе пользователем этого элемента.

Все это активно обсуждается с заказчиком, при этом рисуются эскизы окон, которые претерпевают множество изменений перед тем, как начинают удовлетворять обе стороны. Обсуждение часто доходит до таких частностей, как цвет фона окна и гарнитура шрифта. Чтобы наилучшим образом удовлетворить требования заказчика, в команду разработчика программного продукта часто включается профессиональный дизайнер, в задачу которого входит выработка стиля оформления продукта и рисование окон в этом стиле.

Плодотворный метод определения функциональных требований разработал Айвар Якобсон (Ivar Jacobson). Этот метод входит в так называемый *унифицированный процесс разработки* (RUP, Rational Unified Process), предлагаемый фирмой Rational Software Corporation. Якобсон предложил начинать определение требований с составления всех возможных *действий* (actions) некоторого *действующего лица* (actor) с программным продуктом, с начала сеанса его работы до окончания сеанса. Действующим лицом может быть не только пользователь, но и какой-то алгоритм, выполняющийся в программном продукте, и какая-то внешняя программа.

Каждый такой набор действий образует *сценарий* работы действующего лица. Пользователь может идти разными путями к достижению результата, поэтому он может выполнять разные сценарии для достижения одной и той же цели.

Набор всех сценариев, приводящих к одному результату, называется *use case*. На русский язык этот термин переводят по-разному: прецедент, сценарий использования, вариант использования. Сценарии, образующие прецедент, называются *экземплярами* (instances) прецедента. Вначале в прецеденты вносятся все сценарии, в том числе и неудачные. Затем сценарии каждого прецедента анализируются, обобщаются и собираются в один-два наилучших сценария.

Например, пользователь Интернета может создать экземпляр прецедента "Загрузить файл с сайта", набирая адрес этого сайта в строке адреса браузера. Другой экземпляр прецедента будет создан щелчком кнопки мыши по ссылке в окне браузера. Как видите, в браузере реализовано несколько сценариев прецедента "Загрузить файл с сайта". Набор другого адреса — это другой сценарий того же прецедента. Если адрес верен и загрузка файла допустима, то сценарий завершится успешно, в противном случае сценарий завершится неудачей.

После четкого определения всех прецедентов и выделения самых успешных сценариев входящие в них действия выражаются в графических элементах управления. Элементы управления размещаются в одном или нескольких окнах таким образом, чтобы естественное перемещение по этим элементам и окнам приводило к выполнению наилучших сценариев.

Требования к надежности продукта

Программные продукты решают самые разнообразные задачи. Некоторые задачи, такие как управление химическими процессами, управление движением самолета или космического корабля, жизнеобеспечение тяжелобольного человека, или человека, находящегося в суровых условиях, требуют повышенной надежности функционирования.

Надежность обеспечивается, в первую очередь, конечно, выбором особо надежной аппаратуры, но и программное обеспечение играет здесь огромную роль. В программу следует заложить по возможности простой и прозрачный алгоритм решения задачи, в правильности которого можно быть уверенным. В алгоритм следует включить обработку всех исключительных ситуаций. Разумеется, программа должна быть тщательно отлажена и протестирована. Это, кстати говоря, увеличивает сроки и стоимость ее разработки.

Надежность работы часто обеспечивается дублированием аппаратуры. В таком случае программный продукт должен обеспечивать двойной контроль, сравнение параллельно вычисленных результатов и действия при их несовпадении, своевременное переключение на резервную аппаратуру.

В требованиях к такому программному продукту описываются средства повышения его надежности: тщательная и всесторонняя проверка входных и выходных данных, промежуточный контроль, дублирование вычислений. Здесь же устанавливается гарантированное время реакции на события, время восстановления продукта после сбоя, четко определяются необходимые операции по восстановлению продукта.

Условия эксплуатации продукта

Программный продукт никогда не работает на компьютере в одиночестве. Он действует в окружении других программ и опирается на ту или иную операционную среду. В основе среды лежит операционная система, установленная на компьютере. При разработке всегда подразумевается какая-нибудь операционная система, в которой продукт будет работать. Хорошо, если будет обеспечено взаимодействие с разными версиями операционной системы. Еще лучше, если разработка ведется сразу для нескольких операционных систем.

Операционную систему надо настроить на корректное выполнение продукта. Может потребоваться какой-то определенный выбор значений или определение новых системных переменных для оптимизации работы. Иногда требуется серьезное изменение конфигурации операционной системы.

Часто программный продукт взаимодействует с другими программами. Ему может потребоваться текстовый редактор, электронная таблица, средства электронной почты. Такие программы должны быть предварительно установлены на компьютер, или их установка должна запрашиваться при инсталляции продукта.

Все эти условия должны быть оговорены при определении требований к продукту. Конечно, лучше всего, если не требуется никаких дополнительных средств и конфигурирования. К сожалению, это встречается редко. Поэтому важно сразу же уточнить требования к условиям эксплуатации продукта и придерживаться их в процессе разработки.

Требования к техническим средствам

Каждый программный продукт использует ресурсы компьютера: оперативную память, дисковое пространство, сетевые средства, устройства ввода-вывода. Некоторые программы довольствуются стандартной и наиболее распространенной конфигурацией компьютера, другим требуются дополнительные ресурсы. Завышенные требования сужают область применения программного продукта, хотя зачастую улучшают и ускоряют его работу.

Следует стремиться к уменьшению требований к конфигурации и мощности техники, на которой будет работать будущий программный продукт, но, в

любом случае, надо в самом начале работы как можно точнее выразить их. Это поможет в правильном выборе алгоритма решения задачи и настройке его на заданную конфигурацию.

Требования к установке продукта

При установке программного продукта часто нужно проверить конфигурацию компьютера, изменить ее в некоторой степени, создать каталоги и конфигурационные файлы, сделать запись в системный реестр. Эти действия могут вступить в противоречие с системой безопасности компьютера. Поэтому надо сразу оговорить все возможные изменения, вносимые программным продуктом, определить права доступа к каталогам и файлам, права пользователя, от имени которого производится установка.

Лучше всего, если программный продукт вписывается в стандартную конфигурацию компьютера. Если же для работы требуется особая настройка, то ее надо оговорить в требованиях к установке.

Техническое задание

После определения требований к продукту вырабатывается *техническое задание* (ТЗ) на его разработку. Это очень важный документ. Он должен содержать как можно более подробное описание будущего программного продукта, тщательно согласованное между заказчиком и разработчиком. В нем четко выражаются все требования, записанные не только обычным текстом, но и снабженные чертежами и рисунками.

Техническое задание содержит, кроме того, полное описание графического интерфейса пользователя с приложением рисунков всех окон и управляющих графических компонентов, расположенных в них. Также должны быть перечислены все отчеты, которые будет предоставлять программный продукт, с описанием всех полей, а также электронная или бумажная форма их представления.

Кроме требований к программному продукту, техническое задание обычно содержит требования к документации и оперативной справке, этапы разработки и сроки завершения каждого этапа, а также форму отчетности по каждому этапу. Иногда включается и порядок сдачи-приемки программного продукта, условия его сопровождения и модификации, хотя чаще это оформляется отдельными документами.

В идеале техническое задание должно полностью содержать все требования к будущему продукту и детально описывать их, тогда команде разработчиков остается только в точности его выполнить. Увы, действительность далека от этого идеала. Реальное положение вещей лучше всего отражает шутливый афоризм: "В жизни все не так, как на самом деле". Техническое задание часто меняется в

процессе разработки по многим причинам. Иногда при определении требований упускаются какие-то важные детали. Иногда в середине разработки выясняется, что какую-то конструкцию невозможно встроить в программный продукт. Часто оказывается, что требования устарели, появилась новая операционная среда или новая конфигурация компьютера. Со временем меняется положение дел у заказчика, его фирма расширяется или сокращается, фирма перепрофилируется, вследствие этого уменьшается финансирование проекта или меняются сроки его выполнения.

Поэтому, стремясь к полноте технического задания, следует предусмотреть возможные изменения. Надо быть готовым к этому и проектировать разработку так, чтобы можно было как можно легче вносить изменения в программный продукт. Опытные в таких ситуациях менеджеры часто специально составляют техническое задание в самых общих выражениях, чтобы получить простор для маневра в процессе разработки.

Упражнения

1. Определите требования к программе чтения файлов с диска. Какие типы файлов она будет различать? Как будут записываться имена файлов? Какой интерфейс вы предложите пользователю?
2. Вам предложили автоматизировать библиотечный каталог. Опишите требования к этому программному продукту.
3. Опишите требования к автоматизации работы деканата.
4. Опишите требования к информационной системе вашего факультета.

Проектирование программного продукта

После определения всех требований, составления и утверждения технического задания можно переходить к следующему этапу — проектированию разработки программного продукта. На этом этапе набирается команда разработчиков и назначается менеджер проекта (project manager). Разработчики под руководством менеджера проекта, прежде всего, анализируют техническое задание и формулируют записанные в нем требования к продукту на своем языке. Это может быть более детальное описание в терминах программирования, запись на псевдокоде или на специальном языке описания интерфейсов. Затем они выбирают удобную для себя среду разработки, язык программирования, подыскивают или конструируют алгоритм решения задачи, подбирают структуры данных, способы хранения и представления данных. В результате возникает проект создания программного продукта, в котором отражается план необходимых для этого работ.

Редко удается сразу разработать полный и всеобъемлющий проект. Обычно разработка проходит несколько этапов, на которых проект уточняется и детализируется. Многие стандарты разработки различают эскизный проект, технический проект и рабочий проект. Но работа по созданию проекта начинается с анализа поставленной задачи.

Анализ

Классический метод анализа рассматривает создаваемый продукт как информационную систему или систему управления. Слово "система" очень общее, ему трудно дать точное определение. Для анализа системы важно, что она состоит из нескольких относительно независимых подсистем. Например, информационная система факультета может состоять из подсистемы, обеспечивающей учебный процесс во главе с деканатом, подсистемы, обеспечивающей научную деятельность факультета во главе с ученым советом, подсистемы, обеспечивающей социальные нужды студентов и сотрудников.

Подсистема состоит из одного или нескольких модулей. Каждый модуль обеспечивает решение одной или нескольких задач, взаимодействуя при этом с другими модулями подсистемы. Например, подсистема учебного процесса факультета может состоять из модулей "деканат", "кафедра", "библиотека", "компьютерный класс", "методическая комиссия".

В результате такого анализа выявляется структура системы, поэтому он носит название *структурного анализа* (structured analysis). Есть несколько десятков различных методик структурного анализа, приспособленных к анализу систем разного типа. Наиболее широкое распространение получила методика SADT (Structured Analysis and Design Technique), разработанная в конце шестидесятых годов Дугласом Россом в Массачусетском технологическом институте. В начале семидесятых годов она была с успехом применена в крупном аэрокосмическом проекте. Затем методика SADT использовалась в проектировании телефонных сетей, в системах автоматизированного проектирования (САПР), в разработке автоматизированных систем управления различными производственными процессами. Она до сих пор применяется для анализа самого широкого круга задач и реализации многих проектов.

Методика SADT была применена ВВС США для реализации программы компьютеризации промышленности, названной ICAM (Integrated Computer Aided Manufacturing). В ходе выполнения этой программы потребовалось разработать стандарт взаимодействия промышленных систем. Этот стандарт получил название IDEF (ICAM DEFinition). Оказалось, что он хорошо подходит для анализа и проектирования самых разных деловых и производственных процессов. Стандарт IDEF быстро развивался. На его основе появились разновидности IDEF0, IDEF1, IDEF1X, IDEF3, IDEF4, IDEF5, ABC, приспособленные для решения специфических задач. Каждая разновидность стандарта IDEF уже выпущена в нескольких версиях

Результаты структурного анализа оформляются графически в виде нескольких диаграмм. Каждая методика предлагает свой набор диаграмм и свои правила их оформления, но почти во всех методиках есть диаграммы потоков данных DFD (Data Flow Diagrams), диаграммы переходов состояний STD (State Transition Diagrams), диаграммы сотрудничества ERD (Entity-Relationship Diagrams). Фирмы, вырабатывающие методики, предлагают и свои графические инструменты для их вычерчивания. Большое распространение получили программные продукты ERWin, BPWin фирмы Computer Associates, Rational Rose фирмы Rational Software Corporation, серия программных продуктов SILVERRUN одноименной фирмы.

В последнее время все большее распространение получает *объектно-ориентированный анализ* (OOA, Object-Oriented Analysis), активно развиваемый фирмой Rational Software Corporation. Этот метод анализа опирается на RUP (Rational Unified Process). Он начинает с составления всех возможных прецедентов (use cases) работы будущего программного продукта и создает аналитическую модель. Аналитическая модель, как следует из названия методики, определяет объекты, с которыми будут работать сценарии, входящие во все прецеденты. В число этих объектов, прежде всего, попадают сущности, которыми оперирует программа. Примерами сущностей могут служить списки сотрудников учреждения, его штатное расписание, банковские счета, каталог библиотеки и т. д. Действующие лица (actors) сценариев тоже становятся объектами. Кроме них, выделяются объекты, реализующие действия (actions), выполняемые действующими лицами, и объекты, связывающие действующие лица между собой.

Далее, объекты описываются классами. Выделяются классы сущностей, граничные классы, описывающие взаимодействия различных объектов, управляющие классы, описывающие объекты управления другими объектами. При этом сразу рисуются диаграммы классов, а также диаграммы взаимодействий объектов. В объектно-ориентированном анализе выработано много разновидностей диаграмм, но наиболее часто создаются диаграммы кооперации объектов и диаграммы последовательностей.

Для облегчения и автоматизации анализа разработаны различные программные продукты. Например, фирма Rational Software Corporation предлагает средство анализа и проектирования Rational Rose. С его помощью можно быстро нарисовать нужные диаграммы и даже создать исходный текст программы на одном из наиболее распространенных языков программирования.

Разработка эскизного проекта

Эскизный проект возникает как результат анализа требований, предъявленных к программному продукту. В нем в общем виде формулируются указания по созданию программного продукта. Здесь ставится задача для каждого

разработчика, описываются алгоритм решения задачи, способы взаимодействия создаваемого продукта с другими программами и устройствами ввода-вывода, выбираются структуры данных, определяются способы хранения данных на диске или в базе данных.

Эскизный проект не может быть слишком большим. Он должен быть обозримым, схематичным, четко показывающим основные этапы создания программного продукта. Обычно эскизный проект содержит не больше 5—6 страниц текста. К нему прилагаются диаграммы, рисунки и чертежи, а также календарный план выполнения проекта.

После того как эскизный проект создан, он раздается всем участникам разработки для изучения и обсуждения. Каждый разработчик обдумывает свой участок проекта, вносит свои предложения и дополнения, конкретизирует план выполнения проекта.

Разработка технического проекта

После изучения эскизного проекта всеми заинтересованными лицами наступает время создания технического проекта. В его обсуждении принимает участие вся команда разработчиков под руководством менеджера проекта. Каждый разработчик вносит свои предложения по реализации и улучшению проекта, уточняет и детализирует относящиеся к нему положения проекта, согласует интерфейсы с другими разработчиками.

Технический проект будет рабочим документом на все время реализации проекта, поэтому он должен быть понятен и приемлем для всех программистов. В нем не должно быть недомолвок, двусмысленностей, не должно оставаться пробелов и недоговоренностей.

При разработке технического проекта окончательно определяется конфигурация технических средств, и вся дальнейшая работа ведется с учетом этой конфигурации. Уточняется операционная среда, в которой будет функционировать программный продукт, и системное программное обеспечение. Например, Web-приложение работает в браузере. Браузеры по-разному интерпретируют языки HTML и JavaScript, поэтому надо сразу решить, будет ли программный продукт рассчитан на определенный браузер или он должен работать в любом. В первом случае разработчики могут включить в продукт дополнительные возможности языков HTML и JavaScript, интерпретируемые данным браузером, во втором — должны использовать только стандартные конструкции, что может значительно затруднить разработку.

В техническом проекте уточняются типы и структуры исходных и промежуточных данных, полностью детализируется алгоритм решения задачи. Задача разбивается на модули, которые распределяются среди программистов.

При объектно-ориентированном проектировании в техническом проекте определяются все объекты, необходимые для осуществления проекта и выявляются связи между ними. Полностью выписывается строение каждого

объекта, его поля и методы. Объекты записываются в виде интерфейсов или абстрактных классов, дальнейшая разработка которых поручается конкретным программистам.

После проработки технического проекта каждым участником разработки собираются и обобщаются их уточнения и замечания. Окончательная версия проекта обсуждается командой разработчиков. Менеджер проекта выносит технический проект на утверждение руководством фирмы-разработчика и заказчиком программного продукта. После этого технический проект становится рабочим проектом для группы разработчиков.

Рабочий проект

После утверждения технического проекта он становится основным рабочим документом для команды разработчиков программного продукта. Рабочий проект — это большой, подробный документ, наиболее полно описывающий будущий программный продукт и план его создания. В нем содержатся детальные указания каждому разработчику и команде в целом, определена структура базы данных и других хранилищ данных, которой будут руководствоваться все разработчики. Короче говоря, в рабочем проекте должны содержаться все сведения, нужные каждому разработчику и команде в целом. В частности, в нем должны быть записаны этапы и сроки разработки, чтобы каждый программист твердо знал их.

При объектно-ориентированном проектировании в рабочем проекте должны быть полностью описаны все классы и связи между ними. Это описание можно сделать в виде абстрактных классов или интерфейсов, на языке разработки или на языке описания. Важно, чтобы все участники проекта правильно понимали эту запись и одинаково интерпретировали ее.

Каждому участнику проекта выдается экземпляр рабочего проекта. При всяком изменении рабочего проекта участники получают его новую версию. В настоящее время с развитием Web-технологии, как правило, создается собственный сайт для каждого проекта. Все рабочие документы публикуются на этом сайте, а при каждом их изменении участники проекта получают уведомление по электронной почте.

Упражнения

5. В упражнении 1 этой главы определены требования к программе чтения файлов с диска. Разработайте проект создания этой программы.
6. Разработайте проект автоматизации библиотечного каталога.
7. Проведите анализ работы деканата и разработайте проект его автоматизации.
8. Проанализируйте информационные потоки вашего факультета и спроектируйте компьютерную систему их обработки.

Разработка программного продукта

Процесс разработки программного продукта полностью определяется рабочим проектом, в котором записаны этапы и сроки разработки, а также отчетность, которую должен предоставить каждый разработчик на каждом этапе. Менеджер проекта полностью контролирует процесс разработки, просматривая отчеты и корректируя при необходимости выполнение текущего этапа. Помимо запланированных отчетов менеджер может запрашивать у разработчиков краткие еженедельные и даже ежедневные отчеты.

Разработка программного продукта часто начинается с создания его работающего прототипа, содержащего минимальную функциональность и строгое оформление пользовательского интерфейса. Прототип демонстрируется заказчику. Заказчик одобряет его или вносит свои замечания, изменения и дополнения. Рабочий проект корректируется в соответствии с этими замечаниями и процесс разработки продолжается.

Каждый этап разработки завершается показом результатов заказчику. В рабочем проекте определяется набор приемо-сдаточных испытаний продукта на каждом этапе. Этот набор выполняется в присутствии заказчика. Просмотрев результаты приемо-сдаточных испытаний, заказчик, как правило, вносит свои замечания в проект. Рабочий проект корректируется и разработка продолжается.

Качество и своевременность разработки во многом определяются качеством ее планирования. Тщательно написанный рабочий проект дает возможность программистам сосредоточиться на выполнении проекта, равномерно распределить свои усилия, не отвлекаться на переделки и доработки. Проект, полностью согласованный с заказчиком, почти не меняется. Это позволяет строго следовать этапам разработки, переходить от одного этапа к другому, не возвращаясь к уже законченному и полузабытому коду, написанному на предыдущих этапах.

Как ни печально, в нашем мире нет ничего идеального, и разработка программного продукта не исключение. На практике рабочий проект в ходе разработки меняется, причем меняется чаще и сильнее, чем хотелось бы команде разработчиков. Грамотно составленный проект должен учитывать возможные изменения и дополнения и выделять какое-то время на их выполнение. Однако очень трудно предугадать момент, в который потребуется переделка, поэтому даже опытные разработчики проектов попадают в цейтнот, из которого приходится выходить напряженными усилиями всей команды. Это приводит к спешке и создает нервную обстановку, а в такой обстановке увеличивается вероятность появления ошибок. На исправление ошибок требуется дополнительное время, которого не хватает. Трудности нарастают как снежный ком, в результате страдает качество программного продукта и срываются сроки выполнения проекта. Остается еще раз напомнить о важности этапа проектирования программного продукта.

Отладка

Практически невозможно написать программу без ошибок. Справедливость банальной фразы: "Человеку свойственно ошибаться" начинаешь остро чувствовать, уже написав всего 2—3 программы. Не стоит удивляться тому, что в написанной вами программе есть ошибки. Напротив, отсутствие ошибок вызывает удивление. Поэтому, приступая к разработке, следует сразу же предположить, что придется регулярно исправлять сделанные в ней ошибки, и выделить время на их обнаружение и устранение.

Процесс нахождения и ликвидации ошибок называется *отладкой* (debugging). Интересно происхождение английского термина, который буквально переводится как "устранение жуков". Он возник на заре развития компьютеров, когда никак не могли найти ошибку в одной программе. После долгих поисков оказалось, что в аппаратуру заполз жучок и замкнул электрические проводники таким хитрым образом, что это вызывало ошибку. С тех пор ошибки в программе на жаргоне называются "багами".

Отладка отнимает довольно много времени. По разным оценкам на нее уходит от трети до половины всего времени реализации проекта, поэтому следует серьезно отнестись к этому процессу.

Чтобы уменьшить время отладки, следует уже во время проектирования программного продукта предусмотреть средства, облегчающие и ускоряющие отладку. Прежде всего, надо выбрать как можно более простой и понятный алгоритм решения задачи, чтобы можно было правильно его запрограммировать, а в случае ошибки проследить его выполнение и найти ошибку.

Следует серьезно отнестись к выбору языка программирования. Низкоуровневые языки, такие как язык ассемблера или языки вроде C/C++, дают программисту очень много возможностей, но неразумное использование этих возможностей приводит к очень большому количеству ошибок. Эти ошибки, к тому же, трудно обнаружить и устранить. Языки высокого уровня, такие как Java, C# и другие, напротив, ограничивают действия программиста, но предотвращают появление ошибок.

Очень важно выбрать правильный стиль программирования, единый для всей команды разработчиков. Применение проверенных стандартных конструкций языка резко уменьшает возможность ошибиться. Кроме того, программный код, написанный в едином стиле, легко может прочитать каждый разработчик, а известно, что чужие ошибки легче выявить, чем свои.

Наконец, надо выбрать программный инструмент разработчика, помогающий предотвратить ошибки и легко обнаружить их появление. К таким продуктам относятся отладочные компиляторы, интегрированные с отладчиками среды разработки, специальные программы-верификаторы исходно-

го кода и программы-отладчики машинного кода. Правильный выбор программного средства значительно облегчит последующую отладку.

Ввиду особой важности отладки программного продукта мы рассмотрим этот процесс подробнее в следующей главе.

Тестирование

Некоторые ошибки нелегко выявить, поэтому на каком-то этапе разработки необходимо остановиться и провести тщательное и всеобъемлющее *тестирование* (testing) программного продукта. Тестирование заключается в том, что работа программы проверяется на ряде задач, решение которых известно заранее. Результаты работы программы сравниваются с эталонными результатами, и на основании их совпадения делается вывод о применимости программы к этому классу задач. Кроме того, в процессе тестирования часто проверяется исходный текст программы, что позволяет найти ошибки в алгоритме решения задачи.

Разумеется, тестирование не может доказать отсутствие ошибок, оно только выявляет их наличие. Нельзя сказать, что программа, успешно прошедшая тестирование, свободна от ошибок. Тем не менее, после серьезного тестирования можно запускать программу в промышленную эксплуатацию с большой долей уверенности. Практика показывает, что полностью протестированный программный продукт успешно справляется со своей задачей.

Тестирование не следует путать с отладкой. Задача тестирования — выявить наличие дефектов в программе, а задача отладки — отыскать их местоположение и устранить.

Набор тестов следует подготовить еще до проектирования программного продукта, на этапе определения требований к нему. Именно требования к программе и составляют круг тех задач, которые должны быть выражены в тестовых заданиях. Кроме того, спроектировав систему, уже трудно беспристрастно оценивать ее. Направление мысли волей-неволей будет идти в русле готового проекта, а это помешает всесторонней оценке проделанной работы. Поэтому подготовку тестов и последующее тестирование следует поручить не разработчикам проекта, а профессионально подготовленной команде специалистов-тестировщиков.

Не следует думать, что тестирование завершает разработку, напротив, этот процесс надо начинать на самых ранних стадиях работы. Одна из методик тестирования объектно-ориентированных программных продуктов, называемая *unit-тестированием*, предписывает разрабатывать тестовые программы даже раньше программных модулей самого проекта. Другие методики тоже рекомендуют включать в рабочий проект тестирование программного продукта на каждом этапе его разработки.

Процесс автоматизации коснулся и тестирования. Созданы программные продукты, облегчающие и направляющие этот процесс. Более того, созданы программные продукты, создающие наборы тестов для применения их в тестирующих программах. Необходимость и важность тестирования сейчас осознана в полной мере. Специальность тестировщика становится все более востребованной и уважаемой. Она усложняется и уже сейчас требует специальной подготовки.

Более подробно о методах тестирования мы поговорим в следующей главе.

Сопровождение продукта

Работа с программным продуктом не заканчивается после того, как он передан заказчику и начал работать. Просто наступает следующий этап разработки — этап сопровождения продукта. На этом этапе выявляются и устраняются ошибки, не замеченные при отладке и тестировании программы, вносятся мелкие изменения и дополнения. Часто эти изменения связаны с изменившимися условиями эксплуатации программного продукта: закупкой нового оборудования, установкой новой версии операционной системы или новых драйверов устройств. Иногда в процессе эксплуатации выявляются искажения или упущения, сделанные в проекте по вине фирмы-разработчика или заказчика. Довольно часто требуется обучение персонала, обслуживающего программный продукт.

В этих случаях фирма-разработчик консультирует потребителя программного продукта по телефону или по электронной почте, обучает обслуживающий персонал на своей территории или непосредственно на рабочих местах потребителя. Иногда приходится выезжать к потребителю и на месте решать проблемы, возникшие в процессе эксплуатации.

При обнаружении ошибок в программном продукте приходится исправлять программный код, отлаживать и тестировать соответствующие модули и заново устанавливать продукт у потребителя. Если продукт уже пошел в серию и установлен во многих местах, то приходится писать программу, устанавливающую "заплатку" (patch), именуемую на жаргоне "патчем", на программном продукте и рассылать ее всем потребителям вместе с извинениями и инструкцией по установке.

Процесс сопровождения продукта следует оговорить с заказчиком на этапе определения требований. Запланированное сопровождение протекает легко и естественно, расходы на него определены заранее и включены во все планы фирмы-разработчика. Тем не менее, точный объем работ по сопровождению вычислить заранее невозможно. Поэтому чаще всего сопровождение программного продукта оговаривается отдельным договором или входит в лицензионное соглашение, распространяемое вместе с продуктом. В процессе эксплуатации договор о сопровождении может меняться, а по истечении некоторого времени его можно прекратить.

Модификация продукта

Программный продукт не может долго существовать в первоначальном виде. Информационные технологии активно развиваются, и очень быстро наступает время, когда продукт надо привести в соответствие с изменившимися условиями. Меняются и требования потребителей программного продукта. Им нужны новые, более совершенные программы. Наступает время модификации программного продукта.

Модификация программного продукта предполагает его переделку, зачастую довольно значительную. Поэтому всю работу по созданию продукта приходится пройти заново. Модификация начинается с определения новых требований к продукту. Затем проектируется новая версия продукта. При этом устанавливается, какие модули останутся без изменений, а какие подвергнутся изменению, удалению или полной замене. Тут же оценивается объем изменений и время, необходимое для этого. Кроме того, определяется, какие новые модули надо добавить к программному продукту.

После утверждения проекта команда разработчиков приступает к модификации продукта. Снова проходит разработка, отладка и тестирование. После этого новая версия передается в эксплуатацию, сопровождается, а через некоторое время снова модифицируется.

Многие фирмы-разработчики различают новую версию продукта и его незначительную модификацию. Они отмечают каждый новый выпуск продукта двумя числами, записанными через точку, например, OurProduct v.1.1, CoolProduct 2.3. Первое число означает номер версии продукта, второе — число модификаций данной версии. Какие изменения приводят к выпуску новой версии, а какие считаются только модификацией, решает сама фирма. Некоторые фирмы считают своим долгом выпускать новую версию каждый год, даже если в нее внесены только косметические изменения графического интерфейса с пользователем. Конкурирующие фирмы стараются увеличить номер версии своего продукта, чтобы создать впечатление его зрелости и солидности.

Иногда оба числа означают номер версии, говорят: "Версия 3.2 нашего продукта". В такой нумерации первым числом отмечается "выпуск" (release) программного продукта. Его увеличение означает сильное изменение продукта и происходит не так часто, как появление новой версии.

Некоторые программные продукты отмечаются тремя числами. К номеру версии и модификации добавляется число заплаток, наложенных на данную модификацию. Так, например, нумеруется ядро Linux. Запись "Linux 2.6.5" означает "ядро версии 2.6, на которое наложено 5 патчей".

Цикличность разработки продукта

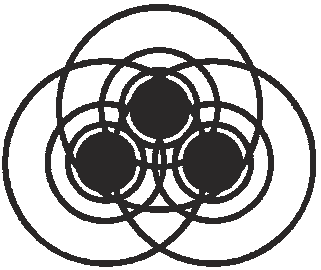
Отслеживая последовательные этапы разработки программного продукта, мы неоднократно замечали, что во время работы на одном этапе часто приходится возвращаться к предыдущим этапам. Так, обнаружение ошибки при

разработке, отладке или тестировании приводит к повторению этапа разработки или даже этапа проектирования. То же может случиться и при сопровождении продукта. Модификация продукта обязательно требует повторения всех этапов разработки.

Таким образом, разработка программного продукта на практике — это не последовательный, а циклический процесс. Приходится 2—3 раза повторить все этапы разработки, прежде чем продукт можно будет передать в эксплуатацию. Конечно, лучше было бы пройти весь путь разработки только один раз, но это практически никогда не удается сделать. Остается только постараться уменьшить число повторений цикла разработки. В этом помогает четкое определение требований к продукту и тщательное проектирование процесса разработки.

Контрольные вопросы

1. Чем компьютерная программа отличается от программного продукта?
2. Что входит в жизненный цикл программного продукта?
3. Какие требования предъявляются к программному продукту?
4. Как связаны графический интерфейс пользователя и требования к продукту?
5. Как происходит анализ информационной системы?
6. Какие документы вырабатываются в результате анализа информационной системы?
7. Что следует включить в проект информационной системы?
8. Какие виды проектов различают специалисты?
9. Какие методы применяются для разработки проектов?
10. В чем заключается отладка программного продукта?
11. Что такое тестирование программного продукта?
12. Чем тестирование отличается от отладки?
13. В чем заключается сопровождение программного продукта?
14. Что программисты называют патчем?
15. Как нумеруются версии программных продуктов?



Глава 20

Методы отладки и тестирования программы

В предыдущей главе, обсуждая жизненный цикл программы, мы отметили, что этап отладки и тестирования программного продукта — это обязательная и существенная часть процесса разработки. Подсчитано, что отладка занимает от трети до половины всего времени, отведенного на разработку. Даже после этого в готовом программном продукте зачастую остаются ошибки, выявляемые уже в процессе его эксплуатации. Поэтому отладке и тестированию надо уделить особое внимание.

Ошибки, сделанные в программе, можно разделить на два типа: синтаксические и семантические. Синтаксические ошибки возникают при нарушении правил записи программы на выбранном языке программирования. Современные компиляторы хорошо выявляют такие ошибки и выводят на экран толковые сообщения о типе ошибки и месте ее обнаружения. Эти сообщения позволяют быстро найти место ошибки и исправить ее. Дело несколько осложняется тремя обстоятельствами.

Во-первых, компилятор указывает не ту строку исходного текста программы, в которой произошла ошибка, а ту, в которой она проявилась. Эта строка может оказаться гораздо ниже, чем та, на которой сделана ошибка. Например, если вы забудете закончить описание класса на языке C++ точкой с запятой, как в следующем примере, то компилятор будет считать, что описание продолжается, и сообщит об ошибке только тогда, когда в этом "продолжении" возникнет какое-нибудь противоречие. Оно может проявиться спустя несколько десятков строк исходного текста, и вы будете ломать голову, недоумевая, какая же в этом месте может быть ошибка. Нужен некоторый опыт, чтобы сообразить, что ошибка находится гораздо выше.

```
class A{
   int n;
```

```
public:
   A(int n){
     this.n = n;
   }
}
class B{
};
```

Во-вторых, ошибка может произойти по разным причинам, компилятор же укажет только одну. Иногда компилятор честно сообщает, что не может определить причину ошибки, написав просто: "Syntax error". В большинстве случаев он указывает наиболее часто встречающуюся, по мнению разработчиков компилятора, ошибку, в вашей программе может оказаться совсем другая, поэтому принимайте сообщения компилятора не как истину, а только как один из возможных вариантов.

В-третьих, некоторые ошибки компилятор не в состоянии обнаружить. К ним относится деление на ноль или переполнение в арифметических выражениях, отрицательное или слишком большое значение индекса массива, и другие ошибки, возникающие в процессе вычислений. Такие ошибки проявляются уже на этапе выполнения (run time) программы и сообщает о них не компилятор, а исполняющая система. Их следует учитывать уже при проектировании, включая в программу обработку исключительных ситуаций.

Синтаксические ошибки обнаруживаются и устраняются довольно легко после нескольких запусков программы. Без исправления этих ошибок программа просто не будет работать. Гораздо труднее исправить семантические ошибки — ошибки в алгоритме работы программы.

Часто встречающаяся семантическая ошибка — неправильная расстановка скобок в выражении. Компилятор может проверить парность скобок, но если число открывающих скобок совпадает с числом закрывающих скобок, то компилятор не может сказать, в нужных ли местах они стоят. Здесь надо быть особенно внимательным и отдавать себе отчет в том, что вы хотите вычислить. Например, переменная `x` может получить значение, вычисленное следующим оператором:

```
double x = 2 * (a - b) / (a + b) + k * 5 * (a + b) / (a * b);
```

Но значение `x` вполне можно вычислить и таким оператором:

```
double x = 2 * (a - b) / (a + b) + k * (5 * a + b) / (a * b);
```

Компилятору обе записи "кажутся" правильными. Выбор того или иного оператора зависит от задачи, в которой вычисляется переменная `x`.

Еще одна распространенная ошибка, не обнаружимая компилятором, возникает из-за неправильной расстановки или отсутствия фигурных скобок.

В следующем примере, очевидно, переменная s, в которой накапливается сумма, должна изменяться при каждой итерации цикла, но компилятор не может "понять", что, фактически, она меняется только один раз: после выполнения цикла.

```
int m, s = 0;
for (int k = 0; k < a.size() - 1; k++)
   m = a[k + 1] - a[k];
   s += 2 * m * m - 3 * m + 2;
```

Чтобы избежать таких ошибок, специалисты рекомендуют записывать тело любого составного оператора в фигурных скобках.

Семантические ошибки выявить гораздо труднее, чем синтаксические. Поскольку компилятор не может их заметить, он не выдает никаких сообщений. Программа может долго работать, не попадая в условия, в которых проявляются семантические ошибки. Бывает, что ошибочные результаты появляются только после нескольких месяцев эксплуатации программы. Поэтому для выявления таких ошибок зачастую приходится прикладывать специальные усилия, а в проекте необходимо предусматривать специальную фазу отладки и тестирования.

Процесс отладки проходит несколько этапов. Ошибку сначала надо обнаружить. Потом найти место ее появления, как говорят, *локализовать* ошибку. После этого ошибку надо устранить. Рассмотрим подробнее каждую стадию отладки.

Обнаружение ошибки

Как уже говорилось выше, большинство синтаксических ошибок обнаруживается компилятором или выявляется сразу же после первых запусков программы. Их нетрудно обнаружить и исправить. Как правило, трудности вызывает обнаружение семантических ошибок. О них обычно и идет речь, когда говорят об отладке.

После того как очевидные ошибки устранены, и программа при запуске не сообщает об ошибках, могут возникнуть следующие ситуации:

1. Программа не дает никаких результатов. Это происходит чаще всего в результате зацикливания или ожидания какого-то события, которое по разным причинам не может наступить.
2. Программа дает неверные результаты. Самый распространенный вариант, возникающий из-за ошибок в алгоритме или из-за ошибок в кодировании алгоритма.
3. Программа дает правдоподобные результаты. Это самый опасный случай, поскольку получаемые значения можно принять за верные результаты и передать в эксплуатацию ошибочную программу.

4. Программа работает правильно. Почти фантастика.

Рассмотрим подробнее каждую ситуацию.

Программа не дает результатов

В этой ситуации надо, прежде всего, добиться все-таки от программы хоть какой-то выдачи. Это достигается добавлением в программу функций вывода `printf()`, `fprintf()`, `puts()` или других функций, выводящих промежуточные значения всех или наиболее важных переменных. Они образуют так называемую *отладочную печать*. Это название появилось в эпоху телетайпов, когда результаты выводились на бумажную ленту. Сейчас отладочные сведения выводятся на экран дисплея или записываются в файл, но слово "печать" осталось в лексиконе программистов, означая выдачу результатов на какое-нибудь устройство вывода.

Что входит в отладочную печать? Прежде всего, надо распечатать исходные данные, чтобы убедиться, что ввод происходит правильно. Затем надо проверить начало и окончание каждого цикла, распечатав начальные значения цикла и значения, полученные по окончании цикла. Печать внутри цикла обычно не делается, она будет выполняться при каждом повторении цикла и приведет к огромной выдаче, в которой будет трудно что-либо понять.

Отладочная печать ставится перед разветвлениями, а также в начало каждой ветви, чтобы убедиться в правильном прохождении условных операторов.

Очень много ошибок возникает при передаче аргументов в функции. Компилятор работает с каждой функцией отдельно, у него нет сведений о других функциях, и он не может проверить правильность фактических параметров. Поэтому отладочная печать ставится в начале каждой функции, чтобы проверить правильность полученных ею аргументов. Кроме того, нужно распечатывать значение, возвращенное функцией, чтобы убедиться в точности ее работы.

После беглого просмотра результатов отладочной печати уже можно увидеть, какие участки кода программа проходит без задержек, а какие приводят к зацикливанию. Например, распечатав в следующем цикле значения `k` и `a.count()`, можно обнаружить, что переменная `k` меняется в цикле два раза, уменьшаясь и увеличиваясь на единицу:

```
while (k < a.size()){
   if (a.count() < current){
      s += a.deposit(abonent(k)).value;
   }else{
      s += a.deposit(abonent(--k)).value;
   }
   k++;
}
```

В результате переменная `k` не увеличивается и программа зацикливается.

Получив отладочную печать и внеся необходимые исправления в программу, мы переходим от первой ситуации к одной из следующих ситуаций.

Программа дает неверные результаты

Эта ситуация встречается чаще всего. Она может возникнуть по самым разным причинам. К ней могут привести и простая ошибка в наборе текста программы, и неверный алгоритм решения задачи. Например, вы можете случайно исказить оператор присваивания:

```
double x = 3.45 * a[k - 1] + 2.5 * a[k + 1];
```

и написать, поставив вместо десятичной точки звездочку:

```
double x = 3.45 * a[k - 1] + 2*5 * a[k + 1];
```

Из-за этой мало заметной ошибки значение переменной `x` изменится в четыре раза.

Такие ошибки легко обнаружить, но очень трудно локализовать. Зачастую приходится делать полную *трассировку* программы, поставив отладочную печать после почти каждого оператора, тем самым тщательно отслеживая изменения переменных. Трассировкой не следует злоупотреблять: в результате выдается масса информации, которую трудно проанализировать. Поэтому на практике к этому методу прибегают только в самых крайних случаях.

Гораздо чаще для обнаружения ошибки в программу устанавливаются *контрольные точки* (breakpoints), в те места, где ошибки наиболее вероятны. Программа останавливается на каждой контрольной точке. После останова можно просмотреть значения переменных и продолжить выполнение программы до следующей контрольной точки. Многие инструментальные средства программирования позволяют даже изменить переменные в контрольной точке и продолжить выполнение программы с другими значениями.

Еще одно средство поиска ошибок — *пошаговое выполнение* программы, при котором она приостанавливается после выполнения каждого оператора. После останова можно просмотреть промежуточные значения переменных и выполнить следующий оператор. Разумеется, полное пошаговое выполнение всей программы займет слишком много времени. Поэтому его обычно начинают с какой-либо контрольной точки и прекращают, как только ошибка найдена.

Еще труднее найти ошибки в алгоритме решения задачи. Они могут накапливаться понемногу от оператора к оператору. По отладочной печати трудно отследить момент появления таких ошибок. Пошаговое выполнение программы тоже не поможет найти ее. В таком случае надо проверять сам алгоритм и убеждаться в его правильности. Методы решения этой непростой задачи обсуждаются в следующей главе.

Программа дает правдоподобные результаты

Это самая коварная ситуация. Получив результаты, близкие к предполагаемым, вы можете подумать, что программа работает правильно, и передать ее в эксплуатацию.

Как убедиться в том, что результаты правильны, а не правдоподобны? Здесь помогает тестирование программы. Тщательно подобранные тесты позволят отличить правдоподобные значения от правильных результатов. Как правило, одного набора тестов недостаточно для убедительного доказательства неверности работы программы. Специалисты-тестировщики всегда подбирают несколько наборов тестов так, чтобы точнее выявить дефекты программы.

Выявить правдоподобные результаты часто помогает прием, заимствованный из математики. Обычно мы проверяем правильность вычисления корня уравнения, подставляя его в уравнение и убеждаясь в том, что левая часть уравнения равна правой его части. Тем не менее, математикам давно известно, что из того, что левая часть уравнения, при подстановке в него приближенно вычисленного корня, почти совпадает с правой частью, вовсе не вытекает, что этот корень близок к настоящему корню уравнения.

Особенность таких некорректных задач заключается в том, что небольшое изменение исходных данных приводит к сильному изменению результата, хотя такой сильный скачок не вытекает из условий задачи. Это дает способ проверки правильности правдоподобных результатов. При тестировании программы надо провести серию испытаний с близкими значениями исходных данных и следить за тем, как меняются результаты ее работы. Неоправданно сильное изменение результатов при незначительном изменении данных должно вызвать сомнение в правильности вычислений, если только это не вытекает из условий задачи.

Локализация ошибки

После обнаружения ошибки надо найти место ее возникновения, как говорят, *локализовать* ошибку. Это не такая простая задача, как кажется поначалу. Ошибка может быть сделана где-то в начале исходного текста программы, а обнаружена позже, спустя несколько десятков строк исходного текста или даже в другой функции. Для локализации приходится делать обратный просмотр исходного текста программы, начиная от места обнаружения ошибки. В этом помогает трассировка программы, расстановка контрольных точек и пошаговое выполнение. Контрольная точка устанавливается в некотором отдалении от места обнаружения ошибки, в том месте, где по вашим соображениям может начаться неправильное выполнение программы. После этого тщательно проверяется участок программы от контрольной точки до места обнаружения ошибки.

Иногда такое обратное отслеживание хода выполнения программы не помогает. Тогда приходится прослеживать ее работу по исходному тексту, просматривая выполнение операторов со значениями, полученными из контрольной печати, и доходя до места обнаружения ошибки. Такое прослеживание часто называют *прокруткой* программы. Прокрутка программы — трудоемкое занятие, ее лучше проводить на небольшом участке программы, сузив предварительно участок прокрутки с помощью отладочной печати.

Когда и прокрутка не помогает, приходится проверять логику выполнения программы. Проверка начинается с места обнаружения ошибки. Просматривается полученная в этой точке отладочная печать, и проводятся рассуждения вроде следующих: "Такие результаты мог дать оператор *A* или оператор *B*. Но оператор *B* в этом случае не может выполняться. Значит, посмотрим на оператор *A*. Оператор *A* дал значение *C*, а оно могло получиться только в ситуации *D*. Чтобы возникла ситуация *D*, надо, чтобы...". На каждом логическом шаге приходится делать отладочную печать и небольшую прокрутку. Привлечение логических рассуждений помогает быстрее проделать эту трудную работу, поскольку концентрирует внимание только на тех переменных, которые могут получить ошибочные значения, а таких переменных немного.

Наконец, каждый программист знает, что при попытке объяснить ошибку, которая никак не поддается локализации, своему коллеге по работе или постороннему консультанту, неожиданно, в процессе рассказа, самому автору программного кода вдруг становится ясно, в чем заключается ошибка. Иногда даже письменное изложение проблемы помогает найти ее решение. Это объясняется тем, что при последовательном изложении проблемы мысли приходят в порядок, выстраивается логическая цепочка, приводящая к решению задачи. Поэтому очень важно периодически обсуждать возникающие проблемы всей командой разработчиков. Такие обсуждения не только позволяют устранять трудности, возникающие в процессе разработки, но и держат программистов в курсе всех задач, решаемых на каждом этапе реализации проекта.

Устранение ошибки

После того как ошибка обнаружена и найдено место ее возникновения, наступает время исправления программы. Некоторые ошибки, такие как неправильно записанные выражения, неверно определенные начальные значения переменных, неправильные входы и выходы из циклов, ошибки в логических условиях, исправить легко. Для этого достаточно изменить один или несколько операторов в исходном коде программы. После этого надо снова запустить те же наборы тестов, чтобы убедиться в том, что ошибки исчезли.

Другие ошибки, например, те, которые возникают из-за неправильно запрограммированного алгоритма, исправить сложнее. Их устранение требует изменения больших кусков программного кода. Такие изменения зачастую вносят в программу новые ошибки. Эти ошибки снова надо обнаруживать, локализовать и устранять. Процесс отладки становится циклическим. Его приходится повторять несколько раз. Как гласит шутливая аксиома отладки: "Каждая последняя ошибка является предпоследней".

Чтобы не попасть в неприятный цикл отладки или быстрее выйти из него, следуйте простому правилу: "Исправлять за один раз только одну ошибку". Тогда вы будете знать, из-за чего возникла новая ошибка, и сможете быстро устранить ее.

Труднее всего исправить ошибки алгоритма, заложенного в программу. Изменение алгоритма часто влечет переработку структуры программы, введение новых классов или значительное изменение существующих классов. Происходит возврат к этапу проектирования.

Наконец, не так уж редко встречается ошибка в определении требований к программе. Программа выдает не те результаты, которые ожидает заказчик программного продукта, да и вообще не то, что он хотел. Можно считать это не ошибкой, а недоговоренностью или недоразумением, но такое положение надо исправлять. Для исправления приходится возвращаться к самому первому этапу разработки и начинать все сначала, уточняя требования к программе.

Средства отладки

Трудоемкость процесса отладки всегда вызывала стремление автоматизировать его. С появлением первых компиляторов стали появляться и *программы-отладчики* (debuggers), называемые на жаргоне программистов "дебаггерами". Они предоставляют программные средства для выполнения основных работ по отладке. С их помощью легко установить контрольные точки, сделать трассировку и пошаговое выполнение программы, просмотреть текущие значения всех или выбранных переменных.

Отладчики тесно связаны с компиляторами. Для улучшения отладки компилятор может вставлять в машинный код дополнительную, отладочную, информацию, которую отладчик использует при прогоне программы. Поэтому отладчики чаще всего поставляются вместе с компиляторами, в одной интегрированной среде разработки IDE (Integrated Development Environment). В меню **Options**, или каком-нибудь другом меню, предназначенном для настройки параметров компилятора в такой интегрированной среде, можно выбрать один из нескольких режимов работы компилятора, в том числе отладочный режим, **Debug**, или окончательный режим, **Release**.

Во время разработки программы надо выбрать отладочный режим компилятора. Тогда можно будет использовать все средства отладчика, которые обычно перечислены в меню **Debug** интегрированной среды. После окончания отладки, когда уже решено передавать программу в эксплуатацию, компилятор надо перестроить на создание окончательной версии (release, build). В режиме **Release** компилятор удаляет из машинного кода отладочную информацию и генерирует оптимизированный рабочий код.

Упражнения

1. Приведите примеры ошибок, которые выявляются не сразу, а после выполнения нескольких следующих операторов.
2. Приведите примеры ошибок, которые не может обнаружить компилятор.
3. Вам нужно найти ошибку в программе, вычисляющей корни квадратного уравнения по обычной школьной формуле. Расставьте в исходном коде отладочную печать.
4. Выясните, каково наибольшее и наименьшее целое число на вашем компьютере. Что происходит при выходе за эти значения?
5. Найдите наибольшее по абсолютной величине действительное число, которое ваш компьютер воспринимает как ноль ("машинный ноль") и ответьте на вопрос, почему сравнение с нулем действительных чисел, вроде if (x == 0.0), часто приводит к ошибке.
6. Проверьте, при всех ли значениях x выполняется тождество $\cos^2(x) + \sin^2(x) = 1$ на вашем компьютере.
7. Найдите с максимальной возможной на вашем компьютере точностью две суммы, отличающиеся порядком следования слагаемых:

   $$S_1 = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{999999} + \frac{1}{1000000}$$

   и

   $$S_2 = \frac{1}{1000000} + \frac{1}{999999} + \ldots + \frac{1}{2} + 1.$$

   Посмотрите, совпадут ли значения S_1 и S_2.

Тестирование

Мы уже не раз говорили, что тестирование заключается в проверке работы программы на ряде задач, решение которых известно заранее. Результаты работы программы сравниваются с эталонными результатами, и на основании их совпадения или несовпадения делается вывод о применимости программы к этому классу задач.

Классическое определение тестирования, данное Гленфордом Дж. Майерсом (Glenford J. Myers), звучит так: "Тестирование — процесс выполнения программы с намерением найти ошибки". Новейшие методы тестирования не только запускают программу с различными исходными и промежуточными данными, но и проверяют ее исходный текст. Поэтому определение Майерса уже устарело и под тестированием сейчас понимаются любые действия, направленные на поиск ошибок в программе.

Разумеется, тестирование не может доказать отсутствие ошибок, оно только выявляет их наличие. Нельзя сказать, что программа, успешно прошедшая тестирование, свободна от ошибок. Тем не менее, после серьезного тестирования можно запускать программу в промышленную эксплуатацию с большой долей уверенности. Практика показывает, что полностью протестированный программный продукт успешно справляется со своей задачей.

Тестирование не следует отождествлять с отладкой, хотя они тесно связаны. Задача тестирования — выявить наличие дефектов в программе, а задача отладки — отыскать их местоположение и устранить ошибки. Сначала с помощью тестирования надо найти ошибки, а потом, во время отладки, устранить их. Иногда тестирование и отладка выполняются одним специалистом в одном процессе, чаще их осуществляют разные люди.

Unit-тестирование

По сути дела, тестированием своего участка программы занимается каждый разработчик программного обеспечения. Написав какой-либо класс, процедуру или несколько процедур, программист обязательно проверяет их работу на характерных для этого кода наборах данных. Появившаяся недавно методика unit-тестирования возводит эту привычку в абсолютное правило. Эта методика предписывает проводить тестирование после написания каждой процедуры или класса. Более того, unit-тестирование обязывает писать тесты еще до создания исходного кода программы!

Допустим, мы решили написать класс комплексных чисел `Complex`. По методике unit-тестирования мы начинаем с того, что пишем пустой класс `Complex` и вместе с ним сразу же пишем класс `TestComplex`, который будет впоследствии содержать тесты.

```
class Complex{};


class TestComplex{
   Complex z;
public:
   TestComplex(Complex z){
      this.z = z;
```

```
   }
   void runTest(){
   }
};
```

Весь этот код сразу компилируется, чтобы убедиться в правильности написанной конструкции. Затем начинаем разработку класса `Complex`. В процессе разработки каждый создаваемый метод класса `Complex` записывается еще и в тестовый класс `TestComplex`, например, следующим образом:

```
#include <iostream>

class Complex{
   double re, im;
public:
   Complex(double a = 0.0, double b = 0.0){
      re = a; im = b;
   }
   double mod(){
      return sqrt(re * re + im * im);
   }
};

class TestComplex{
   Complex z;
public:
   TestComplex(Complex z){
      this.z = z;
   }
   void runTest(double result){
      cout << testMod(result) << endl;
   }
   bool testMod(double result){
      return z.mod() == result;
   }
};

void main(){
   Complex z1(), z2(0.0, 1.0), z3(3.0, -4.0);
```

```
    TestComplex t1(z1), t2(z2), t3(z3);
    t1.runTest(0.0);
    t2.runTest(1.0);
    t2.runTest(5.0);
}
```

После компиляции и отладки этого кода сразу же начинается тестирование. Только после того, как все тесты выполнены успешно, продолжается создание класса.

Такая методика программирования получила название *программирования, управляемого тестами* (TDP, test-driven programming). Она вошла обязательной составной частью в быстро получившую популярность среди разработчиков свободного программного обеспечения методику экстремального программирования XP (extreme programming). Unit-тестирование первоначально возникло в технологии Java. Не удивительно, что и наибольшее развитие получил свободно распространяемый программный продукт JUnit, автоматизирующий unit-тестирование Java-классов. Для разработчиков, пишущих программы на языке C++, по аналогии с этим программным продуктом, создана библиотека классов, названная CppUnit.

Сейчас в распоряжении программистов есть множество программных продуктов, облегчающих unit-тестирование. Более того, есть продукты, генерирующие наборы тестов. Например, фирмой Parasoft Corporation создан и распространяется программный продукт JTest, подготавливающий тесты для JUnit.

Методики тестирования

За полувековую историю развития компьютеров предложено множество методик тестирования. Самая простая рассматривает программу как "черный ящик", внутренняя структура которого неизвестна. Испытатель проверяет работу "черного ящика", подавая на его вход определенные сигналы и наблюдая, как программа реагирует на них. Первым эту методику предложил, пожалуй, Козьма Прутков, сказавший: "Щелкни кобылу в нос, она махнет хвостом".

По методике "черного ящика" каждый тест задает исходные данные программе. Программа выполняется, после чего полученные результаты сравниваются с заранее известными тестовыми значениями. Такая методика покажет отсутствие ошибок только в том случае, когда набор тестов охватит все возможные исходные данные. Поскольку их необъятно много, всеобъемлющий охват исходных данных невозможен. Следовательно, теоретически нет уверенности, что такая методика выявит все ошибки. Кроме того, две ошибки могут взаимно уничтожаться на тестах и не будут выявлены по этой методике.

Другая методика обращается с программой как с "белым ящиком" или, другими словами, "прозрачным ящиком". Она учитывает исходный текст программы. По этой методике набор тестов должен привести к выполнению каждого оператора программы и прохождению каждого из возможных путей выполнения программы хотя бы по одному разу. Эта методика хорошо проверяет логику выполнения программы, но тоже не может выявить все ошибки. Например, некоторые ветви программы могут быть просто упущены в процессе разработки, а такое тестирование не выявит их отсутствие.

Реальные методики тестирования используют оба подхода. Одни методики приближаются к "черному ящику", другие — к "прозрачному ящику". Так или иначе, в настоящее время практически во все наборы тестов включаются тесты, учитывающие исходный текст программы. Это значительно облегчает тестирование.

Пусть, например, надо протестировать программу, находящую площадь треугольника по длинам его сторон a, b, c. Если мы знаем, что эта программа вычисляет отдельно площадь прямоугольного треугольника как полупроизведение его катетов, и отдельно площадь остальных треугольников по формуле Герона, то нам не надо создавать огромное количество тестов. Мы можем разбить все множество исходных данных a, b, c на два класса — образующих прямоугольный треугольник и не образующих его — и прогнать всего по одному тесту для каждого класса.

По своему назначению тесты делятся на несколько групп:

- функциональное тестирование;
- тестирование обращений к базам данных;
- тестирование логики программы;
- нагрузочное тестирование;
- стрессовое тестирование;
- тестирование интерфейса пользователя;
- тестирование безопасности и прав доступа;
- тестирование инсталляции программного продукта.

Рассмотрим подробнее каждую из этих групп тестов.

Функциональное тестирование

Это основной вид тестирования, проверяющий соответствие программы функциональным требованиям, предъявленным к ней на самом первом этапе проектирования программного продукта. Функциональное тестирование призвано убедиться в правильности ввода исходных данных, обработки и вывода результатов, а также в правильности работы всех элементов управления программой. Для этого проигрываются все сценарии использования

программы (use case). При этом в тестах задаются и правильные и неправильные входные данные. Это позволяет проверить реакцию программы на ошибки ввода.

В большинстве функциональных тестов программа рассматривается как "черный ящик", поскольку проверяется работа программы, а не ее реализация. Тем не менее, как уже говорилось выше, знание алгоритма и исходного текста программы может помочь в проверке правильного ее функционирования.

Пусть, например, надо протестировать процедуру, решающую квадратное уравнение $ax^2 + bx + c = 0$ с действительными коэффициентами. Для этого надо подготовить тестовый набор коэффициентов a, b, c, учитывающих все возможные ситуации. Не зная о том, что решение уравнения зависит от знака дискриминанта, мы можем упустить один из тестов, приводящих к положительному или отрицательному дискриминанту. Зато, зная алгоритм нахождения корней, мы можем разбить все тестовые значения коэффициентов на те, которые дают неотрицательный дискриминант, и те, для которых дискриминант отрицателен. В каждой группе можно взять только один набор значений коэффициентов a, b, c. Дополнив этот набор вырожденными и граничными значениями, мы получим полный набор тестов. Он показан в табл. 20.1.

Таблица 20.1. *Набор тестов для программы нахождения корней квадратного уравнения*

a	***b***	***c***	x_1	x_2	**Примечание**
0	0	0	Любое	Любое	Вырожденное значение
0	0	1	Нет	Нет	Вырожденное значение
0	1.0	–1.0	1.0	1.0	Вырожденное значение
1.0	2.0	–3.0	1.0	–3.0	Неотрицательный дискриминант
1.0	0.0	1.0	*i*	*i*	Отрицательный дискриминант
1e308	2.0	–3.0	0	0	Граничное значение
1e–308	2.0	–3.0	Велико	Велико	Граничное значение
1.0	1e308	1.0	Мало	Велико	Граничное значение

Успешно выполнив этот набор, мы можем быть уверены, что обычный школьный алгоритм решения квадратного уравнения работает правильно. Конечно, такую уверенность дает только простота задачи, которую решает тестируемая программа. В реальных условиях убедиться в полноте набора тестов гораздо труднее.

Тестирование обращений к базам данных

Этот набор тестов проверяет правильность связи программы с базами данных, в которых она хранит информацию. Тесты должны проверить все используемые программой обращения к базам данных. Их надо проделать как с верными данными, так и с ошибочными, чтобы убедиться в том, что база данных сохраняет целостность во всех случаях.

Это тестирование предполагает знание методов обращения к базе данных, примененных в программе, а также знание особенностей используемого программой сервера базы данных. Поэтому подход к программе, как к "черному ящику" здесь не годится. Подготовка наборов тестов должна вестись со знанием использованных в программе связей с базой данных.

Тестирование бизнес-логики программы

Тестирование деловых правил, заложенных в программу, призвано проверить правильность ее работы в течение определенного отчетного периода времени: дня, недели, месяца, квартала, года. Такие тесты должны за короткое время пробежать расписание всего отчетного периода и проделать все действия и события за этот период. При этом особое внимание уделяется функциям, обрабатывающим даты и интервалы времени. Для их проверки применяются как тесты, включающие правильные даты, так и тесты с неправильными датами.

Кроме того, в тесты закладывается составление отчетов и других документов, которые должна генерировать программа. Проверяется, происходит ли уведомление пользователей по электронной почте о событиях, происходящих в системе, если, конечно, такие уведомления заложены в деловые правила.

Нагрузочное тестирование

Эти тесты дают рабочую нагрузку на программу. Они должны смоделировать реальную обстановку, в которой будет работать программа: многопользовательский режим, работу по сети, активное использование устройств ввода-вывода, принтеров и графопостроителей. Кроме того, нагрузочные тесты отслеживают реакцию программы на изменение нагрузки, время отклика на события, быстроту связи с базами данных, удаленными источниками информации. Эти параметры должны находиться в пределах, определенных техническим заданием.

Нагрузочное тестирование должно проверить, как программа использует все необходимые ресурсы компьютера, и убедиться в том, что она правильно работает в заданных условиях.

Стрессовое тестирование

Стрессовое тестирование заключается в том, чтобы дать предельные нагрузки на программу с целью проверить ее работу в условиях дефицита ресурсов. Эти тесты должны проверить работу программы при недостатке оперативной памяти, превышении пропускной способности сетевых средств, переполнении каналов ввода-вывода, одновременного доступа большого количества пользователей и т. п. Успешное выполнение стрессовых тестов даст уверенность в том, что программа сохранит работоспособность при перегрузках.

Для программ, рассчитанных на быстрое восстановление после сбоев, стрессовое тестирование должно сымитировать такие сбои, чтобы проверить способность восстановления.

Стрессовые тесты полезны не только для выявления дефектов программы, но и для получения информации о реакции программы на пиковые нагрузки. Эту информацию затем можно включить в инструкцию по эксплуатации программы.

Тестирование интерфейса пользователя

Тестирование интерфейса содержит наибольший объем ручной работы. Тестировщик должен проверить работу всех элементов управления, открыть все окна, имеющиеся в программе, пройти и опробовать все пункты меню. Ему надо попытаться проделать и всевозможные неправильные действия пользователя, чтобы проверить реакцию программы на ошибки использования интерфейса.

Во время тестирования интерфейса следует вводить всевозможные значения во все поля ввода, чтобы проверить, отвечает ли их диапазон требованиям, предъявленным к программе. Необходимо вводить и невозможные значения, например, в числовые поля вводить символьные данные, чтобы просмотреть сообщения системы, сделанные в ответ на ввод неправильных данных.

Тестирование безопасности и прав доступа

Подсистема безопасности требует отдельного тестирования. При этом проверяются регистрация новых пользователей, локальный и удаленный доступ к системе, разграничение прав доступа к ней. Очень часто система строится так, что одни пользователи могут только просматривать доступную им информацию, другие пользователи могут добавлять, редактировать и удалять

ее, а третьи могут менять права пользователей первых двух категорий. Все это обязательно надо протестировать и проверить, правильно ли определяются права пользователей на те или иные действия в системе.

Для тестирования безопасности в системе создаются несколько временных пользователей с разными правами доступа. Тесты выполняются несколько раз от имени каждого из них. В наборе тестов следует предусмотреть все действия пользователя, имеющиеся в программе, и посмотреть, какие из них разрешены пользователю, а какие — нет.

Тестирование инсталляции программного продукта

Первая функция программного продукта, с которой сталкивается пользователь, — это его установка на компьютер. Если установить продукт не удастся, то потребитель откажется от него. Если при установке возникнут трудности, то пользователь потеряет доверие к продукту и к фирме — разработчику этого продукта. Поэтому необходимо тщательно протестировать процесс инсталляции на разном оборудовании, обратив внимание на безошибочное выполнение повторной установки, обновления предыдущей версии, адаптации к разному оборудованию.

Наборы тестов

Набор тестов (test suite) следует подготовить еще до проектирования программного продукта, на этапе определения требований к нему. Именно требования к программе и составляют круг тех задач, которые должны быть выражены в тестовых заданиях. Кроме того, спроектировав систему, уже трудно беспристрастно оценивать ее. Направление мысли разработчика волей-неволей будет идти в русле готового проекта, а это помешает всесторонней оценке проделанной работы. Поэтому подготовку тестов и последующее тестирование следует поручить не разработчикам проекта, а профессионально подготовленной команде специалистов-тестировщиков.

Как составить набор тестов? Это целиком зависит от тестируемой программы и требований к ней. Здесь можно дать только самые общие рекомендации. Во-первых, для каждой из перечисленных выше групп тестов надо составить свой набор тестов. Во-вторых, надо подобрать набор тестов, работающих с программой как с "черным ящиком", рассмотрев все возможные области значений исходных данных. В-третьих, надо изучить исходный текст программы и подобрать тесты для прохождения всех путей алгоритма, заложенного в программу, и тесты для выполнения каждого оператора программы.

Каждый отдельный тест, называемый специалистами *тестовым случаем* (test case), состоит из исходных данных, вводимых в тестируемый программный модуль, и предполагаемого результата, который должен получить модуль.

Результатом не обязательно должны быть значения, возвращаемые модулем. Это могут быть обновления базы данных, изменения в конфигурационных файлах, отправка сообщений по сети. Важно, чтобы это были четко отслеживаемые значения, влияющие на ход выполнения программы.

Процесс тестирования

Не следует думать, что тестирование завершает разработку и тестировать надо уже готовую программу. Напротив, этот процесс надо начинать на самых ранних стадиях разработки. Тестирование может выявить упущения в проектировании программного модуля, а такие упущения лучше устранить пораньше. Кроме того, известно, что исправление одних ошибок часто вносит другие. Хотя общее число ошибок уменьшается, после каждого исправления приходится снова тестировать программу. Таким образом, тестирование вклинивается в процесс разработки. По этим причинам сейчас получила большую популярность методика непрерывного тестирования программы во время ее разработки. Она называется *zero-defect mindset*.

Согласно методике zero-defect mindset, все ошибки подразделяются на несколько уровней по своей грубости. Программист не имеет права добавлять новую функциональность в программу, пока он не исправит все крупные ошибки до определенного уровня. Допустимый уровень ошибки устанавливается для каждого этапа разработки, он повышается по мере завершения работы.

Итак, по всем современным методикам тестирование выполняется прямо в процессе разработки. При тестировании программы, состоящей из нескольких программных модулей, надо выбрать один из двух противоположных стилей тестирования: *тестирование снизу вверх* или *тестирование сверху вниз*.

Тестирование снизу вверх начинает с отдельных подпрограмм: функций и процедур, не вызывающих другие подпрограммы. На их вход подаются тестовые данные, результаты их работы сравниваются с тестовыми результатами. После того как тесты уже не выявляют ошибок в таких подпрограммах, тестировщик переходит к тестированию подпрограмм, вызывающих уже протестированные подпрограммы. При этом выявляются ошибки, возникающие при взаимодействии подпрограмм. Затем тестируются подпрограммы, вызывающие только что протестированные подпрограммы и т. д. Наконец, в последнюю очередь проверяется работа головного модуля всей программы.

Тестирование сверху вниз, напротив, начинает сразу с готового прототипа головного модуля программы. Вместо подпрограмм, вызываемых из головного модуля, подставляются заглушки (stubs), выдающие головному модулю заранее определенные значения. Убедившись в правильности работы такого скелета всей программы, тестировщик начинает по очереди заменять за-

глушки настоящими подпрограммами или их прототипами, пока не дойдет до самых мелких программных единиц, уже не вызывающих никакие другие подпрограммы.

На практике тестировщики редко следуют в чистом виде тому или иному стилю тестирования. Чаще всего одновременно тестируются и отдельные программные единицы, и прототип головного модуля, а окончательная сборка всей программы происходит где-то посередине. В некоторых случаях задача ставится таким образом, что надо протестировать сразу и головной модуль, и какие-то отдельные подпрограммы. В других случаях те или иные модули и программные единицы просто не готовы к тестированию.

Какова бы ни была методика тестирования, сразу же после исправления ошибки прогоняется тот же набор тестов, с помощью которого была найдена ошибка. Это так называемое *регрессионное тестирование*, позволяет убедиться в том, что ошибка исправлена. Если регрессионное тестирование не выявило дефектов в программе, то этот набор тестов можно отложить и перейти к следующему этапу тестирования.

Особенности тестирования объектно-ориентированных программ

Объектно-ориентированный подход к программированию накладывает свой отпечаток и на методику тестирования. Вместо тестирования функций, процедур и прочих программных модулей, производится тестирование классов, связей между ними и иерархий классов.

Количество классов, их строение и взаимодействие между ними определяются на стадии объектно-ориентированного проектирования. На этой стадии создаются диаграммы классов и диаграммы связей. Зачастую по этим диаграммам генерируется исходный программный код в виде абстрактных классов и/или интерфейсов. После этого уже можно провести полноценное тестирование полученной модели. Исправление ошибок на такой ранней стадии позволит значительно сэкономить время отладки и всей разработки в целом. Более того, тестирование поможет выяснить, соответствует ли модель требованиям, предъявленным к программному продукту, и своевременно исправить проект.

Тестирование объектно-ориентированных программ чаще всего выполняется снизу вверх. Вначале проверяется программный код методов класса и тестируются отдельные методы. Затем начинается тестирование класса.

Для тестирования класса пишется тестовая программа, в которой создаются объекты класса с разными значениями его полей. Тестовая программа проверяет выполнение контрактов методами классов. Для этого она обращается ко всем методам класса и отслеживает результаты их выполнения. При проверке работы методов, обращающихся к другим объектам, создаются объек-

ты-заглушки, содержащие только поля и методы, нужные для тестирования основного объекта. Поля объекта-заглушки получают определенные, хорошо узнаваемые значения. Методы объекта-заглушки очень просты, они только сигнализируют каким-нибудь образом обо всех обращениях к ним.

Тестирование шаблонов классов требует особого внимания. Здесь легко допустить ошибку и трудно обнаружить ее. Разработчик должен в точности знать, какой класс создаст компилятор по шаблону, написанному им, а для этого нужен большой опыт работы с этим компилятором. Тестировщик тоже должен знать особенности компилятора, чтобы создать тесты для каждого класса, создаваемого по шаблону.

После того как тестирование отдельных классов уже не выявляет ошибок в них, создаются объекты разных классов, и проверяется их взаимодействие. Выстраивается иерархия классов и тестируется правильность наследования в этой иерархии. Здесь удобнее стиль тестирования сверху вниз, от базовых классов к порожденным классам, от вершины иерархии классов к самым конкретным классам. Для тестирования абстрактных классов специально создаются его реализации-заглушки.

Особую трудность вызывает тестирование классов, использующих полиморфизм. Каждый объект такого класса обладает своим поведением, отличным от поведения других объектов того же типа. Тип объекта маскируется, класс полиморфного объекта трудно определить, поэтому нелегко проверить контракт объекта, тем более что в процессе разработки такие объекты могут легко заменяться другими объектами того же типа. При составлении тестов приходится учитывать будущее поведение полиморфного объекта, а для этого надо хорошо знать разрабатываемый проект.

Средства тестирования

Процесс автоматизации коснулся и тестирования. Множество фирм занимается выпуском программных продуктов, предназначенных для тестирования. Наиболее известны компании: Parasoft Corporation, TestQuest, Optimyz Software, Segue Software, Compuware Corporation, AutomatedQA Corporation. Ими и другими фирмами создано множество программных продуктов, облегчающих и автоматизирующих этот процесс. Мы уже упоминали JUnit, JTest, CppUnit. К ним можно добавить программные продукты Rational Functional Center, Insure++, ITG Center, SilkPerformer, CARS. Это только малая часть средств автоматизации тестирования и генерации тестов.

Такое многообразие программных инструментов тестирования объясняется не только важностью и актуальностью самой задачи тестирования, но и тем, что эта задача сильно специализирована. Почти каждый новый программный продукт ставит новые задачи тестировщикам. Для их решения приходится разрабатывать новое средство автоматизации или значительно моди-

фицировать имеющиеся средства. Некоторые из этих новых средств получают дальнейшее развитие и распространение, другие завершают свой жизненный цикл с завершением тестирования.

Обеспечение качества программного продукта

Задача тестирования — это часть более общей задачи обеспечения качества программного обеспечения, обозначаемой аббревиатурой QA (quality assurance). Тестирование только выявляет уже сделанные ошибки, обеспечение качества призвано предотвратить их. Но это не единственное, чем занимаются отделы QA компьютерных фирм. Обеспечить надлежащее качество программного продукта может только целый комплекс мер. Такие меры мы уже упоминали в предыдущих главах. Это четкое определение требований к продукту, которые не надо будет перерабатывать. Это и продуманное проектирование разработки, не нуждающееся в дальнейшем изменении. Это выбор языка программирования и инструментальных средств программирования, предотвращающих ошибки и дисциплинирующих разработчиков. Это правильный подбор команды разработчиков, которые будут тянуть лямку разработки не так, как лебедь, рак и щука.

Задача специалистов по QA — свести все эти меры воедино, разработать корпоративные стандарты качества, обеспечить качественное проектирование, а затем и качественную реализацию проекта. Во время работы над проектом задачей отдела QA становится сквозной контроль качества разработки.

Фирма-разработчик может выбрать для обеспечения качества своих проектов один из международных стандартов качества: ISO 9000, ISO 9001, CMM, CMMI. Эти стандарты регламентируют и методику разработки, обеспечивающую надлежащее качество конечного продукта, и методику проверки качества на всех промежуточных этапах реализации проекта и по окончании проекта. Фирмы, следующие международным стандартам, могут сертифицироваться международными организациями, разработавшими эти стандарты. Наличие сертификата свидетельствует о надежности фирмы-разработчика и гарантирует заказчику высокое качество сделанного для него программного продукта.

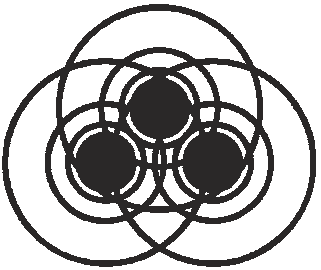
Упражнения

8. Протестируйте программу, вычисляющую значение Y по формуле $Y = \frac{a+b}{a-b} + \frac{1}{ab}$ для всевозможных действительных чисел a и b.

9. Пусть имеется функция `triangleArea(double a, double b, double c)`, вычисляющая площадь треугольника по длинам его сторон. Составьте набор тестов для проверки ее работы.
10. Изменится ли ваш набор тестов, если вы будете знать, что в функции `triangleArea()` площадь треугольника вычисляется по формуле Герона?
11. Напишите набор тестов для нахождения ошибок в функции, которой передается дата рождения сотрудника, а она сообщает, будет ли очередная годовщина его рождения на следующей неделе.

Контрольные вопросы

1. Какова разница между синтаксическими и семантическими ошибками?
2. В чем разница между ошибками времени компиляции и ошибками времени выполнения программы?
3. Что называется отладкой программы?
4. Что такое тестирование программы?
5. Чем отладка программы отличается от ее тестирования?
6. Какие этапы проходит процесс отладки?
7. Какие правила, облегчающие отладку, вы можете предложить?
8. Почему разработчики не должны сами тестировать свой продукт?
9. Почему, тем не менее, unit-тестирование проводят сами разработчики программного продукта?
10. Достаточно ли для полного тестирования проделать набор тестов, в которых каждый оператор программы обязательно выполняется хотя бы один раз?
11. На каком этапе разработки можно начинать тестирование продукта?
12. Какие виды тестирования различают специалисты?
13. Каковы особенности тестирования объектно-ориентированных программ?



Глава 21

Верификация программы

Из материала предыдущей главы уже стало ясно, что ни добросовестная отладка, ни всестороннее тестирование не могут доказать правильность работы программы. С помощью отладки и тестирования можно найти и исправить ошибки, но их отсутствие надо доказывать другими методами. Тестирование показывает верность работы программы только на тестовых наборах, а создать и выполнить тестовые наборы, исчерпывающие все множество данных, невозможно даже для тривиальных программ. Доказательство правильности работы программы для всех исходных и промежуточных данных, так называемая *верификация* (verification) программы, требует теоретического анализа программы и логических рассуждений, сходных с теми, что применяются в математике для доказательства теорем.

Основная идея верификации состоит в том, чтобы доказать соответствие текста программы требованиям, предъявленным к ней. По сути дела, доказывается правильность перевода с языка, на котором изложены требования, на язык программирования, на котором записан исходный текст программы. Чем более точно и строго изложены требования, тем легче провести верификацию программы. Лучше всего, если требования записаны на каком-либо псевдокоде, специализированном языке описаний или начерчены в виде диаграмм.

Для верификации программ разработано несколько подходов.

- *Аналитический подход*, при котором проводится верификация уже готовой программы или отдельного модуля, выполняется сравнение его с требованиями к модулю. Специалисты называют этот процесс *валидацией* (validation) программы. Обычно валидация проводится вместе с тестированием, при этом доказывается правильность наиболее критических участков программы.

- *Синтетический подход*, при котором в начале разработки каждого модуля сразу же проверяется его корректность, идет строгий контролируемый перевод требований на язык программирования. Именно этот подход специалисты называют верификацией программы.
- *Верификация вычислений* доказывает правильность вычисления выражений, разветвлений и циклов, размеров массивов и буферов. Ее своеобразие заключается в том, что это доказательство правильности работы самой программы, а не ее соответствия требованиям.

Хотя теоретики верификации склоняются к синтетическому подходу, на практике гораздо чаще применяется аналитический подход, валидация программы. Каждый разработчик уверен в правильности создаваемого им кода, ведь он долго обдумывал логику выполнения программы и тщательно записывал ее на языке программирования. Поэтому он уверен, что никакой дополнительной проверки и анализа его кода не требуется. Только после того как написанная программа начинает странно себя вести, разработчики приходят к мысли о необходимости ее верификации.

Итак, специалисты различают верификацию и валидацию программы. Мы оставим только первый термин, поскольку он широко распространен и уже вошел во все словари. Слово "валидация" применяется гораздо реже и только в специальной литературе. Может быть, оно тоже будет применяться повсеместно, но пока не будем усложнять текст книги множеством терминов.

Верификация алгоритма

Проще всего доказать правильность работы отдельной процедуры, функции или метода класса. Верификация процедуры, функции или метода сводится к верификации заложенного в них алгоритма, записанного на выбранном языке программирования. Поэтому в первую очередь возникает вопрос: правильно ли исходный текст компилируется в машинный код и отрабатывается компьютером? Это серьезный и важный вопрос, но он относится к разработчикам компилятора. Выбрав тот или иной компилятор, мы надеемся, что он хорошо проверен, и рассчитываем на правильность его работы.

Во вторую очередь возникает вопрос о правильности синтаксиса программы. Этот вопрос легко решается с помощью компилятора. Современный компилятор всеобъемлюще проверяет синтаксис программы и просто не сможет откомпилировать синтаксически неверную программу. Большинство компиляторов можно перевести в отладочный режим, в котором проводится наиболее тщательная проверка. Если компилятор не сообщил об ошибках, то можно быть уверенным, что с точки зрения языка программирования все сделано правильно.

Остается самый сложный вопрос — о семантике процедуры, функции или метода класса, о смысле выполняемых ими действий. Здесь надо, прежде

всего, четко определить то, что называется контрактом метода (*см. гл. 10*). Это, во-первых, условия, которым должны удовлетворять входные данные. Они называются *предусловиями* метода. Во-вторых, условия, накладываемые на результаты работы метода. Это *постусловия* метода. Контракт метода четко определяет два допустимых множества значений: множество входных данных и множество выходных данных. Дальнейшая проверка метода может учитывать только значения, входящие в эти множества, что значительно облегчит верификацию.

Теория верификации использует строгие методы доказательства, изучаемые математической логикой. Для проверки предусловия определяется логическая функция $P(X)$, определенная на множестве входных значений метода M. Эта функция возвращает истину, если набор входных значений X удовлетворяет предусловию метода M, и ложь, если X не удовлетворяет предусловию. Такие функции называются в математической логике *предикатами*. Постусловие выражается предикатом $Q(X, Y)$ от двух переменных: набора входных значений X и набора выходных значений Y метода M.

С точки зрения математической логики, для того чтобы убедиться в правильности работы модуля, надо доказать, что всякий набор входных значений X, удовлетворяющий предикату $P(X)$, преобразуется проверяемым методом M в набор выходных значений Y, удовлетворяющий предикату $Q(X, Y)$. Два предиката P, Q и метод M образуют так называемую *тройку Хоара*. Тройку Хоара тоже можно считать предикатом, дающим истину, если метод выполняет свой контракт. Итак, с точки зрения математической логики верификация метода заключается в доказательстве истинности троек Хоара для данного метода.

К сожалению, в такой общей постановке, для произвольных методов, задача доказательства истинности троек Хоара алгоритмически неразрешима, т. е. нельзя выработать общий алгоритм такого доказательства. Но для некоторых классов процедур, функций и методов разработаны достаточно эффективные методы верификации. Глубокое изучение этих методов требует знания математической логики, теории алгоритмов, теории автоматов. Эти дисциплины изучаются после общего курса "Алгоритмические языки и программирование", поэтому мы не будем давать строгое математическое изложение этих методов, а ограничимся общим описанием.

Метод индуктивных утверждений

Метод индуктивных утверждений, предложенный независимо друг от друга Р. Флойдом (R. W. Floyd) и П. Науром (P. Naur) в 1966 году, позволяет свести доказательство истинности троек Хоара к доказательству некоторого конечного числа утверждений, записанных в терминах алгоритма, реализуемого проверяемой функцией, процедурой или методом.

Большинство алгоритмов содержат одно или несколько разветвлений. Поэтому для разных наборов входных значений X выполнение метода может проходить по разным путям, приводящим к соответствующему выходному набору значений Y. Каждый путь выполнения алгоритма называется *трассой вычислений*. Тройка Хоара для данного метода M будет истинной, если истинны тройки Хоара для каждой трассы вычислений.

Трассы вычислений разбивают все предусловие $P(X)$ на несколько предусловий P_1, P_2, P_3 и т. д., проходящих по определенной трассе. Скажем, для метода нахождения корней квадратного уравнения $ax^2 + bx + c = 0$ есть две трассы вычислений, приводящие к отрицательному или неотрицательному дискриминанту. Соответствующие предусловия определяются знаком выражения $b^2 - 4ac$. К этим двум трассам надо еще добавить трассы вычислений, выполняемых в вырожденных случаях нулевых коэффициентов уравнения.

Итак, остается доказать истинность тройки Хоара для каждой трассы, что достигается простым последовательным доказательством истинности выполнения каждого оператора. Как мы уже договорились выше, ответственность за это возлагается на разработчиков компиляторов.

Дело обстоит сложнее для алгоритмов, содержащих циклы, которые могут привести к появлению огромного количества трасс вычислений. Здесь не обойдешься простым перебором. Надо искать какое-то теоретическое решение. Выход был найден во введении дополнительных условий, названных *индуктивными утверждениями цикла*. В начале или середине каждого цикла выбирается контрольная точка. В начале метода и в его конце тоже выбираются контрольные точки. В каждой контрольной точке K определяется индуктивное утверждение U_K. В первой контрольной точке, в начале метода, это предусловие $P(X)$, в последней, в конце метода, — постусловие $Q(X, Y)$. В остальных контрольных точках K_i, находящихся в циклах, в качестве утверждения U_{Ki} выбирается какой-нибудь инвариант цикла, т. е. выражение, связывающее переменные, вычисляемые в цикле, и не меняющееся при итерациях цикла.

После этого на пути между каждыми двумя соседними контрольными точками K_{i-1} и K_i индуктивно задается некоторое *условие верификации*, обозначим его R_i. Внутри циклов путь замыкается в кольцо, так что точки K_{i-1} и K_i совпадают. Только в начале работы цикла и при последней итерации это кольцо разрывается.

Пусть на пути между двумя соседними контрольными точками лежат операторы A_1, A_2, ..., A_n. Сначала в качестве условия R_i берем истинность утверждения U_{Ki}. Потом просматриваем операторы A_n, A_{n-1}, ..., A_1 в обратном порядке. После просмотра очередного оператора A_j ставим промежуточное условие R_{ij}, заключающееся в истинности вывода (импликации) предыдуще-

го промежуточного условия из постусловия оператора A_j. Окончательное условие R_i состоит в истинности вывода последнего промежуточного условия из утверждения U_{Ki-1}.

Таким образом, условие R_i индуктивно зависит от условий правильного выполнения операторов A_1, A_2, ..., A_n. После наложения условий на все пути данной трассы вычислений, доказательство истинности тройки Хоара сводится к проверке всех условий R_i.

Пример

Рассмотрим для примера алгоритм перевода целого числа M из десятичной системы счисления в двоичную систему, показанный на блок-схеме рис. 1.15. Результат работы алгоритма, двоичное число, записывается в строку символов N. Предусловием алгоритма $P(M)$ служит требование, чтобы на его вход подавались только целые числа M. Постусловие $Q(M, N)$ заключается в том, чтобы результирующая строка символов N содержала только нули и единицы, соответствующие двоичной записи числа M.

Введем три контрольных точки: контрольную точку K_1 в начале блок-схемы, сразу после ввода числа M, контрольную точку K_2 в начале цикла перед условием $M == 0$ и контрольную точку K_3 в конце блок-схемы перед выводом строки N. В контрольной точке K_2 зададим утверждение U_{K2} — какой-нибудь инвариант цикла.

У нас получились три пути. Первый путь ведет от начала алгоритма — контрольной точки K_1 в начало цикла — к контрольной точке K_1. Условием R_1 на этом пути служит предусловие $P(M)$, которое должно быть выполнено по контракту алгоритма. Второй путь проходит внутри цикла от контрольной точки K_2 через операторы цикла опять к контрольной точке K_2. Условием R_2 на этом пути служит инвариант цикла. Он тоже выполняется автоматически. Третий путь идет от контрольной точки K_2 при последнем выполнении цикла к выходу из программы — контрольной точке K_1. Условием R_3 на этом пути является постусловие алгоритма $Q(M, N)$, которое тоже всегда должно выполняться.

Итак, все три условия между контрольными точками выполняются тривиальным образом, что доказывает методом индуктивных утверждений правильность алгоритма перевода целых десятичных чисел в двоичную систему счисления.

Разумеется, в таком простом примере можно обойтись без строгой верификации, но в сложных запутанных алгоритмах проверка условий может быть далеко не простым делом.

Как видите, верификация даже небольшой программы — процесс длительный и трудоемкий. В обычных проектах вряд ли будет выделяться время на такую кропотливую работу. Ее проводят, главным образом, для программ-

ных продуктов с особыми требованиями к надежности или в тех случаях, когда даже после тестирования нет уверенности в правильности работы программы.

Использование верифицированных фрагментов

Метод индуктивных утверждений просматривает весь программный модуль, устанавливая контрольные точки и индуктивно наращивая условия, которые должны выполняться между ними. Между тем, некоторые участки модуля, возможно, уже были верифицированы. Их верификацию можно не проводить вторично.

Такой подход широко применяется в математике. Никто не доказывает теоремы, основываясь на аксиомах. Доказательство новой теоремы всегда опирается на утверждения, уже доказанные ранее.

Трудами многих математиков, логиков и специалистов по компьютерным наукам доказана правильность и даже оптимальность большинства стандартных алгоритмов. Алгоритмы суммирования, нахождения наибольшего и наименьшего элемента массива, линейный и бинарный поиск в массивах, схема Горнера и многие другие, гораздо более сложные алгоритмы уже верифицированы. Правильность фрагментов программ, использующих эти алгоритмы, можно считать доказанными.

Некоторые фрагменты программы могут только незначительно отличаться от стандартных верифицированных алгоритмов. Верификацию таких фрагментов можно упростить, если доказать корректность изменений, внесенных в верифицированный алгоритм. Такая работа постоянно ведется, и в теории верификации программ доказана корректность большого числа преобразований алгоритмов. Например, корректна вставка в верифицированную программу правильно работающего оператора, корректно упрощение инварианта цикла, изменение нижней границы цикла `for`, правильное изменение оператора присваивания.

Знание стандартных алгоритмов, правильность которых уже доказана, и знание корректных правил преобразования алгоритмов сильно облегчают верификацию программ.

Упражнения

1. Пусть имеется функция, вычисляющая площадь треугольника по заданным длинам его сторон. В ней заложена проверка всех вырожденных случаев, отдельно рассматриваются прямоугольный и равносторонний треугольники. Разбейте предусловие этой функции по трассам вычислений.

2. Докажите методом индуктивных утверждений правильность алгоритма Эвклида для нахождения наибольшего общего делителя двух целых чисел.
3. Докажите методом индуктивных утверждений правильность схемы Горнера для вычисления многочлена в заданной точке.
4. Докажите методом индуктивных утверждений правильность алгоритма нахождения наибольшего элемента массива и его индекса в массиве.

Доказательство завершения цикла

Зацикливание программы — очень распространенная и трудно устраняемая ошибка. Запустив программу, мы долго смотрим на экран дисплея, безнадежно ожидая результатов ее работы. Через некоторое время мы убеждаемся, что программа зациклилась, прекращаем выполнение программы и переходим к ее отладке, чтобы найти причину неправильного выхода из цикла.

Тестирование позволяет увериться в том, что циклы правильно работают и завершаются только на тестовых наборах. Но как убедиться, что после нескольких месяцев работы не найдется какое-нибудь немыслимое сочетание значений переменных, при котором цикл не сможет завершиться, и "повесит" программу? Для ответа на этот вопрос разработаны различные методы анализа завершения циклических вычислений. Наиболее распространены два из них: метод Флойда и метод счетчиков.

Метод Флойда

Метод Флойда основан на методе индуктивных утверждений и развивает его. Согласно этому методу в программе также расставляются контрольные точки и определяются условия на путях между соседними контрольными точками. После этого в циклическом пути между парой соседних контрольных точек ставится еще одно условие — условие завершения цикла.

Для его задания сначала вводится функция, определенная на множестве переменных, меняющихся в цикле, и принимающая значения в каком-нибудь упорядоченном множестве, не имеющем никакой бесконечной убывающей или возрастающей последовательности элементов. Например, годится множество натуральных чисел с обычной упорядоченностью по возрастанию чисел, или множество подмножеств конечного множества, упорядоченное по включению множеств. Такая функция называется *ограничивающей функцией*. Ограничивающая функция рано или поздно исчерпает множество своих значений, и это послужит сигналом к завершению цикла.

Например, для проверки завершения цикла при нахождении корня нелинейного уравнения методом бисекции, записанным в листинге 3.2, достаточно взять в качестве ограничивающей функции `fabs(b - a)`, т. е. функ-

цию, возвращающую длину отрезка $[a,b]$. Действительно, эта длина уменьшается вдвое при каждой итерации цикла, но не может стать меньше заданной наперед константы `eps`.

После определения ограничивающей функции индуктивно определяется условие завершения цикла на заданном пути. Это делается так же, как и определение условия индуктивного утверждения, но в индуктивный процесс вовлекается ограничивающая функция.

Наконец, теория верификации утверждает, что цикл не станет бесконечным, если условия завершения для всех циклических путей истинны. В тривиальном примере листинга 3.2 условие завершения цикла `fabs(b - a) < eps` как раз и записано в нем. Оно проверяется автоматически.

Метод счетчиков

Метод счетчиков предлагает вводить в циклы временные переменные, называемые *счетчиками*, которые определяются перед началом цикла и увеличиваются при каждой итерации. Индуктивные утверждения в циклических контрольных точках должны содержать условия ограниченности переменных-счетчиков.

После этого обычным образом применяется метод индуктивных утверждений. При этом вместе с верификацией программы доказывается завершение циклов.

Метод счетчиков кажется проще и удобнее метода Флойда, но при его практическом применении часто бывает трудно определить индуктивное утверждение, которое должно быть инвариантом цикла и должно задавать условие ограниченности счетчика.

Например, для той же программы метода бисекции, записанной в листинге 3.2, можно поставить в цикл переменную-счетчик `count`, просто подсчитывающий число итераций цикла, т. е. принимающий значения 1, 2, 3 и т. д. Тогда условие ограниченности счетчика может быть записано в виде: $\texttt{count} < \log_2(\texttt{(b - a)/eps})$.

Упражнения

5. Докажите завершение цикла в программе, отыскивающей наибольший общий делитель двух целых чисел по алгоритму Эвклида.
6. Докажите завершение цикла в программе, уточняющей корень нелинейного уравнения методом Ньютона (методом касательных).
7. Докажите завершение цикла при сортировке массива методом просеивания, записанном в листинге 4.11.
8. Докажите завершение цикла в методе быстрой сортировки массива, записанном в листинге 5.5.

Автоматизация верификации

Изложенные выше методы верификации похожи на запись алгоритма доказательства верности некоторого логического текста. Это наводит на мысль о том, чтобы написать программу-верификатор, выполняющую такой алгоритм. На вход такой программы подается текст другой программы, ее предусловия и постусловия, а программа-верификатор делает вывод о ее верности. Такие программы действительно разрабатываются с шестидесятых годов прошлого столетия, но из-за необычайной сложности задачи они получили очень ограниченное применение.

Действительно, программе-верификатору необходимо подать на вход предусловие и постусловие. Сразу же возникает вопрос, на каком языке их записать? Этот вопрос вполне разрешим. Условия можно записать на каком-либо языке записи логических высказываний и предикатов.

Далее надо задать индуктивные утверждения. Кто это должен сделать? Подавать их на вход программы или генерировать внутри программы? Индуктивные утверждения зависят от смысла программы, поэтому большинство верификаторов требует задания их в числе входных данных.

Затем надо вывести условия верификации. Программный модуль, выполняющий эту работу, должен уметь делать логические выводы. Такие модули уже созданы, но каждому логическому модулю нужен набор аксиом, из которых он будет делать выводы. Создать набор аксиом для языка программирования уровня C++ — это очень непростая задача.

Наконец, надо провести доказательство истинности условий верификации. Эта задача относится к области искусственного интеллекта и находится еще только в начальной стадии разработки.

По этим и другим причинам процесс верификации автоматизирован только для самых примитивных языков программирования. Если же программа-верификатор написана для верификации программ, написанных на промышленном языке программирования, то оказывается, что она обрабатывает только очень ограниченное подмножество промышленного языка.

Достоверность верификации

Допустим, что мы провели верификацию программы методом индуктивных утверждений или каким-нибудь другим методом. Можно ли утверждать, что в ней нет ошибок? Звучит парадоксально, но бывали случаи, когда в верифицированных программах, правильность которых, казалось бы, была доказана, находились ошибки. Дело в том, что верификация основана на предусловии и постусловии метода. Если они заданы неправильно, то верификация может пройти успешно, а ошибки могут остаться. Даже если

предусловие верно, нет гарантии, что оно будет выполнено при работе программы, а верификация не проверяет реакцию метода на неправильное предусловие.

Далее, проверяемый метод может обращаться к другому методу. Верификация не проверяет контракт вызываемого метода. Даже если контракт верен, могут появиться ошибки в типах или порядке следования аргументов метода. Верификация не учитывает ошибки округления, переполнение, машинный ноль и другие особенности вычислений, которые могут сильно повлиять на выполнение программы. Наконец, человек, доказывающий условия верификации, может ошибиться в доказательстве и счесть ложное условие истинным. Не спасет от такой ошибки и программа-верификатор. В ней тоже могут содержаться ошибки, она сама нуждается в верификации.

Таким образом, сама верификация требует тщательной проверки, каждый ее шаг должен быть истинным, а чтобы достичь этого, нужны двойные усилия и дополнительные затраты времени.

Модельный подход к верификации

Те методы верификации программы, которые мы рассматривали до сих пор, проводят так называемую *дедуктивную верификацию*. Она основана на как можно более строгом и точном описании требований к программе и на дальнейшем математическом доказательстве правильности реализации программой этих требований. Для доказательства привлекаются развитые методы теории множеств, математической логики, теории доказательств.

Дедуктивная верификация — это мощное, но очень громоздкое средство доказательства правильности программы. Успех дедуктивной верификации целиком зависит от точности математического описания задачи, а его очень нелегко сделать. Методы математической логики требуют детального рассмотрения каждого логического вывода, а такие выводы в реальных программах насчитываются сотнями. Математические доказательства трудны, в чем убедился каждый выпускник средней школы.

Все эти сложности приводят к тому, что дедуктивная верификация проводится пока только для сравнительно небольших программ, в основном тех, которые ориентированы на обработку потока данных. Верификация более сложных задач, особенно ориентированных на поток управления, сталкивается со значительными трудностями и требует больших затрат времени.

В таких случаях часто бывает удобнее применить *модельный подход* к верификации. При этом подходе не строится математическое описание задачи, а создается модель программы в виде абстрактной машины или конечного автомата или сети Петри. Потом требования к программе проверяются для каждого возможного состояния автомата.

Модельный подход можно применить не ко всей программе, а к отдельным ее частям, построив для них отдельные модели. Его можно даже применять к еще не готовой программе, построив модель по ее проекту. Модель удобна для доказательства правильности работы программы, ее можно более или менее детализировать, облегчая процесс доказательства.

Однако у модельного подхода есть неустранимый недостаток. Он проводит верификацию модели, а не самой программы, поэтому доказывает только правильность модели. Ценность такого доказательства целиком зависит от построения модели. Если модель слишком близка к программе, то она становится громоздкой и неудобной в использовании, теряются все преимущества модельного подхода. Если модель слишком упрощена, то она не отражает полностью работу программы и возникает сомнение в том, что правильность модели влечет правильность программы.

Итак, успех модельного подхода всецело зависит от построения модели верифицируемой программы. Создание модели требует высокой квалификации и значительного опыта, позволяющего выбрать правильную степень приближения модели к реальной программе и не допустить искажения протекающих в ней процессов.

Для облегчения моделирования часто создают не одну универсальную модель программы, а несколько специализированных моделей, позволяющих доказать те или иные свойства программы. Например, для проверки пригодности программы к решению поставленных перед ней задач моделируются внешние воздействия на программу. Для доказательства стабильности программы создается модель окружающей среды.

Специализированные модели гораздо проще универсальной модели и они могут гораздо точнее моделировать отдельные функции программы. Кроме того, некоторые свойства программы вообще плохо поддаются моделированию. Например, при модельном подходе к доказательству точности работы программы всегда будет происходить потеря точности за счет огрубления при переходе к модели. Поэтому на практике при доказательстве правильности работы больших программных продуктов проводится и дедуктивная верификация и построение большого числа специализированных моделей.

Контрольные вопросы

1. Что называется верификацией программы?
2. Каковы основные подходы к верификации программы?
3. Что такое валидация программы?
4. Что такое тройка Хоара?

5. Где следует расставлять контрольные точки при верификации по методу индуктивных утверждений?
6. Как задаются условия верификации в методе индуктивных утверждений?
7. Как можно доказать отсутствие зацикливания в программе?
8. Что такое дедуктивная верификация?
9. Каковы преимущества и недостатки модельного подхода к верификации?
10. Какие преимущества при верификации дает знание особенностей алгоритма, заложенного в программу?
11. Можно ли полностью доверять изложенным в этой главе методам верификации?
12. Возможна ли автоматизация верификации программ?

Литература

1. Бадд Т. Объектно-ориентированное программирование в действии: Пер. с англ. — СПб.: Питер, 1997. — 464 с., ил.
2. Брукс Ф. Мифический человеко-месяц или как создаются программные системы: Пер. с англ. — М.: Символ-Плюс, 2001. — 304 с.
3. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд.: Пер. с англ. — М.: Издательство "Бином", СПб.: Невский диалект, 1998. — 560 с., ил.
4. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. — М.: Мир, 1989. — 360 с., ил.
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидис Дж. Приемы объектно-ориентированного проектирования: Пер. с англ. — СПб.: Питер, 2001. — 368 с., ил.
6. Керниган Б., Ритчи Д. Язык программирования Си: Пер. с англ. — СПб.: Невский диалект, 2001. — 352 с.
7. Кнут Д. Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: Пер. с англ. — М.: Вильямс, 2000. — 720 с.
8. Кнут Д. Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.: Пер. с англ. — М.: Вильямс, 2000. — 832 с.
9. Кнут Д. Искусство программирования, том 3. Сортировка и поиск, 2-е изд.: Пер. с англ. — М.: Вильямс, 2000. — 832 с.
10. Страуструп Б. Язык программирования C++, 3-е изд.: Пер. с англ. — СПб.: М.: Невский диалект, М.: Издательство "Бином", 1999. — 991 с., ил.

Предметный указатель

A

Abstract method 276
Access methods 238
Actor 430
Allocator 387

B

Bit-fields 191
Bitwise 70
Bpwin 436
Breakpoints 449
Buffer 365
Build 453

C

Capacity 388
Class 225
Class fields 226
Class methods 226
Class template 301
Compound assignment operators 72
Concatenation 354
Contract 237
Current position 161

D

Dangling else 81
Debugger 452
Debugging 440
Deque 394
DFD 436
Dictionary 402

E

ERD 436
Erwin 436
Exception 287
Expression 73
Expression statement 78
Extension 233

F

Fall through labels 101

G

Get method 238

H

High cohestion 239

I

IDEF 435
Idenifiers 64
Incapsulation 139
Inheritance 234
Initialization 106
Input stream 167
Instance 227

L

LIFO 316
Low coupling 239

M

Map 402
Message 236
Modularity 237
Multiple inheritance 277

N

Names 64
Namespace 282

O

Object-oriented programming 139
OOD 140
OOP 139
Output stream 167
Overloading 215
Overriding 235

P

Patch 442
Polymorphism 235, 236, 237
Predicate 411
Promotion 67

Q

QA 465
Quality assurance 465

R

Reference 208
Release 453
Responsibility 237
Runtime 287
RUP 430

S

Set 407
Set method 238
Simple assignment operator 72
Size 388
Stack 315, 392
Standard error stream 168
Statement 77
STD 436
Stderr 367
Stdin 367
Stdout 367
Stream 365
String 349
Structure 180
Structured analysis 435
Subclass 234

T

Tag 180
TDP 456
Template 215
Test suite 461
Test-driven programming 456
Testing 441
Throws 293

U

Union 189
Unit-тестирование 441, 454
Use case 431

V

Validation 467
Vector 387
Verification 467
Virtual base class 280

W

Wide char type 350
Widening 68

Z

Zero-defect mindset 462

А

Абстрактный класс 276
Абстракция 224
Алгоритм 16, 20
✧ Евклида 19
Арифметический сдвиг 72
Ассоциативный массив 402

Б

Бинарная функция 421
Бинарный файл 163
Битовые поля 191
Блок-схема 24
Буфер 365

В

Валидация 467
Вариант использования 431
Вектор 387
Верификация программы 467
Виртуальный базовый класс 280
Входной поток 167
Выражение 73
Выходной поток 167

Г

Генератор 421

Д

Двоичная система счисления 43
Дедуктивная верификация 476
Действительная константа 62
Дек 393, 394
Деструктор 263
Десятичная система счисления 42
Дизъюнкция 69
Директива препроцессору 195, 203, 223, 245, 287, 301, 315, 349, 365, 387, 409, 427, 445, 467, 479
Дистрибутив 428
Длина массива 105
Дополнение 70
Дополнительный код отрицательного целого числа 48
Дружественная функция 281
Дружественный:
✧ класс 280
✧ метод 281

Е

Емкость 388

Ж

Журнальный файл 168

З

Закон Деметра 239
Закрытие файла 162
Закрытый член класса 238
Зацепление 239
Защищенный член класса 240

И

Идентификатор 64
Иерархия 232
Имя 64
Инициализация массива 106
Инициализация полей класса 270
Инкапсуляция 139
Исключительная ситуация 287
Исполнитель алгоритма 19
Итерация 30

К

Класс 225
Код символа 52
Кодировка:
✧ ASCII 53
✧ CP 1251, 57
✧ CP 866, 56
✧ ISO 8859-5, 55
✧ KOI8-R 57
✧ Mac Cyrillic 59
✧ Unicode 60

Кодовая таблица 52
Конечность алгоритма 18
Константа 61
Конструктор копирования 247
Контракт 237
Контрольные точки 449
Конъюнкция 69

Л

Линейный алгоритм 27
Линейный список 398
Логические операции 69
Логический сдвиг 72
Логический тип 204

М

Макрос 197, 203, 223, 245, 287, 301, 315, 349, 365, 387, 409, 427, 445, 467, 479
Манипулятор 376
Массив 105, 106
Массовость алгоритма 17
Международный стандарт ISO 8859, 55
Метка 102
Метод 223
Метод класса 226
Методы доступа 238
Множественное наследование 277
Множество 407
Модульное программирование 138
Модульность 237

Н

Набор тестов 461
Надкласс 234
Наследование 234
Нуль-символ 54

О

Объединение 189
Объект 223
Объектно-ориентированное программирование 139
Объектно-ориентированное проектирование 140
Объектно-ориентированный анализ 140, 436
Объект-функция 421
Однозначность алгоритма 18
ООА 140, 436
ООП 139
Оператор 77
✧ break 100
✧ continue 99
✧ варианта 100
✧ присваивания 79
✧ цикла 93
Операция присваивания 72
Ответственность 237
Открытие файла 161
Открытый член класса 238
Отладка 440
Отображение 402
Отрицание 69
Очередь 396
Очередь с приоритетом 397

П

Перегрузка:
✧ операции 217
✧ функции 215
Переопределение метода 235
Перечислимый тип 178
Побитовая дизъюнкция 71
Побитовая конъюнкция 70
Побитовое исключающее ИЛИ 71
Побитовые операции 70
Повышение типа 67
Подкласс 234
Поле 223
Поле класса 226
Полиморфизм 237, 273
Полиморфный тип 311
Помеченный блок 102
Помеченный оператор 102
Поток 365
✧ ввода-вывода 167
Пошаговое выполнение 449
Предикат 411
Прецедент 431

Принцип KISS 240
Пробельный символ 40
Программирование, управляемое тестами 456
Программный комплекс 428
Программный продукт 428
Пространство имен 282
Прототип функции 210
Процедурное программирование 138
Псевдокод 22

Р

Разветвление 30
Размер 388
Расширение класса 233
Расширение типа 68
Режим:
✧ дозаписи файла 163
✧ записи файла 163
✧ чтения файла 163

С

Связность 239
Сдвиг:
✧ влево 71
✧ вправо 71
Символ 62
Скрытый бит 50
Словарь 402
Составное имя 64
Составные операции присваивания 72
Ссылка 208
Стандартный ввод 367
Стандартный вывод 367
Стандартный вывод сообщений 367
Статический метод 255
Стек 315, 392
Строка 63, 349
Структура 180
Структурное программирование 138
Структурный анализ 435
Суперкласс 234
Схема Горнера 39
Сценарий использования 431
Сцепление строк 354

Т

Текстовый файл 163
Текущая позиция файла 161
Тестирование 441, 454
Техническое задание 433
Тип данных 48
Трасса вычислений 470
Тройка Хоара 469

У

Унарная функция 421
Унифицированный процесс разработки 430
Условие верификации 470
Условная операция 73
Условный оператор 79

Х

Характеристика 51

Ц

Целая константа 61
Целое деление 66
Цикл 30

Ч

Чисто виртуальный метод 276
Члены класса 226

Ш

Шаблон:
✧ класса 301
✧ функции 215
Широкий тип 350

Э

Экземпляр класса 227