Г. Г. Рапаков
С. Ю. Ржеуцкая

Программирование на языке Pascal

- *Базовые понятия программирования*
- *Конструкции языка*
- *Структурное и модульное программирование*
- *Массивы, множества, записи, файлы*
- *Динамические структуры данных*
- *Введение в объектно-ориентированное программирование*

Учебное пособие

bhv

Министерство образования Российской Федерации
Вологодский государственный технический университет

Г. Г. Рапаков
С. Ю. Ржеуцкая

Программирование на языке Pascal

*Рекомендовано УМО
по университетскому политехническому образованию
для студентов высших учебных заведений, обучающихся
по специальности 220400 — "Программное обеспечение
вычислительной техники и автоматизированных систем"*

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.068+800.92Pascal
ББК 32.973.26-018.1я7
Р23

Рапаков Г. Г., Ржеуцкая С. Ю.

Р23 Программирование на языке Pascal. — СПб.: БХВ-Петербург, 2004. — 480 с.: ил.

ISBN 5-94157-401-0

Учебное пособие ориентировано на широкий круг читателей, как начинающих знакомство с программированием, так и имеющих в нем достаточный опыт. Необходимое для новичков изложение азов предмета сочетается в книге с подробным и глубоким описанием тонкостей языка Pascal. Издание насыщено примерами и содержит множество полезных рекомендаций. Особое внимание уделено вопросам стиля в программировании, как линейном, так и объектно-ориентированном. Каждую главу завершают контрольные вопросы и задания, сложность которых дозированно возрастает от начальных глав к конечным. Эти материалы могут быть рекомендованы преподавателям информатики средней и высшей школ в качестве методических.

Для школьников старших классов и студентов

УДК 681.3.068+800.92Pascal
ББК 32.973.26-018.1я7

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Галина Смирнова*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 20.10.03.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 38,7.
Тираж 4000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-401-0

© Рапаков Г. Г., Ржеуцкая С. Ю., 2004
© Оформление, издательство "БХВ-Петербург", 2004

Содержание

Введение **1**

Как работать с книгой 2

Часть I. Введение в программирование. Базовые понятия **5**

Глава 1. Основы алгоритмизации **7**

1.1. Алгоритмизация и требования к алгоритму 7
1.2. Способы записи алгоритмов 8
1.2.1. Описательный способ 8
1.2.2. Блок-схемы алгоритмов 9
1.2.3. Пример блок-схемы алгоритма 11
1.2.4. Следование, ветвление, цикл 13
1.3. Контрольные вопросы и задания 13

Глава 2. Язык программирования. Программа **14**

2.1. Язык программирования 14
2.1.1. Исторический обзор 14
2.1.2. Состав языка программирования. Синтаксис и семантика 19
2.1.3. Описание языка 21
2.1.4. Стандарт и реализации языка 23
2.2. Программа. Программный продукт и его характеристики 25
2.3. Жизненный цикл программы 26
2.4. Контрольные вопросы 28

Глава 3. Система программирования **30**

3.1. Компиляторы и интерпретаторы 30
3.2. Этапы компиляции и компоновки 32
3.3. Отладка программы. Типы ошибок 33
3.4. Состав системы программирования. Понятие интегрированной среды разработки 34
3.5. Системы программирования на основе языка Pascal 35
3.6. Контрольные вопросы 36

Глава 4. Программа и данные. Типы данных **38**

4.1. Основные понятия 38
4.1.1. Исполнение программы 38
4.1.2. Биты и байты 40

4.2. Константы и переменные ... 41
4.3. Типы данных и способы внутреннего представления данных ... 42
4.3.1. Концепция типов данных ... 42
4.3.2. Внутреннее представление данных в памяти компьютера ... 43
4.3.3. Классификация типов ... 48
4.4. Контрольные вопросы и задания ... 49

Часть II. Основы программирования на языке Pascal ... 51

Глава 5. Типы данных и константы языка Pascal ... 53

5.1. Классификация типов. Стандартные типы: порядковые и вещественные ... 54
5.2. Целые типы. Логический тип ... 55
5.3. Символьный тип. Строки символов ... 56
5.4. Вещественные типы ... 58
5.5. Типы данных, определяемые программистом ... 59
5.6. Перечисляемый и интервальный типы ... 60
5.7. Контрольные вопросы и задания ... 62

Глава 6. Программа на языке Pascal ... 64

6.1. Алфавит языка ... 64
6.2. Лексика языка ... 65
6.3. Структура программы ... 69
6.3.1. Общие сведения ... 69
6.3.2. Раздел *uses* ... 71
6.3.3. Раздел описания меток ... 71
6.3.4. Раздел описания констант ... 71
6.3.5. Раздел описания типов данных ... 73
6.3.6. Раздел описания переменных ... 74
6.3.7. Раздел описания процедур и функций ... 75
6.3.8. Раздел операторов ... 76
6.4. Контрольные вопросы и задания ... 77

Глава 7. Простые (линейные) программы ... 79

7.1. Пример учебной программы ... 79
7.2. Ввод и вывод данных ... 80
7.2.1. Инструкции ввода и вывода ... 80
7.2.2. Формат вывода ... 83
7.3. Оператор присваивания ... 84
7.4. Арифметические выражения ... 85
7.4.1. Арифметические операции ... 85
7.4.2. Побитовые операции ... 88
7.4.3. Приоритет операций ... 90
7.4.4. Стандартные арифметические функции ... 91

7.4.5. Типы в арифметических выражениях. Автоматическое преобразование типов ... 93
7.4.6. Явное преобразование типов. Переполнение ... 94
7.5. Полезные формулы ... 97
7.6. Примеры программ ... 97
7.6.1. Вычисления по формулам ... 97
7.6.2. Выделение цифр целого числа ... 98
7.6.3. Перестановка значений переменных ... 99
7.6.4. Определение размера ячейки памяти ... 100
7.7. Контрольные вопросы и задания ... 101

Глава 8. Программирование разветвлений ... 104

8.1. Логические выражения ... 104
8.2. Составной оператор ... 107
8.3. Условный оператор ... 108
8.4. Оператор выбора ... 112
8.5. Оператор безусловного перехода. Пустой оператор ... 114
8.6. Примеры программ ... 117
8.6.1. Разные задачи ... 117
8.6.2. День недели по заданной дате ... 118
8.7. Контрольные вопросы и задания ... 119

Глава 9. Циклические программы ... 123

9.1. Цикл с постусловием ... 124
9.2. Цикл с предусловием ... 129
9.3. Цикл с параметром ... 132
9.4. Вложенные циклы ... 137
9.5. Примеры программ ... 138
9.5.1. Выделение цифр целого числа ... 138
9.5.2. Обработка числовых данных ... 139
9.5.3. Формирование числовых последовательностей ... 140
9.5.4. Вычисление сумм бесконечных убывающих последовательностей. Особенности арифметики с плавающей точкой ... 141
9.5.5. График функции на текстовом экране ... 145
9.5.6. Задачи на перебор всех вариантов ... 146
9.5.7. "Римские цифры" ... 148
9.6. Контрольные вопросы и задания ... 151

Часть III. Структурное и модульное программирование ... 155

Глава 10. Процедуры и функции ... 157

10.1. Общие сведения о подпрограммах ... 157
10.2. Процедуры ... 158
10.3. Механизм передачи параметров ... 161
10.3.1. Параметры-значения ... 162
10.3.2. Параметры-переменные ... 163
10.3.3. Параметры-константы ... 164

10.4. Функции 165
10.5. Область видимости и время жизни переменной 168
10.5.1. Подробнее о распределении памяти 168
10.5.2. Вложенные процедуры и функции 170
10.5.3. Глобальные, автоматические и статические переменные. Определения 171
10.5.4. Побочные эффекты вызова подпрограмм 173
10.5.5. Рекомендации по разработке программ 174
10.6. Рекурсия 174
10.7. Дополнительные сведения о процедурах и функциях 176
10.7.1. Опережающее объявление 176
10.7.2. Параметры-процедуры и параметры-функции 177
10.7.3. Нетипизированные параметры-переменные 179
10.8. Примеры программ 180
10.8.1. Числовые последовательности 181
10.8.2. Численные методы решения математических задач 182
10.8.3. Случайные числа 186
10.8.4. Игра "Ханойские башни" 188
10.9 Контрольные вопросы и задания 189

Глава 11. Структуризация в программировании 192

11.1. Теорема структуры 192
11.2. Основы структурного программирования 193
11.2.1. Методы структурного программирования 193
11.2.2. Пример разработки программы 195
11.3. Контрольные вопросы и задания 200

Глава 12. Библиотечные модули 203

12.1. Компиляция и компоновка программы 203
12.2. Понятие модуля. Преимущества модульного программирования 204
12.3. Структура модуля 205
12.4. Снова о глобальных, локальных и статических переменных 208
12.5. Пример модуля 209
12.6. Раздельная компиляция модулей 212
12.6.1. Компиляция модулей в Turbo Pascal 212
12.6.2. Управление модулями в Delphi. Понятие проекта 213
12.6.3. Распространение готовых модулей 214
12.7. Стандартные модули 214
12.8. Контрольные вопросы и задания 216

Часть IV. Структуры данных и алгоритмы обработки 219

Глава 13. Массивы 221

13.1. Общие сведения 221
13.2. Описание массивов 222
13.2.1. Описание массива в разделе *var* 223

13.2.2. Предварительное описание типа массива 223
13.2.3. Инициализация массивов начальными значениями 224
13.2.4. Подробнее о выделении памяти 225
13.3. Действия над массивами 227
13.3.1. Заполнение массива данными 227
13.3.2. Вывод элементов массива 229
13.3.3. Обработка одномерных массивов 229
13.3.4. Действия с двумерными массивами 233
13.3.5. Перестановки элементов в массиве 236
13.3.6. Сортировка массива 237
13.3.7. Быстрый поиск в упорядоченных массивах 241
13.3.8. Удаление и вставка элементов в массив 242
13.3.9. Умножение матриц 244
13.4. Массивы как параметры подпрограмм 247
13.5. Примеры программ 249
13.5.1. Перевод числа в двоичную систему 249
13.5.2. Решение системы линейных уравнений 249
13.6. Контрольные вопросы и задания 252

Глава 14. Обработка строк текста 256

14.1. Типы данных *char* и *string* 256
14.1.1. Символьный тип 256
14.1.2. Строковый тип 257
14.2. Операции над строками 261
14.2.1. Операция сцепления (+) 261
14.2.2. Операции отношения 263
14.3. Строковые процедуры и функции 264
14.3.1. Процедуры удаления и вставки символов 265
14.3.2. Функции для работы со строками 266
14.3.3. Процедуры преобразования типов 267
14.4. Примеры программ 268
14.4.1. Вставка, удаление и замена фрагментов текста 268
14.4.2. Преобразование строчных букв в заглавные 270
14.4.3. Удаление комментариев из строки программы 271
14.4.4. Вычисление арифметического выражения 274
14.5. Контрольные вопросы и задания 278

Глава 15. Множества 281

15.1. Понятие множества 281
15.2. Операции над множествами 283
15.3. Примеры программ 287
15.3.1. Формирование случайных неповторяющихся чисел 287
15.4. Контрольные вопросы и задания 289

Глава 16. Записи 291

16.1. Работа с записями. Примеры 291
16.1.1. Определение типа "запись". Правила работы с записями 291

16.1.2. Пример использования массива записей 294
16.1.3. Модуль для выполнения операций с комплексными числами 296
16.2. Записи с вариантами. Пример 299
16.3. Контрольные вопросы и задания 303

Глава 17. Файлы 306

17.1. Некоторые сведения о файловой системе 306
17.1.1. Имя и расширение файла 307
17.1.2. Каталоги 307
17.1.3. Устройства 310
17.2. Описание файлового типа 311
17.2.1. Виды файлов. Файловая переменная 311
17.2.2. Указатель. Доступ к файлам 311
17.3. Общая схема работы с файлами 312
17.3.1. Последовательность действий 312
17.3.2. Общие процедуры и функции 313
17.3.3. Процедуры для работы с каталогами 315
17.3.4. Процедуры и функции модуля *dos* 315
17.4. Текстовые файлы 316
17.4.1. Процедуры и функции для текстовых файлов 317
17.4.2. Стандартные текстовые файловые переменные *input* и *output* 320
17.4.3. Задачи на обработку текстовых файлов 321
17.5. Типизированные файлы 326
17.5.1. Процедуры и функции для типизированных файлов 327
17.5.2. Задачи на обработку типизированных файлов 328
17.6. Нетипизированные файлы 335
17.7. Контрольные вопросы и задания 336

Глава 18. Динамические структуры данных 339

18.1. Еще раз о распределении памяти 339
18.2. Указатели. Описание указателей 341
18.2.1. Указатели и адреса 342
18.2.2. Описание указателей 343
18.3. Создание и удаление динамических переменных 344
18.4. Динамически формируемые массивы и строки 346
18.5. Структуры данных на основе указателей 350
18.6. Связанные списки 351
18.6.1. Общие сведения 351
18.6.2. Примеры программ 354
18.6.3. Сравнение связанных списков и массивов 359
18.7. Деревья. Двоичные поисковые деревья 360
18.7.1. Общие сведения 360
18.7.2. Двоичные поисковые деревья. Индексы 361
18.7.3. Пример построения двоичного поискового дерева 363
18.8. Контрольные вопросы и задания 366

Глава 19. Введение в объектно-ориентированное программирование 369

19.1. Абстрактные типы данных 369
19.1.1. Понятие АТД и структур данных 369
19.1.2. Стандартные АТД 370
19.1.3. Принцип абстрагирования 375
19.2. Введение в ООП 375
19.2.1. Принцип сокрытия деталей. Идеи ООП 375
19.2.2. ООП и Pascal 377
19.2.3. Определение класса 377
19.2.4. Область видимости элементов класса 379
19.2.5. Пример описания класса 379
19.2.6. Создание и уничтожение объектов. Конструкторы и деструкторы 382
19.2.7. Механизм наследования 384
19.2.8. Переопределение методов базового класса в потомках 385
19.2.9. Абстрактные классы 388
19.2.10. Свойства *(property)* 389
19.2.11. Применение ООП 392
19.3. Контрольные вопросы и задания 393

Заключение 395

Часть V. Приложения 397

Приложение 1. Компиляторы и системы программирования на основе Pascal 399

П1.1. Особенности языка Delphi 399
П1.2. Другие компиляторы и системы программирования на основе языка Pascal 408
П1.2.1 Free Pascal 409
П1.2.2. GNU Pascal 411
П1.2.3. TMT Pascal 411
П1.2.4. Virtual Pascal 413

Приложение 2. Стандартные модули Turbo Pascal и дополнения к ним 414

П2.1. Принципы формирования изображения 414
П2.2. Модуль *crt* 415
П2.2.1. Процедуры и функции управления экраном 415
П2.2.2. Работа с окнами 419
П2.2.3. Задержка при выполнении программы 421
П2.2.4. Управление клавиатурой 422
П2.2.5. Управление звуком 424
П2.3. Модуль *graph* 426
П2.3.1. Общие сведения 426
П2.3.2. Графические примитивы 429

П2.3.3. Установка цветов и стилей 434
П2.3.4. Окна в графическом режиме 436
П2.3.5. Вывод текста 437
П2.3.6. Сохранение и восстановление битовых образов изображений 438
П2.4. Модуль DOS 440
П2.4.1. Работа с системной датой и временем 441
П2.4.2. Функции для обработки параметров командной строки 442
П2.4.3. Запуск внешних программ из программы на Turbo Pascal 442
П2.5. Дополнения к модулям 444
П2.5.1. Модуль *crtplus* 444
П2.5.2. Модуль для подключения мыши к программе на Turbo Pascal 447
П2.5.3. Дополнение к модулю *graph* — вывод рисунков в формате BMP 449

Приложение 3. Основы практической работы в интегрированной среде Turbo Pascal 455

П3.1. Работа в окне интегрированной среды, текстовый редактор Turbo Pascal 455
П3.1.1. Общие сведения 455
П3.1.2. Начало работы 456
П3.1.3. Создание новой программы 456
П3.1.4. Набор и редактирование текста 457
П3.1.5. Работа с окнами 458
П3.2. Справочная система 459
П3.3. Компиляция программы, поиск и устранение ошибок 460
П3.4. Запуск программы на выполнение, просмотр результатов, отладка 460
П3.4.1. Пример работы с отладчиком 461
П3.5. Перечень ошибок 463

Список литературы 465

Предметный указатель 467

Введение

Книга, предлагаемая вниманию читателя, предназначена для использования в качестве *базового учебного пособия по дисциплине "Программирование на языке высокого уровня"*. Она может быть полезна как для *студентов*, обучающихся по специальностям направления "Информатика и вычислительная техника", так и для студентов других родственных направлений, а также для *школьников старших классов*.

Программирование — интересная, живая, быстро развивающаяся наука. Однако первые шаги при обучении программированию для многих оказываются очень нелегкими. Главное качество программиста — хорошее логическое мышление — развивается только в упорной и кропотливой работе.

С другой стороны, нелегко приходится и преподавателю начального курса программирования. Он понимает, что первые шаги начинающих программистов — основа их будущих успехов, но прекрасно видит и трудности начального этапа обучения. Авторы книги, преподаватели с многолетним стажем, стремились к тому, чтобы оказать максимальную поддержку и ученику, и преподавателю в нелегком деле освоения алгоритмов и программ.

Несмотря на обилие литературы по программированию, трудно найти учебник, который в первую очередь способствовал бы развитию навыков алгоритмизации, оставляя на втором плане особенности конкретного языка программирования. Данная книга представляет собой именно *учебник по программированию*, а не справочник по конструкциям языка программирования.

Нельзя, однако, научиться программировать на всех языках сразу. Поэтому в качестве базового языка был принят строгий и лаконичный Pascal (Паскаль), хорошо зарекомендовавший себя как *первый язык* при обучении. Авторы в дальнейшем будут пользоваться англоязычным названием, поскольку многие диалекты языка не имеют устойчивого перевода (Free Pascal, GNU Pascal и т. д.). Разумеется, все конструкции языка Pascal изложены в полном объеме и пояснены на примерах, так что никакого дополнительного справочника по языку не потребуется.

Тематику книги можно определить как "Конспект начинающего программиста" или "Популярный учебник". Ее объективную оценку могут дать читательская аудитория и специалисты. С точки же зрения авторов их труд отличает от множества других изданий удачное сочетание нескольких факторов:

- *краткости*, свойственной конспекту лекций;
- строгой *систематизации* материала;

- *доступности для усвоения*, присущей отработанному курсу, вобравшему в себя собственный опыт авторов и опыт, почерпнутый ими из многочисленных публикаций;
- *привлечения* внимания читателя к наиболее важным материалам в каждой теме;
- *наличия большого количества примеров с комментариями*, облегчающих обучение и стимулирующих интерес к материалу;
- *наличия контрольных вопросов в конце каждой главы*, которые способствуют проверке и закреплению пройденного теоретического материала;
- *наличия контрольных заданий, которые* могут быть использованы для практической работы со студентами и школьниками и существенно облегчают труд преподавателя.

Представленный учебный материал прошел успешную проверку на занятиях со студентами и школьниками.

Как работать с книгой

Пособие состоит из четырех частей.

В первой части рассматриваются ключевые вопросы алгоритмизации и программирования, вводится система основных понятий и определений. Эта часть не привязана ни к какому конкретному языку программирования, в ней нет примеров программ. Однако для успешного освоения практической части курса необходимо перед началом работы за компьютером внимательно прочитать материал первой части (она невелика по объему) и ответить на контрольные вопросы. Возможно, при первом чтении отдельные моменты покажутся непонятными или неинтересными — не фиксируйте на них внимание и переходите к работе над второй частью.

Главы второй части посвящены разбору конструкций языка высокого уровня и проиллюстрированы большим количеством примеров программ на языке Pascal.

Программы отлажены в среде Borland Pascal, однако они работоспособны и в среде Delphi, нужно только правильно выбрать тип приложения — консольное (Console Application). Все же рекомендуем начинать обучение с простой и испытанной среды Borland (Turbo) Pascal. Если она плохо работает под управлением вашей операционной системы, можно воспользоваться свободно распространяемой средой Free Pascal, которая имеется в сети Internet. Для получения дистрибутива можно, например, использовать адрес **http://www.ru.freepascal.org**. Среда Free Pascal имеет интерфейс, очень похожий на Borland Pascal, однако предназначена для современных операционных систем семейств Windows или UNIX. Входной язык очень похож на

Turbo Pascal, во всяком случае, разработчики системы говорят о совместимости с Turbo Pascal 7.0 (но не с Delphi).

При работе со второй частью книги необходимо добиться полного понимания и уверенных навыков программирования. Рекомендуется выполнить как можно больше заданий, которые приводятся в конце каждой главы. Они обычно расположены в порядке возрастания сложности.

Только после этого можно переходить к третьей части, посвященной вопросам структурного и модульного программирования. Главы этой части помогут выработать хороший стиль программирования, умение грамотно структурировать программы. Эти навыки окажутся полезными в дальнейшем при изучении других языков программирования. Главу 10, посвященную процедурам и функциям, возможно, придется перечитать несколько раз, пока не будут понятны все моменты, иначе неизбежны проблемы с усвоением дальнейшего материала.

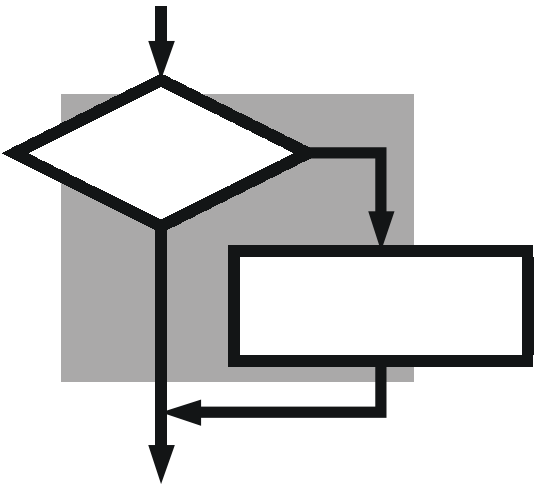
В четвертой части детально разбираются основные структуры данных и алгоритмы их обработки. Ее последние главы, посвященные динамическим структурам данных, абстрактным типам и объектно-ориентированному программированию, не просты для начинающих, т. к. требуют хорошо сформированного алгоритмического мышления. Авторы старались изложить данный материал настолько доходчиво, насколько это возможно для таких сложных абстрактных понятий. Опыт преподавания показывает, что все эти вопросы под силу не только студентам младших курсов, но и развитым ученикам старших классов.

Приложения, завершающие книгу, содержат краткую информацию о компиляторах и системах программирования, работающих на основе Pascal, перечень и описание стандартных модулей Turbo Pascal с дополнениями к ним, а также практическое руководство по работе в интегрированной среде Turbo Pascal.

Таким образом, двигаясь "шаг за шагом", читатель проходит путь от простейших программ до современных концепций и технологий программирования.

По мере накопления знаний и опыта рекомендуется снова возвратиться к более серьезному изучению материала первой части. При повторном чтении читатель посмотрит на этот материал другими глазами — глазами программиста, уже имеющего опыт практической работы, но нуждающегося в систематизации и обобщении знаний.

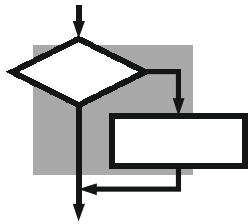
Если все теоретические вопросы и понятия программирования, изложенные в книге, покажутся понятными, интересными и подтвержденными практикой, значит цель авторов достигнута.



ЧАСТЬ I

Введение в программирование. Базовые понятия

Глава 1



Основы алгоритмизации

1.1. Алгоритмизация и требования к алгоритму

Несмотря на развитие новых технологий разработки программ, наиболее важными и фундаментальными понятиями программирования по-прежнему являются *алгоритм* и *алгоритмизация*. По сути дела, решение любой задачи на компьютере можно представить как процесс разработки, описания и выполнения алгоритма.

Алгоритм — подробное описание последовательности действий, расположенных в строгом логическом порядке, которое позволяет решить конкретную задачу. Составление такого пошагового описания процесса решения задачи называется ее *алгоритмизацией*. Слово "алгоритм", по существу, является синонимом таких слов, как "способ", "рецепт" и т. п.

Алгоритм предназначен для конкретного *исполнителя*, это может быть не только компьютер, но и робот, станок с программным управлением, наконец, человек. Элементарные действия (шаги), понятные исполнителю, на которые разбивается алгоритм, называются *инструкциями* или *командами*. Другими словами, алгоритм — конечная последовательность инструкций, которая должна быть выполнена исполнителем для решения поставленной задачи.

Уточним данное определение перечислением требований, предъявляемых к алгоритму:

- *однозначность* — недопустимы инструкции, которые имеют неопределенное или неоднозначное толкование;
- *массовость* — пригодность алгоритма для решения не только данной задачи, а и множества родственных задач, относящихся к общему классу;
- *детерминированность* — повтор результата при повторе исходных данных;
- *корректность* — способность алгоритма давать правильные результаты решения задачи при различных исходных данных;

- *конечность* — решение задачи должно быть получено за конечное число шагов алгоритма, "зацикливание" недопустимо.

Дополнительным требованием является *эффективность* — для успешного решения задачи должны использоваться ограниченные ресурсы (время работы процессора, объем оперативной памяти, быстродействие жесткого диска и др.).

1.2. Способы записи алгоритмов

Разработка алгоритма решения задачи — сложный творческий процесс. Записать алгоритм в виде компьютерной программы без каких-либо предварительных рассуждений может только опытный программист при решении небольшой по объему, четко поставленной задачи. В реальной жизни такие задачи встречаются редко, поэтому обычно разработчик сначала продумывает алгоритм и записывает его в какой-либо удобной форме, а затем реализует алгоритм в виде программы.

При разработке сложных коммерческих программных продуктов часто алгоритмизацию выполняет один человек, а запись (кодирование) алгоритма на языке программирования — другой. Следовательно, необходимо иметь такие методы записи алгоритмов, которые легко воспринимаются человеком, но являются достаточно строгими, чтобы их было нетрудно реализовать на каком-либо языке программирования или, иначе говоря, перевести на язык, понятный компьютеру.

Существуют различные варианты записи алгоритмов. К основным вариантам относятся описательный и графический способы. *Описательным (словесным)* называется способ записи алгоритма на естественном языке (русском, английском и т. д.) или с использованием специального *псевдоязыка*. *Графический* способ отличает компактная и наглядная форма записи в виде специальных графических знаков с указанием связи между ними.

1.2.1. Описательный способ

Запись алгоритма на естественном языке

Это наименее формализованный способ записи, который, в основном, используется для пояснения сути алгоритма и предназначен для чтения или исполнения человеком. При этом степень детализации алгоритма произвольно выбирается разработчиком. В литературе по программированию такой способ применяется очень часто. В данной книге словесный способ, как наиболее удобный для восприятия, будет использоваться для пояснений текстов программ, которые также называются *листингами*.

Обратите внимание, что, например, книгу кулинарных рецептов можно рассматривать как своего рода сборник "гастрономических" алгоритмов, запи-

санных на естественном языке. Таким образом, слово "алгоритм" зачастую выступает как синоним слов "способ", "рецепт" и т. п.

Использование псевдоязыков

Алгоритм, записанный на естественном языке, как правило, недостаточно формализован и может содержать какие-либо неясности или допускать неоднозначное толкование. Желание получить более формальный и вместе с тем наглядный способ записи алгоритма послужило толчком к разработке псевдоязыков.

Псевдоязык предполагает сочетание элементов естественного языка и языка программирования высокого уровня. Обычно предложения естественного языка встраиваются в конструкции формального языка. Получается достаточно строгое описание алгоритма, и вместе с тем оно обычно короче и легче для восприятия, чем текст программы, записанный на языке высокого уровня.

Хорошим примером псевдоязыка является школьный алгоритмический язык, разработанный А. П. Ершовым. К сожалению, не существует никаких стандартных псевдоязыков, поэтому авторы книг по программированию и алгоритмизации, использующие их для записи алгоритмов, вынуждены предварительно давать описание "своего" псевдоязыка. В настоящей книге авторы не будут применять указанный способ и ограничатся небольшим примером записи алгоритма на псевдоязыке, близком к школьному алгоритмическому *(см. разд. 1.2.3)*.

1.2.2. Блок-схемы алгоритмов

Блок-схема — это графическое изображение алгоритма в виде плоских геометрических фигур (блоков), соединенных линиями. Внутри блока записывается действие, которое необходимо выполнить, или условие, которое необходимо проверить. Блок-схема — стандартный способ записи алгоритма. Существует государственный стандарт, содержащий перечень правил построения блок-схем — ГОСТ 19.701—90 (ИСО 5807—85). Дата введения стандарта: 01.01.1992 г.

Основные блоки, которые используются при графической записи алгоритма, изображены на рис. 1.1, где *а* — начало (конец) алгоритма; *б* — блок ввода/вывода; *в* — операционный (вычислительный) блок; *г* — логический (условный) блок; *д—ж* — циклы.

Операционный блок служит для описания действий, логический — для проверки условий. Используя блоки *а—г,* можно описать практически любой алгоритм. Однако для изображения повторяющихся действий (циклов) ГОСТом предлагаются дополнительные блоки.

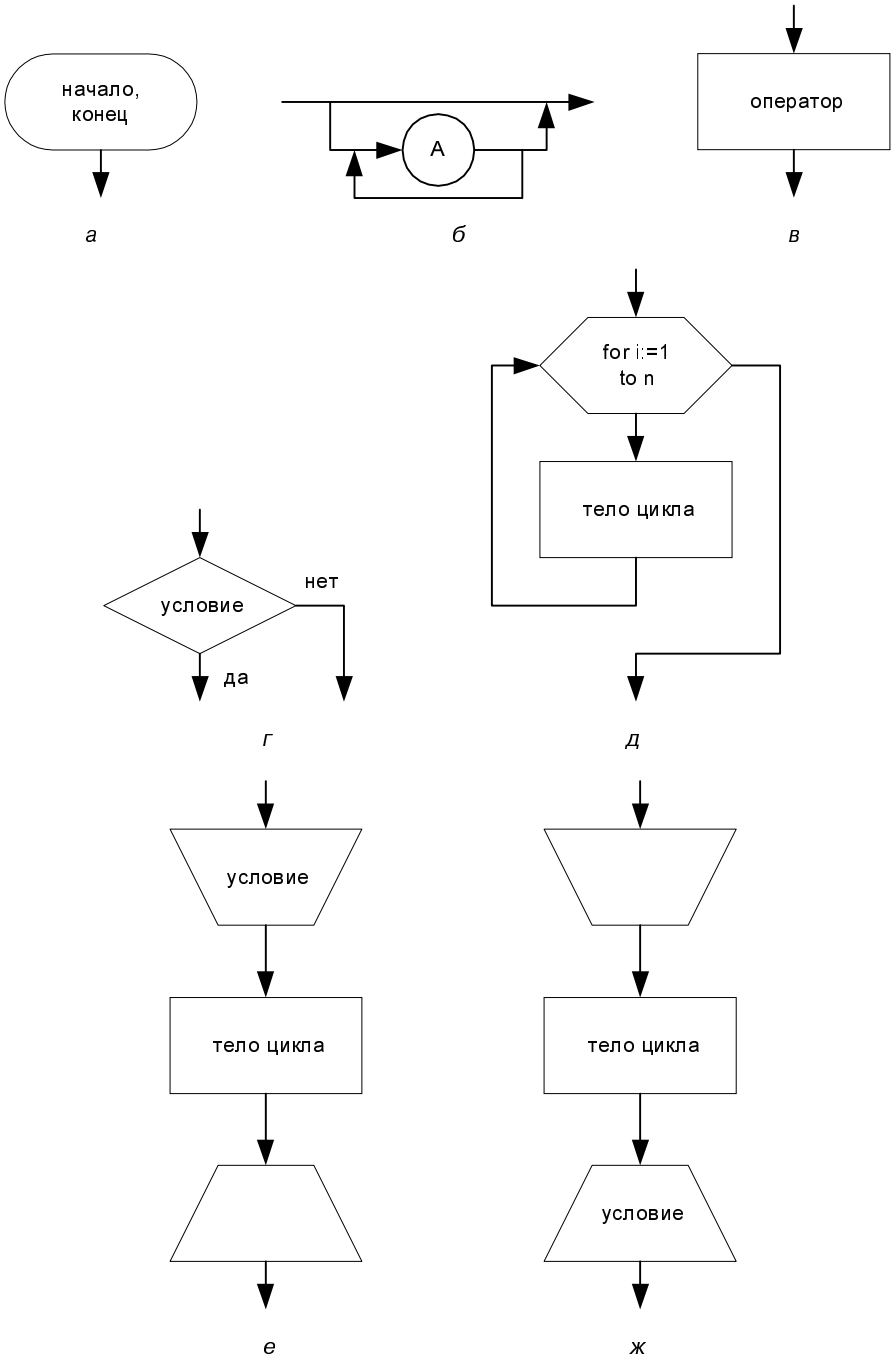


Рис. 1.1. Условные обозначения на схемах алгоритмов

Конструкция *д* специально не регламентируется ГОСТом для использования при изображении циклов, однако приведенное обозначение (шестиугольник) разрешено использовать для различных целей, поэтому такое изображение цикла на блок-схеме часто встречается в литературе по программированию. И наоборот, конструкции *е* и *ж* в литературе практически не встречаются. Авторам они также представляются неудачными. Чаще всего при изображении циклов используются логические и операционные блоки (см. рис. 1.2).

1.2.3. Пример блок-схемы алгоритма

В качестве примера рассмотрим различные способы записи алгоритма игры "Угадай число". Условие игры: игрок должен угадать число, "задуманное" компьютером, — случайное число в диапазоне от 0 до 1000.

Запись алгоритма на естественном языке

Компьютер задумывает случайное число *A*. В свою очередь, игрок вводит число *B*, пытаясь угадать значение *A*.

Выполняется проверка: число *B* строго больше числа *A*?

Если это так, т. е. "да", то компьютер выводит на экран сообщение "много" и просит игрока ввести следующее число, еще раз попытав удачи. Если же результат — "нет", то компьютер выполняет следующую проверку:

B строго меньше *A*? Если ее результат — "да", то выводится сообщение "мало", и компьютер ожидает ввода игроком следующего числа.

Если же результат второй проверки — "нет", это означает, что игрок угадал число, задуманное компьютером. В таком случае выводится сообщение "вы угадали", и игра завершается.

Запись на псевдоязыке

```
Алгоритм Угадай Число
Начало
    A:=случайное число
    Повторяй
        Ввод B
        Если B>A то вывод "много"
        Иначе
        Если A>B то вывод "мало"
```

```
        Иначе
        вывод "вы угадали"
        Все
    До A=B
Конец
```

Блок-схема алгоритма

Алгоритм игры представлен на рис. 1.2.

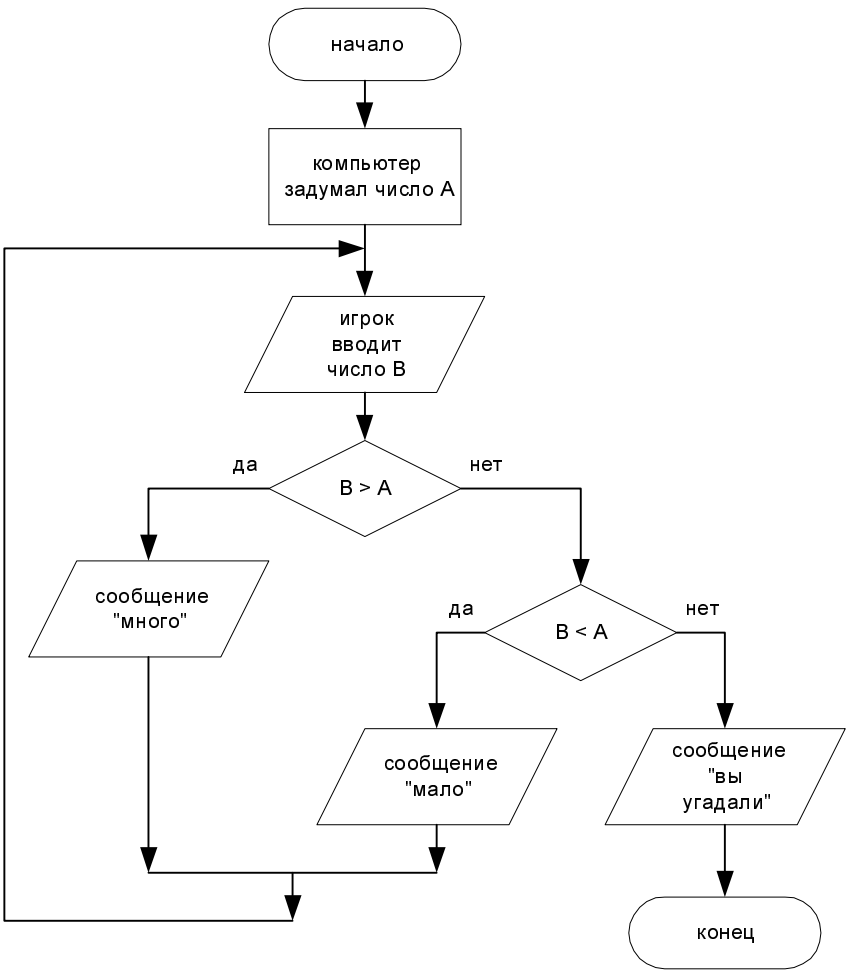


Рис. 1.2. Блок-схема алгоритма игры "Угадай число"

1.2.4. Следование, ветвление, цикл

Алгоритмические структуры (рис. 1.1, *а*, *б*, *в*) образуют линейную последовательность операций, которые выполняются по очереди в порядке записи — *следование*. Программную реализацию такой алгоритмической структуры называют *линейной программой*. Линейные программы обычно предназначены для решения простейших задач, в которых не предусмотрены выбор из нескольких возможных направлений хода программы или циклическое повторение операций.

Возможность альтернативного выбора при выполнении программы предоставляют *ветвления* (рис. 1.1, *г*), при выполнении которых алгоритм может пойти по одной из двух возможных ветвей в зависимости от справедливости проверяемого условия. Иногда выделяют также *обход*, который представляет собой пропуск нескольких шагов алгоритма при выполнении или невыполнении какого-либо условия.

Цикл (рис. 1.1, *д, е, ж*) представляет собой многократно повторяющуюся последовательность шагов алгоритма. Очень часто цикл также изображают, используя логический блок для проверки условия его завершения (такой способ был использован и в алгоритме игры "Угадай число").

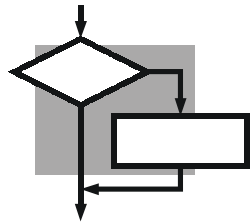
1.3. Контрольные вопросы и задания

1. Дайте определение алгоритма. Каким требованиям он должен удовлетворять?
2. Что такое алгоритмизация?
3. Для кого предназначен готовый алгоритм?
4. Какие варианты записи алгоритмов вы знаете? Дайте их сравнительную характеристику.
5. Выполните следующее задание.

❐ Составьте и запишите блок-схемы алгоритмов:

- вычисления суммы двух целых чисел;
- деления двух чисел столбиком (нахождение частного и остатка);
- деления отрезка пополам с помощью циркуля и линейки;
- перехода улицы через переход, оборудованный светофором;
- приготовления десяти одинаковых бутербродов с сыром.

Глава 2



Язык программирования. Программа

2.1. Язык программирования

Алгоритм, представленный в виде блок-схемы или в словесной форме (даже с использованием псевдоязыка), не пригоден для исполнения *процессором* какого-либо вычислительного устройства (компьютера, робота и др.). В этом случае нужен абсолютно формальный способ записи, не допускающий даже малейшей неточности. Следовательно, нужен особый язык, который позволяет записывать алгоритмы с использованием ограниченного набора конструкций, не допускающих неоднозначного толкования. Такие языки называются языками программирования.

Таким образом, *язык программирования* — совокупность средств и правил представления алгоритма в виде, пригодном для выполнения вычислительной машиной.

Рассмотрим кратко историю языков программирования.

2.1.1. Исторический обзор

Машинный язык

На раннем этапе развития вычислительной техники программы писались на машинном языке — в *машинных кодах*, т. е. так, как их воспринимает процессор компьютера или другого цифрового устройства. Для каждой элементарной операции в этом случае требуется указать ее код и адреса ячеек памяти с данными. Запись выполняется в цифровом виде с использованием двоичной системы счисления. Основными неудобствами такого способа программирования являются следующие:

- исторически сложилось так, что имеется много типов процессоров, отличающихся друг от друга *архитектурой* (устройством) и *системой команд* (набором допустимых инструкций). В результате программа на *машинном языке* годится только для исполнения тем процессором, для которого она написана;

- программу на машинном языке трудно читать даже профессионалу. В такой программе тяжело находить ошибки. Если объем программы превышает критический, программу практически невозможно полностью отладить. Даже если программа доведена до уровня, при котором она отвечает поставленной задаче, малейшие изменения могут вызвать непреодолимые трудности.

В настоящее время машинный язык используется только в исключительных случаях для программирования специальных задач с использованием специализированных вычислительных устройств.

Ассемблер

Первоначальный прогресс в технологии программирования был связан с идеей использования символьных имен (названий) вместо цифровых кодов операций и адресов данных. Язык записи команд, основанный на этой идее, получил название языка Ассемблера. Использование осмысленных названий вместо кодов операций и адресов памяти существенно упрощает процесс программирования и внесения изменений в программу. К сожалению, основной недостаток машинного языка — зависимость программы от конкретного типа процессора — при переходе к Ассемблеру не был устранен, но *это был только первый шаг в правильном направлении.*

Программа, записанная на Ассемблере, не может восприниматься процессором непосредственно (ему нужны только двоичные коды операций и адреса). Следовательно, необходимо предварительное преобразование (перевод) программы с языка Ассемблера на машинный язык. Это делается с помощью специальной программы, называемой *транслятором*, а *процесс преобразования программы в машинные коды называется трансляцией (см. разд. 3.1).* Попутно на транслятор были возложены функции выявления некоторых ошибок в тексте программы, нарушающих правила записи. Такие ошибки называются *синтаксическими.*

Языки программирования высокого уровня

Следующая ступень развития — это *языки программирования высокого уровня.* Их использование позволяет отвлечься от системы команд конкретного типа процессора. Такой язык содержит правила записи программ, которые, с одной стороны, достаточны и удобны для описания алгоритмов решения задач, а с другой стороны, толкуются однозначно и могут быть преобразованы в программы в машинных кодах.

Задача программиста заключается в том, чтобы подготовить правильный текст на языке программирования, а остальное возьмет на себя транслятор, который прочтет этот текст, проверит его на соответствие правилам языка и сформирует программу на машинном языке. Естественно, транслятор должен "знать" систему команд своего процессора и особенности работы уста-

новленного на компьютер программного обеспечения, его *операционной системы*. Это приводит к необходимости разработки нескольких различных трансляторов для одного и того же языка программирования.

Также понятно, что разработка транслятора с языка высокого уровня — задача более сложная и дорогостоящая, чем разработка транслятора с Ассемблера, а программа в машинных кодах, сформированная самым лучшим транслятором, все равно будет уступать по качеству и быстродействию программе, разработанной программистом-профессионалом на Ассемблере. За удобство программирования приходится платить некоторым снижением эффективности разработанной программы.

Говорят, что языки программирования высокого уровня являются *машинно-независимыми*. В противоположность этому, машинный язык и язык Ассемблера — *машинно-зависимые языки*. Иногда их называют *языками программирования низкого уровня*. Конечно, слова "низкий уровень" вовсе не означают низкое качество этих языков. Так подчеркивается их неразрывная связь с аппаратурой компьютера. Во многих задачах, например, при разработке ядра (основной части) операционных систем, где качество машинного кода ставится выше удобства разработки, без языка Ассемблера вообще не обойтись. Тем не менее, везде, где возможен компромисс, язык высокого уровня предпочитают Ассемблеру.

Возможность создания независимых от системы команд компьютера языков программирования привела к их бурному развитию. В настоящее время насчитывается более сотни самых разнообразных языков программирования высокого уровня. Большинство из них предназначены для применения в конкретных областях (например, языки программирования баз данных, языки программирования для Internet и т. д.). Эта группа языков развивается очень быстро, т. к. сфера применения компьютеров постоянно расширяется и возникает потребность как в новых языках программирования, так и в развитии уже имеющихся.

Вместе с тем имеется довольно устойчивая группа хорошо отработанных, испытанных языков универсального назначения, очень близких по принципам, заложенным в их основу, и по используемым конструкциям. На протяжении десятилетий эти языки постепенно совершенствовались, а заодно сближались, перенимая друг у друга наиболее удачные моменты. Однако и сейчас каждый язык имеет массу неповторимых, только ему присущих особенностей, у каждого есть свои поклонники, предпочитающие именно его. Очевидно, в обозримом будущем языки будут существовать и развиваться параллельно, хотя *тенденция к сближению языков универсального назначения налицо*.

Перечислим наиболее популярные языки универсального назначения, соблюдая исторический порядок их появления и указывая их отличительные особенности. Более полный обзор современного состояния языков про-

граммирования можно найти в [20], где рассматриваются и редко используемые языки функционального и логического программирования (Lisp, Prolog и др.).

Fortran. Является первым из языков высокого уровня (1954—1958). В то время основной сферой применения вычислительных машин (термин "компьютер" вошел в обиход гораздо позже) были научно-технические расчеты. Для этой цели и был разработан простой и эффективный язык программирования. Базовые принципы, заложенные в язык Fortran, впоследствии легли в основу других языков программирования высокого уровня. Особенно сильно его влияние на Basic. В настоящее время популярность языка Fortran невелика, но его последние версии — Fortran 77 и Fortran 90 — продолжают использоваться в сфере научно-технических и инженерных расчетов.

Algol. Опубликован в 1960 г. От языка Fortran отличался значительно более строгими правилами синтаксиса, что позволило выявлять больше ошибок еще на этапе трансляции. Программы, написанные на этом языке, более понятны и выразительны, чем Fortran-программы. Благодаря этим качествам Algol нашел применение в научных кругах, в первую очередь среди специалистов по прикладной математике, теоретической и экспериментальной физике. Является непосредственным предшественником языка Pascal. Очевидно, язык Pascal оказался удачнее своего предшественника, поскольку практически его вытеснил.

Basic. Появился на свет в 1964 г. и был задуман как очень простой в изучении язык, который можно использовать как первый язык при обучении программированию. В настоящее время на его основе создан современный язык Visual Basic, используемый как простой и удобный язык широкого назначения, *но не в качестве языка для обучения программированию*. В этой роли он как-то не прижился.

Pascal. Язык разработан в 1967 г. швейцарским ученым Никлаусом Виртом *специально для целей обучения*. Автор несколько усовершенствовал правила языка Algol с целью удобства его освоения и применения, оставив неизменными достоинства — простоту и выразительность текста программы в сочетании с удобством отладки. В продвижении языка большая заслуга принадлежит специалистам фирмы Borland, которые разработали целую серию быстродействующих и эффективных трансляторов с Pascal и внесли ряд дополнений в сам язык.

Профессор Вирт не остановился на создании Pascal. На его основе им были разработаны два новых языка программирования — *Modula* (1970) и *Oberon* (1985), представляющие собой дальнейшее развитие Pascal. К сожалению, обстоятельства сложились так, что эти языки не приобрели той популярности, на которую можно было бы рассчитывать. Ученики Вирта сделали многое для популяризации и распространения языков Modula и особенно

Oberon. Ими был разработан новый диалект *языка* Oberon, названный *Component Pascal* (Компонентный Паскаль). Этим названием подчеркивается неразрывная связь всех языков на основе Pascal. Как в дальнейшем сложится судьба данного языка, пока сказать трудно.

Однако сам Pascal на протяжении многих лет был и остается не только *основным языком для обучения принципам программирования*, но и самым распространенным языком на многочисленных олимпиадах по программированию. На его основе фирмой Borland создана удобная среда разработки Delphi, пользующаяся большой популярностью, особенно в России.

Популярность языка Pascal в нашей стране столь велика, что его название обычно пишут по-русски — Паскаль. Однако авторы книги придерживаются англоязычного обозначения, поскольку названия многочисленных диалектов этого языка *(см. разд. 2.1.4)* не имеют устойчивых русскоязычных эквивалентов (Free Pascal, GNU Pascal, Object Pascal и т. д.).

C. Язык был создан в 1972 г. в компании Bell Laboratories. При разработке этого языка преследовались совсем другие цели, чем при создании языка Pascal. Язык C задумывался как *инструмент для разработки системного программного обеспечения*, т. е. как промежуточный между языками высокого и низкого уровня. От программ, написанных на этом языке, пытались добиться производительности, близкой к производительности программ на Ассемблере, но в то же время старались сохранить возможность переноса программ с одной *компьютерной платформы* на другую, что характерно для языков высокого уровня. При разработке синтаксиса основных конструкций языка преследовалась цель сделать текст программ как можно лаконичнее.

Разумеется, вопросы простоты изучения языка и удобства разработки программ на нем отошли на второй план, поскольку невозможно сразу удовлетворить всем требованиям. В связи с этим *при обучении программированию рекомендуется начинать с языка Pascal.* Хорошо усвоив основные принципы программирования на языке высокого уровня, можно затем по достоинству оценить лаконичность и эффективность программ, написанных на языке C, а само изучение этого языка уже не вызовет проблем.

C++. Объектно-ориентированная версия языка программирования C — C++ — была разработана в 1980 г. Бьорном Страуструпом в компании Bell Laboratories. Сегодня этот язык является одним из наиболее популярных среди профессионалов. Достаточно, например, сказать, что с его помощью была создана операционная система *Windows.*

Развитие языков C/C++ продолжается и в настоящее время. Появляются новые, молодые языки, авторы которых берут за основу наиболее удачные конструкции C/C++, убирая тяжеловесные или устаревшие элементы. Наиболее популярными современными языками программирования на основе C++ являются *Java* и *C#* (читается "Си шарп"). Оба эти языка не очень объемны, но достаточно стройны и удобны в освоении и использовании. К со-

жалению, программы на Java или C# пока уступают по скорости выполнения программам на C, что снижает возможности их использования в качестве языков системного программирования. Однако эти языки с успехом используются в сфере программирования для Internet.

Ada. Этот язык был разработан для создания программных систем с многолетним сроком службы и высокой степенью надежности. Ada создан по заказу и состоит на вооружении Министерства обороны США. На сегодняшний день этот язык считается одним из наиболее сложных, однако свои цели вполне оправдывает.

2.1.2. Состав языка программирования. Синтаксис и семантика

Обычный разговорный язык состоит из четырех основных элементов: символов (букв), слов, словосочетаний и предложений. Количество символов языка, образующих его алфавит, невелико. Количество слов неизмеримо больше, но все же конечно: все слова языка можно перечислить, например, сведя их в толковый словарь. Все словосочетания, а тем более предложения перечислить уже нельзя, но известны правила, по которым они составляются. Правила русского языка, например, изложены в соответствующих учебниках. *Аналогично устроены все языки программирования.*

Состав языка программирования

Язык программирования содержит элементы, перечисленные ниже.

- *Символы* — это основные неделимые знаки, из которых составляются все тексты программ на данном языке. Совокупность всех символов образует *алфавит языка*. Алфавит языка программирования несколько шире, чем алфавит естественного языка, и включает обычно латинские буквы, знаки арифметических операций, символы-разделители и ряд других специальных символов.
- *Лексемы* — это неделимые последовательности символов алфавита (элементарные конструкции), имеющие самостоятельный смысл. Они образуются из основных символов языка, так же как слова обычного языка строятся из букв. Возможны лексемы, состоящие из одного символа, например, знаки операций. Нельзя, однако, провести прямую аналогию между лексемами и словами обычного языка. Дело в том, ч*то любой язык программирования, конечно, имеет определенное количество *зарезервированных (ключевых) слов*, которые составляют *словарь* языка, но таких слов неизмеримо меньше, чем в естественном языке (всего несколько десятков). Однако программист может формировать по определенным правилам собственные слова — *идентификаторы*. Из-за этой особенности текст программы на языке высокого уровня воспринимается сложнее,

чем запись алгоритма на естественном языке. Совокупность лексем и правил их формирования образует *лексику языка*.

- *Выражения* строятся из лексем в строгом соответствии с правилами языка. Они задают порядок вычисления некоторого *значения*. Выражения играют в языке программирования ту же роль, что и словосочетания в обычном языке. Еще более близкий аналог выражений — математические формулы.
- *Операторы* (инструкции или команды языка) задают полное описание некоторого действия, которое необходимо выполнить. Это аналог предложения, выражающего законченную мысль, в обычном языке. Для описания сложного действия может потребоваться группа операторов. В этом случае операторы объединяются в *составной оператор* или *блок*.

Действия, заданные операторами, выполняются над *данными*. Предложения языка, в которых даются сведения о данных, называются *описаниями* или *неисполняемыми операторами*.

Совокупность описаний и операторов языка программирования, реализующая алгоритм решения конкретной задачи, образует программу на данном языке.

Синтаксис и семантика языка

Система правил записи элементов языка программирования (лексем, выражений, операторов) образует его *синтаксис*.

Смысл конструкций языка — это его *семантика*.

Иными словами, синтаксис языка определяет внешний вид отдельных конструкций ("как они выглядят"), а семантика толкует действия, которые выполняет компьютер по этим конструкциям ("что они делают").

Современные языки высокого уровня очень схожи по своей семантике, но имеют некоторые непринципиальные различия в синтаксисе. Отсюда проистекает важный вывод: *начинающим изучать программирование следует обратить внимание именно на семантику языка*. Именно она представляет собой те *базовые знания*, которые не зависят от особенностей конкретного языка. Хорошее знание основ поможет в дальнейшем изучать новые языки программирования в короткие сроки и при этом писать программы хорошего качества независимо от синтаксиса языка.

Справедливости ради следует признать, что изучение правил синтаксиса языка обычно не вызывает затруднений. Понять до конца семантику каждой конструкции гораздо труднее. Еще труднее научиться использовать каждую конструкцию языка по назначению при записи своих собственных алгоритмов. К сожалению (или к счастью), тут можно предложить только один старый испытанный способ — упорную работу за компьютером, разбор многочисленных примеров, решение задач.

2.1.3. Описание языка

В отличие от естественных языков, для каждого языка программирования имеется его строгое описание. Зачем нужно такое описание, очевидно понятно, ведь речь идет о языке, предназначенном для общения с компьютером, где малейшая двусмысленность приводит к ошибкам.

Описание языка есть описание его основных элементов, причем описание каждого элемента языка задается его синтаксисом и семантикой. Синтаксические определения устанавливают правила построения элементов языка. Семантика определяет смысл и правила использования тех элементов языка, для которых были даны синтаксические определения.

Неформальное описание языка

Для описания языка программирования можно использовать обычный естественный язык. Но такое описание (назовем его неформальным) будет страдать теми же недостатками, что и описание какого-либо алгоритма в словесной форме: излишним многословием при опасности неоднозначного толкования. Правда, в отличие от описания алгоритма, описание языка рассчитано на изучение его человеком (программистом), поэтому в литературе по программированию такое описание применяется довольно часто.

Однако разработчики трансляторов (это тоже программисты, но не *прикладные*, а *системные*) предпочитают иметь более формальное описание языка, т. е. такое, которое не допускает никакого неоднозначного толкования и при этом является компактным и удобным в использовании для профессионалов. В соответствии с их пожеланиями уже для описания одного из первых языков (Algol) был придуман специальный сугубо формальный язык описания.

Способы формального описания языка

Сейчас имеется несколько языков описания. Для того чтобы отличать их от языков программирования, которые они описывают, их называют *метаязыками*.

Дадим представление о двух разных метаязыках, которые используются для одних и тех же целей и являются полностью взаимозаменяемыми.

- Формы Бэкуса—Наура (БНФ). Именно этот метаязык был предложен для описания языка Algol, а затем многократно использовался авторами различных языков программирования для формального описания их творений. В настоящее время в основном используется расширенный вариант, который называется РБНФ (расширенные БНФ).
- Синтаксические диаграммы Вирта. Графическое изображение основных конструкций языка.

Формы Бэкуса—Наура позволяют очень строго и компактно описать все элементы языка, используя несколько символов специального назначения — *метасимволы*. При этом каждое понятие языка выводится из более простых понятий.

Самые простые понятия — это символы и зарезервированные слова языка. Они называются *терминальными символами* или *терминалами*. В описаниях терминалы берутся в кавычки или выделяются жирным шрифтом.

Все остальные понятия — *нетерминалы* — постепенно выводятся из терминальных с помощью специальных металингвистических формул, в которых вместо знака равенства используется знак `::=`, читаемый как "есть по определению". Каждый нетерминал заключается в угловые скобки. Метасимвол `|` имеет обычное значение "или".

Расширенная БНФ дополнена еще несколькими полезными метасимволами. Так, запись `{A}` обозначает повторение символа `A` несколько (возможно и нуль) раз. Запись `[A]` обозначает повторение символа `A` нуль или один раз (таким образом удобно обозначать необязательные элементы языка). Кроме того, формулы РБНФ выглядят несколько проще, чем в БНФ, т. к. понятия не берутся в угловые скобки, а знак `::=` заменен на обычное равенство. Терминальные символы берутся в кавычки.

Синтаксические диаграммы Вирта — это графическое описание языка с использованием блоков, соединенных линиями. При этом терминалы обозначаются при помощи кружков (или овалов), а нетерминалы — при помощи прямоугольников. Для каждой конструкции Бэкуса—Наура можно построить эквивалентную синтаксическую диаграмму. Например, конструкциям `{A}` и `[A]` соответствуют синтаксические диаграммы, изображенные на рис. 2.1, *а*, *б*.

Рис. 2.1. Синтаксические диаграммы конструкций `{A}` и `[A]`

Приведем небольшой пример, иллюстрирующий применение этих способов. Пусть необходимо описать такую несложную конструкцию любого языка программирования, как целое число.

Описание с использованием БНФ имеет вид:

```
<Целое>::=<ЦелоеБезЗнака>|<Знак><ЦелоеБезЗнака>
<ЦелоеБезЗнака>::=<Цифра>|<ЦелоеБезЗнака><Цифра>
```

```
<Знак>::=+|-
<Цифра>::=0|1|2|3|4|5|6|7|8|9.
С использованием РБНФ:
Целое=[Знак]Цифра{Цифра}
Знак= "+"|"-"
Цифра="0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
```

Соответствующая синтаксическая диаграмма приведена на рис. 2.2.

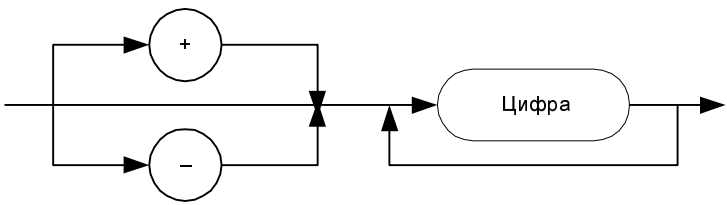


Рис. 2.2. Синтаксическая диаграмма описания целого числа

Неформальное описание этой же конструкции: целое число — это последовательность цифр, которой может предшествовать знак.

Несмотря на то, что неформальное описание — наименее строгое из всех способов описания языка, в данной книге будет использоваться именно этот способ. На это есть, по крайней мере, две причины:

- неформальное описание легче воспринимается начинающими. С целым числом, конечно, все понятно при любом способе описания, но чтение более сложных конструкций требует определенных усилий;
- при неформальном описании можно сделать упор именно на смысл конструкций языка. Вспомним, что основная цель при изучении первого языка программирования — как можно лучше понять его семантику.

Возможные неточности неформального описания будут уточняться с помощью большого количества примеров, для того чтобы полностью исключить возможность неоднозначного толкования.

Однако пример формальной записи более серьезной конструкции языка (арифметического выражения) в книге все же имеется *(см. разд. 14.4.4)*.

2.1.4. Стандарт и реализации языка

Естественные языки создавались веками, поэтому они довольно консервативны. Но определенные изменения с течением времени все же происходят: появляются новые слова и выражения, исчезают устаревшие обороты. Время

от времени филологам приходится пересматривать нормы языка и вносить изменения в словарь и в свод правил языка.

Языки программирования изменяются гораздо более динамично, чем естественные языки, однако нуждаются в стандартизации ничуть не меньше их. При отсутствии единого стандарта языка программирования разработчики трансляторов будут вправе вносить в язык любые изменения, какие они сочтут нужными. В свою очередь разработчики прикладных программ заинтересованы в том, чтобы различные трансляторы одного и того же языка программирования действительно поддерживали *одни и те же правила языка*. Не переписывать же программу заново каждый раз при замене транслятора!

К сожалению, приходится констатировать, что на практике конкретные реализации трансляторов всегда чем-то отличаются от стандарта. Поэтому говорят о наличии различных *реализаций* или *диалектов* одного и того же языка программирования. Как правило, различия в реализациях направлены в сторону расширения стандартных конструкций языка, а не их замены. Обычно документация по конкретному транслятору содержит раздел "Отличия от стандарта", который позволяет разработчикам прикладных программ быстро освоить особенности конкретной реализации языка. Разумеется, предполагается, что прикладные программисты, изучающие документацию, знают стандарт языка.

Первоначально стандарты языков программирования являлись формально чисто американскими и утверждались Американским Национальным Институтом Стандартов (American National Standard Institute — ANSI), а также Институтом по электротехнике и электронике (Institute of Electrical and Electronics Engineers — IEEE). Позднее эти стандарты стали фиксироваться также Международной Организацией Стандартов (International Standard Organization — ISO) и ее подразделением по электротехнике (International Electrotechnical Commission — IEC). Поэтому современные стандарты могут иметь различные обозначения — ANSI/IEEE, ANSI/ISO/IEC и т. д.

Многие современные языки программирования имеют стандарты. Это языки C/C++, C#, Ada, Fortran, Cobol, Pascal. Наиболее доступны стандарты языков C/C++, поскольку по этому вопросу имеется обширная литература, а разработчики трансляторов C и C++ не позволяют себе существенных отклонений от стандарта.

С языком Pascal сложилась несколько иная ситуация. Многочисленные учебники по этому языку обычно содержат описание очень популярной, но не единственной реализации языка *Borland Pascal* (Turbo Pascal). Кроме того, существует обширная литература по программированию в среде Delphi, в которой описывается диалект *Object Pascal*. Поэтому есть смысл перечислить имеющиеся стандарты этого языка.

В настоящее время действуют два официальных международных стандарта Pascal. Первый из них был разработан в 1983 г., но официально утвержден

несколько позже (стандарт Programming Languages — Pascal ANSI/ISO/IEC 7185:1990; English version EN 27185:1992). Он практически полностью совпадает с описанием языка в нотации Вирта. Второй стандарт — это расширенный вариант языка (стандарт Programming Languages — Extended Pascal ANSI/ISO/IEC 10206:1991).

Последний стандарт языка — объектный (Object-Oriented Extensions to Pascal). В отличие от первых двух, он формально не утвержден, но оформлен в виде отчета (ANSI Technical Report) в 1993 г. (ANSI/X3-TR-13-1994).

Реализаций языка Pascal так много, что не представляется возможным перечислить все. Отметим, что современные реализации поддерживают как минимум расширенный Pascal, а большая часть их реализует и объектно-ориентированные расширения. К сожалению, ни один диалект полностью не совпадает со стандартом. Подробнее особенности различных реализаций будут рассмотрены в следующей главе, а также в *приложении 1*. На протяжении всей книги будут оговариваться те расширения языка, которые внесены в последние версии Turbo Pascal 7.0 и особенно Delphi.

Название Object Pascal приживается как-то плохо, поэтому мы употребляем название Delphi, даже если речь идет не о системе, а о языке программирования. В фундаментальном учебнике по языкам высокого уровня Р. Себесты [20] также употребляется словосочетание "язык Delphi".

2.2. Программа. Программный продукт и его характеристики

Программа — это запись (реализация) алгоритма на языке программирования. Программу можно считать законченным и готовым к использованию результатом процесса разработки алгоритма.

В настоящее время в обиход прочно входит термин *программный продукт*, под которым понимается компьютерная программа как продукт человеческой деятельности, выставленный на рынке в качестве товара.

Обычно различают *тиражные* и *заказные* программные продукты. Тиражный программный продукт производится для того, чтобы его могли использовать во многих местах различные пользователи. У него нет заказчиков, а решение о начале разработки принимается исходя из предполагаемого рыночного спроса. Текстовые процессоры, электронные таблицы, системы управления базами данных, электронные словари, русификаторы, переводчики, программы оптического распознавания символов — все это примеры тиражных программ.

Заказной программный продукт создается для одного, редко — для нескольких пользователей или разрабатывается для передачи (продажи) другой организации с целью использования его в качестве составной части ее программного продукта.

Исходя из определения программного продукта, перечислим его основные *характеристики*:

- корректность и безошибочность, т. е. способность правильно решать ту задачу, для которой он предназначен;
- надежность, под которой понимается устойчивость к сбоям, зависаниям и прочим внештатным ситуациям, даже в случае неправильных действий неопытных пользователей;
- простота и удобство эксплуатации (по-другому это называется дружественный пользовательский интерфейс);
- эффективность, т. е. способность обеспечить оптимальное быстродействие для получения результата с требуемой точностью при разумном объеме выделенной памяти. Процессорное время и память — это два важных ресурса компьютера, которые следует экономить;
- переносимость с одной компьютерной платформы на другую ("кросс-платформенность") обозначает, что если на родном компьютере сменили операционную систему (или вообще сменили компьютер), программа должна продолжать функционировать (в крайнем случае, переработка программы должна быть незначительной);
- открытость — это требование сейчас становится все более актуальным. Под открытостью понимают возможность стыковки данного программного продукта с другими продуктами. Например, желательна возможность связи с программами *Microsoft Office* (допустим, для того, чтобы
  передать результаты программы в электронную таблицу *Excel* для построения диаграммы или сводной таблицы, или прочитать исходные данные из текстового документа *Word*).

2.3. Жизненный цикл программы

Исходя из того, что уже было сказано выше, понятно, что важнейшими этапами при создании программного продукта являются разработка алгоритма и его реализация на подходящем языке программирования. К сожалению, эти два этапа не всегда удается выполнить строго последовательно, ведь в процессе реализации могут обнаружиться недоработки, а то и ошибки в самом алгоритме, и его придется доработать (переработать).

На практике дело обстоит еще сложней: ошибки в алгоритме могут быть выявлены как в процессе его реализации, так и при эксплуатации уже готовой программы. Известен, например, случай, когда ошибка в программе формирования случайных чисел с заданными характеристиками была обнаружена через несколько лет после того, как эта программа начала использоваться.

Но и при правильном алгоритме может выясниться, что ошибки были допущены при его реализации. А возможна и такая ситуация — алгоритм и

программа безупречны, но за то время, пока они разрабатывались, изменилась сама постановка задачи (допустим, какие-то функции программы уже не нужны пользователям, а какие-то требуется добавить). И опять придется возвращаться к самому началу, продумывать и реализовывать новые алгоритмы, снова сдавать обновленный вариант заказчику.

Процесс этот будет продолжаться до тех пор, пока программа продолжает эксплуатироваться. В конце концов любой программный продукт заканчивает свое существование, уступая место другим, более современным и совершенным программным средствам.

Периодически повторяющийся процесс разработки, внедрения и эволюции программного продукта принято называть его жизненным циклом.

Современный рынок программного обеспечения требует, чтобы процесс создания программы был быстрым, а ее дальнейшая эксплуатация — долговременной и надежной. Для достижения этого необходима хорошая организация и тщательное планирование всего жизненного цикла.

Принято выделять несколько *этапов (фаз)* жизненного цикла.

- *Постановка задачи* — выполняется специалистом в предметной области на естественном языке (русском, английском и т. д.). Необходимо определить цель будущей разработки, ее содержание и общий подход к решению. На этом этапе обязательно выполняется четкое обоснование необходимости разработки программного продукта. Возможно, что задача решается точно *(аналитически)*, и без компьютера можно обойтись. А возможно, на рынке программного обеспечения уже есть готовый программный продукт, который годится для решения данной задачи и имеет приемлемую стоимость. Особенно важен этот предварительный этап для тиражных программных продуктов, поскольку ошибочное решение без детального анализа рынка приведет к колоссальным финансовым потерям.
- *Анализ задачи и моделирование* — на этом этапе определяется круг будущих пользователей программного продукта и основные функции, которые им необходимы, исходные данные и результаты программы, выявляются ограничения на их значения, выполняется формализованное описание задачи и построение (выбор) математической модели, пригодной для ее реализации на компьютере.
- *Разработка или выбор алгоритмов реализации основных функций программы* — выполняется на основе данных предыдущих этапов. Многие задачи можно решить различными способами, поэтому необходимо проанализировать все варианты и выбрать оптимальный. На этом этапе алгоритмы записываются на одном из языков для записи алгоритмов (допустим, в виде блок-схем), поскольку вопрос о выборе подходящего языка программирования может быть решен позднее.

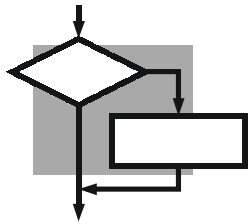
- *Проектирование общей структуры программы* — к началу этого этапа выбор языка программирования и вспомогательного программного обеспечения должен быть сделан. Поэтому с учетом выбранных инструментальных средств формируется модель решения с последующей детализацией и разбивкой на подпрограммы, а также определяется "архитектура" программы и способ хранения информации (набор переменных, массивов данных и т. п.).
- *Кодирование* — запись алгоритма на языке программирования. Очень трудоемкий этап. Современные системы программирования, подобные Delphi, позволяют ускорить процесс разработки программы, автоматически создавая часть ее текста, однако творческая работа по-прежнему лежит на программисте.
- *Отладка и тестирование программы.* Под *отладкой* понимается устранение ошибок в программе. *Тестирование* позволяет вести их поиск и, в конечном счете, убедиться в том, что полностью отлаженная программа дает правильные результаты. Для этого разрабатывается *система тестов* — специально подобранных контрольных примеров с такими наборами параметров, для которых решение задачи известно. Тестирование должно охватывать все возможные ветвления в программе, т. е. *проверять все ее инструкции* и включать такие исходные данные, для которых решение *невозможно.* Проверка особых, *исключительных ситуаций* необходима для анализа корректности. Например, программа должна отказать клиенту банка в просьбе выдать сумму, отсутствующую на его счете. В ответственных проектах большое внимание уделяется так называемой "защите от дурака" *(fool-tolerance)*, подразумевающей устойчивость программы к неумелому обращению пользователя.
- *Публикация* результатов работы, продвижение тиражного программного продукта на рынке или *передача программы заказчику* для эксплуатации.
- *Сопровождение и модернизация программы* — включает консультации представителей заказчика по работе с программой и обучение персонала. Недостатки и ошибки, замеченные в процессе эксплуатации, обычно устраняются разработчиками бесплатно. Модернизация программы в соответствии с изменившимися требованиями заказчика, естественно, потребует от него дополнительных финансовых вложений. Для тиражных программных продуктов выполняется регулярный анализ отзывов потребителей и ведется разработка новых версий программы с учетом их пожеланий.

2.4. Контрольные вопросы

1. Дайте определение языка программирования.
2. Чем был вызван переход от машинного языка к Ассемблеру?

3. Что такое транслятор? Какой процесс называют трансляцией?
4. В чем различие между языками программирования высокого и низкого уровня?
5. Перечислите наиболее популярные языки универсального назначения. Дайте их краткую характеристику.
6. Какой язык был разработан специально для целей обучения?
7. Чем была вызвана необходимость разработки языка С? Укажите его сильные стороны.
8. Перечислите элементы, входящие в состав языка программирования.
9. Что такое синтаксис и семантика языка? На что, в первую очередь, нужно обратить внимание при изучении языка? Почему?
10. Какие способы описания языка программирования вы знаете? Проиллюстрируйте их использование на примере конструкции *идентификатор*. Неформальное определение идентификатора следующее: последовательность латинских букв (включая знак подчеркивания) и арабских цифр, начинающаяся с буквы.
11. Чем вызвана необходимость стандартизации языков программирования? Какие организации занимаются стандартизацией? Что такое диалект языка программирования? Какие реализации языка Pascal вы знаете?
12. Дайте определение программы. Чем различаются тиражные и заказные программные продукты?
13. Перечислите основные характеристики программного продукта.
14. Что понимается под жизненным циклом программы?
15. Прокомментируйте этапы жизненного цикла программы. Какие фазы и в каких случаях могут быть пропущены?

Глава 3



Система программирования

3.1. Компиляторы и интерпретаторы

Текст программы, написанной на любом языке программирования, кроме машинного, должен быть обработан специальной программой — *транслятором (см. разд. 2.1.1)*. Однако у разных трансляторов могут иметься свои особенности в организации процесса трансляции. В настоящее время трансляторы разделяются на две основные группы: *компиляторы* и *интерпретаторы*.

Компилятор — это обслуживающая программа, выполняющая трансляцию (перевод) программы с одного языка программирования на другой. Чаще всего компилятор переводит программу, записанную на исходном языке, в машинные коды. Сам процесс трансляции программы обычно называется *компиляцией*. Результатом компиляции является программа или ее части, представленные на машинном языке (или другом языке низкого уровня).

В свою очередь, группу компиляторов можно разделить на *ассемблеры* и *компиляторы с языков высокого уровня*.

Специфической чертой *ассемблеров* является то, что они осуществляют дословную трансляцию одной символической команды в одну машинную. Иначе говорят, что ассемблер выполняет трансляцию один в один. Команды языка высокого уровня значительно отличаются от команд машинного языка. В языках универсального назначения, таких как Pascal или C, одна команда исходного языка транслируется в 7—10 машинных команд. Однако есть и такие языки, в которых каждой команде может соответствовать 100 и более машинных команд (например, язык логического программирования Prolog).

Интерпретатор — программа, осуществляющая пооператорную трансляцию и выполнение исходной программы. В отличие от компилятора, интерпретатор не порождает на выходе программу на машинном языке. Распознав команду исходного языка, он тут же выполняет ее. Процесс трансляции и выполнения программы интерпретатором называется *интерпретацией*.

И компиляторы, и интерпретаторы имеют свои достоинства и недостатки.

Сначала перечислим достоинства интерпретаторов:

- интерпретатор позволяет начать обработку данных после написания даже одной команды. Это делает процесс разработки и отладки программ более гибким;
- отсутствие выходного машинного кода позволяет не "захламлять" внешние устройства дополнительными файлами;
- основным достоинством интерпретаторов в настоящее время является возможность достаточно легкого приспособления к любым машинным архитектурам и операционным системам. Это свойство иначе называется *"кроссплатформенность"*. Объясняется это тем, что обычно сами интерпретаторы разрабатываются на каком-либо распространенном языке высокого уровня (чаще всего на C или C++), поэтому являются переносимыми между компьютерными платформами. Для переноса программ на интерпретируемом языке практически не требуется никакой переработки при переносе с одной платформы на другую.

Недостатком интерпретаторов является низкая скорость выполнения программ. Обычно интерпретируемые программы выполняются намного медленнее программ в машинных кодах, порожденных при компиляции. Это существенный минус, который сразу резко ограничивает области применения интерпретаторов. *В тех случаях, когда производительность разработанной программы имеет решающее значение, должен быть использован компилятор.*

В принципе, для любого языка программирования можно разработать и компилятор, и интерпретатор. Такие примеры есть: например, для популярного языка Visual Basic имеются как компиляторы, так и интерпретаторы. Однако чаще встречается другая ситуация — для работы на языке предлагаются только компиляторы или только интерпретаторы. Авторам книги не известны интерпретаторы с языка Pascal, но знакомо множество различных компиляторов этого языка. Языки программирования для Internet — *JavaScript*, *VBScript* и ряд других обрабатываются интерпретаторами.

В некоторых современных языках программирования принципы компиляции и интерпретации сочетаются. Это справедливо, например, для популярного языка Java, разработанного компанией Sun Microsystems. Программа на этом языке сначала обрабатывается компилятором. Однако на выходе компилятора порождается не машинный, а некий промежуточный код (его называют *псевдокодом или байт-кодом*), который затем интерпретируется при помощи виртуальной машины Java. Похожий принцип используется и в языке C#, который предлагается компанией Microsoft и представляет собой альтернативу языку Java. Правда, для C# псевдокод, порожденный компилятором, все-таки перед выполнением преобразуется в полноценный машинный код.

Таким образом, налицо попытка совместить достоинства компиляторов (высокая производительность откомпилированной программы) и интерпретаторов (кроссплатформенность). Заметим, что интерпретируемые программы пока еще не достигли такого же быстродействия, как предварительно откомпилированные.

3.2. Этапы компиляции и компоновки

На сегодняшний день наиболее распространенным способом обработки программы на языке высокого уровня является компиляция, поэтому остановимся на этом процессе более подробно.

Обратите внимание, что на выходе компилятора порождается не готовый к исполнению машинный код, а так называемый *объектный код*, который нуждается в дальнейшей обработке.

Дело в том, что программа на языке высокого уровня пользуется готовыми отлаженными фрагментами кода для выполнения различных типовых действий (ввод данных с клавиатуры или из файла, отображение информации на экране, вычисление стандартных математических функций и т. д.). Такие фрагменты кода называются *подпрограммами (см. гл. 10)* и находятся в *библиотеках*, представленных в виде отдельных файлов, расположенных на магнитном диске. Библиотечные подпрограммы предварительно обрабатываются компилятором и хранятся в виде объектных кодов.

Компилятор во время работы распознает названия библиотечных подпрограмм, но не может, да и не должен извлекать нужные подпрограммы из библиотек и вставлять их в сформированный машинный код программы. Этим занимается другая программа — *компоновщик* (редактор связей), которая собирает все объектные коды в полноценный исполняемый код.

Таким образом, для получения исполняемого кода программа на языке высокого уровня должна пройти два этапа обработки:

- компиляция;
- компоновка.

Результаты компиляции и компоновки могут быть сохранены на диске, а могут оставаться только в оперативной памяти. Например, при обработке программ на языке C/C++ практически всегда на диске сохраняются файлы с расширением obj (objective) и exe (executable).

Как правило, при работе в системах на основе Pascal файлы с расширением obj не сохраняются, но могут быть сохранены файлы с расширением exe. Чаще всего такое сохранение выполняется автоматически *(см. разд. П3.4)*. Кроме того, в Pascal имеется замечательная возможность раздельной компиляции отдельных частей программы (модулей), в результате которой сохра-

няются объектные файлы особого формата, имеющие расширение tpu или dcu. Подробная информация о библиотечных модулях и раздельной компиляции представлена ниже *(см. гл. 12)*.

3.3. Отладка программы. Типы ошибок

Типы ошибок. Программирование является творческим процессом, поэтому ошибки неизбежно встречаются даже у опытных программистов. Различают следующие типы ошибок:

- синтаксические ошибки (ошибки компиляции);
- ошибки выполнения;
- ошибки в алгоритме программы или его реализации (семантические).

Синтаксические ошибки возникают при нарушении правил языка. Их обнаруживает компилятор, который и выдает сообщение об ошибке, обычно устанавливая курсор в позицию, где обнаружена ошибка. Исправление синтаксических ошибок может вызвать затруднения только на начальном этапе обучения.

Ошибки выполнения (run-time errors) не нарушают синтаксис языка. Однако они приводят к ошибочным операциям в процессе выполнения программы, таким например, как попытка деления на ноль или извлечения квадратного корня из отрицательного. При возникновении такой ошибки процессор вынужден выполнить аварийную остановку программы, на экран при этом выводится сообщение **Run-time error** и сообщается код ошибки, по которому можно установить ее причину. Коды ошибок выполнения, возникающие при запуске программы на Turbo Pascal, приведены в *приложении 3*.

Часто такие ошибки возникают не при каждом запуске программы, а только при определенных сочетаниях исходных данных. Подобные случаи называются "исключительными ситуациями" или "исключениями" (exceptions). В современных языках программирования имеются средства корректной обработки исключений, которые позволяют избежать ошибок при выполнении программы. В *приложении 1* рассмотрены средства обработки исключительных ситуаций в языке Delphi. Аналогичные средства имеются и в языках C++, Java и др.

Семантические ошибки программы при верных исходных данных и внешне безошибочной работе программы приводят к неверным результатам. Этот тип ошибок наиболее коварен и труден для исправления, т. к. пользователь, получая ошибочный результат, может посчитать его верным, поскольку никаких сообщений об ошибках не было.

Семантические ошибки должен обнаруживать сам программист, проверяя работу программы на заранее подготовленной *системе тестов*, для которых заранее известны результаты. Подготовка такой системы тестов сама по себе представляет серьезную задачу.

Для облегчения поиска семантических ошибок используются специальные программы, называемые *отладчиками*. Отладчик позволяет выполнять программу по шагам, просматривая при этом промежуточные результаты каждого шага. Этот процесс называется *трассировкой*. Если на очередном шаге результат оказывается не таким, на какой рассчитывал разработчик, значит, ошибка локализована. Осталось только понять ее причину и исправить. Правила работы со встроенным отладчиком системы Turbo Pascal содержатся в *приложении 3*. Аналогично можно работать с любым другим отладчиком.

3.4. Состав системы программирования. Понятие интегрированной среды разработки

Имеющихся сведений уже достаточно, чтобы перейти к обсуждению технических деталей процесса разработки и отладки программы на языке высокого уровня. Выделим основные этапы этого процесса (для определенности примем, что при обработке программы используется способ компиляции):

- набор исходного текста программы или редактирование уже имеющегося текста;
- получение исполнимого кода (компиляция и компоновка);
- выполнение программы (целиком или по шагам) и анализ результатов, после которого обычно приходится возвращаться к первому этапу.

Этот циклический процесс обычно повторяется очень много раз, прежде чем программа будет выдавать удовлетворительные результаты для всех исходных данных тестового набора. Поэтому для программиста крайне важно иметь *удобные инструментальные средства, которые могут обеспечить быстрое прохождение всей цепочки действий от набора исходного текста до получения результатов.*

Такие возможности предоставляют специальные программные комплексы, которые называются *системами программирования*.

Можно выделить следующие основные компоненты, присутствующие в любой системе программирования:

- текстовый редактор, обеспечивающий максимум удобств при вводе и корректировке исходного кода программы и имеющий средства работы с файловой системой;

- компилятор и компоновщик, которые обеспечивают процесс преобразования исходного текста в машинные коды (в качестве альтернативы возможен интерпретатор);
- отладчик, который позволяет выполнять программу по шагам;
- набор библиотек с готовыми подпрограммами, которые можно включать в разрабатываемые программы — чем больше таких библиотек, тем удобнее и проще процесс разработки в данной системе программирования;
- набор примеров законченных программ, которые помогают быстрее освоить конкретный диалект языка программирования, демонстрируя некоторые типовые приемы программирования;
- программы-утилиты, предназначенные для выполнения различных вспомогательных функций;
- файлы с документацией.

В современных системах программирования текстовый редактор, компилятор с компоновщиком и отладчик обычно объединены в одном модуле, который называется *интегрированной средой разработки (Integrated Development Environment — IDE)*. Запустив на выполнение такую среду, можно не выходить из нее и никуда не переключаться в течение всего цикла отладки программы (описание интегрированной среды Turbo Pascal имеется в *приложении 3*).

3.5. Системы программирования на основе языка Pascal

Поскольку таких систем программирования существует множество, перечислить все крайне затруднительно. Упомянем только самые популярные.

Безусловным лидером в продвижении языка Pascal является компания Borland. Фирма занимается этой деятельностью с 1983 г., когда была выпущена в свет первая версия системы Turbo Pascal. С тех пор вышел не один десяток версий систем программирования под названиями Turbo Pascal, Borland Pascal, Borland Pascal for Windows, Delphi, Kylix (версия Delphi для операционной системы Linux). Существенный вклад в дело продвижения языка Pascal был внесен выдающимся программистом Андерсом Хейлсбергом, сотрудником фирмы.

Огромный коммерческий успех продуктов фирмы Borland дал основание говорить о наличии так называемого "промышленного" стандарта языка Pascal, под которым понимается входной язык последних версий системы Delphi. Возможно, в будущем этот диалект будет официально утвержден.

При начальном обучении программированию на языке Pascal неофициальным стандартом можно считать версию Turbo Pascal 7.0 или ее несколько расширенный вариант Borland Pascal 7.0. Хотя язык этих систем имеет незначительные отличия от стандарта Extended Pascal, тем не менее он поддерживается разработчиками других компиляторов и систем программирования.

Системы Turbo (Borland) Pascal 7.0 были разработаны для операционной системы *DOS*, поэтому встроенные в них компиляторы работают неэффективно под управлением операционной системы Windows и других операционных систем. Эти компиляторы имеют общее название — 16-разрядные компиляторы (имеется в виду 16 двоичных разрядов — основная единица обработки в операционной системе DOS).

Современные компиляторы для операционных систем Windows и UNIX иначе называются 32-разрядными (единица обработки — 32 двоичных разряда). Они порождают более быстродействующие машинные программы. Например, программа на языке Pascal, откомпилированная в Delphi, будет работать примерно в два раза быстрее, чем та же самая программа, но откомпилированная в Turbo Pascal.

В последнее время получили распространение кроссплатформенные компиляторы и системы программирования на основе Pascal, которые поддерживают большинство наиболее распространенных операционных систем. Наиболее популярными из них являются Free Pascal (компилятор и среда разработки, похожая на Turbo Pascal), компилятор GNU Pascal (свободно распространяемые программные продукты), TMT Pascal (распространяется на коммерческой основе, но стоит недорого, кроме того, есть и бесплатная версия).

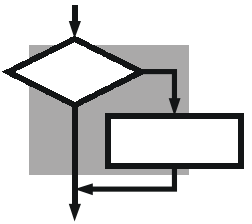
Эти системы ориентированы на стандарт Extended Pascal (хотя и не полностью его поддерживают). Однако все они поддерживают и диалект фирмы Borland, поэтому многочисленные программы, разработанные ранее, можно перекомпилировать в новых системах, выбрав соответствующую платформу, для которой необходимо сформировать машинный код. Кроме того, входной язык каждой из таких систем имеет собственные расширения стандарта, зачастую очень интересные.

3.6. Контрольные вопросы

1. Что такое компилятор и чем он отличается от интерпретатора?
2. В каких случаях удобно использовать интерпретатор? Когда незаменим компилятор?
3. Какие этапы обработки проходит программа на языке высокого уровня для получения исполняемого кода?

4. Чем вызвана необходимость компоновки?
5. Какие типы ошибок, возникающих при программировании, вам известны?
6. Расскажите, как для исправления семантических ошибок используются системы тестов и трассировка программы.
7. Перечислите состав типовой системы программирования. Каково ее назначение?
8. Что такое интегрированная среда разработки?
9. В чем отличие 16-разрядных и 32-разрядных компиляторов языка Pascal?

Глава 4



Программа и данные. Типы данных

Современные системы программирования значительно облегчают рутинную работу программиста, однако не могут освободить его от творческой составляющей его труда. Разработка программы по-прежнему остается серьезной интеллектуальной работой, требующей хорошего понимания фундаментальных принципов работы компьютера. В первую очередь обсудим процесс *исполнения программы* и рассмотрим такие важные понятия, как *данные* и их *типы*.

4.1. Основные понятия

4.1.1. Исполнение программы

Предположим, что программа уже прошла подготовительные этапы компиляции и компоновки. Рассмотрим внимательнее основной процесс, ради которого создавалась программа — выполнение подготовленного машинного кода.

При работе в интегрированной среде разработки машинный код программы после подготовки уже находится в оперативной памяти и готов к исполнению. Если подготовка программы была выполнена заранее и запускается файл с расширением exe из операционной системы, то сначала выполняется *загрузка программы* в оперативную память. При работе с интерпретатором в оперативной памяти находится не весь код программы, а только подготовленный фрагмент программы, но это сути дела не меняет. В любом случае *исполняться может только код, размещенный в оперативной памяти*.

В ходе выполнения программы процессор последовательно получает и исполняет инструкции, из которых состоит программа. Большинство инструкций связано с какой-либо обработкой информации (данных). Все данные, необходимые программе для ее работы, помещаются в отдельную область оперативной памяти, которая называется *областью данных программы*.

Таким образом, при запуске программы на выполнение для нее выделяются две различных области оперативной памяти: *область кода* и *область данных программы*.

При выполнении программы инструкции и данные поступают на регистры процессора. Это быстродействующие элементы памяти, находящиеся в микросхеме процессора. После выполнения процессором очередной инструкции полученные результаты посылаются на хранение в оперативную память или остаются в регистрах, а процессор переходит к анализу и выполнению следующей инструкции. Этот процесс продолжается до тех пор, пока не встретится инструкция завершения программы или не возникнет неустранимая ошибка выполнения. Схематически (немного упрощенно) процесс выполнения программы представлен на рис. 4.1.

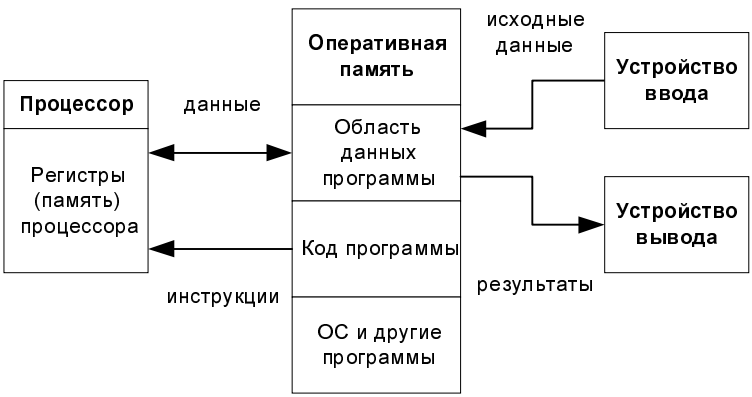


Рис. 4.1. Исполнение программы

Данные, предназначенные для обработки, называются *исходными данными* и задаются (вводятся) обычно в начале выполнения программы. Программа по ходу выполнения может запрашивать недостающие исходные данные.

Исходные данные могут поступать от различных *устройств ввода*. Основной способ задания исходных данных для компьютерной программы — ввод их с клавиатуры. Выбор какого-либо пункта меню, щелчок мышью на определенной кнопке на экране — также способы ввода исходных данных. Помимо этого, программа может считывать исходные данные из файлов на диске, принимать их от сканирующего устройства или других устройств ввода данных.

Независимо от способа ввода исходных данных, они обязательно помещаются в область данных программы в оперативной памяти и, как правило, сохраняются в ней до конца выполнения программы (или до тех пор, пока они нужны программе).

В процессе выполнения программы исходные данные преобразуются в *результаты*. Результаты обычно сначала поступают в оперативную память, а затем выводятся на экран, принтер или другие *устройства вывода*, а также могут быть записаны в файлы на диске, отосланы по электронной почте и т. д.

По ходу выполнения программы в область данных программы могут быть помещены *промежуточные* данные, которые остаются в пямяти до тех пор, пока в них есть необходимость.

По окончании работы программы вся выделенная для нее память освобождается.

Архитектура компьютера (т. е. структура и принципы взаимодействия отдельных блоков), которая изображена на рис. 4.1, иначе называется архитектурой фон Неймана. Название дано в честь одного из разработчиков данной архитектуры, известного математика Джона фон Неймана.

Так устроено большинство современных компьютеров, во всяком случае, персональные компьютеры, с которыми мы имеем дело. Языки программирования высокого уровня, которые рассматриваются в книге, разработаны именно для компьютеров с архитектурой фон Неймана. Их конструкции преобразуются компилятором в машинный код, приспособленный для эффективного исполнения в такой архитектуре.

В настоящее время используются и другие архитектуры компьютеров, для которых разрабатываются другие языки программирования. Примером другой архитектуры могут служить многопроцессорные компьютеры с параллельными вычислениями. Однако это предмет отдельного курса.

4.1.2. Биты и байты

Обратите внимание:

Вся информация (и программа, и ее данные) хранится в памяти компьютера одинаково — в *двоичной* форме в виде последовательности *битов* (от *binary digit*, двоичная цифра). Биты иначе называют двоичными разрядами.

Каждый бит может принимать значение одной двоичной цифры — *ноль* или *единица*. Восемь битов объединены в *байт*. Байт — это минимальная единица памяти, которая может быть выделена под переменную или инструкцию программы.

Каждый байт оперативной памяти имеет свой порядковый номер, который называется его *адресом* (адреса начинаются с нуля).

Максимальное число, которое можно записать при помощи восьми двоичных цифр, — это 11111111, что соответствует десятичному числу 255; минимальное — 0. Поэтому значением байта может быть любое целое число от 0 до 255 (всего 256 различных чисел, что составляет 2^8).

Различные данные могут занимать один или несколько байт. Участок памяти, в котором хранится одно значение, в программировании принято называть *ячейкой памяти*. Название это достаточно условное, т. к. никаких границ между ячейками в памяти нет. Адресом ячейки называют адрес ее самого первого байта. Таким образом, *каждый элемент данных имеет свой адрес в памяти*. Программа на машинном языке содержит в своих инструкциях конкретные адреса памяти, откуда следует брать данные и куда их помещать.

К счастью, в языках высокого уровня данные обозначаются при помощи символических имен (идентификаторов), а всю работу по переводу символических имен в адреса оперативной памяти берет на себя транслятор.

4.2. Константы и переменные

Каждый элемент данных в программе является константой или переменной.

Константами называются элементы данных, значения которых в процессе выполнения программы не изменяются. Самые простые примеры констант — названия дней недели или месяцев, значение ускорения свободного падения и т. п. Такая информация может понадобиться программе для работы, но нет необходимости заставлять пользователя вводить эти данные с клавиатуры или тратить время на чтение их из дискового файла. *Значения констант задаются непосредственно в тексте программы* при ее разработке.

Переменные, в отличие от констант, могут менять свои значения при выполнении программы. Например, значения процентов по различным видам банковских вкладов или тарифы за пользование услугами мобильной связи постоянно пересматриваются, поэтому будет грубой ошибкой со стороны программиста задавать эти значения в программе в виде констант.

Принято поступать по-другому: такие данные обозначаются символическими именами и являются *переменными*. Для их хранения отводится *ячейка* в оперативной памяти компьютера, а сами значения переменных при выполнении программы вводятся в качестве исходных данных или формируются по ходу выполнения алгоритма.

Использование переменных в программе позволяет записать алгоритм решения задачи в общем виде и сделать программу пригодной для *многократного использования с различными исходными данными*.

Таким образом, переменная — *ячейка оперативной памяти компьютера, которой присвоено определенное имя (идентификатор). Содержимое ячейки (значение переменной) может меняться по ходу выполнения програмы*. Каждое *новое значение*, записанное в ячейку памяти, *"затирает"* предыдущее значение, поэтому *в любой момент времени* переменная имеет только одно, *текущее*,

значение. Значение переменной в определенный момент времени иногда называют *состоянием переменной*.

В математике значение переменной в рамках определенной задачи неизменно. Именно поэтому высказывание `a := a + 1` математик сочтет неверным (если знак `:=` будет трактоваться им как равенство). Тем не менее, для программиста это абсолютно правильная конструкция; она задает вычисление суммы содержимого ячейки `a` и числовой константы `1` и занесение полученного результата в ту же ячейку `a`.

Языки высокого уровня позволяют давать имена не только переменным, но и константам, и именование констант считается хорошим стилем программирования. Например, ускорение свободного падения можно обозначить буквой g, как принято в физике, присвоить ему известное значение 9,81 и далее использовать символическое обозначение.

Обратите внимание — механизм использования констант и переменных различен, т. к. *константы, как правило, не занимают отдельных ячеек памяти.*

Так, в приведенном примере `a := a + 1` отдельная ячейка памяти будет выделена только под переменную `a`, константа `1` будет *вписана непосредственно в текст программы.*

Замечание

Исключением являются типизированные константы языка Pascal *(см. разд. 6.3.4)*, для которых выделяется память. В языках C/C++ также различают два вида констант: занимающие и не занимающие память.

4.3. Типы данных и способы внутреннего представления данных

4.3.1. Концепция типов данных

Именование констант и переменных в программировании очень похоже на использование символических выражений в алгебре или физике. Однако для того чтобы компилятор смог нормально обработать символические имена, в программе нужно *выполнить описание (определение, объявление) переменных и символических констант.* При описании сообщается о *типе* каждой именованной величины.

Тип определяет множество значений, которые могут принимать данные (константы и переменные), а также совокупность операций, допустимых над этими значениями.

Например, значения 1 и 3 относятся к *целочисленному типу*, и над ними можно выполнять любые арифметические операции. Значения `'отличная'` и

'учеба' принадлежат к *строковому типу* и над ними можно выполнять только одну операцию — *склеивания, или конкатенации* текста (обозначается через +).

В конечном счете, *тип определяет форму внутреннего (машинного) представления данных и размер отводимой для них ячейки памяти.* Известно, что все данные одного типа занимают ячейки памяти одинаковых размеров и внутренняя форма хранения данных одинакова. Это понятно — ведь конкретные значения переменных появляются по ходу выполнения программы, а память под переменные выделяется до заполнения их данными. Поэтому выручить тут может только явное указание типа.

Из всего сказанного выше следует очень важный вывод: *все типы данных имеют ограниченный диапазон значений,* т. к. под каждый тип отводится ячейка определенного размера.

Ячейка памяти — это не автобус, в который при желании всегда поместится лишний человек. Каждый бит может хранить только одну двоичную цифру, поэтому при попытке вместить слишком много бит возникает *переполнение ячейки памяти.*

Этот факт не согласуется с нашими математическими представлениями о бесконечных множествах целых и действительных чисел. Тем не менее, в программировании это фундаментальная истина.

Для понимания особенностей хранения и обработки данных различных типов важно понимать *способы внутреннего представления данных.*

4.3.2. Внутреннее представление данных в памяти компьютера

В различных языках программирования обозначения типов данных и размеры ячеек памяти для них могут незначительно различаться. Допустим, целочисленный тип имеет обозначение `integer` в Pascal и `int` в C/C++ и для него отводится два или четыре байта в зависимости от системы программирования. Однако внутренняя форма представления данных не зависит от конкретного языка программирования и определяется только архитектурой процессора. Напомним, что вся информация хранится только в *двоичной* системе счисления.

Имеются всего две основных формы представления данных в компьютере:

- с фиксированной точкой (в русском переводе иногда употребляют термин "с фиксированной запятой", т. к. речь идет о разделителе, отделяющем целую часть числа от дробной части);
- с плавающей точкой (запятой).

В дальнейшем для определенности будем употреблять слово "точка".

Первый способ употребляется для хранения целых чисел, символов (!) и логических значений. Считается, что десятичная точка при этом зафиксирована в конце числа, т. е. дробной части нет. Второй способ используется для вещественных чисел либо целых чисел, имеющих очень широкий диапазон значений.

Для того чтобы различие двух способов хранения данных стало понятным, вспомним математику и другие точные науки, где также используется два способа записи чисел:

- обычный (127, −3,26 и т. д.);
- в виде произведения двух сомножителей, один из которых — степень с основанием 10 ($1{,}7 \cdot 10^{12}$, $25 \cdot 10^{-8}$ и т. д.). При этом первый из сомножителей принято называть *мантиссой,* а показатель степени — *порядком.*

Второй способ удобен при записи очень больших или очень маленьких чисел. Очевидно, когда решался вопрос о способе хранения вещественных чисел в компьютере, именно такая форма представления показалась наиболее экономичной с точки зрения затрат памяти. Разумеется, в качестве основания степени было выбрано число 2 (а не 10), поскольку все данные хранятся в двоичной системе счисления.

Например:

- целое число 19 в двоичной форме с фиксированной точкой будет записано как 10011 (разложение по степеням двойки: $19 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$);
- вещественное число 15,375 в двоичной форме с плавающей запятой можно представитьть как $1{,}111011 \cdot 2^{11}$, где 1,111011 — мантисса, 11 — порядок. Проверить правильность такого представления можно так:

  $15{,}375 = 1{,}921875 \cdot 2^3$

  $1{,}921875 = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-6}$ (получили 1,111011)

  $3 = 1 \cdot 2^1 + 1 \cdot 2^0$ (получили 11)

Рассмотрим подробнее два способа хранения данных.

Хранение чисел с фиксированной точкой

Можно выделить две разновидности таких данных: целые числа без знака и со знаком.

Целые числа без знака — хранятся в двоичной системе счисления в определенном количестве байтов.

Например, десятичное число 14 в виде одного байта представлено так: 00001110 ($14 = 1 \cdot 2^3 + 1 \cdot 2^2 + ... + 1 \cdot 2^1 + 0 \cdot 2^0$). Старшие ("левые") биты заполнены нулями.

Ячейка памяти для хранения этого числа может быть разных размеров: наиболее распространенные варианты — один, два или четыре байта. Не занятые под цифры старшие байты заполняются нулями.

Целые числа со знаком — для указания знака числа применяется самый левый бит двоичного числа, причем по принятому соглашению ноль обозначает положительное число, а единица — отрицательное. Отрицательные числа при этом преобразуются к виду, который называется *дополнительным кодом числа.*

Алгоритм построения кодов отрицательных чисел таков:

1. Записать модуль числа обычным образом в двоичной системе, а в неиспользуемые старшие биты записать нули.
2. Все нули заменить единицами, а единицы нулями. Тем самым получим обратный код числа.
3. К полученному числу прибавить единицу. Если при этом образуется единица переноса из старшего разряда (переполнение), отбросить единицу переполнения.

Пример. Записать в дополнительном коде число −33, тип — двухбайтовое целое.

Модуль числа в обычном виде	0000000000100001
Обратный код	1111111111011110
Дополнительный код	1111111111011111

Числа в дополнительном коде складываются точно так же, как и беззнаковые, при этом единица переноса из старшего разряда отбрасывается.

Вычитание одного числа из другого сводится к сложению первого числа с дополнительным кодом второго, т. е. тоже к сложению.

Например, сложим 33 и −33.

```
 0000000000100001 +
 1111111111011111
1|0000000000000000
```

В данном случае отбросили единицу переполнения, в результате чего получили 0.

Замечание

При выборе типа данных необходимо знать диапазон величин, которые могут храниться в n битах памяти. Для *знаковых целых* справедливо следующее соотношение. Число N может храниться в n битах памяти, если $-2^{n-1} = N = 2^{n-1} - 1$.

Представление с плавающей точкой

В этом случае хранимое число представлено в виде произведения $m \cdot 2^p$ и следовательно в одной ячейке памяти требуется хранить не одно значение, а два — мантиссу m и порядок p, причем оба числа могут иметь знак. Разумеется, основание степени (двойка) нигде не хранится, но подразумевается.

Например, $0{,}375 = 0{,}25 + 0{,}125 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}$. В двоичном представлении с плавающей точкой это число можно записать как $0{,}011 \cdot 2^0$. В данном случае $m = 0{,}011$; $p = 0$.

Для увеличения количества значащих цифр мантиссу обычно подвергают *нормализации*.

Нормализация означает, что мантисса должна удовлетворять условию $0{,}5 \cdot m < 1$ (возможен вариант $1 \cdot m < 2$), т. е. ненулевая мантисса любого хранимого числа с плавающей точкой должна начинаться с двоичной единицы.

В приведенном выше примере мантисса не нормализована, т. к. имеет ноль слева. После выполнения нормализации получим, например, $m = 0{,}11$; $p = -1$. При этом мантиссу можно хранить как целое двоичное число 11 (поскольку целую часть мантиссы, которая всегда равна нулю, хранить не имеет смысла).

Число байт для хранения мантиссы и порядка может быть различным (в целом размеры ячейки памяти для хранения числа с плавающей точкой могут составлять 4, 6, 8, 10 байт).

Замечание

Реальные способы хранения чисел с плавающей точкой несколько сложнее, чем изложенные выше. Например, в IBM-совместимых персональных компьютерах вместо порядка числа хранится его *характеристика,* которая получается из порядка по определенной формуле и всегда имеет только положительные значения, кроме того, вообще не хранится первая цифра мантиссы, т. к. она всегда равна 1. Однако эти тонкости сути дела не меняют.

Сравнение способов представления с фиксированной и плавающей точкой

Все сказанное выше дает основание задуматься о том, что компьютерная арифметика имеет свои специфические законы, знание которых позволит избежать многочисленных проблем, возникающих при отладке программы. Рассмотрим следствия, которые вытекают из приведенных выше способов хранения данных.

- В математике рассматривается *бесконечное* множество целых чисел. Целый тип в языке программирования — это *интервал* целых чисел. Операции над целыми числами определены лишь тогда, когда исходные данные и результат лежат в этом интервале. В противном случае возникает

ситуация, называемая *переполнением*. При целочисленном переполнении, как правило, ошибка выполнения программы не возникает, однако *результат искажается*. Такую ошибку обнаружить всегда непросто, поэтому следует быть особенно осторожным при выборе *типа* целочисленных данных.

- За исключением переполнения все операции над аргументами с фиксированной точкой выполняются *точно* (без погрешности). Исключение может составлять операция деления целого на целое (например, 2/3).
- Все операции над данными с фиксированной точкой выполняются существенно быстрее, чем над данными с плавающей точкой. В последнем случае значительное время тратится на операции *выравнивания порядков* (для возможности выполнения операции *операнды* должны иметь один и тот же порядок).
- В математике вещественные числа — это *бесконечное непрерывное* множество чисел. В памяти компьютера вещественные числа представляются *конечным* множеством значений. Поскольку размеры ячейки памяти для хранения вещественного числа существенно больше размеров ячейки для хранения целого числа, количество различных значений вещественных чисел так велико, что может показаться бесконечным. Например, внутреннее представление шестибайтового действительного числа может дать $2^{48} = 281\,474\,976\,710\,656$ (более чем 10^{14}) возможных комбинаций. Это очень большое число, но все же оно несопоставимо с бесконечным множеством вещественных чисел.
- Вещественные числа всегда представлены в компьютере с некоторой погрешностью. Все операции над вещественными числами также выполняются с погрешностью.
- Каждое вещественное число будет иметь приблизительно одинаковое количество *значащих цифр* в его представлении, т. к. под мантиссу отводится фиксированное количество байт. Как следствие этого, ошибка для очень больших чисел будет больше по абсолютной величине.
- Представители вещественных чисел неравномерно распределены внутри диапазона значений. Их плотность уменьшается с увеличением абсолютного значения числа.
- Нельзя сравнивать на равенство/неравенство вещественные числа обычным способом. Поскольку они представлены в памяти с погрешностью, сравнения их не всегда могут быть абсолютно достоверны. Вместо этого можно сравнивать модуль разности этих чисел с каким-нибудь достаточно маленьким числом.
- Следует быть осторожным при выполнении операций между числами, порядки которых сильно отличаются, например, между очень большими и очень маленькими числами. Не исключены ситуации, когда мантисса

не сможет обеспечить требуемую точность. Например, при сложении 10000000 и 0.00000001 разрядности мантиссы не хватит, чтобы представить число 10000000.00000001, и результат останется равным 10000000.

Замечание

Для уменьшения влияния ошибки округления при выполнении арифметических операций с вещественными числами необходимо иметь в виду следующее. Если складывается много чисел, то необходимо: *разбить их на группы чисел, близких по абсолютному значению*, произвести суммирование в группах, *начиная с меньшего числа*, после чего полученные суммы сложить, опять-таки начиная *с меньшей*. Таким образом, *при работе с вещественными числами при перестановке мест слагаемых сумма может измениться, и иногда значительно.* Аналогичные рекомендации можно дать и для других арифметических операций.

4.3.3. Классификация типов

Несмотря на то, что имеются всего лишь две основные формы представления данных в компьютере, различных *типов данных* в языках высокого уровня гораздо больше. С одной стороны, это хорошо, поскольку программист имеет возможность выбора наиболее оптимального типа для представления тех или иных данных, с другой стороны, большой выбор всегда создает определенные сложности.

Все типы данных, используемые в языках высокого уровня, можно разделить на две большие группы:

- скалярные (простые);
- структурированные (композитные, составные).

Скалярные типы в свою очередь подразделяются на стандартные (базовые) и пользовательские (производные от базовых типов). *Стандартные типы* предлагаются программисту разработчиками языка.

Исходя из внутреннего представления данных, все языки программирования имеют группу *целочисленных* и группу *вещественных* типов. В дополнение многие языки имеют *символьный* и *логический типы*. В некоторых языках имеется *денежный* тип (`money`, `currency`), а также тип *даты* (`date`, `datetime`). Интересно, что все эти типы имеют внутреннее представление с фиксированной точкой. Например, дата обычно хранится в виде целого числа, представляющего количество дней, содержащихся в ней. Деньги обычно хранятся с точностью до копеек или сотых долей копеек.

Пользовательские типы данных образуются из стандартных типов по определенным правилам, и такую возможность предоставляют многие языки программирования.

Структурированные типы имеют в своей основе скалярные типы данных и используются для хранения совокупностей данных. Большинство языков программирования предоставляет стандартный набор структурированных типов:

- массивы (в том числе строки — массивы символов);
- записи;
- дополнительно в Pascal есть множества.

Универсальные языки программирования имеют также *файловые типы,* которые обычно относят к структурированным типам.

4.4. Контрольные вопросы и задания

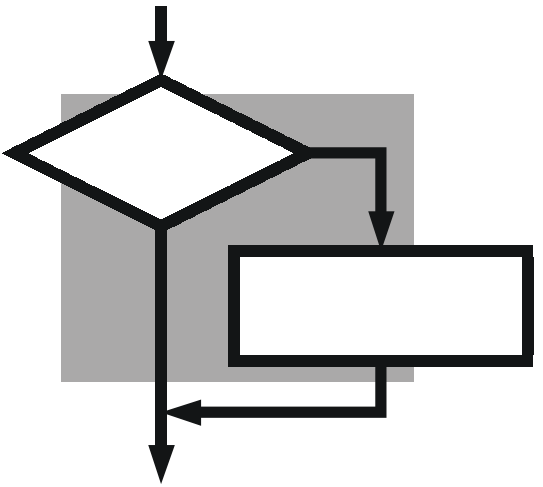
Вопросы

1. Перечислите, какие известные вам устройства, входящие в состав компьютера, используются для ввода/вывода, обработки данных и хранения промежуточных результатов.
2. Дайте определение константы и переменной. Чем они отличаются?
3. Какие единицы измерения информации вы знаете?
4. Что такое ячейка памяти?
5. Что называется адресом переменной?
6. Что такое тип?
7. Чем вызвана необходимость описания данных?
8. В чем состоит различие представления данных с фиксированной и плавающей точкой?
9. Всегда ли арифметические операции над целыми числами выполняются абсолютно точно?
10. Объясните, почему вещественные числа нельзя сравнивать на точное равенство.
11. В чем состоит различие между представлением множества вещественных чисел в математике и вычислительной технике?

Задания

1. Сколько различных комбинаций из нулей и единиц можно записать в 4 битах?
2. Сколько двоичных цифр (бит) необходимо, чтобы закодировать одну школьную оценку?

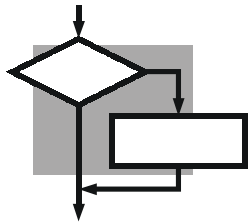
3. Гибкий магнитный диск (дискета) может хранить 1,44 мегабайта информации (1 Мбайт = 2^{20} = 1048576 байт ≈ 1 млн байт). Определите приблизительно, сколько страниц этого учебника можно запомнить на одной дискете.
4. Изображение на экране стандартного цветного монитора с диагональю 15" состоит из 600 строк по 800 точек. Минимальное количество цветов, в которые должна быть окрашена каждая точка экрана для комфортной работы с большинством офисных программ, составляет 256. Какой объем памяти необходим для запоминания одного цветного изображения в этом случае?
5. Запишите в двоичной системе счисления числа 12; 37; 89; 0,75. Запишите в форме с плавающей точкой следующие значения: 0,125; 4,5. Мантисса при этом должна быть нормализована.



ЧАСТЬ II

Основы программирования на языке Pascal

Глава 5



Типы данных и константы языка Pascal

До сих пор изложение строилось таким образом, чтобы фундаментальные понятия программирования не были привязаны к какому-либо конкретному языку высокого уровня. Логика авторов должна быть понятна — языков программирования много и они постоянно развиваются, но определенный теоретический багаж поможет быстро осваивать синтаксис новых языков (известно, что по своей семантике все языки универсального назначения практически идентичны).

И все-таки, нельзя научиться программировать на всех языках сразу, поэтому практическая часть курса должна быть основана на одном из языков высокого уровня. Многолетний опыт преподавания подсказывает, что одним из лучших языков для начального обучения программированию является Pascal. На нем и остановим свой выбор.

Изложение стандарта этого языка в том виде, как он был разработан Виртом, вряд ли полезно, поскольку стандарт в чистом виде не реализован ни в одной системе программирования. Поэтому в книге используются и те дополнения к языку, которые внесла фирма Borland. Обычно сначала разбирается базовый вариант какой-либо конструкции, основанный на диалекте Turbo Pascal 7.0, а затем упоминаются расширения, сделанные в Delphi, если таковые имеются для данной конструкции.

Замечание

Обычно дополнения по Delphi оформлены в виде таких замечаний, поэтому при первом чтении их можно пропустить. Тем не менее, следует четко представлять себе, что промышленный стандарт — это Delphi, и при разработке серьезных программ необходимо пользоваться такими системами, как Delphi.

Начнем изучение с конкретных типов данных, которые предоставляет разработчику этот язык.

5.1. Классификация типов. Стандартные типы: порядковые и вещественные

Язык Pascal имеет традиционный для языка высокого уровня набор типов данных, в который входят как простые (скалярные), так и составные (структурированные) типы. Кроме того, этот язык позволяет легко создавать производные (пользовательские) типы данных на основе уже имеющихся.

В языке Pascal к стандартным типам относятся:

- целочисленные;
- вещественные;
- символьный (литерный);
- логический (булевский).

Целочисленные типы, символьный, логический, а также два пользовательских типа данных (перечисляемый и интервальный) образуют группу так называемых *порядковых типов*. Все порядковые типы имеют внутреннее представление с фиксированной точкой, поэтому у них много общего. Каждому значению порядкового типа можно поставить в соответствие целое число — порядковый номер. Для этих типов имеются специальные функции *(см. разд. 7.4)*.

Все вещественные типы имеют внутреннее представление с плавающей точкой. Диапазон этих чисел столь велик, что невозможно поставить в соответствие каждому из чисел порядковый номер, поэтому вещественные типы не относятся к порядковым.

Структурированные типы имеют в своей основе скалярные типы данных. К ним относятся:

- массивы;
- множества;
- записи;
- файлы;
- классы (для объектно-ориентированных расширений языка).

В данной главе мы рассмотрим только простые (скалярные) типы данных.

К сожалению, различные системы программирования на основе языка Pascal по-разному выделяют память для данных одних и тех же типов. Так, 16-разрядные компиляторы, разработанные в свое время для операционной системы DOS, выделяют два байта для наиболее популярного целочисленного типа `integer`, более поздние 32-разрядные компиляторы для операци-

онных систем Windows и UNIX выделяют четыре байта. То же самое можно сказать и о наиболее часто используемом вещественном типе `real` (6 байтов или 8 байт).

Тем не менее, обозначения типов данных во всех системах программирования полностью совпадают и способы записи констант различных типов также идентичны. Поэтому сведения, приведенные ниже, окажутся полезными при работе в любой среде программирования. Размеры ячейки памяти (а следовательно, и диапазон значений) приводятся применительно к популярным системам Turbo Pascal (Borland Pascal).

Замечание

В некоторых системах программирования (например, Delphi) даже вводятся понятия логического и физического типов, причем разработчик может сам выбирать физический тип, т. е. размер ячейки памяти, для каждого из логических типов (разумеется, только в допустимых пределах). Подробнее о логических и физических типах Delphi см. *приложение 1*.

5.2. Целые типы. Логический тип

Целые типы

Целочисленных типов данных в языке Pascal довольно много. Они различаются размером выделяемой ячейки памяти и способом представления (целое без знака или целое со знаком).

Таблица 5.1. *Целочисленные типы данных в Turbo Pascal*

Название целого типа	Диапазон возможных значений	Память, байт
`byte` (байтовый)	0—255	1
`shortint` (короткий целый)	–128—127	1
`integer` (целый)	–32 768—32 767	2
`Word` (слово)	0—65 535	2
`longint` (длинный целый)	–2 147 483 648—2 147 483 647	4

Типы `byte`, `word` представлены как целые без знака.

Целочисленные константы

Константы целых типов записываются в общепринятой математической форме. Например, 356, −5 и т. д.

Целочисленные данные могут быть представлены как в десятичной, так и в шестнадцатеричной *системах счисления*. Признаком *шестнадцатеричной* константы является знак доллара — $, после которого следуют шестнадцатеричные цифры (допустимый диапазон — от $0 до $FFFFFFFF).

Замечание

В десятичной системе счисления для представления числа используются десять цифр от 0 до 9. Для шестнадцатеричной их шестнадцать — от 0 до 9 и буквы: A (это шестнадцатеричная цифра, соответствующая числу 10 обычной десятичной системы счисления), B (это — 11), C (это — 12), D (это — 13), E (это — 14), F (это — 15). Например: 123 ($123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$) — целое десятичное число, а $7B ($7B = 7 \cdot 16^1 + B \cdot 16^0$) — число, равное ему, в шестнадцатеричной форме записи.

Логический тип

Логический (булевский) тип имеет всего два значения: `true` (да — истина, 1) и `false` (нет — ложь, 0) (табл. 5.2). Переменные логического типа занимают в памяти один байт, хотя для их хранения хватило бы и одного бита (байт — минимально возможный размер ячейки памяти, отводимой под переменную).

Над данными логического типа нельзя выполнять обычные арифметические операции, для них определены логические операции и операции сравнения. Интересно, что логические значения упорядочены, т. е. в операциях сравнения `true > false`.

Таблица 5.2. *Логический (булевский) тип данных*

Тип	Диапазон возможных значений	Память, байт
`boolean` (булевский)	`true`, `false`	1

5.3. Символьный тип. Строки символов

Символьный (литерный) и строковый типы представляют данные, являющиеся *символами* и их последовательностями — *строками* (табл. 5.3). В памяти компьютера символы хранятся в виде их *числовых кодов* (беззнаковые целые числа от 0 до 255). Понятно, что каждый символ занимает ровно один байт в памяти.

Числовые коды преобразуются в буквы и другие символы лишь в момент их вывода на экран или принтер. Соответствие между символом и его кодом задается при помощи *кодовой таблицы*, которая находится в памяти компьютера и используется при выводе символов *(см. приложение 2)*.

Таблица 5.3. *Символьный тип данных*

Тип	Диапазон возможных значений	Память, байт
`Char` (символьный, литерный)	Символы кодовой таблицы	1

Запись символов. Специальные и управляющие символы

В том случае, если в программе требуется использовать значение символьной переменной или константы, его необходимо заключить в апострофы или записать с использованием знака #, за которым следует код символа.

Например, `'A'` обозначает букву A, `';'` — точку с запятой, `' '` — пробел, `#32` или `#$20` являются также символами пробела (32 — это код, соответствующий пробелу, а шестнадцатеричное число 20 равно десятичному 32).

Символьные константы упорядочены по кодам. Например, `'a'<'b'` — истина.

Рекомендуется применять # (знак номера) только для *специальных (служебных) символов,* которые не отображаются на экране и имеют мнемонические сокращения, унаследованные из прошлого. Некоторые из них могут использоваться программистом для выполнения определенных действий:

- `#07` (BEL) — подача короткого звукового сигнала;
- `#08` (BS) — удаление символа слева от курсора и смещение курсора на одну позицию назад, соответствует клавише <BackSpace>;
- `#09` (HT) — горизонтальная табуляция: смещение курсора в позицию, кратную 8, плюс 1 (9, 17, 25 и т. д.), соответствует клавише <Tab>;
- `#10` (LF) — перевод строки, курсор смещается по вертикали вниз на одну строку;
- `#11` (VT) — вертикальная табуляция;
- `#12` (FF) — прогон страницы;
- `#13` (CR) — возврат каретки или перевод строки, выполняет перемещение курсора в начало следующей строки экрана (соответствует клавише <Enter>);
- `#26` (SUB) — конец файла, вводится нажатием комбинации клавиш <Ctrl>+<Z>;
- `#27` (ESC) — конец работы, символ соответствует клавише <Esc>;
- `#32` (BL) — пробел

и т. д.

Запись строк символов

Последовательность символов, заключенная в апострофы, является *строкой* и относится к типу `string`. Причем сами апострофы не входят в состав строки, а лишь указывают на то, что все заключенные в них символы следует рассматривать как единое целое — *строковую константу*. Если в состав строки потребуется включить сам апостроф, достаточно написать его дважды подряд. *Строчные и прописные буквы в составе строки различаются*, т. к. им соответствуют различные коды. Максимальная длина строки — 255 символов. Символы внутри строки нумеруются от 1 до значения длины строки.

Например: `'Язык программирования Turbo Pascal'`, `'12345'`, `'A+B'`. Более подробно строки и действия над ними будут рассматриваться ниже *(см. гл. 14)*.

5.4. Вещественные типы

Основные вещественные типы данных приведены в табл. 5.4.

Таблица 5.4. *Вещественные типы данных в Turbo Pascal*

Название вещественного типа	Диапазон возможных значений (плюс-минус)	Количество значащих цифр	Память, байт
`single` (с одинарной точностью)	1,5e – 45 — 3,4e38	7—8	4
`real` (вещественный)	2,9e – 39—1,7e38	11—12	6
`double` (с двойной точностью)	5,0e – 324—1,7e308	15—16	8
`extended` (с повышенной точностью)	3,4e – 4932—1,1e4932	19—20	10
`comp` (сложный)	–2e63 + 1—2e63 – 1	19—20	8

Замечание

При разработке в системах Turbo Pascal или Borland Pascal программ, критичных ко времени счета, следует заменять тип `real` на `single`, `double` или `extended`. Скорость вычислений при этом увеличивается не менее чем в 2—3 раза. Но пользоваться этими типами несколько сложнее, т. к. необходимо изменить способ компиляции, используемый по умолчанию. Сделать это можно двумя методами: при включенной директиве комилятора`{$N+}` или установив переключатель для соответствующего пункта меню интегрированной среды **Options | Compiler | 8087/80287**.

Тип comp, являясь вещественным (приблизительно от $-9{,}2 \cdot 10^{18}$ до $9{,}2 \cdot 10^{18}$), фактически представляет "большое" целое число со знаком, сохраняющее 19—20 значащих десятичных цифр. В то же время в выражениях он полностью совместим с любым другим вещественным типом. Наиболее подходящая область для его применения — это бухгалтерские расчеты.

Формы записи вещественных констант

Вещественные числа могут записываться двумя способами — в общепринятой форме и форме с мантиссой и порядком. Общепринятая форма предполагает запись по обычным правилам арифметики. Целая часть отделяется от дробной *десятичной точкой*, а не запятой, как в математике. Если точка отсутствует, число считается *целым*.

Запись вещественного числа в форме с *мантиссой* и *порядком* использует степень десяти (например: $25 \cdot 10^{-3}$) и удобна для записи очень больших и очень маленьких чисел. При этом число изображается так: пишется мантисса, знак умножения опускается, вместо основания 10 пишется буква *e*, а следом указывается порядок (показатель степени). Буква *e*, предшествующая порядку, читается как "умножить на 10 в степени".

Например, 123.456 или −11.9 — общепринятая форма, а 5.18e + 02 (518) или 10e − 03 (0,01) — форма с мантиссой и порядком.

Примеры *неправильной* записи вещественных чисел:

- 123 — отсутствует десятичная точка;
- 1,23 — запятая вместо точки;
- 0.123 − 03 — отсутствует обозначение порядка *e*;
- 12.34e1.2 — порядок числа должен быть целым.

Обратите внимание: *любое вещественное число хранится в памяти компьютера в форме с плавающей точкой независимо от способа его записи.*

5.5. Типы данных, определяемые программистом

О языке Pascal говорят, что он строго *типизирован* — программист должен описать все объекты программы, указывая их типы, и использовать объекты только в соответствии с этими типами. Может показаться, что такой подход не способствует творчеству, ограничивая программиста. На самом деле он предотвращает анархию, помогая создавать надежные и качественные программы. Принуждая программиста к аккуратности при описании объектов

программы, Pascal избавляет его от необходимости искать и исправлять ошибки при выполнении, что гораздо труднее.

Начинающим программистам обычно хватает стандартных типов этого языка. А для более опытных программистов, стремящихся улучшить качество своих программ, язык Pascal предоставляет средства определения новых типов, производных от стандартных типов. Такие типы иначе называются *пользовательскими*. Дадим первое представление о создании пользовательских типов.

Из определения типа следует, что задать тип данных — значит определить множество его допустимых значений. Поэтому определение типа имеет вид:

```
ИмяТипа = выражение, описывающее множество допустимых значений.
```

В простейшем случае выражение справа от знака равенства может представлять собой имя стандартного типа, например:

```
int=integer    или
integer=longint
```

В первом случае просто заменили длинное имя `integer` более коротким `int`. Во втором случае изменили (переопределили) стандартный тип `integer`, придав ему диапазон типа `longint`.

Замечание

Второй случай возьмите на заметку, поскольку это самый быстрый способ изменить диапазон значений всех переменных какого-либо стандартного типа.

Более подробно об определении типа *см. разд. 6.3*.

Далее рассмотрим два пользовательских типа, которые применяются довольно часто.

5.6. Перечисляемый и интервальный типы

В Pascal имеются два дополнительных порядковых типа, которые относятся к пользовательским типам:

- интервальный тип или диапазон;
- перечисляемый тип.

Они используются для того, чтобы еще больше ограничить количество значений, принимаемых переменными этого типа, и позволяют:

- значительно улучшить наглядность программы;

- оптимально выделить память под переменные, т. к. в этом случае компилятор сам подбирает наиболее подходящий размер ячейки памяти;
- облегчить поиск ошибок (благодаря возможности контроля тех значений, которые получают соответствующие переменные).

Интервальный тип задается своим минимальным и максимальным значениями и может быть определен на основе любого порядкового типа:

```
МинимальноеЗначение .. Максимальное значение
```

Например: `1..12` (номер месяца может принимать значения от 1 до 12) или `'a'..'z'` (буквы латинского алфавита — от a до z).

Для каждой операции с переменной интервального типа автоматически выполняется проверка: остается ли значение переменной внутри установленного для нее диапазона. Например, при попытке присвоить номеру месяца значение ноль будет выведено сообщение об ошибке.

Автоматическая проверка объявленных границ позволяет программисту не отвлекаться на организацию собственного контроля, что является существенной выгодой от использования интервального типа.

Перечисляемый тип ограничен еще больше, поскольку он задается перечислением своих значений. Сами значения указываются через запятую, а весь список заключается в круглые скобки. Первое значение имеет порядковый номер 0, второе — 1 и т. д. (при необходимости до 65535).

Например: `color=(red,blue,green,black)`.

В приведенном примере создается новый перечисляемый тип данных `color`. Переменные этого типа могут принимать всего 4 значения: `red`, `blue`, `green` и `black`.

Для значений перечисления одного и того же типа допустимы операции сравнения. Упорядочение осуществляется по номеру элемента в описании типа.

Например, будет истинно выражение `blue < green`, т. к. `blue` имеет меньший номер по порядку в описании типа, чем `green`.

В языке Pascal применение перечисляемого типа ограничено тем, что значения данных этого типа нельзя вводить с клавиатуры или выводить на какое-либо устройство вывода, а также над ними нельзя выполнять обычные арифметические операции. Их можно только присваивать переменной перечисляемого типа.

Обычно этот тип применяется для хранения промежуточных данных, что позволяет сделать текст программы более выразительным и понятным (см. листинги 9.19, 10.14, 14.9).

5.7. Контрольные вопросы и задания

Вопросы

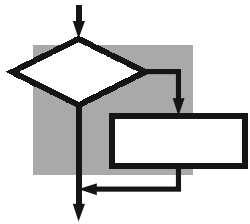
1. Чем отличаются друг от друга порядковые и вещественные типы?
2. Как вы думаете, почему в среде Turbo Pascal тип `integer` имеет размер два байта, а в среде Delphi — четыре байта?
3. Когда возникает переполнение?
4. Как записывается целое число в шестнадцатеричной системе счисления?
5. Где применяется булевский тип данных, какие он принимает значения, сколько места требуется для его размещения в памяти?
6. В каких целях в программе могут использоваться служебные символы?
7. Как выполняется запись строк символов в программе?
8. В каких случаях используются типы `single`, `double`, `extended`, `comp`?
9. Что такое пользовательские типы данных, чем они отличаются от стандартных?
10. Приведите примеры перечисляемого и интервального типов.

Задания

1. Какие утверждения неправильны:
   - 156 — целое число?
   - 137 — шестнадцатеричное целое число?
   - 5e3 — целое число?
   - 125.78 — вещественное число?
   - $2AF — шестнадцатеричное число?
   - `'Строка'` — строковое значение?
2. Какие из приведенных чисел будут представлены в форме с плавающей точкой: 155, 20.5e + 02, 123.456, 78e2, 53.7e − 04?
3. Какие из следующих утверждений неправильны:
   - для диапазона `1..260` лучше всего подходит тип `byte`;
   - для диапазона `0..75000` лучше всего подходит тип `word`;
   - для диапазона `'a'..'z'` лучше всего подходит тип `char`;
   - для вещественных переменных обычно применяется тип `real`;
   - значение 32 000 входит в тип `integer`.

4. Какой тип лучше всего подходит для данных диапазона: `6..90`, `40..45`, `10..65000`, `0..10000`?
5. Какой идентификатор типа описывает самый широкий диапазон данных?
6. Какие из следующих соотношений неправильны:
   - 6.22e + 02 = 622?
   - 20e − 03 = 0.02?
   - 2347.6e − 03 = 2.34760?
   - 0.2e03 = 2000.0?
   - 1200e + 03 = 12000.0?
7. Какое из следующих утверждений неправильное:
   - перед шестнадцатеричными числами записывается знак #?
   - перед кодом символа записывается знак #?

Глава 6



Программа на языке Pascal

6.1. Алфавит языка

Программа на языке Pascal формируется с помощью конечного набора знаков, образующих *алфавит языка*, и состоит из:

- прописных и строчных букв латинского алфавита (A, B, ..., Z, a, b, ..., z) и знака подчеркивания;
- десятичных (0, 1, ..., 9) и шестнадцатеричных цифр (0, 1, ..., 9, A, B, ..., F).

Кроме того, в алфавит включаются *специальные символы* (табл. 6.1) и комбинации специальных символов — они образуют *составные символы* (табл. 6.2).

Таблица 6.1. *Специальные символы*

Символ	Название	Символ	Название
+	Плюс	{ }	Фигурные скобки
-	Минус	.	Точка
*	Звездочка	,	Запятая
/	Дробная черта	:	Двоеточие
=	Равно	;	Точка с запятой
>	Больше	'	Апостроф
<	Меньше	#	Номер
[ ]	Квадратные скобки	$	Знак денежной единицы
( )	Круглые скобки	^	Каре
@	Коммерческое а		Пробел (не имеет обозначения)

Таблица 6.2. *Составные символы*

Символ	Название	Символ	Название
:=	Присваивание	<=	Меньше или равно
<>	Не равно	>=	Больше или равно
..	Диапазон значений	(..)	Альтернатива [ ]
(* *)	Альтернатива { }		

6.2. Лексика языка

Неделимые последовательности знаков алфавита образуют *лексемы*. В языке Pascal различают такие виды лексем:

- константы *(см. гл. 5)*;
- зарезервированные слова;
- идентификаторы;
- знаки операций;
- разделители;
- комментарии.

Зарезервированные слова, идентификаторы, знаки операций и разделители

Зарезервированные слова языка Pascal являются составной частью языка, имеют фиксированное начертание и несут в программе определенный смысл (табл. 6.3). Иначе их называют *служебными* или *ключевыми словами*.

Таблица 6.3. *Зарезервированные слова*

Слово	Смысл слова	Слово	Смысл слова
Absolute	Абсолютный	Constructor	Конструктор
And	Логическое "и"	Destructor	Деструктор
Array	Массив	Div	Деление нацело
Asm	Ассемблер	Do	Выполнять
Begin	начало блока	Downto	Уменьшить до
Case	Вариант	Else	Иначе
Const	Константа	End	Конец блока

Таблица 6.3 *(окончание)*

Слово	Смысл слова	Слово	Смысл слова
Export	Экспорт	or	Логическое "или"
External	Внешний	packed	Упакованный
File	Файл	procedure	Процедура
For	Для	program	Программа
Function	Функция	record	Запись
Forward	Опережающий	repeat	Повторять
Goto	Переход на	set	Множество
If	Если	shl	Сдвиг битов влево
Implementation	Реализация	shr	Сдвиг битов вправо
In	В (входит в)	string	Строка
Inherited	Унаследованный	then	То
Inline	Основной	to	Увеличивая
Interface	Интерфейс	type	Тип
Interrupt	Прерывание	unit	Модуль
label	Метка	until	До
library	Библиотека	uses	Использовать
mod	Остаток от деления	var	Переменная
nil	Отсутствие	while	Пока
not	Логическое "не"	with	С
object	Объект	xor	Исключающее "или"
of	Из		

Внутри зарезервированных слов пробелы использовать запрещено.

Идентификаторами называют *имена (названия)* переменных, констант, типов данных и других объектов программы, о которых речь пойдет в следующих разделах. Различают стандартные идентификаторы и идентификаторы, определенные пользователем (пользователь, как всегда, — программист).

Стандартные идентификаторы служат для обозначения заранее определенных разработчиками языка типов данных, констант, процедур и функций. Например, стандартный идентификатор `sin` в выражении `sin(x)` вызывает функцию, вычисляющую синус угла `x`.

Стандартные идентификаторы по своему назначению схожи с зарезервированными словами языка, однако между ними есть разница: *любой из стандартных идентификаторов допускается переопределять.* Пользователь может написать свою собственную функцию с именем `sin` или определить константу или переменную с таким именем. Часто это ведет к ошибкам, например, в случае с синусом стандартная функция вычисления синуса становится недоступной программе. Поэтому *стандартные идентификаторы лучше использовать по их непосредственному назначению.*

Идентификаторы пользователя применяются для обозначения констант, переменных, типов и других объектов, определенных самим программистом. Тип идентификатора пользователя должен быть указан в описательной части программы, до его использования *(см. разд. 6.3)*.

Для написания идентификаторов применяются следующие правила:

- состоят из букв, цифр и знака подчеркивания; специальные символы, в том числе и пробел, не допускаются. Буквы русского алфавита не могут входить в состав идентификатора Pascal, их можно использовать только в строковых константах;
- начинаются с буквы или знака подчеркивания. Только для метки допускается использование целого числа без знака;
- между двумя идентификаторами должен стоять, по крайней мере, один разделитель;
- нельзя использовать имена, совпадающие по написанию с приведенными ранее зарезервированными словами. Крайне нежелательно также переопределение стандартных идентификаторов;
- при написании имен можно использовать как прописные, так и строчные буквы. Компилятор не делает различий между ними. Например, `MYVAR`, `MyVar`, `myvar` — это три различных варианта написания имени одной и той же переменной.

В программах на Pascal часто используют такой способ написания: первая буква каждого слова прописная, остальные — строчные (например, `Textcolor`). Однако в примерах этой книги будут использоваться, в основном, строчные буквы, что позволит упростить для пользователя процесс ввода текста программ.

Обычно начинающие программисты особо не задумываются над тем, какими именами пользоваться им в своих программах, и применяют короткие однобуквенные обозначения типа `a` или `x`. Но надо четко понимать, что понятные и осмысленные идентификаторы делают текст программы более выразительным и наглядным.

Опытные программисты не жалеют времени на продумывание системы имен в своих программах. Они обычно дружат с английским языком, поэтому часто используют в качестве имен английские слова или их сокращения. Авторы не уверены, что все читатели в совершенстве владеют английским, поэтому в качестве имен мы используем или очень известные английские слова (которые нужно знать) или русские слова и их сокращения, записанные английскими буквами. Например, для обозначения количества можем воспользоваться переменной kol вместо красивых английских слов quantity или amount (это синонимы). Если читатель хорошо знает английский, лучше все-таки использовать слова английского языка.

Имена, используемые в программе, должны быть *уникальными*, то есть в данном блоке программы один идентификатор не должен использоваться для обозначения более чем одной переменной, константы и т. д. Если это требование не выполняется, на экран выводится сообщение об ошибке.

Например: `metka13`, `Blok_15` — допустимые имена.

Примеры *неправильной* записи имен:

- `3DGraph` — начинается с цифры;
- `Nomer.Doma` — содержит точку;
- `blok#1` — содержит специальный символ;
- `My Program` — содержит пробел;
- `Div` — зарезервированное слово.

Знаки операций — это обозначения операций над данными различных типов. Значения, к которым применяется операция, называются ее *операндами*. Примеры операций — `+`, `<`, `div`.

Разделителями служат — пробел, символ конца строки, точка с запятой, точка, запятая, двоеточие, круглые и квадратные скобки. Разделители, стоящие внутри строковой константы, воспринимаются не как разделители, а как ее часть.

Комментарии и директивы компилятора

В любом месте программы, где разрешен разделитель, можно записать пояснительный текст — *комментарий*. Он не обрабатывается компилятором и не включается в исполняемый exe-файл.

Текст комментария ограничен символами `{ }` или `(* *)`:

`{ ТекстКомментария }` или `(* ТекстКомментария *)`.

Допускается следующая вложенность комментария:

```
{ Текст (* ТекстКомментария2 *) Комментария 1 }
```

или

```
(*Текст { ТекстКомментария2 } Комментария1*)
```

Следует знать:

- комментарии удобно использовать при *отладке программы* для *временного исключения* группы операторов, которая, будучи заключена в `{ }` или `(* *)`, воспринимается как комментарий и, следовательно, не выполняется;
- комментарии необходимо отличать от *директив компилятора*, которые используются программистом для управления *режимами компиляции*. Директивы, как и комментарии, заключаются в фигурные скобки, но имеют отличительный признак в виде символа `$`. По умолчанию они находятся в состоянии, гарантирующем минимальный объем исполнимого кода и минимум времени компиляции. Несколько директив компилятора могут находиться в одной строке. Примеры директив компилятора:
  - `{$R+}` — проверять выход за границы диапазонов;
  - `{$I-}` — отмена контроля операций ввода/вывода;
  - `{$F+}` — формировать дальний тип вызова процедур и функций.

6.3. Структура программы

6.3.1. Общие сведения

В общем случае программа имеет вид:

```
{ описательная часть }
begin
   { исполнительная часть }
end.
```

Описательная часть не выполняет никаких действий и служит, в основном, для правильного выделения памяти под данные, используемые в программе. Последовательность действий (инструкций) по обработке данных содержится в исполнительной части программы. В редких случаях описательная часть может отсутствовать. Без исполнительной части программа бессмысленна.

Заголовок

В начале программы может находиться *заголовок*, состоящий из зарезервированного слова `program`, имени программы и параметров, с помощью которых программа взаимодействует со своим внешним окружением.

```
program ИмяПрограммы(input, output);{стандартные файлы ввода/вывода}
```

Имя программе присваивается самим программистом для удобства работы с ней. Оно позволяет отличать одну программу от другой, используя только заголовок и не анализируя подробно последующий текст. Более удобным

способом является использование комментария, помещенного в начало программы. Поэтому в примерах, приведенных в книге, заголовок программы отсутствует. Имя программы никак не связано с именем файла, содержащим ее текст.

Разделы программы

Программа на языке Pascal состоит из следующих *разделов*:

- ❒ { заголовок }
- ❒ { описательная часть }
  - раздел подключаемых библиотечных модулей;
  - раздел объявления меток;
  - раздел объявления констант;
  - раздел объявления типов;
  - раздел объявления переменных;
  - раздел объявления процедур и функций;
- ❒ { исполнительная часть }
  - раздел инструкций (операторов) программы, заключенный между словами `begin` и `end`;
  - признак конца программы — . (точка).

Описательная часть предназначена для объявления всех встречающихся в программе данных и других объектов.

В *исполнительной части* (разделе операторов) записывается последовательность исполняемых операторов. *Каждый оператор выражает действие, которое необходимо выполнить.*

Структура программы

В самом общем виде структура программы имеет вид:

```
program ИмяПрограммы;
uses
  ИмяМодуля1,..;
label
  ИмяМетки1,..;
const
  ИмяКонстанты = ЗначениеКонстанты;
type
  ИмяТипа = ЗначенияТипа;
```

```
var
  ИмяПеременной : Тип;
  { объявления процедур и функций программиста }
begin
  { инструкции основной программы }
end.
```

Обратите внимание — разделы описания могут встречаться в программе любое количество раз и следовать в произвольном порядке (кроме раздела uses, который всегда расположен после заголовка программы). Любой раздел, кроме раздела операторов, может отсутствовать. *Главное, чтобы все описания объектов программы были сделаны до того, как они будут использованы.*

6.3.2. Раздел *uses*

Раздел uses позволяет подключать *стандартные и пользовательские библиотечные модули*. Он начинается с зарезервированного слова uses и имеет следующий вид:

```
uses
  ИмяМодуля1, ИмяМодуля2, ...;
```

Например: `uses crt;`

О модуле crt и о других модулях речь пойдет ниже *(см. гл. 12)*.

6.3.3. Раздел описания меток

Перед любым оператором Pascal в тексте программы можно поставить *метку*, что позволяет выполнить прямой переход на этот оператор с помощью оператора goto из любого места программы *(см. разд. 8.5)*. Метка состоит из имени и следующего за ней двоеточия, после которого и располагается помеченный данной меткой оператор. Все метки, используемые в программе, должны быть описаны. Если же метка описана, но не используется, то ошибки при этом не возникает. Раздел описания меток начинается с зарезервированного слова label и имеет следующий вид:

```
label
       ИмяМетки1, ИмяМетки2, ...;
```

6.3.4. Раздел описания констант

Хранение констант не требует памяти и компилятор помещает их значения прямо в текст исполняемой программы. Каждая константа принадлежит к

определенному типу данных, однако при определении константы его обычно не указывают.

Обратите внимание — тип констант автоматически опознается по форме их записи *(см. гл. 5)*.

Именованные константы

Раздел описания констант начинается с зарезервированного слова const (от латинского *constants* — постоянный) и имеет следующий вид:

```
const
  ИмяКонстанты = ЗначениеКонстанты;
```

Например:

```
const
  g=9.8;            { вещественная константа }
  count=maxint/2+1;{maxint - зарезервированная константа, см. табл. 6.4}
  nmax=100;         { целая константа }
  nmin=-nmax;       {можно использовать ранее определенные константы}
  s='абвгд';        { строковая константа }
  kod=$123;         { шестнадцатеричная константа }
```

Обратите внимание — при определении констант применяется знак =, а не :=.

Идентификатор константы можно употреблять при определении следующих за ней констант.

Замечание

Интересно, что если в правой части определения константы используется выражение, то оно вычисляется не во время *выполнения* программы, а при ее *трансляции*.

Типизированные константы

В языке Pascal существуют так называемые *типизированные константы*. Такое название вызвано тем, что при их описании указывается тип:

```
const
  ИмяКонстанты : Тип = Значение;
```

Например:

```
const
  ocenka : byte = 5;
  predmet : string = 'Информатика';
```

Для хранения типизированных констант память выделяется как под обычные переменные. В языке Turbo Pascal значения типизированных констант можно *изменять* по ходу выполнения программы, т. е. это, по сути, не константы, а переменные с заранее заданным начальным значением. Иначе такие переменные называются *инициализированными*. Это не очень хороший подход, поскольку ключевое слово `const` традиционно принято использовать только для обозначения констант.

В системе Delphi данный недостаток исправлен. Там типизированные константы — это *константы, занимающие память*. Их значение нельзя менять по ходу программы. Однако при описании переменных их можно инициализировать начальными значениями по тем же самым правилам, которые приняты для типизированных констант.

Например: `var x:integer=5; {только в Delphi!}`

Замечание

Типизированные константы могут быть любого типа, кроме файлового *(см. гл. 17)*.

Обратите внимание — использование констант позволяет поменять одновременно множество значений по всему тексту программы (а именно, столько значений, сколько раз встречается в тексте каждая константа), внеся изменения только в одном месте программы — разделе `const`. Используйте именованные константы везде, где это возможно.

Зарезервированные константы

Без предварительного описания в программе можно использовать значения *зарезервированных* или *предопределенных констант* (табл. 6.4).

Таблица 6.4. *Зарезервированные константы*

Идентификатор	Тип	Значение	Описание
`True`	`boolean`	`true`	Истина
`False`	`boolean`	`false`	Ложь
`maxint`	`integer`	32 767	Максимальное целое
`maxlongint`	`integer`	2 147 483 647	Максимальное длинное целое

6.3.5. Раздел описания типов данных

В языке Pascal существует *механизм создания новых типов данных (см. разд. 5.5)*. Каждое новое определение типа задает множество значений и связывает с этим множеством имя типа. Определение новых типов выпол-

няется в разделе описания типов данных. В простых программах этот раздел часто отсутствует, т. к. в этом случае хватает и стандартных типов.

Раздел начинается с зарезервированного слова type и имеет вид:

```
type
  ИмяТипа1 = ОписаниеТипа1;
  ИмяТипа2 = ОписаниеТипа2;
```

Например:

```
const min=1; max=31;
type
num_month=1..12;
num_day=min..max;
days=(monday,tuesday,wednesday,thursday,friday,saturday,sunday);
```

В приведенных примерах определяются два интервальных типа и один перечисляемый тип. Далее идентификаторы типов можно использовать для описания переменных.

6.3.6. Раздел описания переменных

Все переменные, используемые в программе, должны быть перечислены в *разделе описания переменных*. Описание должно предшествовать использованию переменной. После того как переменная описана, она может быть опознана компьютером, а в тексте программы к ней можно обратиться по имени. Однако содержимое переменной после описания еще не определено.

Раздел начинается с зарезервированного слова var и выглядит так:

```
var
  ИмяПеременной1,, ИмяПеременнойN: ТипПеременной;
```

Например:

```
var
   x1,x2: integer; y1: longint; { целочисленные переменные }
   sum: real; root: double;     { вещественные переменные }
   znak: char;                  { символьная переменная }
   flag: boolean;               { логическая переменная }
   day: days;                   { переменная пользовательского типа days }
   Season: (Winter, Spring, Summer, Autumn); { переменная перечисляемого
                                               типа, определяемого при
                                               описании этой переменной }
   Month: 1..12;                { переменная интервального типа }
```

Замечание

Обратите внимание на два последних описания. Оказывается, пользовательские типы данных можно определять непосредственно в разделе описания переменных `var`. Такие типы называются *анонимными,* т. к. не имеют имени. Однако хорошим стилем считается определять новый именованный тип в разделе `type`.

Рекомендации:

- если в программе используются переменные разных типов, то зарезервированное слово `var` (от англ. *variable* — переменная) лучше записать только один раз, а затем привести списки имен переменных каждого типа (как в приведенном выше примере);
- после объявления переменной в качестве комментария рекомендуется указывать ее назначение.

6.3.7. Раздел описания процедур и функций

Данный раздел используется в программах, которые с целью удобства программирования были разбиты на более мелкие части — подпрограммы. *Подпрограммой* называется программная единица (часть программы), имеющая имя, по которому она может быть вызвана из других частей программы.

Подпрограммы бывают двух видов: *процедуры* и *функции*, которые, в свою очередь, делятся на *стандартные* и *определенные пользователем*. Стандартные процедуры и функции являются частью языка и вызываются без предварительного описания. Описание процедур и функций пользователя выполняется в разделе описания процедур и функций.

В общем случае подпрограмма имеет ту же структуру, что и программа.

Объявление процедуры:

```
procedure ИмяПроцедуры (ФормальныеПараметры);
  { описательная часть процедуры }
begin
  { Инструкции исполнительной части процедуры }
end;
```

Объявление функции:

```
Function ИмяФункции (ФормальныеПараметры) : ТипРезультата;
  { описательная часть функции }
begin
  { Инструкции исполнительной части функции }
  ИмяФункции := Результат;
end;
```

Подробнее о процедурах и функциях будет рассказано ниже *(см. гл. 10)*.

6.3.8. Раздел операторов

Раздел операторов является основным, т. к. именно в нем над предварительно описанными константами, переменными, значениями функций выполняются действия, позволяющие получить результат, ради которого и создавалась программа. Раздел начинается зарезервированным словом `begin` и далее следуют операторы языка, *отделенные друг от друга точкой с запятой*. Завершают раздел зарезервированное слово `end` и *точка*:

```
begin
  Оператор1;
  Оператор2;
  ...
  ОператорN;
end.
```

Обратите внимание — операторы выполняются последовательно в порядке записи сверху вниз и слева направо. В одной строке может быть записано сразу несколько операторов, разделенных точкой с запятой, а длинный оператор может занимать несколько строк экрана (но без разрыва ключевых слов).

Если между двумя операторами отсутствует точка с запятой, то это приведет к возникновению ошибки.

Например, запись вида

```
x:=1
y:=2;
```

несмотря на то, что операторы присваивания записаны в разных строках программы, будет воспринята компилятором как:

```
x:=1y:=2;
```

В итоге получается "оператор", в котором используются два знака присваивания и неправильный идентификатор `1y` (имя не может начинаться с цифры).

Замечание

Если за оператором следуют ключевые слова `end` или `until` (это слово используется в операторе цикла), то точку с запятой можно не ставить. Однако не советуем привыкать к такому способу написания, поскольку во многих других языках программирования точка с запятой является завершающей частью оператора, и опускать ее нельзя.

Операторы языка Pascal можно разделить на простые и сложные.

Простые операторы не содержат внутри себя других операторов. *Сложные (структурные)* операторы представляют собой конструкции, содержащие простые операторы.

К простым относятся следующие операторы: присваивания, перехода, пустой оператор, а также инструкции ввода и вывода *(см. разд. 7.2)*. К сложным операторам относятся: составной оператор, условный оператор, операторы цикла, оператор выбора, оператор присоединения в записях.

Как видите, набор операторов не так уж велик. Он соответствует стандартному набору операторов любого языка высокого уровня. В следующих главах мы подробно разберем назначение каждого из операторов.

6.4. Контрольные вопросы и задания

Вопросы

1. Из чего состоит алфавит языка?
2. Что такое разделитель? Какие разделители вы знаете?
3. В чем отличие зарезервированных слов от стандартных идентификаторов?
4. Перечислите правила написания идентификаторов пользователя.
5. Укажите, в каких целях используются комментарии, а в каких — директивы компилятора.
6. Чем отличаются друг от друга описательная и исполнительная части программы?
7. Может ли в программе отсутствовать раздел операторов?
8. Как операторы отделяются друг от друга?
9. Какова цель использования раздела `uses`?
10. Как описывается метка и для чего она используется?
11. Как опознается тип констант?
12. Выполните сравнение именованных, типизированных и зарезервированных констант с обычными константами.
13. Как объявляют типы и переменные в языке Pascal?
14. Чем вызвана необходимость использования раздела описания процедур и функций?
15. Как оформляется раздел операторов?

Задания

1. Что в данном списке можно рассматривать как идентификатор: `FIO`, `ФИО`, `3456`, `X`, `>=`, `&`, `$`, `Summa`, `_Result`?
2. Укажите какие идентификаторы проще воспринимаются при чтении и почему: `Klass1`, `Klass_1`, `summadoxoda`, `SummaDoxoda`, `nomerdoma`, `Nomer_Doma`?

3. Сколько в следующем списке зарезервированных слов: `X`, `Program`, `Y`, `Summa`, `MyMoney`, `Произведение`, `George`, `begin`, `end`, `if`, `repeat`, `Read`?
4. Какие имена из перечисленных в списке являются правильными? Почему? `Zarplata`, `Сумма`, `Summa Nalogov`, `Teach_Kurs`, `12Kurs2`, `Summa_Elemen$tov`?
5. Попробуйте написать самую короткую программу.
6. Какая программа из двух, представленных ниже, является правильной? Почему?

```
program MyProgram;
begin
   Writeln('Hello, world!');
end.
```

или

```
program MyProgram;
begin
   X:=Y+1;
end.
```

7. Какие из комментариев неправильные?

```
{ Комментарий 1 };
(*Комментарий 2 *);
{{Комментарий в комментарии 1}};
(* {Комментарий в комментарии 2} *);
{(* Комментарий в комментарии 3*)};
(*(*Комментарий в комментарии 4*)*);
```

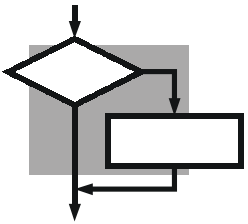
8. Есть ли причины к невыполнению следующей программы?

```
program Short;
begin
end.
```

9. В тексте приведенной ниже программы содержатся три ошибки. Исправьте их.

```
program Ошибки!;
begin
   Summa:=6+8;
end;
```

Глава 7



Простые (линейные) программы

7.1. Пример учебной программы

Создадим первую учебную программу вычисления суммы двух целых чисел. Сценарий взаимодействия человека и компьютера можно предложить следующий. Компьютер запрашивает у человека значение первого целого числа, человек набирает значение на клавиатуре и нажимает клавишу <Enter>, компьютер считывает введенное число и записывает в память под именем a. Затем компьютер запрашивает значение второго целого числа, считывает его и записывает в память под именем b. После этого компьютер выполняет сложение чисел a и b, записывает результат в память под именем sum, выводит на экран сообщение Сумма чисел... и значение величины sum (листинг 7.1).

Листинг 7.1. Вычисление суммы двух целых чисел

```
{ Учебная программа вычисления суммы двух целых чисел }
program example;          { заголовок программы }
var
   a,b,sum: integer;      { переменные a,b,sum — целые }
begin                     { начало программы }
   { вывод запроса на экран }
   write('введите значение целого числа a: ');
   { ввод значения a с клавиатуры }
   readln(a);
   { вывод запроса и ввод значения b }
   write('введите значение целого числа b: ');
   readln(b);
   { вычисление переменной sum }
   sum:=a+b;
   { вывод ответа }
   write('сумма чисел ',a,' и ',b, ' = ',sum);
end.                      { конец программы }
```

Кратко прокомментируем текст.

- В данной программе использованы следующие зарезервированные слова:
  - `program` — заголовок программы, который можно было бы опустить;
  - `var` — начало объявления переменных;
  - `integer` — указание, что переменные `a`, `b`, `sum` — целые числа;
  - `begin`, `end` — начало и конец программы.
- Для ввода исходных данных и вывода результатов использованы инструкции:
  - `write` — для вывода на экран;
  - `readln` — для ввода исходных данных.
- Для вычисления суммы использована инструкция присваивания (`sum:=a+b`).
- Программа разбивается на строки для удобства восприятия. Принято начинать строки с нескольких пробелов, тогда слова `begin` и `end` хорошо выделяются.
- Комментарии заключены в фигурные скобки. Все комментарии в данном примере можно убрать, при этом программа останется работоспособной.
- Рассмотрим используемые в данной программе операторы более подробно.

7.2. Ввод и вывод данных

7.2.1. Инструкции ввода и вывода

Для ввода и вывода данных в языке Pascal предусмотрены следующие процедуры ввода/вывода: `read`, `readln`, `write` и `writeln`. Названия означают "читай", "читай строку" (read line), "пиши", "пиши строку" (write line) соответственно.

Замечание

Часто эти процедуры называют операторами по аналогии с другими операторами языка, хотя, строго говоря, называть процедуры операторами неверно. В стандарте Pascal не определены операторы ввода и вывода, поэтому для этой цели были разработаны стандартные процедуры. Разница между оператором и процедурой Pascal состоит в том, что имя процедуры, в отличие от зарезервированных слов оператора, можно переопределить. То есть можно написать, например, свою собственную процедуру с именем `write` или описать переменную с таким именем. Но нельзя назвать переменную, например, именем `begin`. В дальнейшем, говоря о вводе/выводе, будем использовать термины "инструкция" или "команда", т. к. они имеют более широкое толкование, чем термин "оператор".

Ввод данных

Часто первыми действиями, выполняемыми программой, являются действия по вводу данных. *Ввод данных* — это передача исходных данных программы в оперативную память компьютера для обработки. Основные устройства ввода — клавиатура и дисковый файл *(см. гл. 17)*.

Вспомним, что описание переменных служит для выделения памяти. Выделенные ячейки памяти остаются незаполненными до тех пор, пока в них не будет записано какое-то значение. *Ввод данных — один из способов задания значений переменных.*

После окончания ввода значения переменных известны и их можно использовать в дальнейших вычислениях. В противном случае они не определены и, следовательно, непригодны для использования.

Инструкцию ввода с клавиатуры можно записать в одной из форм:

```
read(x1, x2,...,xN);
readln;
readln(x1, x2,...,xN);
```

где `x1`, `x2`, ..., `xN` — список ввода, содержащий имена переменных допустимых типов (`integer`, `real`, `char`, `string`). При этом первые две инструкции, выполненные последовательно, эквивалентны третьей.

Например:

```
var i:integer; a:real; ch:char;
begin
  readln(i,a);
  readln(ch);
...
```

Следует знать:

- инструкция `readln` при вводе с клавиатуры предпочтительнее `read`, т. к. полностью освобождает *буфер клавиатуры* — рабочую область памяти, в которой временно хранятся введенные с клавиатуры символы. Инструкция `read` оставляет в буфере клавиатуры код клавиши <Enter> (нажатие которой завершает процесс ввода), что может привести к неправильной работе следующей команды ввода;
- в одной инструкции `read` или `readln` можно записать несколько переменных, разделенных запятыми. При выполнении программы значения, вводимые с клавиатуры, можно разделять пробелом либо символом табуляции (клавиша <Tab>) или нажимать клавишу <Enter> после ввода каждого из значений;

- инструкция `readln;` (без переменных) обычно записывается в конце программы и служит для создания паузы, которая длится до нажатия пользователем клавиши <Enter>. В противном случае после окончания работы программы окно с текстом программы закроет экран с полученными результатами;
- тип данных, вводимых во время работы программы, должен соответствовать типу переменной, указанной в инструкции ввода. Это замечание касается в первую очередь числовых данных, в записи которых не должно быть недопустимых символов.

В случае ввода недопустимых значений исходных данных программа завершает работу и на экран выводится сообщение об *ошибке ввода/вывода*. Например, если программа была запущена из системы разработки Turbo (Borland) Pascal, выведется сообщение **Error 106: Invalid numeric format** (Ошибка 106: Неверный числовой формат). Если программа была запущена из *операционной системы* — сообщение **Run time error 106** (Ошибка времени выполнения 106). Номера ошибки в разных системах программирования могут различаться. В следующих разделах мы покажем, как избежать неприятной ситуации возникновения ошибки ввода/вывода *(см. разд. 9.1 и 14.3.3)*.

Выполнение инструкции ввода не связано с появлением поясняющих надписей на экране. Для вывода сообщений используется оператор `write`.

Вывод данных

Вывод данных — это передача данных после обработки из оперативной памяти на устройство вывода (экран, принтер, файл на диске).

Инструкция вывода на экран записывается в одной из следующих форм:

```
write(y1, y2,...,yN);
writeln;
writeln(y1, y2,...,yN);
```

где `y1, y2,...,yN` — *список вывода*. Первые две инструкции, выполненные последовательно, эквивалентны третьей.

Например:

```
write(a + b);
writeln('Сумма равна:', sum);
```

Следует знать:

- инструкции `write` и `writeln` предназначены для вывода констант различных типов, значений переменных или выражений. Число параметров — произвольно;

- из констант наиболее часто выводятся строки текста (вспомним, что строковые константы заключаются в апострофы);
- если в инструкции вывода записано выражение, то сначала будет произведено его вычисление, а затем выполнен вывод полученного результата. Подробнее о вычислении арифметических выражений рассказано ниже *(см. разд. 7.4)*;
- процедура вывода `writeln` аналогична `write`. Отличие заключается в том, что после вывода курсор автоматически переходит в начало новой строки;
- инструкция `writeln` (без параметров) переводит курсор в начало следующей строки. Таким способом можно, например, отделять результаты работы программы друг от друга одной или несколькими пустыми строками.

7.2.2. Формат вывода

При использовании оператора вывода для представления на экране значения целого типа выделяется столько позиций экрана (разрядов), сколько требуется для записи числа. Вывод значения вещественного типа осуществляется в форме с использованием степени десяти. Такой вид результатов на экране часто не устраивает пользователей, поэтому программисту следует позаботиться о приемлемом *формате вывода*.

В инструкциях вывода имеется возможность записи выражения, определяющего *ширину поля вывода* для каждой выводимой переменной или константы:

```
write(y1:w:d, y2:w:d,...,yN:w:d);
writeln(y1:w:d, y2:w:d,...,yN:w:d);
```

где `w` задает общую ширину поля вывода; `d` — место под дробную часть; `w` и `d` — константы или выражения целого типа. Параметр `d` указывается только для выражений вещественного типа и в этом случае результат выводится в общепринятой форме.

Обратите внимание:

- если `w`, заданное программистом, мало, то при выводе ширина поля будет *увеличена*. Часто задают `w = 0` для автоматического подбора ширины поля вывода;
- если `w`, наоборот, слишком велико, то выводимое значение прижимается к правому краю поля вывода, а слева выводятся пробелы;
- если мало `d`, то производится *округление*.

Например, для

```
c:=1.234; write('C=':10, c:7:3);
```

результат на экране имеет вид:

xxxxxxxxC=xx1.234

где **x** — это пустая позиция (пробел).

7.3. Оператор присваивания

Операторы `read` и `readln` позволяют задавать значения переменных, предназначенных для хранения исходных данных. Их обработка в программе связана с хранением промежуточных значений и результатов. *Оператор присваивания* задает значения переменных в ходе выполнения программы и имеет вид:

```
ИмяПеременной := выражение;
```

Знак `:=` читается как "присвоить значение". Частным случаем выражения, стоящего в правой части, являются переменные и константы.

Например:

```
sort:=1; cena:=12.34; x:=x+1; y:=x;
result:=sin(a)+cos(b); name:='модель1';
```

Оператор присваивания можно считать основным оператором языка Pascal, т. к. именно в нем выполняются практически все действия по обработке данных.

Следует знать:

- в зависимости от типа результата различают три вида выражений: арифметические, логические и символьные:
  - *арифметические выражения* служат для обработки числовых данных и в результате их вычисления получается число (пример: `x+5`, где `x` — переменная числового типа);
  - *логические выражения* служат для сравнения различных данных и для других логических действий (пример: `x>5` — результат такого выражения имеет логический тип и принимает значения `true` или `false`);
  - *символьные выражения* нужны для обработки текстов (пример: `'x'+'y'`, результат — строка `'xy'`);
- результат, полученный при вычислении выражения, находящегося в правой части, должен быть *совместим по типу* с переменной, которой он присваивается. При нарушении соответствия выводится сообщение об ошибке **Type mismatch** (Несоответствие типов).

В языке Pascal нельзя с помощью одного оператора присваивания присвоить нескольким переменным одно и то же значение.

Например, нельзя записать: `i:=j:=k:=0;`. Необходимо использовать три оператора: `i:=0;`, `j:=0;`, `k:=0;`.

Оператор присваивания и процедуры ввода — основные способы задания и изменения значений переменных. Однако в языке Pascal есть еще две дополнительные процедуры, которые позволяют изменять значения целочисленных переменных:

- процедура `dec(x,n)` уменьшает значение целочисленной переменной `x` на `n`. Например, `x:=10; dec(x,2);{результат: 8};`. При отсутствии необязательного параметра `n` процедура принимает вид `dec(x)`, а значение `x` уменьшается на единицу;
- процедура `inc(x,n)` увеличивает значение целочисленной переменной `x` на `n`. Например, `x:=10; inc(x,3);{результат: 13}`. При отсутствии необязательного параметра `n` процедура принимает вид `inc(x)`, а значение `x` увеличивается на единицу.

Действие этих процедур полностью аналогично действию оператора присваивания, например, `inc(x);` можно заменить оператором `x:=x+1;`.

7.4. Арифметические выражения

Рассмотрим подробно арифметические выражения, т. к. именно с их помощью выполняются все вычисления в программе. О логических выражениях будет рассказано в *гл. 8*, о символьных — в *гл. 14*.

Результатом *арифметического выражения* является целое или вещественное число. *Выражение* задает порядок действий над элементами данных и состоит из:

- операндов (констант, переменных, функций);
- круглых скобок;
- знаков операций.

7.4.1. Арифметические операции

Операции определяют действия, которые надо выполнить над операндами. В отличие от традиционной математической записи необходимо указывать все знаки операций.

Например, в выражении `5*(2*x+3*y)` все символы операции умножения — `*` (звездочка) должны присутствовать в явном виде.

Все операции делятся на унарные и бинарные. *Унарные* операции относятся к одному операнду, *бинарные* — связывают два.

Например, `-a` — унарная операция, `a+b` — бинарная. При использовании двух знаков операций нежелательно, чтобы они стояли рядом: `a*-b`. Лучше заключить второй операнд в скобки: `a*(-b)`. Хотя результат от этого не изменится, поскольку приоритет унарного минуса выше умножения (см. табл. 7.2).

В табл. 7.1 приведены сведения об основных арифметических операциях. Для упрощения использованы только два основных числовых типа: `integer` и `real`.

Таблица 7.1. *Основные арифметические операции, типы операндов и результата*

Операция	Знак	Тип	
		операндов	результата
Бинарные операции			
Сложение	+	Real	Real
		Integer	Integer
Вычитание	-	Real	Real
		Integer	Integer
Умножение	*	Real	Real
		Integer	Integer
Деление	/	Integer	Real
		Real	Real
Целочисленное деление	Div	Integer	Integer
Остаток от деления	Mod	Integer	Integer
Арифметическое И	And	Integer	Integer
Побитовый сдвиг влево	Shl	Integer	Integer
Побитовый сдвиг вправо	Shr	Integer	Integer
Арифметическое ИЛИ	Or	Integer	Integer
Арифметическое побитовое сложение по модулю 2	Xor	Integer	Integer
Унарные операции			
Сохранение знака	+	Real	Real
		Integer	Integer

Таблица 7.1 *(окончание)*

Операция	Знак	Тип	
		операндов	результата
Унарные операции			
Отрицание знака	-	Real	Real
		Integer	Integer
Арифметическое отрицание	Not	Integer	Integer

Замечание

Операция + используется также для работы со строками *(см. гл. 14)*. Операции +, –, * используются для работы с множествами *(см. гл. 15)*.

Обратите внимание — в языке Pascal результат операции деления всегда имеет вещественный тип, даже если оба операнда целые. Однако для выполнения целочисленного деления имеются две специальные операции.

Операции *div* и *mod*

Целочисленное деление `div` (от англ. *division* — деление) отличается от обычной операции деления тем, что возвращает целую часть частного, а дробная часть отбрасывается — `13 div 3 = 4`, а не 4, (3).

Например:

```
14 div 5 = 2
1 div 3 = 0
17 div -5 = -3
-17 div 5 = -3
-17 div -5 = 3
```

Взятие остатка от деления `mod` (от англ. *modulus* — мера) вычисляет остаток, полученный при выполнении целочисленного деления.

Например:

```
10 mod 5 = 0
11 mod 5 = 1
17 mod -5 = 2
-17 mod 5 = -2
-17 mod -5 = -2
```

Операнды операций `div` и `mod` — целые числа. Взаимосвязь между операциями `div` и `mod` проста. Для `a>0` и `b>0` справедливо:

```
a mod b = a - (a div b) * b
(a div b) * b + (a mod b) = a
```

Следует знать — операцию `mod` можно использовать для того, чтобы узнать, кратно ли целое `a` целому `b`. А именно, `a` кратно `b` тогда и только тогда, когда `a mod b = 0` (см. листинг 8.6).

7.4.2. Побитовые операции

В языке Pascal можно выполнить ряд побитовых арифметических операций над целыми операндами (в побитовых операциях каждый бит обрабатывается по отдельности). Операнды могут быть представлены в десятичной или шестнадцатеричной форме, т. к. действия все равно выполняются над их двоичным внутренним представлением. Все побитовые операции выполняются очень быстро, поэтому опытные программисты используют их в тех случаях, когда высокая скорость выполнения программы имеет решающее значение.

Арифметическое И (`and`) производит логическое побитовое умножение операндов в соответствии с правилом *(таблицей истинности)*:

- `1 and 1 = 1`
- `1 and 0 = 0`
- `0 and 1 = 0`
- `0 and 0 = 0`

Например, вычислим результат выражения `a and b`, если `a=12` и `b=22`. Пусть `a` и `b` являются двухбайтовыми целыми числами. Их двоичное представление:

```
0000000000001100   { первый операнд (двоичная форма числа 12) }
0000000000010110   { второй операнд (двоичная форма числа 22) }
0000000000000100   { результат (двоичная форма числа 4) }
```

Логическое сложение (`or`) производит побитовое сложение операндов в соответствии с правилом *(таблицей истинности)*:

- `1 or 1 = 1`
- `1 or 0 = 1`
- `0 or 1 = 1`
- `0 or 0 = 0`

Например, вычислим результат выражения `12 or 22`:

```
0000000000001100   { первый операнд (двоичная форма числа 12) }
0000000000010110   { второй операнд (двоичная форма числа 22) }
0000000000011110   { результат (двоичная форма числа 30) }
```

Арифметическое побитовое сложение по модулю 2, или исключающая дизъюнкция, исключающее ИЛИ (xor) производит побитовое сложение операндов в соответствии с правилом *(таблицей истинности)*:

- `1 xor 1 = 0`
- `1 xor 0 = 1`
- `0 xor 1 = 1`
- `0 xor 0 = 0`

Например, вычислим результат выражения `12 xor 22`:

```
0000000000001100   { первый операнд (двоичная форма числа 12) }
0000000000010110   { второй операнд (двоичная форма числа 22) }
0000000000011010   { результат (двоичная форма числа 26) }
```

Сдвиг влево (`k shl n`) восстанавливает в качестве результата значение, полученное путем сдвига на `n` позиций влево (в сторону старших разрядов) двоичной формы числа `k`. Дает тот же результат, что и умножение на 2^n.

Например, вычислим результат выражения `2 shl 7`. Для этого необходимо сдвинуть каждый бит двоичной записи числа 2 на 7 позиций влево и перевести результат в десятичную форму:

```
0000000000000010 { двоичная форма числа 2 }
0000000100000000 { результат сдвига влево (двоичная форма числа 256)}
```

Или в десятичной форме: `2 shl 7 = 256 = 2*2`^{7}.

Сдвиг вправо (shr) выполняется аналогично, но в противоположном направлении (в сторону младших разрядов двоичной формы). Результат как при делении на 2^n.

Например, вычислим результат выражения `160 shr 2`:

```
0000000010100000 { двоичная форма числа 160 }
0000000000101000 { результат сдвига вправо(двоичная форма числа 40)}
```

Или в десятичной форме: `160 shr 2 = 40 = 160 div 2`^{2}.

Обратите внимание — операции сдвига выполняются намного быстрее, чем соответствующие операции деления и умножения на степень двойки.

Замечание

Биты, уходящие при сдвиге за пределы разрядной сетки компьютера, теряются, а освободившиеся двоичные разряды заполняются нулями. Применяя операции сдвига к целым со знаком, нужно иметь в виду, что старшим битом является бит знака. Правый сдвиг присваивает ему нулевое значение, что означает знак +.

Арифметическое отрицание (not) вызывает побитовую *инверсию* — получение обратного значения. Например, `not 0 = -1`.

Пример использования побитовой операции `XOR` содержится в листинге 7.6.

Кроме того, в Pascal определена операция получения указателя (адреса) `@`. Подробнее об указателях будет рассказано ниже *(см. гл. 18)*.

7.4.3. Приоритет операций

Последовательность выполнения операций в составе выражения происходит с учетом их *приоритета (старшинства)*. В табл. 7.2 приведен порядок выполнения всех основных операций.

Таблица 7.2. *Порядок выполнения основных операций*

Операция	Приоритет	Вид операции
Унарный минус, `not`, `@`	Первый (высший)	Унарная операция
`*`, `/`, `div`, `mod`, `and`, `shl`, `shr`	Второй	Операции типа умножения
`+`, –, `or`, `xor`	Третий	Операции типа сложения
`=`, `< >`, `<`, `>`, `<=`, `>=`, `in`	Четвертый	Операции отношения

Замечание

Операция `in` (включено в) используется для данных типа множеств `set` *(см. гл. 15)*.

Обратите внимание — операции с равным приоритетом выполняются слева направо. Выражение, заключенное в скобки, перед выполнением вычисляется как отдельный операнд. При наличии вложенных скобок вычисления выполняются, начиная с самых внутренних. В тексте программы необходимо проверять парность расстановки скобок: число открывающих скобок должно быть равно числу закрывающих скобок.

Например:

- в выражении `3+4*5` умножение `*` выполняется перед сложением `+`, потому что его приоритет выше. Для изменения порядка используйте скобки: `(3+4)*5`.
- выражения `a*b/c` и `(a*b)/c` вычисляются одинаково, т. к. приоритеты умножения и деления равны, а знак операции умножения стоит левее знака деления. Для изменения порядка действий необходимы скобки: `a*(b/c)`.

7.4.4. Стандартные арифметические функции

В арифметических выражениях часто используются стандартные функции, приведенные в табл. 7.3.

Таблица 7.3. *Некоторые стандартные функции, типы значений аргумента и результата*

Стандартная функция	Выполняемое действие	Тип	
		аргумента	результата
abs(x)	| x |	real	Real
		integer	integer
sqr(x)	X^2	real	Real
		integer	integer
sqrt(x)	$X^{1/2}$	real	Real
		integer	Real
exp(x)	e^x	real	Real
		integer	Real
Ln(x)	Ln(x)	real	Real
		integer	Real
Pi	число пи	—	Real
sin(x)	Sin(x)	real	Real
		integer	Real
cos(x)	Cos(x)	real	Real
		integer	Real
arctan(x)	Arctg(x)	real	Real
		integer	Real

Вызов стандартной функции осуществляется путем указания в нужном месте программы имени функции (abs, ln, exp и др.) и ее аргумента, заключенного в круглые скобки. После вычисления значения функции ее вызов заменяется результатом и расчет содержащего ее выражения продолжается дальше.

Следует знать:

- аргумент прямых тригонометрических функций sin и cos задается в радианах. Для преобразования значения угла из радианной меры в градусную необходимо умножить величину угла на число 180/pi. Для перевода

значения угла из градусной меры в радианную необходимо умножить величину угла на число `pi/180` (см. листинг 7.2);

- результат функции `arctan` получается в радианах.

Помимо приведенных в табл. 7.3 функций, в арифметических выражениях также используются следующие стандартные функции:

- функция `random(диапазон)` возвращает случайное число `x`, удовлетворяющее условию `0<=x<диапазон`. Тип аргумента и результата — `word`. В том случае, если нам необходимы целые случайные числа из диапазона `a<=x<b`, мы можем получить их, используя выражение `random(b-a)+a`. Если параметр `диапазон` не указан, то `random` возвращает число `x` в диапазоне `0<=x<1`. Тип результата — `real`. В том случае, если нам необходимы вещественные случайные числа из другого диапазона: `a<=x<b`, мы можем задать его при помощи `random*b+a`. Перед первым обращением к функции `random` необходимо с помощью вызова процедуры `randomize` инициализировать программный генератор случайных чисел. В противном случае при каждом запуске программы датчик будет выдавать одни и те же числа. Эту особенность можно использовать при отладке программы;
- функция `frac(x)` вычисляет дробную часть `x`. Аргумент и результат — `real`. Например, `write(frac(0.25*11):4:2); {результат 0.75}`;
- функция `int(x)` вычисляет целую часть `x`. Аргумент и результат — `real`. Например, `write(int(422.117):4:2); {результат 422.00}`; Таким образом, `x=int(x)+frac(x)`.

Для работы с данными порядковых типов в языке Pascal имеются три дополнительные функции — `ord`, `pred`, `succ`:

- `ord(s)` — функция возвращает *порядковый номер* значения `s` в списке значений, определенном типом `s`. Результат имеет тип `longint`. Для целых типов функция возвращает само значение `s`, для символьного типа — код символа `s`, для остальных порядковых типов — порядковый номер `s`;
- `pred(s)` — функция возвращает элемент, *предшествующий* `s` в списке значений типа. Тип результата совпадает с типом `s`. Если предшествующего `s` элемента не существует, возникает программное прерывание;
- `succ(s)` — функция возвращает значение, *следующее за* `s` в списке значений типа. Тип результата совпадает с типом `s`.

Еще три дополнительных функции используются для выполнения *операций над байтами целочисленного операнда*. Их результат также имеет целочисленный тип.

- Функция `hi(i)` выделяет старший байт значения `i` и помещает его в младший байт результата. Старший байт результата равен нулю. Например, `x:=hi($1020); { после выполнения x=$0010 }`.

- Функция `lo(i)` выделяет младший байт значения `i` и помещает его в младший байт результата. Старший байт результата равен нулю. Например, `x:=hi($7893); { после выполнения x=$0093 }`.
- Функция `swap(i)` обменивает содержимое младшего и старшего байтов целочисленного выражения `i`. Например, `x:=swap($7893); { после выполнения x=$9378 }`.

Наконец, еще одна очень полезная функция — `sizeof(it)` — вычисляет размер памяти, которую занимает указанная переменная или тип `it`. Результат — целое. Например, `sizeof(x)` — количество байт, которое занимает переменная `x`; `sizeof(integer)` — количество байт, которое отводится под любую переменную типа `integer`. Пример использования этой функции содержится в листинге 7.7.

7.4.5. Типы в арифметических выражениях. Автоматическое преобразование типов

Если в абстрактной математике знак арифметической операции всегда обозначает одно и то же действие, однозначно определяющее результат при конкретных значениях операндов, то в программировании результат операции зачастую зависит от *типа исходных данных* (см. табл. 7.1). Одни и те же операции выполняются по-разному над целыми и действительными числами, т. к. эти числа имеют принципиально различное внутреннее представление (с фиксированной и плавающей точкой). Об этом подробно рассказывалось в *разд. 4.3.2.*

Языки программирования высокого уровня разрешают использование в одном выражении операндов различных целых и вещественных типов. Например, `5+0.5` — правильное выражение, хотя оно содержит операнды целого и вещественного типов. Говорят, что вещественные и целые типы являются *совместимыми.*

Для вычисления выражения, содержащего данные разных типов, транслятор порождает дополнительные машинные команды *преобразования типов* в соответствии с правилом:

данные преобразуются к типу с более широким диапазоном значений.

Например, если складываются переменные типа `integer` и типа `real`, то перед выполнением сложения переменная целого типа будет преобразована к действительному типу. Это действие называется *автоматическим или неявным преобразованием типа.*

Замечание

Таким образом, если *хотя бы один* из операндов арифметического выражения является *вещественным числом или в выражении есть хотя бы одна операция деления*, то результат всегда *вещественное число.*

То же самое правило действует и при *присваивании* значения переменной:

- если переменная имеет тип с более широким диапазоном значений, чем тип присваиваемого значения, то произойдет *автоматическое преобразование* типа;
- если переменная имеет более узкий диапазон значений, то компилятор выдаст сообщение об ошибке **Type mismatch** (Несоответствие типов).

Например:

```
var x: integer; y:real;
begin
x:=5; y:=0.5; y:=y+x; { пока все правильно }
x:=y; { будет выдано сообщение об ошибке, т. к. в правой }
      { части — действительное число, и его не разместить в }
      { ячейке из двух байтов, которая была определена для }
      { хранения значения целочисленной переменной x }
```

В приведенном примере во избежание ошибки необходимо выполнить *явное преобразование* значения переменной `y` к целому типу. Это можно сделать двумя способами: либо отсечением дробной части действительного числа, либо округлением его до ближайшего целого.

Функции *trunc* и *round* выполняют следующие преобразования типов:

- функция `trunc(x)` возвращает ближайшее целое число, меньшее или равное вещественному `x` для `x>=0` и большее или равное `x` для `x<=0` (от англ. *truncate* — усекать). Таким образом, выполняется *отбрасывание* десятичных знаков после точки. Аргумент — `real`, результат — `longint`.

  Например, `trunc(6.7);{результат:6}, trunc(-1.6); {результат:-1}`
- функция `round(x)` возвращает значение `x`, *округленное* до ближайшего целого числа (от англ. *round* — круглый). Аргумент — `real`, результат — `longint`.

  Например, `round(6.7); {результат:7}, round(-1.6); {результат:-2}`.

  В приведенном выше примере вместо `x:=y;` следовало написать: `x:=trunc(y);` или `x:=round(y);`.

7.4.6. Явное преобразование типов. Переполнение

Использование функций `trunc` и `round`, у которых аргумент принадлежит одному, а результат другому типу, называется *явным преобразованием типа*. В Pascal может использоваться и более общий механизм явного преобразования типов — *автоопределенное преобразование типа*. При автоопределен-

ном преобразовании типов явно указывается тип, к которому надо выполнить преобразование, например, `b:=integer(a);`.

Рассмотрим следующую программу:

```
const a: integer=32767;   { максимально возможное значение типа integer }
var x,y: real;
begin
   x:=a+2;                { переполнение с потерей результата }
   y:=longint(a)+2;       { автоопределенное преобразование типа }
   writeln(x:8:0, y:8:0);
end.
```

При выполнении в среде Turbo Pascal программа выведет:

```
-32767   32769
```

Как видим, значение переменной `x` неправильное, поскольку при выполнении сложения произошло *переполнение*. Результат `32769` вышел за пределы типа `integer`. При вычислении переменной `y` было выполнено явное преобразование переменной `a` к типу `longint`, а константа `2` была автоматически преобразована к этому типу, поэтому при сложении переполнения не произошло.

Замечание

При выполнении той же программы в среде Delphi переполнения не произойдет. Вообще при использовании 32-разрядных компиляторов данный пример неактуален, поскольку такие компиляторы выделяют по четыре байта и под тип `integer`, и под тип `longint`. Тем не менее, с появлением современных компиляторов опасность переполнения не исчезла, хотя и несколько уменьшилась за счет существенного расширения диапазона типа `integer`.

Директивы проверки *{$Q+}* и *{$R+}*

При переполнении для переменных вещественных типов выдается сообщение об ошибке **Float point overflow** (Переполнение с плавающей точкой). Для переменных целых типов программист имеет возможность включать и выключать процедуру проверки переполнения, используя *директивы компилятора*.

Например, рассмотрим следующую программу:

```
var a,b: longint; c: real;
begin
   a:=400000; b:=400000; c:=a*b;
   writeln(a,' * ', b,' = ',c:10:2);
end.
```

При выполнении программа выведет строку:

```
400000 * 400000 = 1086210048.00
```

Причина неверного результата состоит в том, что при умножении результат вышел за пределы типа `longint`, несмотря на то, что в левой части находится переменная типа `real`, которая могла бы вместить полученный результат.

Для *включения проверки переполнения* при выполнении арифметических операций необходимо использовать директиву компилятора `{$Q+}`. В этом случае при выполнении программы из нашего примера на операторе `c:=a*b;` будет выдано сообщение об ошибке: **Arithmetic overflow** (Ошибка арифметического переполнения). В Delphi — сообщение **Integer overflow**.

Для *включения проверки выхода порядковой переменной за границы типа* (см. табл. 5.1—5.4) необходимо использовать директиву компилятора `{$R+}` .

Например, рассмотрим следующую программу:

```
var
   k: integer;
   i: byte; l: 224..265; j: 24..65;
begin
   k:=32000;   { в двоичной форме - 111110100000000 }
   l:=k;   i:=k;   j:=k;
   writeln(' l = ',l,' i = ',i,' j = ',j);
end.
```

При выполнении программа выведет строку:

```
l = 32000 i=0 j = 0
```

Здесь переменные `i` и `j` оказались равны нулю потому, что для их хранения отводится только один байт, а у числа `32000` в младшем байте его представления как раз находится `0`. Если же в примере использовать директиву `{$R+}`, то при выполнении на операторе `i:=k;` будет выдано сообщение об ошибке: **Range check error** (Ошибка диапазона).

Обратите внимание — использование директивы `{$R+}`, к сожалению, не всегда позволяет обнаружить переполнение при работе с типами `integer` и `longint`. *Видимым признаком переполнения* является появление *отрицательных значений* в ситуациях, где их не должно быть (переполнение портит знаковый разряд).

Директивы `{$R+}` и `{$Q+}`обычно помещают в самое начало программы, хотя в принципе их можно поставить в любое место программы, где разрешен разделитель.

Замечание

Директивы проверки по умолчанию выключены — `{$R-Q-}`. Включение проверки увеличивает размер кода и время выполнения программы. Поэтому в процессе отладки проверку необходимо использовать для обнаружения ошибок в программе, а перед созданием исполняемого exe-файла отлаженной программы ее можно отключить.

7.5. Полезные формулы

Возведение в степень

Вычисление степени числа выполняется в Pascal с использованием свойств логарифмов:

$$c = a^b \Rightarrow \ln(c) = b\ln(a) \Rightarrow c = e^{b\ln(a)} \Leftrightarrow \exp(b\ln(a)).$$

Таким способом нельзя возвести в степень отрицательное число. Можно возвести в степень модуль этого числа, а знак обработать отдельно.

Логарифм с произвольным основанием

Для вычисления логарифма с основанием a используем:

$$\log_a(x) = \ln(x) / \ln(a)$$

Тригонометрические функции

В Pascal определены только три тригонометрические функции: `sin`, `cos`, `arctg` (табл. 7.3). Для вычисления остальных тригонометрических функций необходимо использовать известные соотношения:

- $\mathrm{tg}(x) = \sin(x)/\cos(x)$;
- $\mathrm{ctg}(x) = \cos(x)/\sin(x)$;
- $\mathrm{csc}(x) = 1/\sin(x)$;
- $\arcsin(x) = \mathrm{arctg}\left(\dfrac{x}{\sqrt{1-x^2}}\right)$;
- $\arccos(x) = \mathrm{arctg}\left(\dfrac{\sqrt{1-x^2}}{x}\right) = \dfrac{\pi}{2} - \arcsin(x)$;
- $\mathrm{arcctg}(x) = \dfrac{\pi}{2} - \mathrm{arctg}(x)$.

7.6. Примеры программ

7.6.1. Вычисления по формулам

Листинг 7.2 содержит программу вычисления площади треугольника по двум сторонам и углу между ними. Угол вводится в градусах и переводится в радианы.

Листинг 7.2. Вычисление площади треугольника

```
var
   a,b: real;   { длины сторон }
   angle: real; { величина угла в градусах }
   area: real;  { площадь треугольника }
begin
   writeln('Вычисление площади треугольника');
   write('Введите длины двух сторон треугольника в одной строке (см.):');
   readln(a,b);
   write('Введите угол между сторонами в градусах: ');
   readln(angle);
   { переводим угол в радианы }
   angle:=angle*pi/180;
   { area=a*h/2, где h (высота треугольника) может быть }
   { вычислена по формуле: h=b*sin(angle) }
   area:=a*b*sin(angle)/2;
   writeln('Площадь треугольника:', area:7:3,' кв.см.');
   readln;
end.
```

7.6.2. Выделение цифр целого числа

Листинг 7.3 содержит программу, вычисляющую сумму цифр трехзначного числа. Программа иллюстрирует применение функций `div` и `mod`.

Листинг 7.3. Вычисление суммы цифр трехзначного числа

```
var i,first,second,third,sum: integer;
begin
   write('Введите целое трехзначное число: ');readln(i);
   first :=i div 100;      { выделение первой цифры числа }
   second:=i div 10 mod 10;{ выделение второй цифры числа }
   third :=i mod 10;       { выделение третьей цифры числа }
   sum:=first+second+third;
   writeln('Сумма цифр числа ',i,' = ',sum);
end.
```

7.6.3. Перестановка значений переменных

Под перестановкой двух переменных понимают обмен значениями этих переменных. Допустим, до перестановки было: a=1, b=2, а после перестановки будет: a=2, b=1. Поставим себе задачей реализовать эту перестановку средствами языка Pascal. (Пока поверьте на слово, что такие перестановки занимают весьма значительную часть компьютерного времени в системах хранения и поиска информации.)

Для решения задачи введем дополнительную *(буферную)* переменную (что-то типа камеры хранения, куда на время помещается одна из переменных). Программа представлена в листинге 7.4. Для определенности пусть переменные имеют тип integer.

Листинг 7.4. Перестановка двух переменных (вариант 1)

```
var a,b,c:integer;
begin
   write('Введите a и b: '); readln(a,b);
   c:=a; {поместили значение a в буферную переменную c}
   a:=b; {старое значение переменной a заменилось на новое}
   b:=c; {в переменную b помещаем старое значение a из переменной c}
   writeln('a=',a,' b=',b); readln;
end.
```

Если хорошо подумать, то можно предложить вариант перестановки без использования третьей переменной.

Листинг 7.5. Перестановка двух переменных (вариант 2)

```
var a,b:integer;
begin
   write('Введите a и b: '); readln(a,b);
   a:=a+b; {в a теперь сумма  a и b}
   b:=a-b; {из суммы вычли b, получилось a}
   a:=a-b; {из суммы вычли a, получилось b}
   writeln('a=',a,' b=',b); readln;
end.
```

Вторая программа позволяет сэкономить одну переменную, но при этом возможна неприятная ситуация в случае, если переменные принимают очень большие значения (при суммировании возможно переполнение). Никакого

переполнения не произойдет, если заменить операцию арифметического сложения на побитовую операцию XOR (исключающее ИЛИ) *(см. разд. 7.4.2)*.

Листинг 7.6. Перестановка двух переменных (вариант 3)

```
var a,b:integer;
begin
   write('Введите a и b: '); readln(a,b);
   a:=a xor b; {поразрядное исключающее ИЛИ,}
   b:=a xor b; {если его выполнить второй раз,}
   a:=a xor b; {то снова получим первоначальное значение}
   {можете в этом убедиться}
   writeln('a=',a,' b=',b); readln;
end.
```

В действительности при обработке данных применяется почти исключительно первый способ, два других — это всего лишь упражнения для развития логического мышления. Экономить на нескольких байтах памяти — все равно что экономить на спичках.

7.6.4. Определение размера ячейки памяти

Следующий листинг представляет совсем короткую программу, которая позволит программисту быстро сориентироваться в новой системе программирования. Как известно, различные компиляторы могут по-разному выделять память для данных, так вот, приведенный ниже листинг 7.7 позволит узнать размер ячейки памяти для основных типов данных Pascal.

Листинг 7.7. Вывод размера ячейки памяти для различных типов данных

```
begin
writeln('integer: ',sizeof(integer),'; real: ',sizeof(real),'; char:
',sizeof(char)); readln;
end.
```

При выполнении в среде Turbo Pascal программа выведет строку:

```
integer: 2; real: 6; char: 1
```

В среде Delphi:

```
integer: 4; real: 8; char: 1
```

7.7. Контрольные вопросы и задания

Вопросы

1. Что такое ввод и вывод данных?
2. Почему инструкция `readln` при вводе с клавиатуры предпочтительнее `read`?
3. Почему при завершении программы рекомендуется использовать инструкцию `readln;` (без переменных)?
4. Что произойдет, если при вводе числовых данных будет допущена ошибка: вместо числа пользователь программы случайно введет другой символ, например, букву?
5. В чем отличие процедуры вывода `writeln` от `write`?
6. Зачем используется формат вывода?
7. Какие типы выражений, используемых в операторе присваивания, вы знаете?
8. В каких случаях при выполнении присваивания возникает несоответствие типов?
9. В чем состоит особенность выполнения операции деления?
10. Чем отличаются друг от друга операции `div` и `mod`? Приведите примеры.
11. Как определяется порядок выполнения операций в составе выражения?
12. Как выполняется вызов стандартной арифметической функции?
13. В чем состоит особенность использования типов в арифметических выражениях?
14. Зачем используются функции преобразования типов `trunc` и `round`?
15. Чем отличаются явное и автоматическое преобразования типа?
16. Назовите видимый признак арифметического переполнения. Расскажите об использовании директив компилятора `{$Q+}` и `{$R+}`.

Задания

Составить программу для вычисления по формулам. Предусмотреть задание исходных данных при помощи инструкции ввода.

1. Треугольник задан длинами сторон a, b и величиной угла C. Найти сторону c и площадь S данного треугольника:

$$c = \sqrt{a^2 + b^2 - 2ab\cos(C)}\,; \quad S = \sqrt{p(p-a)(p-b)(p-c)}\,,$$

где $p = \dfrac{a+b+c}{2}$.

2. Предусмотреть ввод угла α в градусах и перевод его в радианы:

$$\alpha_{рад} = \alpha_{гр} \cdot \frac{\pi}{180}.$$

3. Треугольник задан длинами сторон. Найти длины высот:

$$h_a = 2S/a; \qquad h_b = 2S/b; \qquad h_c = 2S/c.$$

4. Треугольник задан длинами сторон. Найти длины медиан:

$$m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}; \qquad m_b = \frac{1}{2}\sqrt{2a^2 + 2c^2 - b^2};$$

$$m_c = \frac{1}{2}\sqrt{2a^2 + 2b^2 - c^2}.$$

5. Треугольник задан длинами сторон. Найти длины биссектрис:

$$\beta = \frac{2}{b+c}\sqrt{bcp(p-a)}; \quad \beta = \frac{2}{a+c}\sqrt{acp(p-b)};$$

$$\beta = \frac{2}{a+b}\sqrt{abp(p-c)}.$$

6. Треугольник задан длинами сторон. Найти углы треугольника:

$$\hat{A} = 2\operatorname{arctg}\sqrt{\frac{(p-b)(p-c)}{p(p-a)}}; \qquad \hat{B} = 2\operatorname{arctg}\sqrt{\frac{(p-a)(p-c)}{p(p-b)}};$$

$$\hat{C} = 2\operatorname{arctg}\sqrt{\frac{(p-a)(p-b)}{p(p-c)}}.$$

7. Даны два угла треугольника A, B и высота h_a. Вычислить площадь треугольника и две стороны по формулам:

$$\hat{C} = \pi - \hat{A} - \hat{B}; \quad S = \frac{h_a{}^2 \sin(A)}{2\sin(B)\sin(C)}; \qquad b = \frac{h_a}{\sin(C)}; \qquad a = \frac{2S}{h_a}.$$

8. Даны два угла треугольника B, C и высота h_a. Вычислить длины всех сторон по формулам:

$$a = h_a\left(\frac{1}{\operatorname{tg}(B)} + \frac{1}{\operatorname{tg}(C)}\right); b = \frac{h_a}{\sin(C)}; c = \sqrt{a^2 + b^2 - 2ab\cos(C)}.$$

9. В треугольнике заданы сторона a, углы B и C. Найти площадь S, стороны b и c:

$$\hat{A} = \pi - \hat{B} - \hat{C}; b = \frac{a\sin(B)}{\sin(A)}; C = \frac{a\sin(C)}{\sin(A)}; S = \frac{a^2\sin(B)\sin(C)}{2\sin(A)}.$$

10. В треугольнике заданы сторона a, высота h_a и угол C. Найти стороны b, c и площадь S:

$$b = \frac{h_a}{\sin(C)}; \qquad c = \sqrt{a^2 + b^2 - 2ab\cos(C)}; \qquad S = \frac{h_a a}{2}.$$

11. Треугольник задан координатами своих вершин. Найти площадь треугольника:

$$\hat{B} = 2\operatorname{arctg}\sqrt{\frac{(p-a)(p-c)}{p(p-b)}}; \qquad S = \frac{ac\sin(B)}{2}.$$

Расстояние между точками $A(x_1, y_1)$ и $B(x_2, y_2)$, расположенными на плоскости, рассчитывается как:

$$|AB| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

12. Треугольник задан координатами своих вершин. Найти периметр треугольника.

13. Вокруг треугольника со сторонами a, b и c описана окружность. Определить ее радиус R, угол треугольника A и площадь, ограниченную стороной a и радиусами, проведенными в вершины B, C.

$$A = \arccos\frac{b^2 + c^2 - a^2}{2bc}; \qquad R = \frac{a}{2\sin(A)}; \qquad S = \frac{R^2\sin(2A)}{2}.$$

14. В окружность радиуса R вписан правильный треугольник. Определить длину его стороны a, площадь S и площадь круга S_0, вписанного в треугольник:

$$a = R\sqrt{3}; \qquad S = \frac{3R^2\sqrt{3}}{4}; \qquad S_0 = \frac{\pi R^2}{4}.$$

15. Определить дальность полета тела, запущенного с начальной скоростью V_0 с высоты h под углом α к горизонту, пренебрегая сопротивлением воздуха. Траектория движения тела описывается уравнениями:

$$Y = -gt^2 / 2 + V_{0y}t + h; x = V_{0y}t,$$

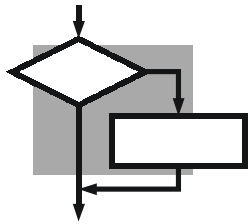
где $g = 9{,}81 м/с^2$, а V_{0x}, V_{0y} — компоненты скорости вдоль осей X и Y.

Замечание: в момент падения тела $Y = 0$.

16. Составьте программу вычисления напряжения на каждом из последовательно соединенных участков электрической цепи сопротивлением $R1$, $R2$, $R3$ Ом, если сила тока при напряжении U вольт составляет I ампер.

17. Составьте программу вычисления значения силы тока I на участке, состоящем из двух параллельно соединенных резисторов сопротивлением $R1$ и $R2$, если напряжение на концах этого участка равно U.

Глава 8



Программирование разветвлений

Далеко не все программы являются линейными. Очень часто встречается ситуация, когда требуется *выбрать* между двумя или более вариантами действий в зависимости от заданного условия. Такая алгоритмическая конструкция называется *ветвлением*. Другая конструкция — *обход* — позволяет *пропустить* несколько шагов алгоритма в зависимости от справедливости проверяемого условия.

8.1. Логические выражения

Проверяемое условие обычно записывается в виде *логического выражения,* состоящего из одного или нескольких *отношений* (табл. 8.1).

Таблица 8.1. *Основные операции отношения*

Операция	Название	Выражение	Результат
=	Равно	A = B	true, если A равно B
< >	Не равно	A <> B	true, если A не равно B
>	Больше	A > B	true, если A больше B
<	Меньше	A < B	true, если A меньше B
>=	Больше или равно	A >= B	true, если A больше или равно B
<=	Меньше или равно	A <= B	true, если A меньше или равно B
In	Принадлежность	A in B	true, если A находится в списке B

Операции отношения выполняют сравнение двух операндов и определяют, истинно значение отношения — `true` или ложно — `false`.

Используя отношения, при помощи знаков *логических операций* получают более сложные *логические выражения*. Результат проверки любого условия, каким бы сложным оно ни было, всегда имеет логический тип — `boolean`.

Например, в результате выполнения оператора `p:=x<5;` логическая переменная `p` получит значение `true`, если текущее значение переменной `x` меньше или равно 5, иначе — `false`.

Таблица 8.2. *Основные логические операции*

Операция	Действие	Выражение	A	B	Результат
not	Логическое отрицание	not A	True		False
			False		True
and	Логическое И (конъюнкция)	A and B	True	True	True
			True	False	False
			False	True	False
			False	False	False
or	Логическое ИЛИ (дизъюнкция)	A or B	True	True	True
			True	False	True
			False	True	True
			False	False	False
xor	Исключающее ИЛИ	A xor B	True	True	false
			True	False	true
			False	True	true
			False	False	false

Приоритет операций

Если в логическом *(булевском)* выражении не использованы круглые скобки, то операции выполняются в порядке их старшинства или убывания приоритетов (см. табл. 7.2). Операции одинакового старшинства выполняются слева направо. Для задания явного порядка используются круглые скобки.

Обратите внимание, в Pascal логические операции имеют более высокий приоритет, чем операции отношения. Поэтому каждое *простое условие в логическом выражении обязательно заключается в скобки*.

Например, для целых m, mmax, n, nmax выражение `m > mmax and n > nmmax` вызовет сообщение об ошибке, т. к. сначала будет выполняться операция `mmax and n`. Определим приоритет, расставив скобки, и получим правильное выражение: `(m > mmax) and (n > nmmax)`. Его вычисление выполняется в следующей последовательности: сначала определяется значение подвыражения `(m > mmax)`, затем `(n > nmmax)` и лишь после этого выполняется логическая операция and.

Замечание

Очевидно, что для получения требуемого результата логическое выражение в ряде случаев может быть вычислено не полностью. В данном примере значение выражения `(m > mmax) and (n > nmmax)` будет равно `false`, если `m <= mmax`, независимо от соотношения `n` и `nmmax`. Аналогично значение выражения `x or y` при `y = true` будет равно `true` независимо от значения операнда `x` (и наоборот). Для сокращения времени вычисления логических выражений используется директива компилятора {$B+} (ее надо поместить в начало программы). По умолчанию выполняется полное вычисление логических выражений. В меню интегрированной среды Turbo Pascal представлена соответствующая команда **Options | Compiler | Complete boolean evaluation**. Если опция включена, в булевском выражении будут оцениваться все условия, в противном случае оценка выражения прекращается, как только результат всего выражения становится очевидным (т. е. выполняется как можно скорее).

Необходимо знать следующие тонкости:

- в языке Pascal нельзя записать двустороннее неравенство вида `1<x<2`. Необходимо воспользоваться логическим выражением `(x>1)and(x<2)`;
- нельзя также записать `x=y=z`. Необходимо: `(x=y)and(x=z)`;
- для записи условия, заключающегося в том, что `x` не лежит в диапазоне от -2 до 2, можно использовать:

  `not((x>-2)and(x<2))` или `(x<=-2)or(x>=2)`
- условие принадлежности точки кругу единичного радиуса с центром в начале координат: `sqr(x)+sqr(y)<1`.

Логические выражения могут использоваться в инструкциях присваивания и вывода.

Например, рассмотрим следующую программу:

```
const a=1; b=2;
var flag: boolean;  { flag – логическая переменная }
begin
   flag:=a>b;       { результат сравнения присваивается переменной flag }
```

```
   writeln('a = ',a,'; b = ',b);
   writeln('a > b — ',flag,'; a < b — ',a < b);
end.
```

При выполнении программа напечатает строки:

```
a = 1; b = 2
a > b — FALSE; a < b — TRUE
```

Но, конечно, основное назначение логических выражений — организация ветвлений, когда в зависимости от значения логического выражения выполняется та или иная группа операторов. Прежде чем переходить к правилам записи ветвлений, нам остается выяснить, как можно объединить в группу несколько операторов. В языке Pascal для этой цели используется составной оператор.

8.2. Составной оператор

Составной оператор представляет собой группу из произвольного числа операторов, отделенных друг от друга точками с запятой. Группа ограничена *операторными скобками* `begin` и `end`.

```
begin
   Оператор1;
   Оператор2;
...
   ОператорN;
end;
```

Обратите внимание — составной оператор *воспринимается как один оператор* и обычно используется в том месте, где по правилам языка может стоять только один оператор, а требуется использование нескольких.

Поскольку зарезервированные слова `begin` и `end` есть не что иное, как операторные скобки, то составной оператор можно мысленно представить в следующем виде: `(Оператор1; Оператор2; ...; ОператорN)`. Вряд ли читатель поставит точку с запятой после открывающей и перед закрывающей скобками. Этого же правила следует придерживаться при записи составного оператора.

Замечание

В программе на Pascal каждому `begin` соответствует свой `end`, но не наоборот, т. к. на `end` заканчиваются конструкции, начинающиеся с `case` *(см. разд. 8.4)* и `record` *(см. гл. 16)*.

Теперь, наконец, можно перейти к реализации ветвлений. В Pascal, как и во многих других языках, для этой цели имеются два оператора — *условный оператор* и *оператор выбора*.

8.3. Условный оператор

Оператор (инструкцию) `if` можно записать двумя способами.

- Вариант 1:

```
if Условие
   then
      begin
         { Эти инструкции выполняются, }
         { если Условие истинно }
      end
   else
      begin
         { Эти инструкции выполняются, }
         { если Условие ложно }
      end;
```

- Вариант 2:

```
if Условие
   then
      begin
         { Эти инструкции выполняются, }
         { если Условие истинно }
      end;
```

Во втором случае говорят о *сокращенной форме* условного оператора (алгоритмическая конструкция *обход*).

Ключевые слова `if`, `then`, `else` обозначают "если", "то", "иначе" соответственно. *Выполнение* условного оператора начинается с вычисления условия. *Если* оно *истинно* (`true`, "да", 1), то выполняется *оператор, стоящий после* служебного слова `then`. Если условие *ложно* (`false`, "нет", 0), то выполняется *оператор, стоящий после* служебного слова `else`, а в сокращенной форме условного оператора — выполняется следующая по порядку за оператором условия инструкция.

Действие условного оператора можно пояснить с помощью блок-схем (рис. 8.1, *а*, *б*).

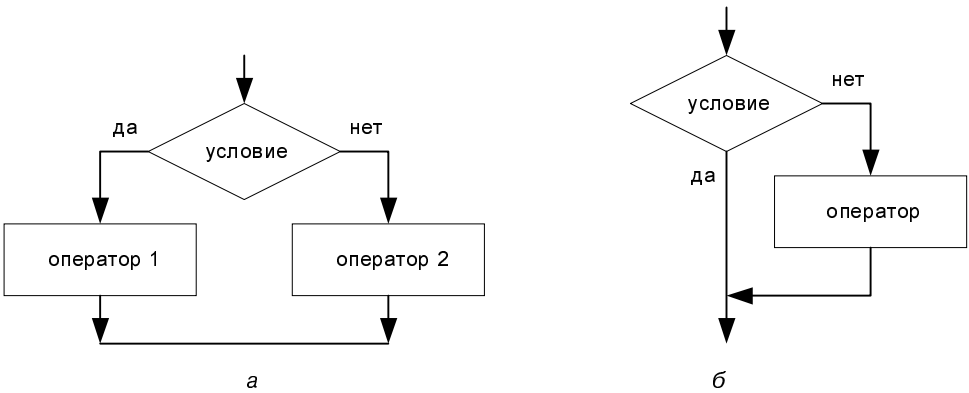


Рис. 8.1. Условные обозначения на схемах алгоритмов: *а* — ветвления и *б* — обхода

Например:

```
if a>b then writeln ('a больше b')
        else writeln ('a меньше или равно b');
```

Или:

```
if (a>=0)and(a<10) then writeln('однозначное число');
```

Один оператор `if` может входить в состав другого оператора `if`. В этом случае говорят о *вложенности операторов*.

- Вариант 1:

```
if Условие1 then
   if Условие2 then Оператор1
               else Оператор2
          else Оператор3;
```

- Вариант 2:

```
if Условие1 then Оператор1
          else if Условие2 then Оператор2
                           else Оператор3;
```

Второй вариант применяется чаще.

Например, листинг 8.1 содержит программу, которая сравнивает возраст брата и сестры и выводит соответствующее сообщение.

Листинг 8.1. Сравнение возрастов (вариант № 1)

```
var age1,age2: integer;
begin
   write('Введите возраст брата: '); readln(age1);
   write('Введите возраст сестры: ');readln(age2);
   if age1>age2 then writeln('Брат старше.')
               else if age1<age2 then writeln('Сестра старше.')
                                 else writeln('Они близнецы.');
end.
```

Запись вложенных условных операторов — непростая задача и часто вызывает затруднения у начинающих программистов. В этом случае лучше составлять программу таким образом, чтобы она не содержала вложенных условных операторов. Например, изменим текст предыдущей программы так, чтобы он содержал сокращенные условные операторы, не вложенные один в другой (листинг 8.2).

Листинг 8.2. Сравнение возрастов (вариант № 2)

```
var age1,age2: integer;
begin
   write('Введите возраст брата: '); readln(age1);
   write('Введите возраст сестры: ');readln(age2);
   if age1>age2 then writeln('Брат старше.');
   if age1<age2 then writeln('Сестра старше.');
   if age1=age2 then writeln('Они близнецы.');
end.
```

Очевидно, что первый вариант получился немного короче, а второй — проще для восприятия. Иногда второй вариант требует записи более сложных условий. Что ж, будем учиться справляться и с вложенными операторами, и со сложными условиями.

Следует знать:

- при вложенности операторов каждое `else` соответствует тому `then`, которое непосредственно ему *предшествует*;
- конструкций со степенью вложенности более 2—3 необходимо *избегать* из-за сложности их анализа при отладке программы;

- если в сложном условном операторе проверяемые Условие1, Условие2… не влияют друг на друга, т. е. последовательность их вычисления безразлична, в тексте программы их рекомендуется располагать в определенном порядке. Условие, с наибольшей вероятностью принимающее значение true, должно стоять на первом месте, с меньшей вероятностью — на втором, и т. д. Это ускорит выполнение программы.

Часто в условных операторах используют составной оператор begin end. Если между begin и end находится *только одна инструкция*, слова begin и end лучше не писать.

В пояснение рассмотрим программу, которая вычисляет частное от деления двух чисел.

Листинг 8.3. Вычисление частного двух целых чисел

```
var
   a,b: integer;    { операнды — целые числа }
   result: real;    { результат — вещественное число }
begin
   write('Введите значение делимого a: '); read(a);
   write('Введите значение делителя b: '); read(b);
   if b=0           { контроль ввода }
   { условие выполнено }
   then writeln('Неверные исходные данные: делитель — ноль')
   { условие не выполнено }
   else
{ составной оператор нужен для объединения двух команд в единое целое }
      begin         { начало составного оператора }
         result:=a/b;
         writeln('Частное чисел ',a,' и ',b,' = ', result:7:3);
      end;          { конец составного оператора }
end.
```

Обратите внимание — при использовании в программе составного оператора обычно стараются слово end располагать строго под словом begin, чтобы легче было проверять соответствие операторных скобок. Авторы, в целях экономии строк, иногда будут располагать слово begin в одной строке с else (или then), помещая при этом закрывающее end строго под (перед) else. При таком способе наглядность не теряется.

8.4. Оператор выбора

Обычно при написании программы не рекомендуется использовать многократно вложенные друг в друга условные операторы `if`, т. к. программа становится громоздкой и ее трудно понимать. Считается, что число уровней вложения не должно превышать двух-трех. Но как быть, если необходимо проверить достаточно много условий? Для этих целей в языке Pascal существует специальный оператор выбора `case`.

Инструкция `case` имеет вид:

```
case Выражение-селектор of
СписокКонстант1: begin
                    { Инструкции 1 }
                 end;
СписокКонстант2: begin
                    { Инструкции 2 }
                 end;
СписокКонстантN: begin
                    { Инструкции N }
                 end
else
   begin
   { Инструкции }
   end;
end;
```

Выполнение оператора `case` начинается с вычисления *выражения-селектора*. Инструкции, помещенные между `begin` и `end`, выполняются в том случае, если значение выражения, стоящего после слова `case`, *совпадает* с константой из соответствующего списка. Если это не так, то выполняются инструкции, следующие после `else`, расположенные между `begin` и `end`. Если `else` отсутствует, выполняется оператор программы, следующий за `case`.

Обратите внимание — в конце оператора `case` стоит ключевое слово `end`, для которого нет парного слова `begin`. Оператор `End` располагают строго под `case`.

Например:

- селектор целочисленного типа:

```
case i of
   1:   z:=i+10;
   2:   z:=i+100;
   3:   z:=i+1000;
end;
```

- селектор интервального типа:

```
case i of
   1..10:    writeln('Число ', i:4,' в диапазоне  1 – 10');
   11..20:   writeln('Число ', i:4,' в диапазоне 11 – 20');
   21..30:   writeln('Число ', i:4,' в диапазоне 21 – 30')
else('Число вне диапазона')
end;
```

Следует знать:

- инструкция `case` является обобщением оператора `if` и используется для выбора одного из *нескольких* направлений дальнейшего хода программы;
- выбор последовательности инструкций программы осуществляется во время ее выполнения в зависимости от текущего значения *выражения-селектора.*
- выражение-селектор должно иметь *порядковый тип* (чаще всего — `integer`, реже — `char`, `boolean` или один из пользовательских типов). Использование *вещественного и строкового* типа *недопустимо*;
- список констант выбора может состоять из произвольного количества значений или диапазонов, отделенных друг от друга запятыми. Границы диапазона записываются двумя константами через разграничитель "`..`". Тип констант должен совпадать с типом селектора;
- константы выбора внутри одного оператора выбора должны быть различны, в противном случае выполняется первая "подходящая" ветвь. В разных операторах выбора разрешается использовать одинаковые константы выбора;
- точку с запятой можно не ставить после последнего элемента списка выбора (как обычно перед `end`).

Проиллюстрируем сказанное на примере программы, выдающей по номеру месяца соответствующее ему время года.

Листинг 8.4. Пример использования оператора выбора

```
var number: integer;
    season:string;
begin
  write('Введите номер месяца'); readln(number);
  case number of
    12,1,2,: season:='зима';
    3..5: season:='весна';
```

```
    6..8: season:='лето';
    9..11: season:='осень';
    else
        season:='не сезон';
  end; {конец оператора Case}
  writeln(season); readln;
end.
```

8.5. Оператор безусловного перехода. Пустой оператор

Иногда в программах с разветвлениями используют еще один оператор Pascal, который называется *оператором перехода*. Этот оператор используется в том случае, если после выполнения очередного оператора надо выполнить не следующий по порядку, а другой, помеченный для этого *меткой*.. Оператор перехода имеет вид:

```
goto ИмяМетки;
```

Например:

```
label   metka01; { метка должна быть обязательно описана }
...
begin
...
metka01 : Оператор; {любой оператор, помеченный меткой }
...
       goto metka01; {оператор перехода}
end.
```

Следует знать:

- метка, на которую передается управление, должна быть описана в разделе описания меток того блока (основной программы, процедуры или функции), в котором эта метка используется. Переход возможен только в пределах блока;
- в соответствии с правилами *структурного программирования* оператор `goto` следует применять *как можно реже*, т. к. он *усложняет понимание* логики программы. В небольших по размеру программах это не очень заметно, но если сам оператор и метка, на которую выполняется переход, отстоят далеко друг от друга, разобраться в программе будет нелегко. В некоторых языках программирования этот оператор вообще отсутствует *(см. разд. 11.1)*.

Пустой оператор

Этот оператор не содержит символов и не выполняет действий. Он может быть помечен меткой, поэтому часто используется в сочетании с оператором перехода. *Например:*

```
begin
...
   goto metka; { переход в конец блока }
...
   metka:      { пустой оператор помечен меткой }
end.
```

Лишняя точка с запятой в разделе операторов, образующая пустой оператор, не создает ошибочной ситуации.

Например, запись вида:

```
x:=1;; y:=2;
```

будет интерпретирована следующим образом: сначала идет оператор присваивания `x:=1;`, затем пустой оператор (еще одна точка с запятой, зачем она поставлена — это уже другой вопрос), а вслед за ним — еще один оператор присваивания `y:=2;`.

Очевидно, что от написания лишних знаков пунктуации следует воздерживаться. В разделе описаний несколько записанных подряд точек с запятой вызовут ошибку компиляции.

Применение пустого оператора вполне уместно только в одной ситуации — если мы ставим точку с запятой после оператора, за которым следует слово `end`, то на самом деле записываем лишний пустой оператор (по правилам перед `end` точка с запятой не ставится). Такое написание не портит внешний вид программы.

Листинг 8.5 содержит программу решения квадратного уравнения с использованием сложных условий. Это единственный пример в книге, который иллюстрирует применение оператора `goto` и пустого оператора. Разумеется, при решении задачи можно было бы обойтись без них, но авторам хотелось дать пример осмысленного применения `goto`. Программа решения квадратного уравнения с использованием подпрограммы-функции (без оператора перехода) приведена в листинге 10.4.

Листинг 8.5. Решение квадратного уравнения (вариант № 1)

```
label labl01;
var
   a,b,c: real;       { коэффициенты уравнения }
```

```
   x1,x2: real;        { корни уравнения }
   d: real;            { дискриминант }
begin
   writeln('введите значения коэффициентов a, b, c'); readln(a,b,c);
   if (a=0)and(b=0)and(c=0) then begin
      writeln('корень — любое число'); goto labl01;
   end;
   if (a=0)and(b=0)and(c<>0) then begin
      writeln('корней нет'); goto labl01;
   end;
   if (a=0)and(b<>0)and(c<>0) then begin
      x1:=-c/b; writeln('единственный корень:',x1:7:3); goto labl01;
   end;
   d:=b*b-4*a*c;      { вычисление дискриминанта }
   if d>=0 then begin
      x1:=-b+sqrt(d)/(2*a); x2:=-b-sqrt(d)/(2*a);
      writeln('корни уравнения:');
      writeln('x1=',x1:7:3); writeln('x2=',x2:7:3);
   end
            else writeln('вещественных корней нет');
   labl01:            { пустой оператор помечен меткой }
end.
```

Самый простой способ убрать `goto` из данного примера — это заменить команды перехода стандартной процедурой `halt`. Процедура `halt` используется для *досрочного завершения программы*, когда это необходимо (например, в случае неверных исходных данных или в особой ситуации, как в примере с квадратным уравнением). Процедуру `halt` авторы в своей программисткой практике иногда используют, а про `goto` забыли с того момента, как перешли с таких языков, как Fortran и Basic, на язык Pascal.

Замечание

Последние версии языков Fortran и Basic также позволяют легко обходиться без `goto`.

В *гл. 11* мы рассмотрим примеры замены оператора `goto` более удачными конструкциями.

8.6. Примеры программ

8.6.1. Разные задачи

Во многих задачах нам приходится проверять число на четность или на кратность его другому числу, поэтому следует напомнить об операции mod (остаток от деления).

Листинг 8.6. Проверка числа на четность

```
var n: integer;
begin
   write('Введите целое число: '); readln(n);
   write('Число ',n,' — ');
   if n mod 2=0 then writeln('четное') else writeln('нечетное');
end.
```

Комментарий

Для проверки на нечетность можно использовать функцию odd:

```
if odd(n) then writeln('нечетное') else writeln('четное');
```

Листинг 8.7 содержит программу, которая проверяет, может ли существовать треугольник с заданными сторонами. Такую проверку нужно обязательно выполнять перед расчетом различных элементов треугольника по трем заданным сторонам. В общем виде эту мысль можно сформулировать так: перед вычислениями всегда следует проверить исходные данные на корректность.

Из геометрии известно, что сумма любых двух сторон треугольника должна быть больше третьей. Программа иллюстрирует использование сложного условия.

Листинг 8.7. Проверка существования треугольника с заданными сторонами

```
var a,b,c: real;
begin
   write('Введите три действительных числа: '); readln(a,b,c);
   if (a+b>c)and(a+c>b)and(b+c>a) then
      writeln('Треугольник с такими сторонами существует!')
   else
      writeln('Треугольник с такими сторонами не существует...');
end.
```

Следующий листинг (листинг 8.8) содержит программу, которая выбирает максимальное число из трех заданных чисел. Выбрать наибольшее из двух чисел легко, из трех же — сложнее, т. к. каждое число нужно сравнить уже с двумя оставшимися. Но не будем торопиться записывать сложные условия, а поищем более короткое решение. Сначала найдем наибольшее из двух чисел, а затем сравним результат с третьим числом. Для этих действий потребуется только одна дополнительная ячейка памяти (переменная max).

Листинг 8.8. Наибольшее из трех чисел

```
var a,b,c,max:integer;
begin
writeln('введите три числа '); readln(a,b,c);
if a>b then max:=a else max:=b;
if c>max then max:=c;
writeln('наибольшее из чисел ', max); readln
end.
```

Аналогично находится и наименьшее из чисел, достаточно только заменить знак ">" на знак "<".

8.6.2. День недели по заданной дате

Листинг 8.9 содержит программу, которая определяет день недели известной даты (по современному европейскому календарю).

Листинг 8.9. Определение дня недели по известной дате

```
var d,m,y: integer; n: longint;
begin
   writeln('Введите день, месяц, год даты (например: 3 12 1964)');
   readln(d,m,y);
   if(m>=2)then m:=m+1
           else
              begin
                 m:=m+13; y:=y-1;
              end;
   n:=trunc(365.25*y)+trunc(30.6*m)+d-621050; n:=n-trunc(n/7)*7+1;
   case n of
      1: write('понедельник');
      2: write('вторник');
```

```
      3: write('среда');
      4: write('четверг');
      5: write('пятница');
      6: write('суббота');
      7: write('воскресенье');
   end;  writeln;
end.
```

8.7. Контрольные вопросы и задания

Вопросы

1. Какая алгоритмическая конструкция называется ветвлением? Приведите примеры ее использования.
2. Приведите примеры логических выражений. Из чего они состоят?
3. К какому типу данных всегда принадлежит результат проверки логического выражения? Приведите примеры его использования в инструкциях присваивания и вывода.
4. Перечислите основные операции отношения, используемые в Pascal.
5. Перечислите основные логические операции, используемые в Pascal.
6. Как определяется порядок выполнения операций в составе логических выражений?
7. Поясните, почему каждое простое условие в логическом выражении необходимо заключать в скобки. Приведите примеры.
8. Что такое составной оператор? Каковы причины его использования? В чем состоит особенность расстановки точек с запятой при записи составного оператора?
9. Чем отличаются друг от друга ветвление и обход? Поясните с использованием блок-схем алгоритмов. Приведите примеры листингов, содержащих варианты использования инструкции `if`.
10. Что произойдет, если проверяемое условие — ложно, а часть оператора, стоящая после служебного слова `else`, отсутствует?
11. В каком случае условные операторы называют вложенными? Что такое уровень вложенности?
12. Почему возникла необходимость использования оператора выбора `case`? Можно ли обойтись вообще без оператора `case`?
13. Перечислите основные особенности использования оператора `case`.

14. Что представляет собой оператор безусловного перехода? Укажите его назначение и особенности применения. Почему рекомендуется избегать оператора `goto`?
15. Для чего используется в программе пустой оператор? Приведите примеры.

Задания

1. Составьте программу, иллюстрирующую возможность применения компьютера в книжном магазине. Компьютер запрашивает стоимость книг, сумму денег, внесенную покупателем. Если сдачи не требуется, он выводит на экран "спасибо"; если денег внесено больше, выводит "возьмите сдачу" и указывает сумму сдачи; если денег недостаточно, выводит "добавьте" и указывает размер недостающей суммы.
2. Вводятся два произвольных числа A и B. Определить, совпадают ли их знаки.
3. Вводятся два произвольных числа A и B. Определить, делится ли большее из этих чисел на меньшее без остатка или нет.
4. Вводятся три произвольных числа A, B и C. Проверить, не является ли одно из них суммой двух других.
5. На плоскости задана прямоугольная система координат с осями X и Y, которая делит плоскость на 4 квадранта. Составить программу, которая вводит координаты точки (x, y) и сообщает, в каком квадранте (четверти) или на какой оси расположена эта точка.
6. На плоскости задана прямоугольная система координат с осями X и Y. Составить программу, которая вводит координаты точки (x, y) и сообщает, принадлежит ли эта точка кругу с центром в начале координат и радиусом r (значение которого также вводится с клавиатуры) или лежит на границе его окружности. Помните, что уравнение окружности — $r^2 = x^2 + y^2$.
7. На окружности с центром в точке (x_0, y_0) задана дуга с координатами начальной (x_1, y_1) и конечной (x_2, y_2) точек. Определите номера квадрантов, в которых находятся начальная и конечная точки.
8. На плоскости задана прямоугольная система координат с осями X и Y. Составьте программу, которая вводит координаты точки (x, y), и сообщает, принадлежит ли эта точка прямоугольнику с центром в начале координат и длинами сторон x_1 и y_1, значения которых вводятся с клавиатуры.
9. Даны круг и квадрат. Составьте программу, определяющую по введенным значениям длин стороны квадрата и радиуса круга, верность утверждения "Круг вписан в квадрат". Используйте логическую переменную, принимающую значение `true`, если утверждение истинно, и значение `false`, если утверждение ложно.

10. Прямоугольник задан своими вершинами: $A(X_1, Y_1)$, $B(X_1, Y_2)$, $C(X_2, Y_2)$ и $D(X_2, Y_1)$. При этом $X_1 < X_2$, а $Y_1 < Y_2$. Написать программу для вычисления площади той части прямоугольника, которая лежит в первом квадранте декартовой системы координат.

11. Заданы координаты трех точек $A(X_1, Y_1)$, $B(X_2, Y_2)$, $C(X_3, Y_3)$. Проверить, лежат ли они на одной прямой.

12. Даны действительные положительные числа A, B, C, X, Y. Выяснить, пройдет ли кирпич с ребрами A, B, C в прямоугольное отверстие со сторонами X и Y. Просовывать кирпич в отверстие разрешается только так, чтобы каждое из его ребер было параллельно или перпендикулярно каждой из сторон отверстия.

13. По введенному номеру года определить, високосный он или нет. Помните, что номер високосного года делится на 4 нацело. Если последние две цифры в записи номера года — нули, а первые две образуют нечетное число, то год не является високосным, например, 1900 г.

14. Составьте программу для вычисления числа дней в месяце. Даны: номер месяца — целое число от 1 до 12, признак високосного года — целое число A, равное 1 для високосного года и 0 в противном случае.

15. Составьте программу, определяющую, пройдет ли график функции $y(x) = 5x^2 - 7x + 2$ через заданную точку с координатами (x, y).

16. Определите полярные координаты точки, заданной координатами (x, y) в прямоугольных координатах, по приведенным ниже формулам. При вычислении угла необходимо учесть, что значение x может быть равно 0, а угол может находиться в разных четвертях.

$$\varphi = \operatorname{arctg}(y / x), \qquad r = \sqrt{x^2 + y^2}\,.$$

17. Дано действительное число A. Вычислить $F(A)$, если:

$$F(x) = \begin{cases} -1, & x < 0; \\ 0, & x = 0; \\ 1, & x > 0. \end{cases}$$

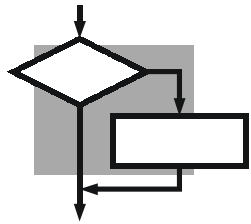
18. Решить систему уравнений по формулам Крамера для любых вводимых значений коэффициентов. Предусмотреть анализ решения в случае, когда определитель системы равен нулю.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1; \\ a_{21}x_1 + a_{22}x_2 = b_2, \end{cases}$$

$$x_1 = \frac{b_1 a_{22} - b_2 a_{12}}{a_{11}a_{22} - a_{12}a_{21}}, \qquad x_2 = \frac{a_{11}b_2 - a_{21}b_1}{a_{11}a_{22} - a_{12}a_{21}}.$$

19. Вводятся два целых двузначных числа *A* и *B*. Определить, у которого из них больше сумма цифр. Например, для числа 128 сумма цифр равна 11, для числа 345 сумма цифр равна 12.
20. Задать с клавиатуры трехзначное число. Из цифр этого числа найти наибольшее и наименьшее.
21. Задать с клавиатуры трехзначное число. Проверить, является ли это число палиндромом. Помните, что палиндромом называется такая запись числа, при которой слева направо и справа налево число выглядит одинаково.
22. Составьте программу, вычисляющую по введенному вами значению текущего времени (часов, минут, секунд) угол (в градусах) между положением часовой стрелки в начале суток и ее положением в текущий момент. Например, если текущее время составляет 3 ч 30 мин 00 с, то этот угол составит 105°108'.

Глава 9



Циклические программы

В программах, связанных с обработкой данных или другими сложными вычислениями, часто выполняются *циклически повторяющиеся действия*.

Например, при необходимости присвоить начальное значение нескольким сотням переменных тяжело и неразумно "вручную" набирать в тексте программы сотни операторов ввода или присваивания. Циклы позволяют записать такие действия в *компактной* форме. Поэтому они являются одной из важнейших алгоритмических структур.

Цикл представляет собой последовательность операторов, которая выполняется *неоднократно*.

В языке программирования Pascal имеется стандартный набор из трех разновидностей цикла — *цикл с постусловием* (инструкция `repeat`), *цикл с предусловием* (инструкция `while`) и *цикл со счетчиком* (инструкция `for`). Хорошо, что у программиста есть возможность выбора наиболее подходящего средства реализации алгоритма, но при изучении циклов часто возникают затруднения при выборе оператора цикла.

Следует знать:

- подавляющее большинство задач с циклами можно решить разными способами, используя при этом любой из трех операторов цикла;
- часто решения, использующие разные операторы цикла, оказываются равноценными, однако в некоторых случаях все же предпочтительнее использовать какой-то один из операторов;
- *самым универсальным* из всех операторов цикла считается `while`, поэтому в случае затруднений с выбором можно отдать предпочтение ему;
- цикл `repeat` имеет очень *простой и понятный синтаксис*, поэтому с него удобно начинать изучение циклов;
- цикл `for` обеспечивает удобную запись циклов с *заранее известным числом повторений*;
- при неумелом использовании циклов любого типа возможна ситуация, когда компьютер не сможет нормально закончить цикл (в таком случае

говорят, что программа *"зациклилась"*). При работе в среде Turbo Pascal для выхода из подобной ситуации используется комбинация клавиш <Ctrl>+<Break>;

- если это не помогает, есть и крайнее средство — <Ctrl>+<Alt>+<Del>. Одновременное нажатие этих трех клавиш или кнопки Reset, расположенной на системном блоке, позволяет *перезагрузить* компьютер, при этом данные, относящиеся к работающей программе, будут утеряны.

9.1. Цикл с постусловием

Оператор повтора `repeat` состоит из *заголовка* (`repeat`), *тела* и *условия окончания* (`until`). Ключевые слова `repeat`, `until` обозначают "повторяй" и "пока" соответственно.

```
repeat
   { Инструкции }
until Условие выхода из цикла;
```

`Условие выхода из цикла` — это выражение логического типа: *простое выражение отношения* или *сложное логическое выражение.*

Цикл работает так: *вначале* выполняется тело цикла — инструкции, которые находятся между `repeat` и `until`, *затем* проверяется значение `Условия выхода из цикла`. В том случае, если оно равно `false` (ложь), т. е. не выполняется, — инструкции цикла повторяются еще раз. Так продолжается до тех пор, пока условие не станет `true` (истина). На рис. 9.1 приведена блок-схема оператора повтора `repeat`.

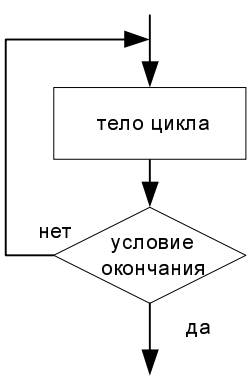


Рис. 9.1. Условное обозначение на схемах алгоритмов оператора `repeat`

Для примера давайте вернемся к игре "Угадай число" (см. рис. 1.2). Напомним — ее условие состояло в том, что игрок должен угадать число (назовем его comp), "задуманное" компьютером — случайное число в диапазоне от 0 до 1000. Процесс продолжается до тех пор, пока значение переменной igrok, которое вводится с клавиатуры, не совпадет со значением переменной comp (листинг 9.1).

Листинг 9.1. Игра "Угадай число"

```
var
   comp: integer;       { число, "задуманное" компьютером }
   igrok: integer;      { вариант игрока }
begin
   randomize;           { инициализация датчика случайных чисел }
   comp:=random(1000); { компьютер загадал число }
   repeat
      write('Введите число: '); readln(igrok);
      if igrok>comp then writeln('Слишком много...')
                    else if igrok<comp then writeln('Слишком мало...')
                                       else writeln ('Вы угадали!');
   until igrok=comp;
end.
```

Хорошим примером использования оператора repeat является *повтор фрагмента программы или всей программы*, производимый для того, чтобы выполнить необходимые действия, но уже с другими исходными данными. Чтобы каждый раз не перезапускать программу, удобно "зациклить" ее при помощи repeat, как показано в следующем примере.

```
var flag: char;
begin
   repeat
      { Повторяемый фрагмент }
      writeln('Повторить (Y/y) ?'); readln(flag);
   until upcase(flag)<>'Y';
end.
```

Комментарий

Функция upcase преобразует строчную букву в прописную.

Следует знать:

- *число повторений* операторов (инструкций) цикла `repeat` определяется в ходе работы программы и во многих случаях заранее неизвестно;
- для успешного завершения цикла `repeat` в его теле обязательно должны находиться инструкции, выполнение которых *влияет* на условие завершения цикла, иначе цикл будет выполняться бесконечно — программа зациклится. Другими словами, *переменная*, которая участвует в условии выхода из цикла, обязательно должна *изменяться* в теле цикла;
- для управления циклом `repeat` удобно использовать функции `succ` или `pred` *(см. разд. 7.4)* и процедуры `inc` или `dec` *(см. разд. 7.3)*.

  Например, рассмотрим следующую программу:

```
var i: integer;
begin
   i:=0;
   repeat
      i:=succ(i);   { или inc(i); или i:=i+1; }
      write(i,' ');
   until i>=10;
   writeln;
   i:=10;
   repeat
      write(i,' ');
      i:=pred(i);   { или dec(i); или i:=i-1; }
   until i<=0;
end.
```

  При выполнении программа выведет на экран строки:

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
```

- цикл `repeat` — это цикл с *постусловием*, т. е. инструкции тела цикла будут выполнены *хотя бы один раз*. Далее мы увидим, что циклы двух других разновидностей при определенном стечении обстоятельств могут вообще ни разу не выполниться. Поэтому цикл `repeat` удобно использовать в тех случаях, когда тело цикла гарантированно *должно выполниться хотя бы один раз*;
- нижняя граница операторов тела цикла четко обозначена словом `until`, поэтому нет необходимости заключать эти операторы в операторные скобки `begin` и `end`. В то же время наличие операторных скобок не будет являться ошибкой. Обычно слово `until` располагают строго под `repeat`, а каждую строку тела цикла начинают с отступов.

Оператор `repeat` часто используют для *проверки правильности ввода исходных данных*, т. к. это именно тот самый случай, когда тело цикла гарантированно выполняется один раз, а при наличии ошибки в исходных данных — более одного раза.

Например, предположим, что по условию задачи исходное данное должно быть двузначным числом. Программа будет повторять запрос на его ввод до тех пор, пока не получит то, что ей требуется. Рассмотрим соответствующий фрагмент программы:

```
var x: integer;
begin
...
repeat
   write('Введите двузначное число ');readln(x);
until (x>9)and(x<100);
...
end.
```

Обратите внимание — цикл `repeat` удобно использовать для *обработки ошибки ввода* при несоответствии типа введенных данных типу переменной. Стандартный способ обработки ошибок ввода состоит в том, что программа завершает работу и на экран выводится сообщение об ошибке (**Invalid Numeric Format** или **Invalid Numeric Input**).

Во многих случаях это совсем не удобно для пользователя программы, который нечаянно ввел с клавиатуры недопустимый символ. Поэтому в Pascal предусмотрена возможность отключения стандартного контроля ошибок ввода при помощи директивы компилятора `{$I-}`, а контроль правильности ввода выполняет стандартная функция `IOResult`. Эта функция имеет нулевое значение, если последняя операция ввода/вывода закончилась успешно, и ненулевое значение, равное коду ошибки, если ввод/вывод закончился неудачей.

Замечание

После вызова функция `IOResult` сбрасывает свое значение, поэтому обычно ее значение сразу присваивается какой-либо переменной. Рекомендуется после анализа результата, возвращаемого `IOResult`, снова установить стандартный контроль ошибок ввода/вывода с помощью директивы `{$I+}`.

Например:

```
var x: integer; ok: boolean;
   begin
...
      repeat
         writeln('Введите целое x');
```

```
        {$i-}  { отмена стандартного контроля операций ввода/вывода }
        readln(x); ok:=(ioresult=0);
        { обработка ошибки }
        if not ok then writeln('Ошибка ввода. Повторите.')
        {$i+}  { включение стандартного контроля операций ввода/вывода }
      until ok;
...
   end.
```

Листинг 9.2 содержит программу "Тест по таблице умножения", которую мы будем использовать и в дальнейшем для демонстрации возможностей различных операторов цикла. Сценарий программы простой — компьютер формирует два случайных сомножителя при помощи функции `random`, выводит вопрос на экран, а затем сравнивает ответ, введенный с клавиатуры, с правильным значением. В результате использования оператора цикла получается полноценная тестирующая программа, которая также будет выводить и результаты тестирования. Обратите внимание, что команда инициализации датчика случайных чисел `randomize` выполняется до начала цикла, т. к. она должна быть выполнена только один раз. Инструкции для вывода результатов теста выполняются после выхода из цикла.

Листинг 9.2. Тест по таблице умножения (вариант № 1)

```
var s1,s2,otvet,kol,prav: integer; yn: char;
begin
   randomize;   { инициализация датчика случайных чисел }
   repeat
      kol:=kol+1;
      s1:=random(18)+2; s2:=random(18)+2;
      write('Сколько будет ',s1,'*',s2,'? '); readln(otvet);
      if otvet=s1*s2 then begin
         write('Правильно! '); prav:=prav+1;
      end
      else write('Неверно...');
      write('Продолжим тест?(Y/N)'); readln(yn);
   until (yn='n')or(yn='N');
   writeln('Результаты теста: ');
   write('Задано вопросов: ',kol,'. Правильных ответов: ',prav,'.');
   readln;
end.
```

9.2. Цикл с предусловием

Оператор повтора `while` состоит из *заголовка и тела цикла*. Ключевые слова `while` и `do` обозначают "до тех пор, пока" и "выполняй" соответственно.

```
while Условие выполнения цикла do
begin
   { Инструкции }
end;
```

Условие выполнения цикла — это выражение логического типа. Тело цикла — оператор, чаще всего составной. Если тело цикла представляет собой одиночный оператор, то ключевые слова `begin` и `end` лучше не писать, хотя их наличие не будет ошибкой.

Оператор `while` аналогичен оператору `repeat`, но проверка значения `Условие выполнения цикла` производится в самом начале оператора — если значение условия равно `true` (истина), то *выполняются инструкции цикла*, находящиеся между `begin` и `end` и снова вычисляется выражение `Условие выполнения цикла`. Так продолжается до тех пор, пока значение `Условие выполнения цикла` не станет равно `false` (ложь). На рис. 9.2 приведена блок-схема оператора повтора `while`.

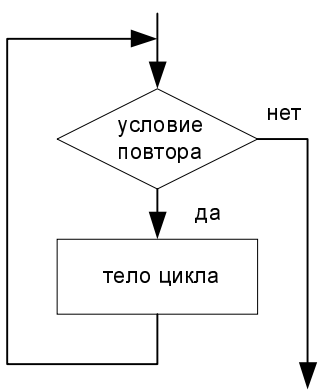


Рис. 9.2. Условное обозначение на схемах алгоритмов оператора `while`

Взаимосвязь операторов `while` и `repeat` можно показать, выразив `while` через `repeat`: оператор `while Условие do Оператор;` *эквивалентен* оператору `if Условие then repeat Оператор until Not Условие;`

Для примера рассмотрим фрагмент программы суммирования чисел от 1 до 10. В данном примере использование всех видов цикла равноценно:

```
...
s:= 0; i:= 1;
   while i<=10 do   { находим сумму чисел от 1 до 10 }
      begin
         s:=s+i;
         i:=i+1;    { изменение переменной управления циклом }
      end;
...
```

Следует знать:

- *число повторений* операторов (инструкций) цикла `while` определяется в ходе работы программы и, как правило, заранее неизвестно;
- для успешного завершения цикла `while` в его теле обязательно должны быть инструкции, оказывающие *влияние* на условие выполнения цикла;
- для управления циклом `while`, как и циклом `repeat`, удобно использовать функции `succ` или `pred` и процедуры `inc` или `dec` *(см. разд. 9.1)*.
- цикл `while` — это цикл с *предусловием*, т. е. инструкции тела цикла *могут быть не выполнены ни разу*, если проверяемое условие ложно с самого начала. Исходя из этого, цикл `while` считают *самым универсальным видом* цикла;
- цикл `while` обычно используется в тех же задачах, что и `repeat` (в зависимости от личного вкуса программиста). Удобнее всего использовать его в тех случаях, когда возможны ситуации невыполнения цикла;
- в операторе цикла `while` точка с запятой никогда не ставится после зарезервированного слова `do`. Если неопытный программист по ошибке поставит эту точку с запятой, то сообщения об ошибке выдано не будет, но компилятор будет считать, что тело цикла представляет собой пустой оператор *(см. разд. 8.5)*. Результаты программы при этом будут неверными;
- ключевые слова `begin` и `end`, ограничивающие тело цикла, обычно располагают по одной вертикальной линии, используя отступы в каждой строке тела цикла. В целях сокращения количества строк программы иногда применяют такой способ записи:

```
while Условие выполнения цикла do begin
   { Инструкции }
end; {end располагается строго под while}
```

Листинг 9.3 содержит программу "Тест по таблице умножения", измененную таким образом, чтобы при необходимости можно было отказаться от выпол-

нения теста и корректно выйти из программы. Такая ситуация может возникнуть, например, при случайном запуске программы, не преследующем цель проверки знаний. Цикл while в таком варианте наиболее предпочтителен.

Листинг 9.3. Тест по таблице умножения (вариант № 2)

```
var s1,s2,otvet,kol,prav: integer; yn: char;
begin
   randomize;
   write('Начнем тест?(Y/N)'); readln(yn);
   while (yn<>'n') and (yn<>'N') do begin
      kol:=kol+1;
      s1:=random(18)+2; s2:=random(18)+2;
      write('Сколько будет ',s1,'*',s2,'? '); readln(otvet);
      if otvet=s1*s2 then begin
            write ('Правильно! '); prav:=prav+1;
      end
      else write ('Неверно...');
      write('Продолжим тест?(Y/N)'); readln(yn);
   end;          { конец цикла }
   if kol>0 then begin
      writeln('Результаты теста: ');
      write('Задано  вопросов: ',kol);
      writeln('. Правильных ответов: ',prav,'.'); readln;
   end; { выводим результаты, только если тест состоялся }
end.
```

Листинг 9.4 содержит программу, которая выводит на экран таблицу значений некоторой функции. Вывод выполняется в два столбца: первый — значения аргумента, второй — значения функции.

$$f(x) = \lg(3) + x\sqrt{5\sin\left(\frac{\pi x}{3}\right)}.$$

Листинг 9.4. Таблица значений функции

```
uses crt;
var x,y,z,lg3,a,b,dx: real;
begin
   clrscr;
   write('Введите начальное значение аргумента: '); readln(a);
```

```
   write('Введите конечное значение аргумента: '); readln(b);
   write('Введите шаг табулирования: '); readln(dx);
   writeln('---------------':20);
   writeln('x':9,' | ':4,'y':4);
   writeln('---------------':20);
   lg3:=ln(3.0)/ln(10.0);   { вычисление lg(3) }
   x:=a;
   while x<(b+dx/2) do begin
      z:=sin((pi*x)/3);
      if(z < 0) then writeln(x:10:3,'   |  функция не определена':22)
      else begin
         y:=lg3+x*sqrt(5.0*z);
         writeln(x:10:3,'   | ',y:7:3);
      end;
      x:=x+dx;
   end;
   writeln('---------------':20);
end.
```

Комментарий

При разработке подобных программ следует *учитывать область определения функции* и в случае необходимости организовать вывод сообщения — "функция не определена".

9.3. Цикл с параметром

Этот вид оператора цикла также называют *циклом со счетчиком*. В нем важную роль играет *переменная-параметр*, которая на каждом шаге цикла автоматически изменяет свое значение ровно на *единицу* — поэтому ее и называют *счетчиком*.

Инструкцию `for` можно реализовать двумя способами.

Вариант 1 (с увеличением счетчика):

```
for Счетчик := НачальноеЗначение to КонечноеЗначение do
   begin
      { Инструкции }
   end;
```

Ключевые слова `for`, `do` обозначают "для", "выполняй" соответственно. Строка, содержащая `for...do`, называется *заголовком цикла*, оператор,

стоящий после do, образует его *тело*. Очень часто тело цикла — составной оператор. Если тело цикла представлено одиночным оператором, то begin и end не пишутся.

На блок-схеме алгоритма цикл с параметром удобно обозначать следующим образом (рис. 9.3).

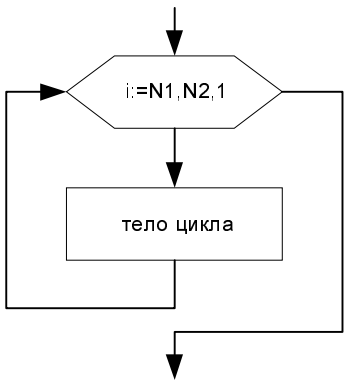


Рис. 9.3. Условное обозначение на схемах алгоритмов цикла с параметром

Такое удобное обозначение, к сожалению, не вполне соответствует ГОСТу на блок-схемы алгоритмов *(см. разд. 1.2)*, хотя явно ему и не противоречит.

Инструкции между begin и end выполняются такое число раз, какое определяет выражение [(КонечноеЗначение – НачальноеЗначение) + 1].

Это соответствует всем значениям счетчика от начального до конечного включительно.

Если НачальноеЗначение больше, чем КонечноеЗначение, то инструкции между begin и end не выполняются ни разу. Это позволяет сделать вывод о том, что цикл с параметром *является разновидностью цикла* с *предусловием*.

Например, выполнение цикла

```
for i:=10 to 14 do write(i: 3);
```

выведет на экран последовательность чисел в виде:

```
 10 11 12 13 14
```

Вариант 2 (с уменьшением счетчика):

```
for Счетчик := НачальноеЗначеиие downto КонечноеЗначение do
begin
   { Инструкции }
end;
```

Инструкции между `begin` и `end` выполняются такое число раз, какое определяет выражение `[(НачальноеЗначение – КонечноеЗначение) + 1]`.

Если `НачальноеЗначение` меньше, чем `КонечноеЗначение`, то инструкции между `begin` и `end` не выполняются ни разу.

Например, выполнение цикла

```
for i:=14 downto 10 do write(i: 3);
```

выведет на экран последовательность цифр в виде:

14 13 12 11 10

Если переменная-счетчик имеет символьный тип `char`, то оператор

```
for ch:= 'a' to 'e' do write (ch: 2);
```

выведет на экран последовательность букв в виде:

a b c d e

а оператор

```
for ch:= 'e' downto 'a' do write (ch: 2);
```

выведет на экран последовательность букв в виде:

e d c b a

Следует знать:

- оператор (инструкция) `for` используется для организации циклов с *фиксированным*, заранее известным или определяемым во время выполнения программы числом повторений;
- переменная-счетчик должна быть *порядкового типа:* чаще — целочисленная, реже — символьного, логического или перечисляемого типов. *Использование вещественного типа недопустимо*;
- *начальное* и *конечное* значения параметра цикла могут быть константами, переменными, выражениями и должны принадлежать к *одному и тому же типу* данных. Начальное и конечное значения параметра цикла *нельзя изменять во время выполнения цикла*;
- после нормального завершения оператора `for` значение параметра цикла равно конечному значению (в некоторых компиляторах (Delphi) — на единицу больше). Если оператор `for` не выполнялся, значение параметра цикла не определено;
- *параметр цикла* `for` может изменяться (увеличиваться или уменьшаться) каждый раз при выполнении тела цикла *только на единицу*. Если нужен *другой шаг* изменения параметра, предпочтительнее циклы `repeat` и `while`.

Например, рассмотрим фрагмент программы, в котором предпринята попытка "обмануть" оператор `for` и получить изменение параметра `i` на 2 на каждом шаге цикла (единица прибавляется автоматически и еще одна единица прибавляется в теле цикла).

```
for i:= 1 to 10 do
begin
   write(i:4); i := i + 1;
end;
```

В данном случае на экране будут выведены числа

1 3 5 7 9

Однако настоятельно *не рекомендуем* пользоваться таким приемом. Стоит только немного видоизменить заголовок цикла предыдущего примера и взять в качестве конечного значения 9, а не 10: `for i := 1 to 9 do`, как ваша программа не сможет нормально выйти из цикла — "зациклится", т. к. в момент проверки условия выхода из цикла `i` никогда не будет равно 9 (сначала 8, а потом сразу 10). Компилятор Delphi вообще не допускает подобных действий. При попытке откомпилировать приведенный выше фрагмент программы будет выведено сообщение об ошибке **Assignment FOR-Loop variable "i"** на операторе `i:=i+1`.

Для досрочного выхода из цикла можно использовать оператор `goto` или стандартную процедуру `break`. Стандартная процедура `continue` позволяет прервать выполнение тела любого цикла и передает управление на его заголовок, заставляя цикл немедленно перейти к выполнению следующего шага.

Приведем пример фрагмента программы с досрочным выходом из цикла:

```
...
for i := 1 to 45 do
begin
   f:=f +i;
   if (f>100)or(i=39) then break;
end;
...
```

Но все-таки проявлением лучшего стиля программирования в подобных случаях будет использование циклов `repeat` и `while`.

Листинг 9.5 содержит программу "Тест по таблице умножения", измененную таким образом, чтобы она задавала ученику ровно пять вопросов, а в конце тестирования выставляла оценку по пятибалльной системе. Естественно, в такой постановке задачи наилучшим решением будет являться цикл `for`.

Листинг 9.5. Тест по таблице умножения (вариант № 3)

```
var s1,s2,otvet,k,prav: integer;
begin
   randomize;
   for k:=1 to 5 do begin
      s1:=random(18)+2; s2:=random(18)+2;
      write('Сколько будет ',s1,' * ',s2,'? '); readln(otvet);
      if otvet=s1*s2 then begin
         write('Правильно! '); prav:=prav+1;
      end
      else write('Неверно...');
   end;
   writeln('Ваша оценка: ',prav); readln;
end.
```

Еще один пример (листинг 9.6) — программа, которая суммирует числа, введенные с клавиатуры. Если количество чисел известно заранее, то наилучшим решением является цикл `for`.

Листинг 9.6. Суммирование любого количества целочисленных значений

```
var x,n,i: integer; sum: real;
begin
   write('Введите количество чисел: '); readln(n);
   writeln('Вводите числа');
   sum:=0;
   for i:=1 to n do begin
      write('Введите значение ',i,'-го слагаемого: ');
      readln(x);       { ввести очередное значение x }
      sum:= sum+x;
   end;
   writeln('Сумма введенных чисел = ',sum:7:0);readln;
end.
```

Комментарий

Обратите внимание, что сумма чисел имеет вещественный тип, хотя слагаемые — целые числа. Это не ошибка — таким способом можно бороться с возможным переполнением.

9.4. Вложенные циклы

В теле любого оператора цикла могут находиться другие операторы цикла. При этом цикл, содержащий в себе другой, называют *внешним*, а цикл, находящийся в теле первого, — *внутренним (вложенным)*. Правила организации внешнего и внутреннего циклов такие же, как и для простого цикла.

Обратите внимание — при программировании вложенных циклов необходимо соблюдать следующее дополнительное условие: *все операторы внутреннего цикла должны полностью располагаться в теле внешнего цикла.*

Рассмотрим задачу вывода на экран таблицы умножения, решение которой предполагает использование вложенных циклов. С использованием цикла `for` соответствующий фрагмент программы имеет вид:

```
var i, j : byte;
begin
   for i:=1 to 9 do begin
      for j:=1 to 9 do
         write(i,' * ',j,' = ',i*j:2);
      writeln; {перевод строки — во внешнем цикле}
   end;
end.
```

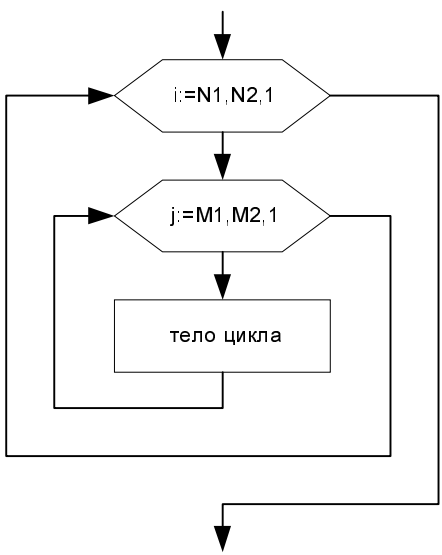


Рис. 9.4. Условное обозначение
на схемах алгоритмов вложенных циклов с параметром

На блок-схеме (рис. 9.4) наглядно показано, что при вложении циклов внутренний цикл выполняется *полностью* от начального до конечного значения параметра при *неизменном* значении параметра *внешнего* цикла. Затем значение параметра *внешнего* цикла изменяется на *единицу*, и опять от начала и до конца выполняется *вложенный* цикл. И так до тех пор, пока значение параметра внешнего цикла не станет больше конечного значения, определенного в операторе `for` внешнего цикла.

Рассмотрим еще один пример с вложенными циклами. В 1912 г. американский флаг "Былая слава" имел 48 звезд (по одной на штат) и 13 полос (по одной на колонию). Листинг 9.7 содержит программу, которая выводит на экран изображение "Былой славы" 1912 г., используя только символы * (звездочку) и _ (символ подчеркивания). В наше время штатов больше, следовательно, увеличилось и количество звезд.

Листинг 9.7. Вывод на экран американского флага "Былая слава"

```
uses crt;
var row,col: integer;
begin
   clrscr;   { очистка экрана }
   for col:= 1 to 19 do write('__');   { верхняя граница полотнища }
   writeln;
   for row:= 1 to 13 do begin
      for col:= 1 to 19 do
         { в строках с 1 по 6 и столбцах с 1 по 8 выводится 48 звезд }
         { иначе — линия }
         if (col<9)and(row<7) then write('* ') else write('__');
      writeln;
   end;
   readln;
end.
```

9.5. Примеры программ

9.5.1. Выделение цифр целого числа

Листинг 9.8 содержит программу, которая определяет количество и сумму цифр в записи любого натурального числа, не превосходящего `maxint`, (см. табл. 6.4), а также выводит запись числа в обратном порядке, опуская незначимые нули в его начале. Подобная задача для трехзначного числа приво-

дилась в листинге 7.4. Если количество цифр числа неизвестно, то такую задачу можно решить, только используя цикл (лучше `repeat` или `while`).

Листинг 9.8. Выделение цифр числа (вариант с циклом)

```
var k,s,n: integer; a: longint;
begin
   write('Введите натуральное число <= ',maxint,' : ');readln(n);
   k:=0; s:=0; a:=0;
   repeat
      k:=k+1;
      s:=s+n mod 10;
      a:=a*10+n mod 10;
      n:=n div 10
   until n=0;
   writeln('Число цифр: ',k,', сумма цифр: ',s);
   writeln('Запись в обратном порядке: ',a);
end.
```

9.5.2. Обработка числовых данных

Листинг 9.9 содержит программу, которая вычисляет среднее арифметическое чисел и определяет минимальное и максимальное число в последовательности из заданного количества действительных чисел, вводимых с клавиатуры. Выполнение этих действий не требует запоминания в памяти всех введенных чисел, все числа по очереди заносятся в одну переменную `a`, и сразу же обрабатываются. Обратите внимание, как постепенно изменяются переменные `max` и `min`, в которых на каждом шаге цикла оказываются самое большое и самое маленькое из всех чисел, введенных к этому моменту.

Листинг 9.9. Определение среднего, минимума и максимума в группе чисел

```
var
   a: real;           { очередное число }
   n: integer;        { количество чисел }
   sum,average: real; { сумма и среднее арифметическое }
   min,max: real;     { минимальное и максимальное число }
   i: integer;        { счетчик цикла }
begin
   write('Введите количество чисел последовательности: '); readln(n);
```

```
   writeln('Вводите последовательность');
   write('->'); readln(a); { вводим первое число последовательности }
   { предположим, что: }
   min:=a;          { первое число является минимальным }
   max:=a;          { первое число является максимальным }
   sum:=a;
   { введем остальные числа }
   for i:=1 to n-1 do begin
      write('->'); readln(a);
      sum:=sum+a;
      if a < min then min:=a;
      if a > max then max:=a;
      end;
   average:=sum/n;
   writeln('Количество чисел: ',n);
   writeln('Среднее арифметическое:',average:7:3);
   writeln('Минимальное число: ',min:7:3);
   writeln('Максимальное число:',max:7:3);
end.
```

Комментарий

Для решения поставленной задачи нет необходимости хранить введенные числа. Однако зачастую, особенно для последовательности случайных чисел, важно знать не только их среднее значение, но и так называемое среднеквадратичное отклонение, характеризующее степень "разброса" относительного среднего. Для его вычисления требуется предварительно найти среднее. Таким образом, числовую последовательность необходимо будет *запомнить при первом вводе и использовать повторно*. Для этого используются *массивы (см. гл. 13)*.

9.5.3. Формирование числовых последовательностей

В математике большую роль играют последовательности чисел с определенными свойствами. Формирование таких последовательностей — хороший пример использования циклов, поэтому рассмотрим одну из них.

В 1202 г. итальянский математик Леонард Пизанский, известный под именем Фибоначчи, предложил следующую задачу. Пара кроликов каждый месяц дает приплод — двух кроликов (самца и самку), от которых через два месяца уже получается новый приплод. Сколько пар кроликов будет через месяц, два, год и т. д., если в начале года мы имели одну пару только что родившихся кроликов?

Обратим внимание на то, что числа, соответствующие количеству пар кроликов по месяцам, составляют последовательность 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Каждый из членов этой последовательности, начиная с третьего, равен сумме двух предыдущих членов. Эта последовательность называется рядом Фибоначчи, а ее члены — числами Фибоначчи. Числа Фибоначчи имеют много интересных свойств. *Ряд Фибоначчи* определяют так: $F_0 = F_1 = 1$; $F_n = F_{n-1} + F_{n-2}$.

Листинг 9.10 содержит программу, которая выводит n членов ряда Фибоначчи.

Листинг 9.10. Программа вычисления членов ряда Фибоначчи

```
var fn,fn1,fn2,k,n: integer;
begin
   write('Введите число членов ряда Фибоначчи: '); readln(n);
   fn1:=1; fn:=0;
   for k:=1 to n do begin
      fn2:=fn1; fn1:=fn;
      fn:= fn1+fn2;
      writeln(fn);
   end; readln;
end.
```

Комментарий

Для того чтобы сделать программу компактной, необходимо, чтобы все члены ряда вычислялись в соответствии с одними и теми же правилами. В ряду Фибоначчи исключение составляют первые два члена. Для того чтобы вычислить их в соответствии с указанными правилами, необходимо искусственно продолжить ряд влево, пополнив его двумя фиктивными членами: $F_{-1} = 1$, $F_0 = 0$. В начале процедуры задаются значения этих фиктивных членов. Все подлинные числа Фибоначчи вычисляются в цикле по одним правилам.

В связи с тем, что ряд Фибоначчи является быстрорастущим, его 24-й член уже превышает верхнюю границу диапазона двухбайтового `integer`.

9.5.4. Вычисление сумм бесконечных убывающих последовательностей. Особенности арифметики с плавающей точкой

Многие из математических величин или значений функций могут быть выражены как *суммы бесконечных последовательностей.*

Например:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + \frac{(-1)^{n-1}}{2n-1} + \ldots,$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots,$$

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

где *n!* — *факториал* натурального числа *n*, т. е. произведение чисел 1, · 2, ..., · *n*.

Чем больше членов ряда участвует в сложении, тем более точным получается искомое значение. Разность полученной суммы ряда и действительной суммы называется *погрешностью сложения*. Эту погрешность можно оценить различными способами. Часто оценивают *n*-й член — если он достаточно мал, т. е. меньше некоторого данного числа `eps`, считается, что найденная сумма удовлетворяет требованиям по точности.

Листинг 9.11 содержит программу, которая вычисляет косинус угла, используя формулу, приведенную выше. Конечно, можно не мучиться и воспользоваться стандартной функцией вычисления косинуса, но задача хороша для тренировки и, кроме того, дает представление о вычислении стандартных функций.

Листинг 9.11. Вычисление косинуса угла

```
const eps =1e-7;   { точность вычисления }
var x,s,u :real; n: word;
begin
   write('Введите аргумент в градусах: '); readln(x);
   x:=x*pi/180;    { переводим угол в радианы }
   s:=1; u:=1; n:=0;
   repeat
      n:=n+2;
      u:=-u*x*x/((n-1)*n);
      s:=s+u;
   until abs(u)<eps;
   writeln('cos(',x*180/pi:6:2,') = ',s:7:3);
end.
```

Листинг 9.12 содержит программу, которая вычисляет частичные суммы n членов ряда 1, −1/3, +1/5, −1/7, +... В пределе (при бесконечно большом n) значение суммы членов этого ряда составляет π/4, поэтому таким способом можно вычислить число π.

Листинг 9.12. Вычисление частичной суммы членов ряда

```
var
   x: real;       { член ряда }
   summ: real;    { частичная сумма }
   n: word;       { количество суммируемых членов }
   i: word;       { счетчик цикла }
begin
   write('Введите кол-во суммируемых членов ряда: '); readln(n);
   x:=1; summ:=0;
   for i:=1 to n do begin
      summ:=summ+1/x;
      if x>0 then x:=-(x+2) else x:=-(x-2);
   end;
   writeln('Сумма ряда:   ',summ:10:6);
   writeln('Значение pi/4:',pi/4:10:6);
end.
```

Комментарий

На примере этой программы попробуем проиллюстрировать особенности машинной арифметики с плавающей точкой *(см. разд. 4.3.2)*. Как известно, для уменьшения влияния ошибки округления суммирование нужно начинать с последних членов ряда, имеющих наименьшую величину. При "обратном" порядке суммирования слагаемые меньше отличаются друг от друга по величине, чем при прямом, поэтому точность вычисления полной суммы будет выше. Для реализации *суммирования в обратном порядке* используем цикл с уменьшением счетчика.

Соответствующий фрагмент программы имеет вид:

```
...
   summ:=0;
   for i:=n downto 1 do
      begin
         x:=1/(2*i – 1);
         if (i mod 2) = 0 then x:=-x;
         summ:=summ+x;
      end;
...
```

Для смены знака слагаемых выполняется проверка номера слагаемого на четность при помощи функции mod: все члены ряда с четными номерами отрицательны.

Разумеется, для данного примера, в отличие от серьезных расчетов, различие в значениях суммы при прямом и обратном суммировании очень мало, но все же заметно. При 10 000 членов ряда его сумма равна:

```
7.85373163397708E-0001 { для цикла for..downto };
7.85373163409531E-0001 { для цикла for..to };
7.85398163397448E-0001 { значение pi/4 }.
```

Для того чтобы оценить *точность* любых вычислений (в том числе и рассмотренных выше), необходимо знать, сколько верных десятичных цифр (знаков) обеспечивает данное представление числа. Листинг 9.13 содержит программу, которая выводит *число десятичных знаков*, выделяемых для представления действительного числа.

Листинг 9.13. Число десятичных знаков для представления действительного числа

```
const base=10;  { основание десятичной системы счисления }
      a=1.0;    {  число a имеет один точный знак }
var p: real;    { 1.0, 0.1, 0.01, 0.001,... }
    n: integer; { число знаков }
begin
   p:=1.0; n:=0;
   repeat
      n:=n+1;
      p:=p/base;
   until a=a+p;
   writeln(n-1);
end.
```

Комментарий

Цикл заканчивается, когда 1.0=1.0…01. Это значит, что последняя цифра числа 1.0…01 исчезла. Результатом выполнения программы в среде Turbo Pascal является число 12, в среде Delphi — 19. Вспомним, что 16-разрядные компиляторы выделяют подтип `real` 6 байт, а 32-разрядные — 8 байт.

В приведенном примере представляет интерес и переменная `p` — это первое появившееся значение, прибавление которого к единице не изменяет ее. В арифметике с плавающей точкой имеется специальное понятие —"машинное эпсилон". Это такое минимальное, не равное нулю число, после прибавления которого к единице компьютер все еще выдает результат, отличный от единицы. Следующий листинг (листинг 9.14) находит это значение.

Листинг 9.14. Определение "машинного эпсилон"

```
var eps: real;
begin
   eps:=1.0;
   while eps/2+1 > 1 do eps:=eps/2;
   writeln('Машинное epsilon = ', eps);
end.
```

Комментарий

Результаты этой программы полностью согласуются с результатами предыдущей: примерно 10^{-12} для шестибайтового `real` и 10^{-19} для восьмибайтового.

9.5.5. График функции на текстовом экране

Листинг 9.15 содержит программу, которая при помощи символа * (звездочка) выводит на экран график функции $\sin(x)/\exp(\alpha \cdot x)$, симметрично относительно вертикали экрана. Ширина поля изменяется от строки к строке; в каждой строке звездочка прижимается к правому краю поля. Разумеется, для вывода графика обычно используются графические средства, а данный пример просто иллюстрирует еще одну возможность использования оператора цикла.

Листинг 9.15. Вывод на экран графика функции $\sin(x)/\exp(\alpha \cdot x)$, используя символ *

```
const offset=40;     { график симметричен относительно вертикали экрана }
      scale=15;      { амплитуда }
      degreestep=40; { частота }
      alpha=0.15;
var i: integer; k: real;
begin
   k:= degreestep*pi/180; { переводим угол в радианы }
   for i:= 0 to 23 do
      writeln('*':round(offset+scale*sin(k*i)/exp(alpha*k*i)));
   readln;
end.
```

К задачам с различными действиями над функциональными зависимостями мы еще вернемся в следующей главе, где покажем, как сделать подобные программы более универсальными.

9.5.6. Задачи на перебор всех вариантов

Особенностью компьютерных вычислений является чрезвычайно высокое быстродействие, поэтому некоторые задачи, которые не имеют аналитического решения (или имеют очень сложное решение) компьютер без труда решит "тупым" перебором всех возможных вариантов. Поэтому большую группу циклических программ составляют так называемые задачи на полный перебор. Приведем два примера таких задач.

Листинг 9.16 содержит программу, которая решает следующий числовой ребус:

```
  охохо { = О0О0О + Х0Х0 = О*Ч10101 + Х*Ч1010 }
+ ахаха { = А*Ч10101 + Х*Ч1010 }
=ахахах { = А0А0А0 + Х0Х0Х = А*Ч101010 + Х*Ч10101 }
```

Необходимо определить, какую цифру обозначает каждая из букв О, Х и А.

Листинг 9.16. Решение числового ребуса

```
var o,a,x,ox,ax: longint;
begin
   for o:=1 to 9 do       { o и a — начальные цифры, поэтому они }
      for a:=1 to 9 do    { могут принимать значения от 1 до 9 }
         for x:=0 to 9 do { x может принимать значения от 0 до 9 }
            begin
               ox:=o*10101+x*1010;
               ax:=a*10101+x*1010;
               if (ox+ax)=(a*101010+x*10101) then
                  writeln(ox,'+',ax,'=',ox+ax);
            end;
end.
```

Комментарий

В вычислении участвуют три цифры. Программа перебирает все $9 \cdot 9 \cdot 10 = 810$ вариантов ребуса и выводит единственный ответ: 90909+10101=101010.

Листинг 9.17 содержит программу, которая решает старинную задачу: сколько можно купить быков (бык стоит 10 рублей), коров (по 5 рублей) и телят (за 0,5 рубля), если на 100 рублей надо купить 100 голов скота?

Листинг 9.17. Задача о покупке скота

```
var b,k,t: integer;
begin
   for b:=0 to 10 do      { ограничение по быкам }
      for k:=0 to 20 do   { ограничение по коровам }
         begin
            t:=100-(b+k); { число телят из второго уравнения системы }
            if 20*b+10*k+t=200 then { условие - первое уравнение системы}
               begin
                  writeln('Куплено быков',b:3);
                  writeln('Куплено коров',k:3);
                  writeln('Куплено телят',t:3);
               end;
         end;
end.
```

Комментарий

Условие задачи можно записать в виде системы двух уравнений с тремя неизвестными: $10b + 5k + 0{,}5t = 100$ и $b + k + t = 100$. Причем обе части первого уравнения можно умножить на 2 и получить: $20b + 10k + t = 200$. В общем случае такая система имеет бесконечное число решений. Но в нашем примере их число конечно: ответом может быть только неотрицательное целое число — за 100 рублей нельзя купить более 10 быков или 20 коров или 200 телят. Решение задачи сводится к тому, что одно неизвестное (количество телят) выражается из второго уравнения системы через два других неизвестных. После этого во вложенном цикле проверяется при переборе b и k только первое уравнение системы.

Если вычесть второе уравнение системы из первого, получим, что решение задачи сводится к одному уравнению с двумя неизвестными: $19b + 9k = 100$ и имеет следующий вид:

```
   for b:=0 to 10 do
      for k:=0 to 20 do
         if 19*b+9*k=100 then
               begin
                  writeln('Куплено быков',b:3);
                  writeln('Куплено коров',k:3);
                  writeln('Куплено телят',100-(b+k):3);
               end;
```

Замечание

Мы решили так называемое *линейное диофантово уравнение*. Общий метод решения *нелинейных диофантовых уравнений* до сих пор неизвестен, но их целочисленные решения также можно найти перебором.

9.5.7. "Римские цифры"

Листинг 9.18 содержит программу, которая представляет заданное натуральное число при помощи римских цифр. При этом 1000 обозначается M; 500 обозначается D, 100 — C, 50 — L, 10 — X, 5 — V, 1 — I.

Листинг 9.18. Запись натурального числа римскими цифрами

```
var n: 0..3999;
    k: (M,CM,D,CD,C,XC,L,XL,X,IX,V,IV,I);
    s: integer;
begin
   write('Введите исходное арабское число: '); readln(n);
   write('Результат: ');
   for k:=M to I do begin
      case k of
         M :s:=1000;CM:s:=900;D :s:=500;
         CD:s:=400; C :s:=100;XC:s:= 90;
         L :s:= 50; XL:s:= 40;X :s:= 10;
         IX:s:=  9; V :s:=  5;IV:s:=  4;
         I :s:=  1
      end;
      while n-s>=0 do begin
         n:=n-s;
         case k of
            M :write('M'); CM:write('CM');D :write('D');
            CD:write('CD');C :write('C'); XC:write('XC');
            L :write('L'); XL:write('XL');X :write('X');
            IX:write('IX');V :write('V'); IV:write('IV');
            I :write('I')
         end; {case}
      end; {while}
   end; {for}
end.
```

Комментарий

В качестве исходных в программе рассматриваются натуральные числа от 0 до 3999, для представления которых используется переменная n интервального типа. Значения переменной k перечисляемого типа — искомые римские цифры.

Прямая задача перехода от арабских к римским цифрам с алгоритмической точки зрения представляет меньший интерес, чем обратная. С помощью приведенного ниже примера можно познакомиться с алгоритмом реализации так называемых *автоматных распознавателей*, которые используются, например, при разработке трансляторов. Листинг 9.19 содержит программу, которая наглядно демонстрирует простой алгоритм распознавания текста. Для того чтобы декодировать выполненную по обычным правилам римскую запись чисел, содержащих любое количество цифр X в начале, а также цифр V и I, необходимо составить следующую *таблицу переходов состояний* (табл. 9.1).

Таблица 9.1. *Таблица переходов состояний для декодирования римских чисел*

Символ	X		V		I	
состояние 1	n:=10	state:=2	n:=5	state:=3	n:=1	state:=6
состояние 2	n:=n+10	state:=2	n:=n+5	state:=3	n:=n+1	state:=6
состояние 3	ok:=false		ok:=false		n:=n+1	state:=4
состояние 4	ok:=false		ok:=false		n:=n+1	state:=5
состояние 5	ok:=false		ok:=false		n:=n+1	state:=7
состояние 6	N:=n+8	state:=7	n:=n+3	state:=7	n:=n+1	state:=5
состояние 7	ok:=false		ok:=false		ok:=false	

Рассмотрим пример расшифровки числа XIV. Процесс начинается с состояния 1 (первой строки таблицы). Первый символ — X, поэтому смотрим столбец X и находим `n:=10; state:=2`. Итак, полагаем n равным 10, после чего смотрим вторую строку (т. к. `state:=2`). Теперь обращаемся к столбцу, определяемому вторым символом — I, и находим `n:=n+1; state:=6`. Значение n, таким образом, становится 10 + 1 = 11. Смотрим шестую строку (`state:=6`). В столбце V находим `n:=n+3; state:=7`. Значение n становится равным 11 + 3 = 14. Смотрим седьмую строку (`state:=7`) и замечаем, что любая следующая цифра X, V или I будет ошибкой (например, XIVX).

Листинг 9.19. Декодирование римской записи числа

```
var n,state: integer; symbol: char; ok: boolean;
begin
   state:=1; ok:=true; n:=0;
```

```
while not eoln do begin
   read(symbol);
   if((symbol='X')or(symbol='V')or(symbol='I'))
   then
      case state of
         1: case symbol of   { первая строка таблицы соответствия }
               'X': begin n:=10; state:=2 end;
               'V': begin n:=5; state:=3 end;
               'I': begin n:=1; state:=6 end;
            end;
         2: case symbol of   { вторая строка }
               'X': begin n:=n+10; state:=2 end;
               'V': begin n:=n+5; state:=3 end;
               'I': begin n:=n+1; state:=6 end;
            end;
         3: case symbol of   { третья строка }
               'X','V': ok:=false;
               'I': begin n:=n+1; state:=4 end;
            end;
         4: case symbol of   { четвертая строка }
               'X','V': ok:=false;
               'I': begin n:=n+1; state:=5 end;
            end;
         5: case symbol of   { пятая строка }
               'X','V': ok:=false;
               'I': begin n:=n+1; state:=7 end;
            end;
         6: case symbol of   { шестая строка }
               'X': begin n:=n+8; state:=7 end;
               'V': begin n:=n+3; state:=7 end;
               'I': begin n:=n+1; state:=5 end;
            end;
         7: ok:=false;   { седьмая строка }
      end    {case state}
      else begin
         if ok then writeln(n:2)
         else writeln('Число введено с ошибкой');
```

```
            state:=1; ok:=true;
         end;    {else}
      end;   {while not}
end.
```

Комментарий

Перед нажатием клавиши <Enter>, завершающей ввод данных, необходимо ввести точку или пробел.

Функция `eoln` проверяет, не является ли очередной символ буфера клавиатуры символом конца строки. Если является, то функция возвращает значение `true`, если нет — `false`. Логическое выражение `not eoln` обеспечивает выполнение цикла `while`, т. е. декодирование текста до тех пор, пока не будет нажата клавиша <Enter>.

Для работы с цифрами M = 1000, D = 500, C = 100 и L = 50 таблицу нужно расширить.

Более серьезный пример реализации автоматного распознавателя содержится в листинге 14.8.

9.6. Контрольные вопросы и задания

Вопросы

1. Что такое цикл? Обоснуйте необходимость его применения.
2. Какие разновидности циклов, существующих в языке Pascal, вам известны?
3. Как можно выйти из ситуации "зацикливания" программы? Что при этом происходит с ее данными?
4. Как можно использовать оператор `repeat` при вводе исходных данных? Приведите примеры.
5. В чем отличие операторов `repeat` и `while`? Когда удобнее использовать цикл `while`?
6. При помощи какого оператора можно досрочно выйти из цикла? Перейти к следующему шагу выполнения цикла?
7. В каких случаях оператор `for` предпочтительнее использовать для организации циклов?
8. Какие ограничения по типу накладываются на переменную-счетчик оператора `for`? По величине шага?
9. Что такое вложенные циклы? Какие дополнительные условия необходимо соблюдать при их организации?

Задания

1. Составьте программу для вывода таблицы умножения целых чисел от 1 до 9 на любое целое число, которое вводится с клавиатуры.
2. Составьте программу, которая выводит все четные числа в диапазоне от 2 до 100 включительно и вычисляет их сумму.
3. Составьте программу вычисления суммы всех двузначных чисел.
4. Составьте программу для вычисления и вывода на экран таблицы значений функции. Вывод выполните в два столбца: первый — значения аргумента, второй — значения функции. При разработке программы следует учитывать область определения функции и в случае необходимости организовать вывод сообщения — "функция не определена" (табл. 9.2).

Таблица 9.2. *Таблица задания на вычисление значений функции*

№	Функция	Аргумент		
		Начальное	Конечное	Шаг
1.1	$F = \frac{3x + 4}{x^2 - 5x - 9}$	0,1	1,25	0,1
1.2	$F = \frac{x^2 - 3x + 2}{\sqrt{2x^3 - 1}}$	3	5,5	0,3
1.3	$F = \frac{62x^2 + 11x - 7}{\sqrt{2 - \lg x}}$	20	115	9

5. Составьте программу для вычисления суммы бесконечного сходящегося ряда. Суммирование прекратить при появлении в сумме слагаемых, имеющих абсолютную величину, меньшую заданной погрешности, значение которой вводится с клавиатуры.

2.1	$S = \frac{1}{3} + \frac{1}{6} + \frac{1}{9} + \frac{1}{12} + \ldots$
2.2	$S = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots$
2.3	$S = 1 - \frac{1}{3^2} + \frac{1}{5^2} - \frac{1}{7^2} + \ldots$

6. Составьте программу вычисления значения биномиального коэффициента

$$C_m^n = \frac{m!}{n!(m-n)!} = \frac{m(m-1)(m-2)...(m-n+1)}{n!}.$$

7. Составьте программу, которая требует у пользователя ввести числовой пароль. Если пароль правильный, то все строки экрана заполняются сообщением **Доступ разрешен**. После третьей неудачной попытки ввести пароль программа должна завершить работу с выдачей сообщения "В доступе отказано".

8. Составьте программу получения в порядке убывания всех делителей данного числа.

9. Составьте программу определения наибольшего общего делителя двух натуральных чисел.

10. Составьте программу определения наименьшего общего кратного двух натуральных чисел.

11. Найдите наибольшее и наименьшее значение функции $f(x) = 3x^2 + x - 4$, если на заданном интервале $[x_{min}; x_{max}]$ аргумент изменяется с шагом Δx.

12. В компьютерном классе имеется N рабочих мест. В течение дня первый компьютер работал m часов, а каждый последующий на 10 минут больше, чем предыдущий. Определите дневной фонд рабочего времени класса.

13. Билет называется "счастливым", если в его номере (шестизначном) сумма первых трех цифр равна сумме трех последних. Подсчитайте число "счастливых" билетов, у которых сумма цифр равна 13.

14. С клавиатуры по очереди вводятся координаты N точек. Определите, сколько из них попадает в круг радиуса r с центром в точке (x_0, y_0).

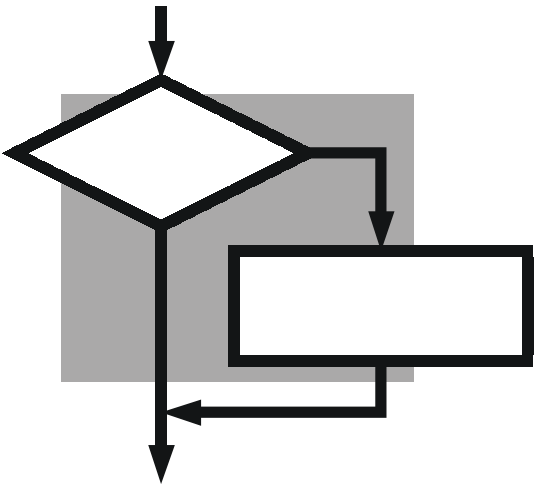
15. Каждая бактерия делится на две в течение одной минуты. В начальный момент имеется одна бактерия. Известно, сколько она весит. Составьте программу, которая рассчитывает количество бактерий, образующихся в результате деления через заданное количество минут. Определите вес получившейся биомассы.

16. Составьте программу вывода на экран всех простых чисел, не превосходящих заданного N. *Простым* называется натуральное число больше единицы, имеющее только два делителя: единицу и само это число.

17. В древности японский календарь использовал 60-летний цикл, состоявший из пяти 12-летних, которые обозначались цветами: зеленый, красный, желтый, белый и черный. Каждый год из числа двенадцати носил название определенного животного: крысы, коровы, тигра, зайца, дракона, змеи, лошади, овцы, обезьяны, петуха, собаки и свиньи. Составьте программу, которая вводит номер любого года нашей эры и печатает его

название, если известно, что, например, 1984 г. был годом зеленой крысы и началом очередного цикла.

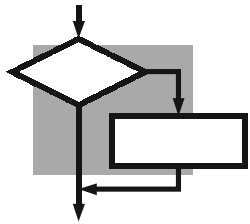
18. Определите k — количество трехзначных натуральных чисел, сумма цифр которых равна S. Число S задается с клавиатуры.
19. Напечатайте в убывающем порядке трехзначные числа от n_1 до n_2, в десятичной записи которых нет одинаковых цифр.
20. Выведите на экран все числа, которые являются полными квадратами в диапазоне от n_1 до n_2.
21. Определите, для какого наибольшего числа n можно вычислить $n!$, пользуясь типом `integer`.
22. Определите, какую наибольшую степень числа 3 можно вычислить, пользуясь типом `word (longint)`.
23. При вычислении суммы $\sum_{k=1}^{n} \frac{(-2)^k}{(k!)^2}$ для хранения значений слагаемых и суммы используется тип `real`, а для k и n — тип `longint`. Определите наибольшее допустимое значение n.
24. Составьте алгоритм и напишите программу, которая определяет сдачу с 50 рублей, возвращаемую покупателю, совершившему покупку стоимостью менее 50 рублей. Предполагается, что стоимость покупки выражается целым числом рублей. Количество банкнот (достоинством 10 рублей) и монет (достоинством 1, 2 и 5 рублей) должно быть минимальным.



ЧАСТЬ III

СТРУКТУРНОЕ И МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Глава 10



Процедуры и функции

10.1. Общие сведения о подпрограммах

Если в программе имеется несколько одинаковых (или очень похожих) фрагментов, то возникает естественный вопрос: нельзя ли оформить повторяющийся фрагмент в виде отдельного блока, а затем обращаться к нему в программе столько раз, сколько требуется. Аналогичная идея возникает и при отладке больших программ — если разделить программу на отдельные самостоятельные фрагменты, то отладить ее по частям будет проще. Эти разумные мысли имеют поддержку во всех языках высокого уровня, которые предоставляют программисту такое удобное средство разбиения программы на части, как *подпрограммы*.

Само название *подпрограмма* говорит о том, что это часть программы, которая оформляется как *самостоятельная программная единица*. Подпрограмма обязательно должна иметь *уникальное имя* (идентификатор), это имя и употребляется для *вызова подпрограммы*. Понятно, что такой вызов можно выполнять *многократно*, более того, одни и те же подпрограммы можно использовать в разных программах, отладив их код только один раз.

Многие программы, приведенные в этой книге, невелики по размерам и содержат десятки строк текста. Написать эти программы можно и без подпрограмм. Иное дело — создание проектов, насчитывающих тысячи и десятки тысяч строк, в которых задействованы целые команды программистов. Писать такие программы как единое целое, без разделения на самостоятельные фрагменты, просто невозможно.

В программах на Pascal используются подпрограммы двух видов: *процедуры* и *функции*. Имея один и тот же смысл и аналогичную структуру, они несколько различаются назначением и способом их использования.

Все процедуры и функции, в свою очередь, подразделяются на две группы:

- стандартные (встроенные);
- определенные пользователем.

Встроенные (стандартные) процедуры и функции входят в стандартные библиотеки и могут вызываться по имени *без предварительного описания* (например, процедуры ввода и вывода `read` и `write`). Некоторые наиболее часто используемые стандартные процедуры и функции были представлены выше *(см. гл. 7)*. Их наличие существенно облегчает разработку программ, и мы ими уже много раз пользовались.

Однако многие *специфичные* для данной программы действия не находят прямых аналогов в библиотеках Pascal, и тогда программисту приходится разрабатывать свои собственные процедуры и функции.

Процедуры и функции пользователя пишутся *самим* программистом и помещаются в *раздел описаний процедур и функций (см. разд. 6.3)*. Их вызов для выполнения записывается *в разделе операторов* основной программы или другой подпрограммы.

10.2. Процедуры

Процедура — это независимая именованная часть программы, которую после *однократного* описания можно *многократно* вызывать по *имени* из последующих частей программы. Вызов процедуры оформляется как отдельный оператор, процедура не может выступать как операнд в выражении.

Структура процедуры повторяет структуру программы, это "программа в миниатюре", состоящая из заголовка и тела. *В отличие от программы, для процедур и функций наличие заголовка обязательно.*

Заголовок состоит из зарезервированного слова `procedure`, идентификатора *(имени)* процедуры и *необязательного* списка *формальных параметров. Тело процедуры* по своей структуре аналогично обычной программе:

```
procedure ИмяПроцедуры(ФормальныеПараметры);
   { Описательная часть процедуры }
begin
   { Инструкции исполнительной части процедуры }
end;
```

Обратите внимание — в конце тела процедуры, как и в конце программы, стоит `end`, однако после `end` ставится *точка с запятой, а не точка.*

Для обращения к процедуре используется *оператор вызова процедуры*. Он состоит из *имени* процедуры и необязательного списка *фактических параметров*, отделенных друг от друга запятыми:

```
ИмяПроцедуры(ФактическиеПараметры);
```

При вызове процедуры работа главной программы *приостанавливается* и начинает выполняться *вызванная процедура*. Когда процедура выполнит свою

задачу, программа *продолжится* с оператора, следующего за оператором вызова процедуры.

Для *принудительного выхода* из процедуры в ее теле записывается оператор завершения exit, который обеспечивает выход во внешний блок (обычно — в основную программу).

Листинг 10.1 содержит программу с простейшей процедурой без параметров, которая выводит на экран горизонтальную линию и 3 раза вызывается в программе, выводящей таблицу перевода из сантиметров в дюймы.

Листинг 10.1. Программа перевода сантиметров в дюймы

```
var inch: integer;   { дюймы }
    sm: real;        { сантиметры }
{ процедура для вывода горизонтальной линии }
procedure horline;
var i: integer;
begin
   for i:=1 to 30 do write('-');
   writeln;
end;
{ основная программа }
begin
   horline;
   writeln('|    дюймы    |       см      |');
   horline;
   { считаем и выводим строки таблицы }
   for inch:=1 to 10 do begin
      sm:=2.54*inch;
      writeln('|',inch:13,'|', sm:14:3,'|');
   end;
   horline; readln;
end.
```

Усовершенствуем процедуру вывода линии, введя в нее два параметра: длину выводимой строки и символ, которым рисуется линия (листинг 10.2).

Листинг 10.2. Программа, демонстрирующая вывод линий

```
procedure horline(len:integer; s:char);
{ len — размер линии в символах, s — символ, которым рисуется линия }
```

```
var i: integer;
begin
   for i:=1 to len do write(s);
   writeln;
end;
{ основная программа выводит три совершенно разных линии }
begin
   horline(10,'-');
   horline(20,'*');
   horline(30,'#');
end.
```

Эти несложные примеры позволяют сделать первые выводы.

- Если в программе есть повторяющиеся фрагменты, оформление их в виде процедур позволит сократить текст основной программы.
- При использовании осмысленных и понятных имен процедур текст основной программы становится не только более коротким, но и более понятным и выразительным.
- Использование процедур позволяет упростить процесс внесения изменений в повторяющиеся фрагменты, т. к. исправления делаются *один раз* в процедуре, а не несколько раз в программе. Допустим, в примере из листинга 10.1 необходимо изменить символ, которым рисуется линия. Тогда достаточно внести только одно изменение в процедуру `horline`, чтобы достичь цели. Очень часто начинающие программисты при внесении одинаковых изменений в повторяющиеся фрагменты кое-где забывают внести изменения, и программа начинает неправильно работать. При использовании многократного вызова одной и той же процедуры такая ошибка в принципе исключена.
- Параметры для процедуры аналогичны исходным данным для программы. Использование параметров позволяет сделать процедуру более универсальной.

Результат работы процедуры из последнего примера (линия) в данном случае выводится на экран. Очевидно, можно представить себе ситуацию, когда процедура выполняет какие-либо вычисления, но по определенным причинам нет смысла сразу выводить результаты на экран. Тем не менее, они нужны программе для дальнейшей работы. В этом случае результаты работы процедуры можно передать в программу через параметры, как и исходные данные.

Для пояснения напишем еще одну простую процедуру, которая из заданной суммы денег (действительное число) выделяет рубли и копейки (два целых

числа). Конечно, можно вывести результат на экран в процедуре, но разумнее сохранить его в двух переменных, через параметры передать в программу, а затем использовать для любых действий, в том числе и для вывода. Кстати, скорее всего результаты этой процедуры потребуются для дальнейших расчетов (в банковских операциях расчеты выполняются с точностью до копейки), да и выделять рубли и копейки приходится довольно часто.

```
procedure rubkop(summa:real; var rub,kop:integer);
begin
   rub:= trunc(summa); kop:= round((summa-rub)*100);
end;
```

Обратиться к данной процедуре можно, например, так:

```
rubkop(123.76, r,k);
```

где `r` и `k` — переменные целого типа, в которые будут занесены результаты. Вообще, это могут быть переменные целого типа с любыми именами.

В общих чертах, наверное, уже понятно, что параметры служат для обмена данными между программой и подпрограммой. Программа передает подпрограмме какие-то входные данные (если они требуются), а от подпрограммы получает результаты в виде значений переменных. Иногда подпрограмма не требует входных данных, иногда не возвращает результатов, а иногда и вообще не содержит параметров, как в самом первом из приведенных примеров.

Передача параметров — дело непростое, имеющее много нюансов, трудных для начинающих. Попробуем разобрать все по порядку.

10.3. Механизм передачи параметров

Параметры, которые указываются при описании процедур и функций, называются *формальными*. Название "формальные" параметры получили в связи с тем, что они нужны только для записи алгоритма, а при вызове подпрограммы на их место будут *подставлены конкретные фактические параметры*.

Соответствие между формальными и фактическими параметрами обеспечивается выполнением следующих требований:

- формальных и фактических параметров должно быть *одинаковое количество*;
- *порядок следования* фактических и формальных параметров должен быть *один и тот же*;
- *тип* фактического параметра *должен быть совместим* с типом соответствующего ему формального параметра (о совместимости типов см. *разд. 7.4*).

Обратите внимание — в языке Pascal типом любого параметра в списке формальных параметров может быть только стандартный или ранее объявленный тип.

Например, нельзя объявить процедуру с заголовком:

```
procedure proc(m,n: integer; k: 1..5);
```

Для передачи в процедуру параметра k необходимо предварительно описать его тип в разделе описаний основной программы.

```
type range = 1..5;
procedure proc(m,n: integer; k: range);
...
```

Имеются два основных вида параметров, которые отличаются способом их передачи в подпрограмму:

- параметры-значения;
- параметры-переменные.

Начиная с версии Turbo Pascal 7.0, появилась еще одна разновидность параметров — параметры-константы. В Delphi они также используются.

10.3.1. Параметры-значения

Параметры-значения используются только для передачи исходных данных из основной программы в процедуру или функцию. По умолчанию параметры подпрограммы считаются именно параметрами-значениями.

Обратите внимание — в данном случае подпрограмме передается лишь *значение параметра, которое помещается в переменную, специально созданную для этой цели.* Таким образом, в подпрограмме используется *копия фактического параметра.* В теле подпрограммы формальные параметры могут изменяться, но это никак не отразится на фактических параметрах, т. к. *меняется их копия.*

Поэтому параметры-значения *нельзя использовать для передачи результатов* из подпрограммы в основную программу. К сожалению, нарушение этого требования — это типичная ошибка начинающих, которая не обнаруживается компилятором, но выражается в неверных результатах работы программы. Ошибки, которые трудно обнаружить, лучше совсем не делать, поэтому будьте внимательны к параметрам подпрограмм.

Зато фактическим параметром-значением может быть не только константа или переменная с заданным значением, но и *произвольное выражение. Перед вызовом* подпрограммы фактические параметры вычисляются, и полученное значение передается в подпрограмму — выполняется *подстановка значений.*

Для пояснения рассмотрим следующую программу:

```
var a,b: real;
{ процедура вычисления и вывода }
procedure square(x,y: real); { x,y — параметры-значения }
```

```
begin
   x:= x*x; y:= y*y;
   writeln(x:7:3,'   ',y:7:3);
end;      { конец процедуры }
begin
   a:=1.0; b:=3.0;
   square(a,b); { вызов процедуры с фактическими параметрами a и b }
   writeln(a:7:3,'   ',b:7:3);
end.
```

При выполнении программа выведет на экран:

```
1.000     9.000
1.000     3.000
```

При вызове процедуры square с фактическими параметрами a, b значения этих параметров один раз копируются в соответствующие формальные параметры x, y, и дальнейшие преобразования формальных параметров x, y внутри процедуры square уже никак не влияют на значения переменных a, b.

10.3.2. Параметры-переменные

Параметры-переменные необходимо использовать, прежде всего, для *возврата результатов работы* подпрограммы в основную программу.

Обратите внимание — в списке формальных параметров перед ними ставится ключевое слово var.

Каждому формальному параметру-переменной должен соответствовать фактический параметр обязательно в виде *переменной*. Выражение здесь недопустимо, т. к. результаты подпрограммы могут быть сохранены только в переменных.

В этом случае при вызове подпрограммы ей *передается адрес фактического параметра в памяти* и в дальнейшем подпрограмма работает именно с этой ячейкой памяти, а не с копией, как при использовании параметра-значения.

Вернемся к нашему последнему примеру, но используем теперь параметры-переменные.

```
var a,b: real;
procedure square(var x,y: real); { x,y — параметры-переменные }
begin
   x:= x*x; y:= y*y;
   writeln(x:7:3,'   ',y:7:3);
end;
```

```
begin
   a:=1.0; b:=3.0;
   square(a,b);
   writeln(a:7:3,'   ',b:7:3);
end.
```

При выполнении программа выведет на экран:

```
1.000     9.000
1.000     9.000
```

Обратите внимание — *исходные данные* в подпрограмму могут передаваться как через параметры-значения, так и через параметры-переменные, а *результаты работы* подпрограммы возвращаются в программу *только* через параметры-переменные.

Замечание

Начинающий программист может поторопиться сделать такой вывод: если параметры-переменные могут быть использованы и для исходных данных, и для результатов, то, пожалуй, без лишней путаницы стоит записывать все параметры со словом `var`. Не советуем привыкать к подобной практике. Причины разъясним чуть позже.

10.3.3. Параметры-константы

Мы выяснили, что параметр можно передавать или по значению, или по адресу в памяти. А какой еще способ можно изобрести?

Конечно, способов всего два, а параметры-константы передаются по адресу, как параметры-переменные, но значения их запрещено изменять в подпрограмме. За этим строго следит компилятор, при нарушении запрета он выдает сообщение об ошибке **Invalid variable reference** (Недопустимое обращение к переменной).

При описании подпрограммы перед параметрами-константами добавляется служебное слово `const`, например: `procedure proc(const p:integer);`.

Параметры-константы — это очень хороший способ передачи исходных данных в подпрограмму, т. к. он не требует дополнительных затрат памяти для хранения копий (во многих случаях для хранения адреса требуется меньше памяти, чем для хранения самого значения). При этом сами исходные данные в своих ячейках памяти остаются в неприкосновенности. Главное, не лениться и добавлять в заголовок процедуры волшебное слово `const` перед входными параметрами (хотя сами авторы в своих примерах часто забывают слово `const`, считая, что на нескольких байтах памяти много не сэкономишь).

Помимо всего, использование параметров-констант — это еще и признак хорошего стиля программирования, а также средство некоторого ускорения работы программы (со стековой памятью работа обычно выполняется немного медленнее, чем с памятью глобальных переменных).

10.4. Функции

Если результатом подпрограммы является только одно значение, то имеет смысл оформить такую подпрограмму не в виде процедуры, а в виде функции. *Функция пользователя* во всем аналогична процедуре, за исключением двух отличий:

- функция передает в программу результат своей работы — единственное значение, носителем которого является имя самой функции;
- имя функции может входить в выражение как операнд.

Функция, определенная пользователем, состоит из заголовка и тела функции. *Заголовок* содержит зарезервированное слово `function`, *имя* функции, *необязательный* список *формальных* параметров и, обратите внимание — в отличие от процедуры, *тип* возвращаемого функцией значения. *Тело функции* по своей структуре аналогично обычной программе:

```
function ИмяФункции(ФормальныеПараметры) : ТипРезультата;
   { Описательная часть функции }
begin
   { Инструкции исполнительной части функции }
   ИмяФункции := Результат;
end;
```

Обратите внимание — в разделе операторов функции должен находиться по крайней мере один оператор, который присваивает ее имени значение *результата работы* функции. Если таких присваиваний *несколько*, то результатом функции будет значение *последнего* выполненного оператора присваивания. Если же такой оператор отсутствует, то значение, возвращаемое функцией, *не определено* (компилятор не выдает сообщения об ошибке, но результат программы, скорее всего, будет неверным).

В отличие от процедуры, вызов функции не оформляется в виде отдельного оператора. Обращение к функции пользователя осуществляется аналогично обращению к стандартной функции, т. е. *посредством ее использования в выражении*. Можно считать, что разработка собственных функций позволяет расширить список стандартных функций, доступных программе.

Замечание

В Turbo Pascal 7.0 и Delphi разрешено оформлять вызов функции пользователя в виде отдельного оператора. Однако такой вызов — скорее исключение, чем правило.

В качестве примера опишем функцию, выполняющую возведение любого числа в целую неотрицательную степень (вспомним, что в языке Pascal отсутствует подобная *стандартная* операция). С помощью данной функции определим число байтов, содержащихся в килобайте, мегабайте и гигабайте, используя известные соотношения: 1 Кбайт = 2^{10} байт, 1 Мбайт = 2^{20} байт, 1 Гбайт = 2^{30} байт (листинг 10.3).

Листинг 10.3. Программа возведения числа в целую неотрицательную степень

```
function power(const x: real; const n: byte):real;
{ функция возведения числа x в целую неотрицательную степень n }
var i: integer; res:real;
begin
   res:=1;
   for i:=1 to n do res:=res*x;
   power:=res; {не забыли присвоить результат функции}
end;
{ основная программа }
begin
   writeln('1 Kb = ',power(2,10):4:0,' byte');
   writeln('1 Mb = ', power(2,20):7:0,' byte');
   writeln('1 Gb = ', power(2,30):10:0,' byte');
   readln;
end.
```

Замечание

Кстати, в некоторых диалектах Pascal имеется стандартная функция power с такими же параметрами, но оба параметра — действительные числа. Для возведения в любую действительную степень можно воспользоваться свойствами логарифмов *(см. разд. 7.5)*.

Рассмотрим еще один пример, который существенно расширит наши представления о функциях. Ранее уже приводилась программа, реализующая решение квадратного уравнения (см. листинг 8.5). Выполним ту же задачу с использованием функции (листинг 10.4).

Листинг 10.4. Решение квадратного уравнения (вариант № 2)

```
function kvadrur(const a,b,c: real; var x1,x2: real): integer;
{ функция для решения квадратного уравнения }
{ a,b,c — коэффициенты уравнения, x1,x2 — корни уравнения }
{ значение функции — количество корней или -1, если a=0 }
var d: real;   { дискриминант }
begin
   if a=0 then kvadrur:=-1
   else
      begin
         d:=b*b-4*a*c;
         if d<0 then kvadrur:=0   { вещественного решения нет }
             else
                begin
                   if d>0 then kvadrur:=2  { два разных корня }
                          else kvadrur:=1; { корни одинаковые }
                   x1:=(-b+sqrt(d))/(2*a);
                   x2:=(-b-sqrt(d))/(2*a);
                end;
      end;
end; {конец функции}
var a,b,c: real;   { коэффициенты уравнения }
   x1,x2: real;   { корни уравнения }
begin
   writeln('Введите в одной строке коэффициенты a, b, c'); readln(a,b,c);
   case kvadrur(a,b,c,x1,x2) of
     -1: writeln('Уравнение не квадратное');
      0: writeln('Уравнение не имеет вещественных корней');
      1: writeln('x=',x1:7:3,', корни одинаковы');
      2: writeln('x1=',x1:7:3,'; x2=',x2:7:3);
   end; readln;
end.
```

Обратите внимание — функция `kvadrur` возвращает не один результат, а три, причем один из них (количество корней) передается как результат функции и используется в выражении оператора `case`, а переменные `x1` и `x2` *передаются как параметры-переменные.*

Анализ приведенной программы дает основания считать функции *более универсальным видом подпрограмм*, чем процедуры, тем более что в современных версиях Pascal разрешен прямой вызов функций (по типу вызова процедуры). В некоторых языках (C/C++ и им подобных) вообще отсутствуют процедуры, и единственным видом подпрограмм являются функции.

Тем не менее, в таких языках, как Pascal, в функциях довольно редко используются параметры-переменные, т. к. для этой цели лучше подходят процедуры. По крайней мере, все стандартные функции запрограммированы таким образом, что они имеют только параметры-значения и возвращают один результат. Очевидно, что и собственные функции стоит писать по аналогии со стандартными. Однако возможность использования в функции параметров-переменных возьмите на заметку. При дальнейшем изложении материала мы в примерах иногда будем обращаться к таким функциям.

Мы подошли к обсуждению довольно сложных вопросов, касающихся использования подпрограмм. Следующие разделы потребуют повышенного внимания, и разобраться в них нужно обязательно. Дело в том, что основные принципы использования подпрограмм одинаковы во всех языках программирования, и полученные знания пригодятся независимо от того, каким языком вам придется пользоваться в дальнейшем. В крайнем случае можно сначала пропустить непонятные места с тем, чтобы вернуться к ним позже.

10.5. Область видимости и время жизни переменной

Все объекты (метки, константы, типы, переменные, процедуры и функции), которые описываются в теле подпрограммы, являются *локальными объектами*, т. е. доступны *только* в пределах этой подпрограммы и недоступны вызывающей программе. Иначе говорят, что они "видимы" в той подпрограмме, в которой описаны. Попробуем разобраться, как выделяется память для данных подпрограммы.

10.5.1. Подробнее о распределении памяти

При запуске любой программы для нее формируется специальная область памяти, которая называется *программным стеком (подробнее об устройстве стековой памяти см. разд. 18.6.2)*. Эта область используется для хранения локальных переменных подпрограмм, которые помещаются в стек непосредственно при вызове подпрограммы.

Параметры подпрограммы также помещаются в стек, причем для параметров-значений в стек помещается *само значение*, а для параметров-переменных — *адрес фактического* параметра, в котором хранится значение. Это

объясняет все рассмотренные ранее различия между параметрами-значениями и параметрами-переменными (параметрами-константами).

Кроме того, в стек помещается *точка возврата*, т. е. адрес инструкции вызывающей программы, к которой нужно вернуться после окончания работы подпрограммы.

Как только подпрограмма закончит работу, вся память, отведенная для нее в стеке, считается свободной и в дальнейшем может использоваться для данных других подпрограмм. Так обеспечивается *независимость* подпрограмм друг от друга и от основной программы.

Стек иначе называют *автоматической памятью программ* (очевидно, потому, что она автоматически выделяется и освобождается). Аналогично локальные переменные иначе называются автоматическими.

Все объекты, описанные в вызывающей программе, называются *глобальными*. Память под глобальные переменные выделяется при компиляции программы. Эта часть памяти называется *сегментом данных*. Иначе ее называют *глобальной* или *статической памятью*.

Глобальные переменные находятся в сегменте данных от начала до конца выполнения программы, поэтому *ими можно пользоваться и в программе, и во всех подпрограммах*, к которым обращается программа.

Следует знать:

- *обмен данными* между программой и подпрограммой может производиться не только через параметры, но и *через глобальные переменные*;
- если *одно и то же имя* определено и в программе, и в подпрограмме, то внутри подпрограммы глобальный объект *недоступен*, т. е. он как бы *экранируется (маскируется)* локальным объектом с таким же именем;
- если *одно и то же имя* определено в *нескольких* подпрограммах одной программы, то в каждой подпрограмме это будут разные локальные объекты, не зависящие друг от друга (если подпрограммы не вложены одна в другую).

Для пояснения рассмотрим следующую программу:

```
var a,b,c,d: integer;
procedure p(x: integer; var y: integer);
var c: integer;
begin
   c:=1;d:=1;x:=1;y:=1;
   writeln(x:3,y:3,c:3,d:3);
end;
begin
   a:=0;b:=0;c:=0;d:=0;
```

```
   p(a,b);
   writeln(a:3,b:3,c:3,d:3);
end.
```

При выполнении программа напечатает строки:

```
1  1  1  1
0  1  0  1
```

Комментарий

x — формальный параметр-значение, которому соответствует фактический параметр a=0. В процедуре его значение заменится на 1, после чего результат будет напечатан. На переменной a это никак не отразится, и после выполнения процедуры a по-прежнему будет равно нулю. y — параметр-переменная, поэтому при выполнении процедуры вместо y действия будут проводиться с переменной b, которая получит значение 1. c — локальная переменная, которая маскирует глобальную переменную с основной программы. d — глобальная переменная.

10.5.2. Вложенные процедуры и функции

В Pascal допускается *любой уровень вложенности процедур и функций*. Процедура, описанная в основной программе, в свою очередь, может содержать описания внутренних процедур и функций. При этом объекты, описанные в вызывающей процедуре, являются *глобальными по отношению к вызываемой процедуре*.

Схематическое изображение структуры блоков некоторой программы на Pascal приведено на рис. 10.1.

Для доступа к объектам, описанным в различных блоках, требуется соблюдать следующие правила:

- имена объектов, описанных в некотором блоке, считаются известными в пределах данного блока, включая и все вложенные блоки;
- имена объектов, описанных в блоке, должны быть уникальны в пределах данного блока и могут совпадать с именами объектов из других блоков;
- если в некотором блоке описан объект, имя которого совпадает с именем объекта, описанного в вышестоящем блоке, то это последнее имя становится недоступным в данном блоке.

Если применить эти правила к схеме на рис. 10.1, можно сказать, что объекты, описанные в блоке *B*, известны (видимы), кроме самого блока *B*, еще и в блоках *C* и *D*, но невидимы в блоке *A*. Объекты, описанные в блоке *F*, видимы только в пределах этого блока.

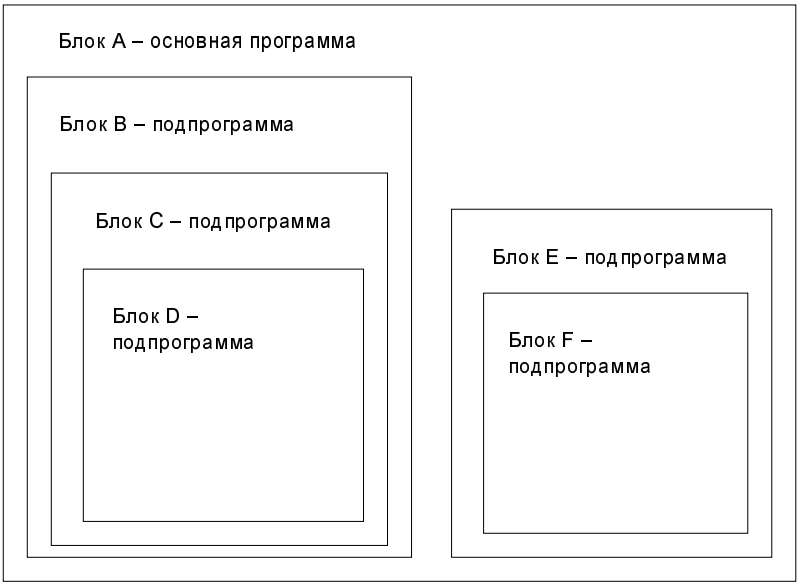


Рис. 10.1. Иллюстрация к области действия параметров

10.5.3. Глобальные, автоматические и статические переменные. Определения

Все-таки попробуем определить такие фундаментальные понятия программирования, как область видимости и время жизни переменных. Выше мы только коснулись их. Пока получается такая картина: глобальные переменные видимы во всех блоках и "живут" от начала до конца работы программы. Локальные (автоматические) переменные видимы только в тех блоках, в которых определены (и во вложенных блоках). Они автоматически создаются в стеке при входе в блок и уничтожаются при выходе из блока. Получается, что время их жизни и область их видимости тесно связаны между собой.

На самом деле все несколько сложнее, поскольку имеются и другие виды переменных. Переменные, которые получили название *статических*, имеют локальную область видимости, но "живут" от начала до конца работы программы, т. к. память для них выделяется не в стеке, а в сегменте данных. Для программиста это означает следующее — значения статических переменных при выходе из подпрограммы не теряются и при следующем вызове подпрограммы можно будет использовать старое значение.

В стандарт Pascal статические переменные не включены, но в диалектах фирмы Borland они есть и оформляются как локальные типизированные константы, которые в Turbo Pascal играют роль переменных, инициализиро-

ванных начальными значениями *(см. разд. 6.3)*. Например, рассмотрим такую программу:

```
procedure a;
const x:integer=0;
begin x:=x+1; write(x:2); end;
begin
   a; a; a; readln;
end.
```

При выполнении программы в среде Turbo Pascal будет выведено: **1 2 3**, что соответствует утверждению о том, что выделение памяти и начальная инициализация переменной x выполняются только при первом вызове процедуры a.

Замечание

При выполнении этой же программы в среде Delphi компилятор выдаст сообщение об ошибке, поскольку в Delphi типизированные константы — это просто константы, занимающие память, и их значение нельзя менять. Однако память под них все-таки отводится в сегменте данных как для статических переменных. С помощью небольшого трюка можно обмануть компилятор, обратившись к переменной x непосредственно по ее адресу *(подробнее о работе с адресами см. разд. 18.1)*:

```
procedure a;
const x:integer=0;
var p:^integer;{так можно описать переменную для хранения адреса}
begin
   p:=@x;{используем операцию взятия адреса}
   p^:=p^+1;{обращаемся к переменной по адресу p, т. е. к x}
   write(x:2);
end;
begin
   a; a; a; readln; {снова увидим на экране 1 2 3}
end.
```

Убедительный совет — никогда не используйте подобных трюков в серьезных программах. Обманывать всегда нехорошо, даже если мы обманываем компилятор.

В *гл. 12*, посвященной модулям, будет рассмотрен еще один случай использования статических переменных в программах на Pascal, с нашей точки зрения, более логичный и понятный.

А теперь вернемся к определениям. *Следует знать:*

- область видимости переменной — это программный блок или совокупность блоков, в которых можно использовать данную переменную;
- время жизни — это время, в течение которого переменной поставлена в соответствие ячейка памяти.

Материал этого раздела еще расширяет наши понятия о переменных. Теперь мы уже можем перечислить полную "шестерку" характеристик (атрибутов) переменной. Это: имя, тип, значение, адрес в памяти, область видимости и время жизни [20]. Разумеется, такое толкование не противоречит первоначальному определению переменной как ячейки памяти. Пожалуй, стоит уточнить это определение только одним словом: переменная — это *абстракция ячейки памяти*. К принципу абстракции в программировании мы еще вернемся.

10.5.4. Побочные эффекты вызова подпрограмм

Побочными эффектами называют изменения глобальных переменных при вызове подпрограмм. Побочным эффектом вызова функции считается изменение параметров-переменных (если они имеются).

Для пояснения рассмотрим следующую программу:

```
var a,b: integer;
function d(x: integer): integer;
begin
   d:=a+x; a:=a+1;
end;
begin
   a:=1; b:=d(1)+d(a);
   writeln(b);   { результат 5, т.к. d(a) вызывается раньше, чем d(1) }
   a:=1; b:=d(a)+d(1);
   writeln(b);   { результат 6, т.к. d(1) вызывается раньше, чем d(a) }
end.
```

Комментарий

Глобальная переменная a изменяется в теле функции d. Поэтому оказывается, что `d(1)+d(a)` ≠ `d(a)+d(1)`.

Иногда программисты сознательно используют побочные эффекты для достижения нужных результатов. Однако не советуем увлекаться такой возможностью, поскольку в этом случае программы становятся очень неудобными для понимания и внесения изменений. К тому же нет гарантий, что

любой компилятор одинаково справится с программой, содержащей трюки, подобные описанному выше.

10.5.5. Рекомендации по разработке программ

Учитывая сказанное выше, примите следующие советы:

- при разработке программ с большим количеством подпрограмм всегда стремитесь сократить число глобальных переменных до минимума, давайте им осмысленные имена и держите каждую из них под постоянным контролем;
- все рабочие переменные, которые используются в подпрограмме для реализации ее алгоритма, описывайте как локальные; при этом имена переменных могут быть любыми, но желательно не совпадающими с именами глобальных переменных (во избежание лишней головной боли);
- если планируется использование подпрограммы в нескольких разных программах, в ней вообще не должны использоваться глобальные переменные, т. е. она должна быть полностью *автономной* и должна обмениваться данными с основной программой только через параметры;
- все входные параметры подпрограммы описывайте как параметры-значения, а еще лучше — как параметры-константы, а с параметрами-переменными будьте особенно внимательны и осторожны;
- избегайте вложенных процедур, т. к. при их использовании вероятность нечаянно изменить значение переменной резко возрастает.

10.6. Рекурсия

Рекурсия — это такой способ организации вычислительного процесса, при котором процедура или функция в ходе выполнения обращается *сама к себе.*

Рекурсия широко применяется в математике. В качестве примера дадим рекурсивное определение суммы первых n натуральных чисел. Сумма первых n натуральных чисел равна сумме первых $(n - 1)$ натуральных чисел плюс n, а сумма первого числа равна 1. Или: $S_n = S_{n-1} + n$; $S_1 = 1$.

Опишем функцию, которая вычисляет сумму, пользуясь данным определением:

```
function sum(n: integer): integer;
begin
   if n=1 then sum:=1
          else sum:=sum(n-1)+n;
end;
```

Текст функции получился необычным и может вызвать сомнения в его правильности: суммируется большое количество слагаемых, но при этом не используется цикл. Все правильно, операция суммирования будет повторяться ровно `n-1` раз, поскольку функция `sum` обращается `n-1` раз к самой себе, при этом параметр каждый раз уменьшается на единицу. В конце концов параметр станет равным 1, и рекурсивные вызовы закончатся, после чего все уже вызванные функции одна за другой завершат свою работу.

Обратите внимание на следующие обстоятельства.

Во-первых, рекурсивная процедура или функция содержит всегда, по крайней мере, одну терминальную ветвь и условие окончания (`if n=1 then sum=1`).

Во-вторых, при выполнении рекурсивной ветви (`else sum:=sum(n-1)+n`) процесс выполнения подпрограммы приостанавливается, но его переменные не удаляются из стека. Происходит новый вызов подпрограммы, переменные которой также помещаются в стек, и т. д. Так образуется последовательность прерванных процессов, из которых выполняется всегда последний, а по окончании его работы продолжает выполняться предыдущий процесс. Целиком весь процесс считается выполненным, когда стек опустеет, или, другими словами, все прерванные процессы выполнятся.

Рекурсия *полезна*, прежде всего, в случаях, когда основную задачу можно разделить на подзадачи, имеющие ту же структуру, что и первоначальная задача.

Не следует путать рекурсию с итерацией, которая реализуется при помощи циклов.

Большинство алгоритмов можно реализовать двумя способами: итерацией и рекурсией. Вспомним пример с суммой, который очень легко реализуется при помощи цикла.

Особенности рекурсии:

- рекурсивная форма организации алгоритма обычно выглядит *изящнее* итерационной и дает более *компактный текст программы*;
- *недостатки* рекурсии состоят в следующем:
  - если глубина рекурсии очень велика, то это может привести к переполнению стека;
  - рекурсивные алгоритмы, как правило, выполняются более медленно;
- при рекурсивном программировании велика вероятность ошибок зацикливания. В целях повышения безопасности работы рекомендуется:
  - использовать директиву компилятора `{$S+}` для *включения проверки переполнения стека*;
  - использовать директиву компилятора `{$R+}` для *включения проверки диапазона*.

Рекурсия — это очень мощный инструмент решения многих задач. Но пользоваться им следует достаточно сдержанно и осторожно. Если задача имеет очевидное нерекурсивное решение, то следует избрать именно его. Вряд ли рекурсию можно рекомендовать для нахождения суммы. Рекурсивная функция sum приведена нами лишь в целях иллюстрации механизма рекурсии.

Листинг 10.5 содержит красивое рекурсивное решение задачи выделения цифр целого положительного числа, которая уже решалась при помощи цикла (см. листинг 9.8). В данном примере цифры выводятся в обратном порядке.

Листинг 10.5. Вывод цифр целого положительного числа в обратном порядке

```
var n: integer;
procedure reverse(n: integer);
begin
   write(n mod 10);
   if (n div 10) <> 0 then reverse (n div 10)
end;
begin
   writeln('Введите целое положительное число <= ',maxint); readln(n);
   reverse(n); writeln; readln;
end.
```

10.7. Дополнительные сведения о процедурах и функциях

10.7.1. Опережающее объявление

Помимо *прямых рекурсий*, рассмотренных выше, существуют рекурсии с так называемым *косвенным* вызовом, при котором одна подпрограмма вызывает другую подпрограмму, вызывающущю, в свою очередь, первую подпрограмму.

А поскольку подпрограмма обязательно должна быть описана до ее вызова, должно использоваться *опережающее описание заголовка подпрограммы.* Описывается один только заголовок, вслед за которым ставится зарезервированное слово forward (опережающий), а полное описание *процедуры* помещается *далее* в тексте программы. Такое описание называется *определяющим. В определяющем описании можно не указывать формальные параметры,* т. к. они уже известны из опережающего описания.

Например:

```
var a,b: real;
{ опережающее описание процедуры p1 }
procedure p1(x: real); forward;
{ описание процедуры p2 }
procedure p2(y: real);
begin
...
p1(a);   { вызов процедуры p1 }
...
end;
{ текст процедуры p1 без описания предварительно описанных параметров }
procedure p1
begin
...
   p2(a);   { вызов процедуры p2 }
...
end;
begin
   p2(a); p1(b);   { вызов процедур p2 и p1 }
end.
```

Как видно из примера, опережающее описание процедуры `p1` позволило использовать обращение к ней из процедуры `p2`, т. к. при трансляции компилятору еще до вызова процедуры `p1` из опережающего описания становятся известны ее формальные параметры, и он может правильно организовать ее вызов.

Пример косвенной рекурсии и опережающего объявления приводится в *разд. 14.4.4*.

10.7.2. Параметры-процедуры и параметры-функции

В языке Pascal параметрами подпрограмм могут быть не только переменные, но и подпрограммы. Для обеспечения такой возможности вводятся *процедурные типы*.

Как только процедурный тип определен, можно объявлять переменные этого типа. Такие переменные называются *процедурными переменными* и могут быть использованы двумя способами:

- в качестве формальных параметров при описании процедур и функций;

- в качестве переменной, которой можно присвоить имя процедуры или функции.

В обоих случаях встает вопрос о совместимости типов. Для того чтобы считаться *совместимыми*, процедурные типы должны иметь *одинаковое число параметров одинаковых типов*, причем в случае функции типы результатов должны быть идентичными. Кроме того, процедура или функция, которая используется в качестве параметра подпрограммы или чье имя присваивается процедурной переменной, должна удовлетворять следующим требованиям:

- она не должна быть стандартной процедурой или функцией. Ниже мы покажем, как легко обойти это ограничение;
- она не должна быть вложенной функцией;
- в среде Turbo Pascal она должна быть откомпилирована с директивой {$F+} — дальний вызов (это связано с особенностями операционной системы DOS).

Параметры процедурного типа особенно полезны в ситуациях, когда над несколькими процедурами или функциями выполняются *общие действия* (например, строится таблица или изображается график нескольких функций).

Рассмотрим простой пример. Пусть надо вывести таблицы значений трех функций: `Sin(x)`, `Sin(x)*Cos(x)` и `Sin(x)/x`. Напишем незамысловатую процедуру вывода таблицы значений функции, использующую функцию в качестве параметра. Тогда для решения задачи достаточно будет вызвать ее три раза с разными значениями параметров. Текст программы приведен в листинге 10.6.

Листинг 10.6. Демонстрация процедурных типов

```
type func=function(x:real):real; {}
{$F+} {эта директива нужна только для 16-разрядных компиляторов}
function f1(x:real):real;
begin f1:=sin(x); end;{стандартную функцию оформляем как обычную функцию}
function f2(x:real):real;
begin f2:=sin(x)*cos(x); end;
function f3(x:real):real;
begin f3:=sin(x)/x; end;
{$F-} {отключаем дальний вызов функций, если он был включен}
{вывод таблицы произвольной функции f на отрезке [a,b]с шагом h}
procedure fun_table(f:func;a,b,h:real);
var x:real;
begin
   x:=a;
   while x<=b do begin
```

```
      writeln(x:8:2,'   ',f(x):8:2); x:=x+h
   end;
   writeln;
end; {процедуры вывода таблицы функции}
begin
{выводим таблицы трех функций f1, f2 и f3 на разных отрезках}
   fun_table(f1,0,3.14,0.5);
   fun_table(f2,0,1.57,0.2);
   fun_table(f3,-5,5,2);
   readln;
end.
```

Замечание

Если требуется вывести не все таблицы, а лишь одну из них на выбор, можно воспользоваться процедурной переменной. Тогда основная программа примет вид:

```
var f:func; k:integer;
  begin
  write('Введите номер функции '); readln(k);
  case k of
     1:f:=f1; 2:f:=f2; 3:f:=f3
  end;
   fun_table(f,0,3.14,0.5); readln;
end.
```

Еще один пример использования процедурных типов имеется в листинге 10.11.

10.7.3. Нетипизированные параметры-переменные

Параметр считается *нетипизированным*, если его тип в заголовке подпрограммы не указан и при этом соответствующий ему фактический параметр может быть переменной любого типа. Нетипизированными могут быть только параметры-переменные.

Поскольку Pascal — строго типизированный язык, в котором при выполнении любых операций обязательно контролируется допустимость типов операндов, внутри подпрограммы нетипизированный параметр в большинстве

случаев приводится к какому-то определенному типу. Сделать это можно двумя способами:

- автоопределенным преобразованием типов;
- наложенным описанием переменной определенного типа.

При автоопределенном преобразовании *(см. разд. 7.4)* тип задают явно, например:

```
procedure proc(var x);
y:=integer(x);
```

Для наложения переменной определенного типа используют ключевое слово `absolute`, за которым следует имя ранее определенной переменной. Таким способом выполняется совмещение в памяти данных с разными именами и, возможно, разных типов. Например:

```
procedure proc(var x);
var r:real absolute x;
```

В данном примере переменные `r` и `x` имеют один и тот же адрес в памяти. Можно считать, что одной ячейке памяти присвоены два разных имени.

Необходимость приведения нетипизированных параметров к конкретному типу ограничивает их применение и связывает руки программисту. Поэтому мы не можем рекомендовать их частому использованию. Однако интересный пример использования нетипизированного параметра все же будет приведен в *гл. 13* (см. листинг. 13.9, процедура `print3`).

Замечание

Существуют ситуации, когда преобразование нетипизированного параметра к конкретному типу не требуется. В таких случаях в подпрограмме над этим параметром не должно выполняться никаких операций, требующих контроля типа.

10.8. Примеры программ

Умение грамотно и логично выделять подпрограммы ("структурировать" программу) требует определенного опыта и интуиции. К сожалению, невозможно сформулировать четкие правила, соблюдая которые, ты обязательно получишь идеально структурированную программу. Поэтому основное внимание авторов при изложении материала было сосредоточено на тех возможностях, которые предоставляет язык Pascal. Способы же использования этих возможностей можно уяснить только на примерах. Самое главное для программиста — привыкнуть использовать подпрограммы даже в небольших задачах, а опыт и интуиция обязательно появятся со временем.

10.8.1. Числовые последовательности

Листинг 10.7 содержит программу, определяющую количество сверхпростых чисел в натуральном ряду чисел, не превышающих 1000.

Замечание

Сверхпростым называется число, если оно простое, и число, полученное из него записью цифр в обратном порядке (палиндром), тоже будет простым (напомним, что простое число не имеет других делителей, кроме единицы и самого себя). Например, 13 и 31 — сверхпростые числа.

Листинг 10.7. Определение количества сверхпростых чисел

```
var x,k: integer; { x  - параметр цикла,k - количество простых чисел}
function prost(y: integer): boolean;
{проверка, является ли y простым числом}
var i: integer;
begin
   prost:=true;
   for i:=2 to round(sqrt(y)) do if y mod i=0 then prost:=false;
end;
function povorot(y: integer): integer; {получение палиндрома числа}
begin
   if y<10  then povorot:=y else
   if y<100 then povorot:=y mod 10*10+y div 10 else
                 povorot:=y mod 10*100+y mod 100 div 10*10+y div 100;
end;
begin
   writeln('Сверхпростые числа':50);
   k:=2;   { 2 и 3 - простые числа }
   for x:=4 to 999 do
      if prost(x) then
         if prost(povorot(x)) then begin
            writeln(x);   { вывод сверхпростого числа }
            k:=k+1;
            if k mod 24=0 then readln; {пауза - экран заполнен}
         end;
   writeln('всего найдено ',k,' сверхпростых чисел < 1000'); readln;
end.
```

Комментарий

В этой задаче было очень легко выделить подпрограммы, поскольку очевидны две самостоятельные подзадачи: проверка на простоту и поворот числа. Использование вложенного вызова функции `prost(povorot(x))` позволило записать текст программы компактно.

Наверное, текст функции проверки числа на простоту вызвал удивление — почему проверяется делимость только на числа, не превышающие квадратного корня из проверяемого числа. Можно доказать, что этого достаточно. Действительно, если число имеет делитель больший корня, то частное от деления (а оно тоже делитель) непременно будет меньше корня.

10.8.2. Численные методы решения математических задач

Как известно, далеко не все математические задачи имеют простое аналитическое решение, но во многих случаях можно получить ответ с приемлемой точностью, воспользовавшись численными (приближенными) методами решения. Для примера рассмотрим три типовых задачи: решение нелинейного уравнения, поиск экстремума функции и вычисление определенного интеграла функции.

Общим в решениях этих задач является то, что при разработке программы крайне нежелательно привязываться к определенной функциональной зависимости, поскольку используемые методы носят универсальный характер. Идеальным способом задания функции является ввод функциональной зависимости с клавиатуры как строки текста, представляющей из себя правильное арифметическое выражение, например, `sin(x)` или `sin(x)*(x+2)`. Но с таким вариантом на данном этапе обучения нам не справиться (подход к решению задачи вычисления арифметического выражения, задаваемого в виде строки текста, см. в *разд. 14.4.4*).

Найдем простой, но достойный выход из положения. Будем задавать функциональную зависимость непосредственно в тексте программы в правой части оператора присваивания. Но при этом оформим этот оператор присваивания как отдельную функцию. Тогда все остальные части программы будут содержать обращения только к этой функции. Получается, что для изменения вида функциональной зависимости достаточно заменить только одну строку в теле функции (правую часть оператора присваивания) и перекомпилировать программу. Без оформления данного оператора присваивания в виде отдельной функции для изменения функции пришлось бы несколько раз исправлять одно и то же арифметическое выражение в разных местах программы, что очень неудобно.

Разумеется, реализация каждого из численных методов оформлена в виде отдельной процедуры или функции с параметрами без использования гло-

бальных переменных, так что ее можно скопировать без переработки в любую другую программу. Основная программа имеет чисто демонстрационный характер и выполняет ввод/вывод и вызов подпрограмм.

Листинг 10.8 содержит программу, вычисляющую значение корня нелинейного уравнения $f(x) = 0$ методом половинного деления (дихотомии). Идея метода основана на следующем утверждении: если $f(x)$ непрерывна на отрезке $[a, b]$ и $f(a)f(b) < 0$, то корень уравнения (или нечетное число корней) попадает в этот отрезок. Для определения значения корня можно сузить диапазон поиска, поделив исходный отрезок пополам. Из двух образовавшихся отрезков выбирают тот, для которого функция на концах имеет противоположные знаки и т. д. Процесс деления отрезка пополам продолжается до тех пор, пока длина отрезка больше заданной погрешности. Метод является медленным, но обладает гарантированной сходимостью и прост в реализации. В случае нескольких корней будет найден один из них.

Листинг 10.8. Вычисление корня уравнения методом половинного деления

```
function f(const x: real): real;
begin
   f:=4-exp(x)-2*x*x; {эту строку заменить для решения другого уравнения}
end;
procedure koren(const a,b,eps: real; var x: real; var error: boolean);
{a,b — границы отрезка,eps — погрешность вычисления корня,
x — корень уравнения, error — признак ошибки, если корень не найден}
var fa,fc,c: real;
begin
   if f(a)*f(b)>0 then begin error:=true; exit; end; {выход по ошибке}
   fa:= f(a);       { расчет f на левом конце отрезка }
   while b-a > eps do begin
      c:=(a+b)/2; fc:=f(c);  { делим отрезок пополам }
      if fa*fc <= 0 then  b:=c
      else begin a:=c; fa:=fc;  end;
   end;{цикла}
   x:=a; error:=false;
end; {процедуры вычисления корня}
const eps=1e-6;
var a,b,x:real; e:boolean;
begin
   write('Введите границы интервала '); readln(a,b);
   koren(a,b,eps,x,e);
```

```
   if not e then writeln(x:0:5)
   else writeln('Нет корня или четное количество корней');
   readln;
end.
```

Замечание

Можно было бы оформить процедуру вычисления корня в виде функции, возвращающей логическое значение (найден или не найден корень) и передающей сам корень в качестве параметра-переменной.

Следующий пример — поиск экстремума (для определенности, минимума) функции методом золотого сечения (листинг 10.9). Суть метода состоит в выделении на отрезке $[a, b]$ точек x_1 и x_2, производящих золотое сечение этого отрезка, для которого выполняются следующие пропорции:

$$\frac{b-a}{b-x_1} = \frac{b-x_1}{x_1-a}; \qquad \frac{b-a}{x_2-a} = \frac{x_2-a}{b-x_2}.$$

Решая эти два уравнения, получаем:

$$x_1 = b-(b-a)\cdot \text{tau}; \qquad x_2 = a+(b-a)\cdot \text{tau}; \qquad \text{tau} = \frac{\sqrt{5}-1}{2} \approx 0{,}618.$$

Тогда, сравнивая $f(x_1)$ и $f(x_2)$, можно определить, в каком интервале находится минимум функции ($[a, x_2]$ или $[x_1, b]$), и сузить тем самым отрезок $[a, b]$. Процесс продолжается до тех пор, пока длина отрезка не станет меньше погрешности eps.

Листинг 10.9. Поиск минимума функции методом золотого сечения

```
function f(const x:real):real;
begin
   f:=sqr(x-5)+2;{эту строку заменить при поиске минимума другой функции}
end;
procedure getmin(const a,b,eps:real; var x,y:real);
{a,b — границы отрезка, на котором имеется локальный минимум, eps — погрешность вычисления, x,y — координаты найденного минимума функции}
var x1,x2:real;
const tau=0.618;
begin
   while b-a>eps do
   begin
      x1:=b-(b-a)*tau;    { tau подобрано таким образом, чтобы }
```

```
      x2:=a+(b-a)*tau;    { x1 и x2 производили золотое сечение отрезка }
      if f(x1)<=f(x2) then b:=x2   { сокращаем интервал [a,b] }
      else a:=x1;
   end;
   x:=a; y:=f(a);
end;
var a,b,x,y:real;
const eps=0.001;
begin
   writeln('Введите значения a и b (границы интервала):');
   readln(a,b);
   getmin(a,b,eps,x,y);
   writeln('Минимум функции находится в точке ',x:0:3);
   writeln('Он равен ',y:0:2); readln
end.
```

В математике и физике часто приходится вычислять приближенное значение определенного интеграла. Одним из наиболее распространенных методов численного интегрирования является метод Симпсона. Суть метода заключается в том, что подынтегральная функция заменяется отрезками парабол и интеграл представляется в виде суммы площадей фигур, расположенных под всеми этими параболами. Программа (листинг 10.10) вычисляет площадь фигуры, ограниченной на координатной плоскости графиками двух функций $f_1(x)$ и $f_2(x)$ сверху и снизу, прямыми $x_1 = a$ и $x_2 = b$ слева и справа. В математике такая фигура называется *криволинейной трапецией*. Например, если $f_1(x) = 2x$, $f_2(x) = x$, $x_1 = 0$ и $x_2 = 1$, то фигура, площадь которой вычисляется, представляет собой треугольник с вершинами x, y: (0,0), (1,1), (1,2). Использование параметра процедурного типа позволяет записать алгоритм в компактной форме.

Листинг 10.10. Вычисление определенного интеграла методом Симпсона

```
{ описание процедурного типа func }
type func = function(x: real): real;
var
   s,square: real;{ значение интеграла и площади криволинейной трапеции }
   tn,tk: real; { границы отрезка } n: integer; { количество интервалов }
{$F+} {только для 16-разрядных компиляторов}
function f1(t: real): real;
begin f1:=2*t end;{ подынтегральная функция #1 (пример)}
function f2(t: real): real;
```

```
begin f2:=t; end; { подынтегральная функция #2 (пример)}
{$F-}
function simps(f: func; a,b: real; n: integer): real;
{ f — функция, a,b, — границы отрезка, n — количество интервалов }
var sum,h: real; j: integer;
begin
   if odd(n) then n:=n+1; {кол-во интервалов должно выражаться четным }
   {числом, т. к. параболу можно провести через три точки}
   h:=(b-a)/n; sum:=0.5*(f(a)+f(b));
   for j:=1 to n-1 do sum:=sum+(j mod 2+1)*f(a+j*h);
   simps:=2*h*sum/3
end; {функции интегрирования}
begin
   write('Введите нижнюю и верхнюю границы отрезка: '); readln(tn,tk);
   write('Число разбиений отрезка: '); readln(n);
   square:=simps(f1,tn,tk,n)-simps(f2,tn,tk,n);
   writeln('Площадь криволинейной трапеции равна S1-S2: ',square:10:7);
   readln
end.
```

10.8.3. Случайные числа

В программах на Pascal для генерации случайных чисел используется стандартная функция `random` *(см. разд. 7.4)*. Многократные тестирования подтвердили хорошее качество ее работы. Листинг 10.11 имеет чисто демонстрационный характер и реализует один из алгоритмов формирования последовательности случайных, а точнее сказать, псевдослучайных чисел.

Листинг 10.11. Формирование случайной числовой последовательности

```
var i,s: integer;
function next(var seed: integer): integer;
   const multiplier=37; increment=3; cycle=64;
begin
   next:=seed;
   seed:=(multiplier*seed+increment) mod cycle;
end;
begin
   s:=16; { можно взять любое целое значение из отрезка [0,63] }
   for i:=1 to 64 do write(next(s):4);
end.
```

Комментарий

Функция next, будучи вызвана с параметром s, например равным 16, возвращает число 16. Но одновременно она изменяет значение переменной seed на 19. Если функцию вызвать снова уже с полученным значением s, она возвратит число 19 и изменит значение s на 2. Продолжая вызывать функцию next, мы получим последовательность целых чисел, начинающуюся с исходного значения s.

Замечательным свойством этой последовательности является то, что каждое значение от 0 до 63 встречается в ней ровно один раз. Более того, шестьдесят пятый вызов функции next даст число 16 и начнет новый цикл. Другими словами, начиная с любого желаемого целого, функция генерирует фиксированную перестановку чисел от 0 до 63.

Похожие приемы обычно и используются для генерации "случайных" чисел. Часто их называют *псевдослучайными* — для того, чтобы подчеркнуть их предсказуемость.

Стандартная функция random возвращает случайное число, *равномерно распределенное* в заданном диапазоне значений, т. е вероятность появления любого из чисел одинакова. Однако давайте рассмотрим задачу, в которой использование random приводит к неудовлетворительным результатам. Пусть требуется сформировать ряд случайных чисел, соответствующих возможным величинам роста взрослого мужчины. Предположим, что эта величина меняется в диапазоне от 1,5 до 2 метров. Функция random возвратит значения, *равномерно* распределенные в этом интервале. Но мы знаем, что средний рост мужчины составляет 1,75 метра. Это значение и близкие к нему встречаются гораздо чаще, чем, например, 1,5 или 2 метра. Такое распределение случайных значений называют *нормальным*. Листинг 10.12 содержит программу, которая использует простейший алгоритм генерации случайных чисел со стандартным нормальным распределением. В данном случае числа лежат в диапазоне [−6,6), но наибольшее количество значений будет находиться в районе нуля. Нетрудно добавить в функцию два параметра, которые будут задавать нужные границы диапазона.

Листинг 10.12. Генератор случайных чисел с нормальным распределением

```
var i: integer;
{ функция — генератор случайных чисел с нормальным распределением }
function normrandom: real;
var s: real; i: byte;
begin
   s:=0;
   for i:=1 to 12 do s:=s+random;
   {можно сделать число повторений и больше 12 - чем больше, тем лучше}
   normrandom:=s-6.0; {вычитать надо половину от числа повторений цикла}
end;{функции}
```

```
begin
   randomize;                 { инициализация генератора случайных чисел }
   for i:= 1 to 192 do        { последовательность случайных чисел: }
      write(normrandom:10:3);{ вывод по 8 чисел в каждой из 24 строк }
end.
```

10.8.4. Игра "Ханойские башни"

В качестве примера использования *вложенных процедур* рассмотрим программу, в которой реализован известный алгоритм древней игры *"Ханойские башни"* (листинг 10.13).

Имеются три стержня (№ 1, № 2, № 3), на одном из которых (например, на правом — № 3) насажены диски разных размеров, причем диски располагаются так, чтобы стержень с дисками напоминал башню, т. е. снизу располагаются самые большие диски, а наверху маленькие. Цель игры — перенести башню с правого стержня (№ 3) на левый (№ 1), причем за один раз можно перенести лишь один диск и только так, чтобы диск с меньшим диаметром был насажен на диск с большим диаметром. Дополнительное условие: снятый диск нельзя отложить — его необходимо сразу же надеть на другой столб. Средний стержень (№ 2) является временным "буфером" для хранения перемещаемых дисков.

Листинг 10.13. Программа "Ханойские башни"

```
type position=(left,centre,right); { перечисляемый тип }
var n: integer;
procedure movedisk(from,tol: position);   { перемещение диска }
   procedure writepos(p: position);
   begin
      case p of
         left  : write('1');
         centre: write('2');
         right : write('3');
      end;
   end;
begin
   writepos(from); write('->'); writepos(tol); writeln
end;
procedure movetower(hight: integer; from,tol,work: position);
begin
```

```
   if hight>0 then
      begin
         movetower(hight-1,from,work,tol); { перемещение всех колец,
            кроме нижнего, на вспомогательную ось }
         movedisk(from,tol); { перемещение нижнего кольца }
         movetower(hight-1,work,tol,from); { перемещение этих же колец
            на конечную ось со вспомогательной }
      end;
end;
begin
   writeln('Введите число колец: '); readln(n) ;
   { вызов процедуры с передачей ей фактических параметров }
   movetower(n,right,left,centre);
end.
```

10.9 Контрольные вопросы и задания

Вопросы

1. В чем различие между стандартными и определенными пользователем подпрограммами?
2. Что называется параметром и каково его назначение? В каких случаях список параметров отсутствует?
3. Какова последовательность событий при вызове процедуры или функции?
4. Какой оператор используется для принудительного выхода из подпрограммы?
5. В чем состоят отличия параметров-значений от параметров-переменных?
6. Какое из приведенных утверждений верно?
   - процедура — частный случай функции;
   - функция — частный случай процедуры;
   - ни одна не является частным случаем другой.
7. Какие объекты программы называются локальными, а какие глобальными?
8. Почему локальные переменные иначе называются автоматическими?
9. Что такое область видимости и время жизни переменных?
10. Какие переменные называются статическими?

11. Что означает маскирование глобального объекта локальным? Приведите пример.
12. В чем состоят преимущества и недостатки локализации?
13. Как избегать нежелательных побочных эффектов подпрограмм?
14. Что такое рекурсия? В чем состоят достоинства и недостатки рекурсии?
15. Что такое опережающее объявление? В каких случаях оно требуется?
16. Каковы особенности использования параметров-процедур и параметров-функций?
17. Почему нетипизированные параметры в подпрограмме на языке Pascal обычно приходится приводить к определенному типу?

Задания

1. Составьте программу вычисления площади поверхности S, объема V и длины экватора L для любой из планет солнечной системы. Форму планет будем считать шаром с радиусом r. Вычисление площади, объема и длины экватора оформите отдельными функциями: $S = 4 \cdot r^2$, $V = 4/3 \cdot r^3$, $L = 4 \cdot r$.
2. Составьте программу поиска наибольшего из четырех чисел с использованием подпрограммы поиска большего из двух. Рассмотрите использование процедур и функций.
3. Даны координаты вершин многоугольника (x_1, y_1, x_2, y_2,x_{10}, y_{10}). Определите его периметр. Вычисление расстояния Δ между вершинами оформите подпрограммой. Рассмотрите использование процедур и функций.

$$\Delta = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

4. Даны отрезки a, b, c и d. Для каждой тройки этих отрезков, из которых можно построить треугольник, выведите на печать его площадь. Если треугольник не существует — организуйте печать соответствующего сообщения.
5. Даны координаты (x_1, y_1, x_2, y_2, x_3, y_3) вершин треугольника и координаты (x_4, y_4) некоторой точки, расположенной внутри него. Найдите расстояние от данной точки до ближайшей стороны треугольника. При определении расстояний необходимо учесть, что площадь треугольника может вычисляться как через три его стороны, так и через основание с высотой. При решении задачи используйте следующие формулы:
   - расстояние между двумя точками на плоскости:

$$\Delta = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2};$$

- площадь треугольника:

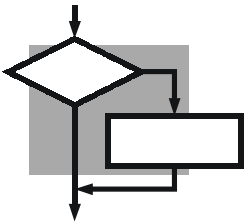
$$S = \sqrt{p(p-a)(p-b)(p-c)}, \qquad p = (a+b+c)/2;$$

- площадь треугольника:

$$S = ah/2.$$

6. Дано натуральное число *N*. Составьте программу, определяющую, есть ли среди чисел n, $n+1$, ..., $2n$ близнецы, т. е. простые числа, разность между которыми равна 2. Используйте процедуру распознавания простых чисел. Напомним, что простым называется натуральное число больше единицы, имеющее только два делителя: единицу и само это число.

Глава 11



Структуризация в программировании

Настало время подвести первые итоги обучения. Конструкции языка программирования уже известны, разобранные примеры программ, надеемся, понятны. Но очевидной стала и еще одна истина — разбор чужих программ и разработка своих собственных это "две большие разницы". Попробуем сделать еще одно теоретическое отступление, на основании которого мы сможем выработать практические рекомендации по правильной организации процесса разработки программы.

11.1. Теорема структуры

В 1965 г. итальянскими математиками К. Бомом и Г. Джакопини была сформулирована теорема структуры (в других переводах: "теорема о структурности", "структурная теорема", "основная структурная теорема"). Примерно к такой же формулировке самостоятельно пришли и специалисты по программированию, в частности Э. Дейкстра, поэтому эту теорему часто называют теоремой Дейкстры.

Теорема утверждает:

"Алгоритм любой сложности можно реализовать, используя только три конструкции: простое следование, бинарное ветвление и повторение".

Материал книги выстроен таким образом, что три эти базовые структуры обозначены очень четко и поэтому формулировка основной структурной теоремы должна быть понятна читателю. Собственно, к подобному выводу можно прийти, изучая на практике конструкции любого языка программирования.

Однако в целях обобщения мы все же еще раз определим три базовые структуры, которые поддерживаются любым процедурно-ориентированным языком высокого уровня:

- *следование* — это последовательность операторов или групп операторов, выполняемых друг за другом в порядке их следования в тексте программы;

- *ветвление* — управляющая структура, которая в зависимости от истинности заданного условия определяет выбор для исполнения одной из двух групп операторов;
- *повторение* — цикл, в котором группа операторов может выполняться повторно, если соблюдается заданное условие.

Существенной особенностью всех этих структур является то, что каждая из них имеет *только один вход и только один выход*, что и обеспечивает логически последовательную структуру программы. Все эти структуры определяются *рекурсивно*, т. е. *допустимо вложение структур* (например, следование или ветвление внутри тела цикла).

К трем главным структурам следует добавить вспомогательные конструкции, которые поддерживаются практически всеми языками программирования:

- неполную альтернативу (сокращенная форма условного оператора);
- множественное ветвление (в Pascal это оператор выбора `case`);
- вызов подпрограммы (процедуры или функции).

Некоторые программисты-теоретики предлагают считать подпрограмму четвертой основной структурой и приводят примеры алгоритмов, которые невозможно реализовать без рекурсивного вызова подпрограмм. Но очевидно, что спор вокруг основной структурной теоремы носит уже чисто теоретический характер, поскольку современное программирование без подпрограмм вообще немыслимо, независимо от того, включены они в формулировку основной структурной теоремы или нет.

11.2. Основы структурного программирования

Появление теоремы структуры дало толчок исследованиям по разработке технологий программирования. В начале 1970-х гг. появляется термин "структурное программирование". Он используется для обозначения совокупности методов и приемов разработки программного обеспечения.

11.2.1. Методы структурного программирования

Идея структурного программирования состоит в утверждении того, что структура программы должна отражать структуру решаемой задачи, а *алгоритм решения задачи должен быть совершенно понятен из текста программы*. Методы структурного программирования включают в себя:

- проектирование программ на основе *метода пошаговой детализации* (нисходящая или восходящая разработка программ);

- ❒ модульное программирование;
- ❒ структурное кодирование.

Большинство специалистов придерживаются точки зрения, что *метод нисходящего проектирования программ* наиболее удобен для решения сложных проблем. Основой этого метода является идея *программирования* "сверху вниз" с постепенной разбивкой исходной задачи на ряд подзадач.

При проектировании программы по этому методу выполняется *последовательное уточнение*: сначала задача определяется в общих чертах, а затем происходит постепенное уточнение ее структуры. На очередном шаге каждая подзадача, в свою очередь, разбивается на ряд других. Решение отдельного фрагмента сложной задачи может представлять собой самостоятельный программный блок — уже знакомую нам *подпрограмму*. Такой процесс детализации продолжается до тех пор, пока не станут ясны все детали решения задачи.

В этом случае программу решения сложной задачи можно представить как иерархическую совокупность относительно самостоятельных подпрограмм. Используя аналогию со строительством, можно отметить, что нисходящий подход соответствует разработке проекта всего здания с последующей детализацией его отдельных частей, а восходящий — проектированию здания с учетом деталей, имеющихся в каталоге (как например, в том случае, когда в распоряжении программиста имеется широкий набор процедур и функций).

При практическом воплощении метода пошаговой детализации обычно пользуются временными "заглушками" — подпрограммами, которые состоят из заголовка и пустого тела `begin end;`. Это позволяет с самого начала выстроить каркас будущей программы, определив назначение и имена основных подпрограмм. Применяя метод пошаговой детализации, программист постепенно наполняет кодом каждую заглушку. При этом каждый раз программа тщательно тестируется, чтобы новая появившаяся подпрограмма не нарушила ее работу.

Такой подход дисциплинирует программистов, заставляя их приложить максимальные усилия на начальном этапе разработки, когда выделяются основные подзадачи и формируются подходы к их решению. Ошибки на этом этапе всегда обходятся очень дорого, поэтому и привлекаются к такой работе самые квалифицированные специалисты. Наполнение кодом отдельных подпрограмм — менее ответственная задача, т. к. в случае неудачной реализации придется переписать только одну подпрограмму, а не всю программу целиком. Эту работу иногда поручают и не очень опытным программистам — надо же с чего-то начинать.

Для того чтобы разработку различных подпрограмм смогли выполнять параллельно разные программисты, в языках высокого уровня имеются средства разбиения программы на части на файловом уровне. Это означает, что

отдельные фрагменты программы можно оформлять в виде отдельных файлов — модулей. Принцип *модульного программирования* будет подробно разбираться в следующей главе, посвященной модулям и их реализации на языке Pascal.

Структурное кодирование призвано обеспечить максимальное удобство для восприятия и понимания программы человеком. При чтении текста программы должна четко прослеживаться логика ее работы. Чтение и понимание программы сильно затруднено при использовании программистом оператора безусловного перехода `goto`. Встретив этот оператор в программе, мы, чтобы уяснить смысл конструкции, принимаемся искать глазами метку, на которую он ссылается. Программирование "без `goto`" подразумевает, что операторы перехода не используются без крайней необходимости. Некоторые современные языки программирования вообще не имеют этого оператора, что согласуется с формулировкой основной структурной теоремы.

Перечислим основные способы полноценной замены оператора `goto` в языке Pascal, которые не нарушают принципов структурного кодирования:

- `halt` — принудительный (досрочный) выход из программы;
- `exit` — принудительный выход из процедуры;
- `break` — досрочный выход из цикла;
- `continue` — немедленный переход к исполнению следующего шага цикла.

Никаких других ситуаций, когда оператор `goto` нельзя заменить ни одной из перечисленных конструкций, в программе не должно быть. Об этом следует позаботиться при разработке алгоритма.

11.2.2. Пример разработки программы

Обычно по поводу всех разумных и правильных мыслей, изложенных в данном разделе, не возникает никаких возражений. Но когда дело доходит до практики, выясняется, что применение метода пошаговой детализации даже для небольшой по объему задачи вызывает затруднения. Попробуем применить этот метод на простом примере.

Мы много говорили о хорошем качестве генератора случайных (псевдослучайных) чисел, который используется стандартной функцией `random`. Давайте разработаем программу, которая поможет оценить степень случайности генерируемых значений, выполнив ряд тестов над последовательностью из заданного количества случайных чисел, формируемых функцией `random` без параметров (напомним, что в этом случае формируются случайные действительные числа в интервале от 0 до 1). Будем выполнять такие тесты:

1. Вычисление среднего значения (при хорошем качестве `random` на большом количестве чисел последовательности должно получиться 0,5).

2. Тест распределения по интервалам. Разобьем отрезок [0,1) на четыре интервала и убедимся, что в каждый из интервалов попадает примерно одинаковое количество случайных чисел. Вообще-то количество интервалов для этого теста должно задаваться произвольно, но пока нами не введено еще понятие массива, ограничимся упрощенным вариантом теста.
3. Тест серий. Будем округлять каждое из случайных значений, получая при этом последовательность из нулей и единиц. Найдем максимальную длину серий подряд идущих единиц и подряд идущих нулей. Другими словами — если мы бросаем монетку, то максимум сколько раз подряд может выпадать орел? А решка? Из жизни у нас есть представления по этому вопросу. А как справится с этой ситуацией `random`?

Поставленная задача не очень объемна и алгоритмически не столь сложна, поэтому она иногда дается в качестве самостоятельного задания начинающим, уже освоившим основные конструкции языка, в том числе и подпрограммы, но не имеющим пока опыта собственной разработки. Интересно наблюдать, как по-разному проходит процесс работы над заданием у разных учащихся.

Некоторые из них сначала разрабатывают и набирают весь текст программы от начала до конца и только после этого запускают программу на выполнение и приступают к отладке. С трудом исправив множество синтаксических ошибок, большинство из них получает результаты, противоречащие здравому смыслу. Найти сразу несколько ошибок в большой программе очень трудно, обычно этот процесс затягивается надолго или требуется помощь преподавателя.

Другая группа учеников, сообразивших, что каждый тест представляет собой довольно независимую задачу, сначала отлаживает три отдельных небольших программы, а затем соединяет их вместе. Фактически они воплощают в жизнь метод программирования "снизу вверх". У них дела идут значительно веселей, чем у первой группы, поскольку результаты каждого теста появляются довольно быстро. Правда, приходится тратить лишнее время на соединение отдельных частей в единое целое.

Наконец, еще один вариант — программа постепенно наращивается, при этом на каждом шаге она запускается на выполнение и все ошибки исправляются постепенно. Как правило, именно учащиеся, избравшие этот вариант, заканчивают работу первыми, т. к. пошли верным путем, применив метод пошаговой детализации.

Рассмотрим возможную последовательность шагов решения нашей задачи.

Шаг 1. Обдумываем структуру программы. Рассуждаем примерно так:

"Хотя каждый из тестов не зависит один от другого, все они выполняются над одной и той же последовательностью случайных чисел. Так будет убеди-

тельнее, да и намного быстрее, поскольку потребуется всего один цикл вместо трех. Значит, заманчивый вариант из трех автономных и независимых процедур без использования глобальных переменных не очень-то подходит. Но и не разделять задачу на части нельзя — с отладкой замучаемся.

Примем компромиссный вариант — выделим в отдельные процедуры без параметров обработку на принадлежность интервалам и на тест серий. Вычисление среднего требует только суммирования чисел, поэтому не стоит оформлять процедуру из одного оператора".

Шаг 2. Пишем каркас программы, определяем необходимые переменные, а вместо процедур используем их заглушки. Все это делается очень быстро, т. к. работа чисто техническая.

```
var x,sum:real; {случайное число и сумма для подсчета среднего}
    n,k,k1,k2,k3,k4:integer; {для цикла и распределения по интервалам}
    intx,count0,count1,max0,max1:integer;{для серий из нулей и единиц}
procedure interval;
begin
{определение принадлежности числа x к одному из интервалов — заглушка}
end;
procedure maxlength;
begin
{обработка x для теста серий — заглушка}
end;
begin
  randomize;
  writeln('Введите количество случайных чисел'); readln(n);
  sum:=0;  k1:=0; k2:=0; k3:=0; k4:=0;
  count0:=0; count1:=0; max0:=0; max1:=0;
  for k:=1 to n do
  begin
     x:=random; {write(x:6:3);}
     sum:=sum+x;
     interval;
     intx:=round(x); {write(intx:2);}
     maxlength;
  end;
  writeln;
  writeln('Среднее значение равно ',sum/n:0:4);
```

```
    write('Количество чисел в интервалах отрезка [0,1): ');
    writeln(k1,' ',k2,' ',k3,' ',k4);
    writeln('Максимальное количество подряд идущих нулей равно ',max0);
    writeln('Максимальное количество подряд идущих единиц равно ',max1);
    readln;
  end.
```

В тело цикла мы добавили операторы вывода самого случайного числа и его округленного значения, которые потребуются при отладке. Эту программу можно смело запускать на выполнение, хотя она еще мало что умеет делать. Тем не менее, избавимся от синтаксических ошибок, проверим правильность вычисления среднего значения и убедимся, что все остальные результаты равны нулю. Неплохо для начала.

Шаг 3. Добавляем в процедуру проверки на принадлежность одному из четырех интервалов (`interval`) ее тело. Код несложный, надо только аккуратно написать условный оператор с вложенными условиями. С этой задачей мы также быстро справляемся:

```
procedure interval;
begin
   if x<0.25 then inc(k1)
   else
      if x<0.5 then inc(k2)
      else
         if x<0.75 then inc(k3)
         else inc(k4);
end;
```

Снова запускаем программу на выполнение и убеждаемся, что функция `random`, как всегда, на высоте. Результаты по интервалам различаются очень незначительно.

Шаг 4. Осталось решить самую сложную задачу — подсчитать размеры максимальных серий из нулей и единиц. Пожалуй, ее можно еще разбить на части, чтобы отладка стала удобнее. Попробуем сначала подсчитать одну только длину максимальной последовательности из нулей (или единиц). Понятно, что одного только текущего значения случайного числа `x` недостаточно и нужно знать его предыдущее значение, чтобы решить, продолжается ли серия подряд идущих цифр или текущее `x` — начало новой серии. Поэтому введем еще одну глобальную переменную, назвав ее `lastx` (предыдущее `x`). Не забудем добавить ее в раздел описаний основной программы. Запишем текст процедуры:

```
procedure maxlength;
begin
```

```
   if intx=0 then count0:=count0+1 else
   if lastx<>intx {начало серии единиц, конец серии нулей} then begin
      if count0>max0 then max0:=count0; {проверили длину серии нулей}
      count0:=0; {готовимся к подсчету длины новой серии нулей}
   end;
   if (k=n) and (count0>max0) then max0:=count0;{самая последняя серия}
   lastx:=intx;{готовимся к обработке следующего числа}
end;
```

При тестировании программы, включающей эту процедуру, обязательно нужно проследить случаи, когда самая длинная серия оказывается самой первой или самой последней. На случай самой первой серии мы присвоили переменной `lastx` значение −1 (или любое другое значение, отличное от 0 и 1), а для обработки самой последней серии пришлось добавить дополнительный условный оператор в текст процедуры.

Шаг 5. Теперь длина максимальной серии из нулей подсчитывается правильно. Может быть, скопировать этот фрагмент процедуры и заменить `count0` и `max0` на `count1` и `max1`?

Стоп! Для этой цели и используются процедуры с параметрами. Оформим соответствующий фрагмент в виде процедуры с параметрами и будем вызывать эту процедуру с разными фактическими параметрами в зависимости от того, обрабатывается в данный момент ноль или единица. Немного модифицировав текст, получим следующий вариант:

```
procedure countlength(var count,max,lastcount,lastmax:integer);
{вспомогательная процедура для подсчета максимальной длины любой серии}
begin
   count:=count+1;
   if lastx<>intx {начало новой серии} then begin
      if lastcount>lastmax then lastmax:=lastcount;
      lastcount:=0;
   end; {проверили длину закончившейся серии и обнулили ее}
   if (k=n) and (count>max) then max:=count;{самая последняя серия}
   lastx:=intx;{готовимся к обработке следующего числа}
end;
procedure maxlength;
begin
   if intx=0 then countlength(count0,max0,count1,max1)
             else countlength(count1,max1,count0,max0);
end;
```

Теперь программа выполняет все свои задачи в полном объеме. Оценим время, затраченное на разработку. Оно засекалось специально и составило 15 минут, включая и тестирование (возможно довести время и до 10 минут, если поставить такую цель). К сожалению, большинство начинающих решает такую или похожую задачи значительно дольше. Но скорость работы довольно быстро набирается со временем, если освоить метод пошаговой детализации. Кстати, учащиеся, которые не сумели справиться с отладкой, решая задачу без разбиения на части, после одного-двух таких заданий быстро соображают, что зашли в тупик, и дальше двигаются уже правильным путем.

В заключение оценим результаты тестирования функции `random`. Можно убедиться, что и тест серий показывает результаты, аналогичные случайному бросанию монеты. Если заменить тип `integer` на `longint` (например, так: `type integer=longint;`), то можно выполнить тестирование и на очень длинных последовательностях случайных чисел. При этом среднее значение составляет 0,5 с точностью 3—4 знака после запятой, а расхождения в распределении по интервалам совсем малы. Максимальные серии имеют длину 20 и даже немного больше. Это вполне согласуется с законами статистики: при выбрасывании монетки кому-то может случайно повезти и 20 раз подряд, но когда-нибудь полоса везений закончится так же, как и последовательность, состоящая из одних единиц.

11.3. Контрольные вопросы и задания

Вопросы

1. Сформулируйте теорему структуры.
2. Что понимают под структурным программированием? Обоснуйте необходимость его применения.
3. В чем заключаются методы нисходящего и восходящего программирования? Каким из них удобнее пользоваться лично вам?
4. Что такое заглушка подпрограммы? Чем удобны заглушки?
5. Что такое структурное кодирование? Используете ли вы оператор `goto`?

Задания

1. Разработайте программу для анализа функций одной переменной на заданном интервале, которая позволяет выполнить несколько различных действий по выбору пользователя:
   - вывод таблицы значений с заданным шагом;
   - нахождение минимального и максимального значений;

- нахождение точек пересечения с осями координат;
- если в вашей системе программирования работает модуль Graph, то построение графика функции (можно взять готовый текст из *разд. П2.3*).

Вид функциональной зависимости задать в отдельной функции пользователя. Для выполнения различных действий можно воспользоваться готовыми процедурами (листинги 10.9, 10.10).

2. Разработайте программу для численного интегрирования различными методами по выбору пользователя:
   - метод прямоугольников;
   - метод трапеций;
   - метод Симпсона.

   Каждый из методов оформить в виде отдельной процедуры. Вид функциональной зависимости задать в отдельной функции пользователя.
3. Разработать программу, проверяющую, лежит ли данная точка внутри треугольника. С клавиатуры вводятся координаты точки и координаты вершин треугольника.

   Указания к решению задачи. Заметим, что если точка находится внутри треугольника, то она должна лежать с каждой его вершиной по одну сторону от прямой, проходящей через противолежащую сторону. Если хотя бы для одной вершины это правило не выполняется, то точка не лежит внутри треугольника. Для того чтобы проверить, лежат ли две точки по одну сторону прямой, рекомендуем по координатам концов отрезка получить уравнение прямой вида $ax + by + c = 0$, точнее его коэффициенты, для каждой стороны треугольника. Тогда точки `(X1,Y1)` и `(X2,Y2)` лежат по одну сторону от прямой $ax + by + c = 0$, если подстановка координат точек в левую часть уравнения прямой дает значения одного и того же знака, т. е. справедливо неравенство `(aX1+bY1+c)*(aX2+bY2+c)>0`.

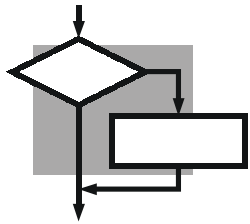
   Разработайте процедуру получения коэффициентов уравнения прямой и функцию проверки того, лежат ли две точки по одну сторону от прямой, заданной своим уравнением. С их помощью вы легко решите задачу.
4. Разработать программу, проверяющую, имеют ли два заданных отрезка общую точку, и если да, то вычислить ее координаты. С клавиатуры вводятся координаты концов отрезков.

Указания к решению задачи. Сначала необходимо получить уравнения прямых, содержащих данные отрезки (см. предыдущую задачу). Затем, если полученные прямые не параллельны `(a1:a2≠b1:b2)`, вычислить координаты их точки пересечения. Для этого нужно решить систему, составленную из

уравнений этих прямых. В завершение нужно проверить, принадлежат ли полученные координаты точки заданным отрезкам или лежат на их продолжении. Если концы отрезка имеют координаты `(X1,Y1)` и `(X2,Y2)`, то координаты `x`, `y` полученной точки должны лежать в интервалах `[X1,X2]` и `[Y1,Y2]` соответственно, иначе точка лежит на продолжении отрезка.

Разработайте процедуру получения коэффициентов уравнения прямой, процедуру решения системы из двух линейных уравнений и функцию проверки, лежит ли данная точка прямой на отрезке, через который проходит данная прямая.

Глава 12



Библиотечные модули

При изучении темы "Процедуры и функции" внимательный читатель, конечно, заметил, что практически все подпрограммы, представленные в качестве примеров в *гл. 10*, имеют универсальный характер. Их можно использовать не только в той программе, частью которой они являются, но и во многих других программах. Действительно, при подборе примеров подпрограмм авторы стремились предложить читателю как можно больше типовых задач универсального характера.

Возникает закономерный вопрос: как проще всего воспользоваться подпрограммами, которые уже были разработаны и отлажены ранее? Разумеется, можно копировать тексты подпрограмм из одной программы в другую, пользуясь буфером обмена, но это утомительное занятие. Кроме того, программы, содержащие множество процедур и функций, становятся громоздкими и неудобными в отладке.

В Pascal есть замечательное средство преодоления этих неприятностей — библиотечные модули. Но прежде чем приступить к изучению модульного программирования, вспомним, как происходит обработка исходного текста программы в системах программирования на основе языка Pascal.

12.1. Компиляция и компоновка программы

Понятие об этапах компиляции и компоновки было введено в *гл. 3 "Системы программирования" (см. разд. 3.2)*. *На этапе компиляции* исходная программа преобразуется в машинный код, но он пока не пригоден для исполнения, т. к. в него не включены коды стандартных процедур и функций. Код программы, полученный после компиляции, называют *объектным кодом*. Стандартные процедуры и функции хранятся в отдельных файлах, называемых *библиотеками*, *которые также являются объектными кодами.*

На этапе компоновки к объектному коду программы добавляется объектный код стандартных процедур и функций из библиотек и в результате он превращается в полноценный исполнимый код (`executable` — отсюда расширение exe).

Компиляция и компоновка — это два различных процесса. Первый из них идет под управлением программы-*компилятора* (Compiler). Процесс компоновки обеспечивается работой программы-*компоновщика* (иначе ее называют редактор связей, Linker), назначение которой состоит в том, чтобы добавить к программе весь недостающий код из других файлов.

Мы напомнили эти положения для того, чтобы сделать важный вывод — если компилятор имеет возможность обрабатывать не только тексты законченных программ, но и *фрагменты* текстов программ, то отдельные части программы можно *компилировать отдельно* и *хранить на диске* в откомпилированном виде. Затем компоновщик соберет объектные коды из разных файлов, и в результате будет получен исполнимый код для всей программы. Такие действия получили название *раздельной компиляции.*

Все компиляторы с языка Pascal поддерживают возможность раздельной компиляции. В языке Pascal для этих целей введено понятие библиотечных модулей (такая возможность появилась в расширенном стандарте Extended Pascal).

12.2. Понятие модуля. Преимущества модульного программирования

Библиотечный модуль оформляется как отдельно компилируемая программная единица, содержащая различные элементы раздела описаний и, возможно, некоторые операторы. В состав модуля входят описания констант, переменных, типов, процедур и функций. На практике часто пользуются термином *модуль*, опуская прилагательное.

Процедуры и функции, содержащиеся в модуле, подключаются к программе, которая их использует, на этапе компоновки. Хранится модуль как в исходном, так и в откомпилированном виде (файлы с расширениями pas и tpu соответственно). В системе Delphi файл с объектным кодом имеет расширение dcu.

Теперь, наконец, можно дать ответ на вопрос, как лучше всего поступать с теми подпрограммами универсального назначения, которые могли бы использоваться неоднократно при решении различных задач. Такие процедуры и функции (а также описания необходимых для их работы типов, констант и переменных) удобно оформлять в виде отдельных модулей.

По ходу работы любой программист обычно накапливает для себя целую коллекцию таких полезных модулей — свою личную библиотеку. Сохраняя ее, он знает, что она позволит ему писать гораздо меньше кода для новых программ, ведь он может *многократно использовать свои старые разработки.*

В этом и заключается один из наиболее фундаментальных принципов современного программирования — *принцип модульности*. На практике реализация данного принципа предполагает выполнение двух условий:

- при разработке программы необходимо выделять подпрограммы таким образом, чтобы можно было воспользоваться уже имеющимися модулями;
- если в личной библиотеке программиста не находится подходящих подпрограмм для решения какой-либо задачи, то разработку новых процедур или функций следует выполнять таким образом, чтобы полученные модули можно было использовать в качестве составных частей для решения других задач.

Таким образом, принцип модульности отлично дополняет структурный подход к разработке программ, позволяя *многократно* использовать разработанные и отлаженные модули. Освоение и применение техники структурного и модульного программирования — задача непростая для начинающего программиста, который пока что в состоянии разрабатывать только короткие программы, для которых преимущества структурного и модульного подхода не очевидны. Тем не менее, каждый разработанный вами собственноручно библиотечный модуль — это шаг в правильном направлении. Все затраченные усилия в будущем окупятся с лихвой!

Чтобы наш совет начинающим выглядел более убедительно, приведем несколько дополнительных аргументов в пользу модульного программирования.

- Модули, в отличие от процедур и функций, включаемых в исходный код программы, могут храниться на диске в откомпилированном виде. В этом случае процесс подготовки программы к выполнению займет меньше времени, т. к. компилироваться будет только основная программа, а код из модулей будет подключаться на этапе компоновки.
- Еще одно немаловажное обстоятельство — при разработке больших программ отдельные модули могут разрабатываться различными программистами, т. к. это относительно автономные программные единицы.
- Для языка Pascal уже накоплено большое количество модулей. Это стандартные модули (некоторые из них будут рассматриваться в *приложении 2)*, а также многочисленные модули, тексты которых приводятся в различных книгах по программированию и/или распространяются на дискетах и компакт-дисках.

Тем не менее, мы начнем с уяснения правил оформления собственных модулей. Они достаточно просты и доступны для начинающего программиста.

12.3. Структура модуля

Исходный текст любого модуля можно разделить на несколько разделов:

- заголовок;
- интерфейсная часть;

- исполняемая часть;
- инициирующая часть.

Собственно программный код располагается в исполняемой части, иногда в инициирующей части. Заголовок и интерфейсная часть задают название модуля и перечисление всех программных элементов, которые предоставляет данный модуль тем программам или другим модулям, которые будут его использовать. Если провести аналогию модуля с книгой, то заголовок и интерфейсную часть можно рассматривать как обложку и оглавление книги. Соответственно, весь основной текст располагается в исполняемой и инициирующей частях.

В общем виде структура модуля выглядит так:

```
unit <имя>; { заголовок модуля }
{$R+}       { возможно, глобальные директивы компилятора}

interface     { начало интерфейсной части }
uses ...      { список модулей }
label ...     { объявления общедоступных меток }
const ...     { объявления общедоступных констант }
type  ...     { объявления общедоступных типов }
var   ...     { объявления общедоступных переменных }
procedure...  { заголовки общедоступных процедур }
function ...  { заголовки общедоступных функций }

implementation     { начало исполняемой части }
uses ...      { используемые при реализации модули }
label ...     { объявления скрытых глобальных меток }
const ...     { объявления скрытых глобальных констант }
type  ...     { объявления скрытых глобальных типов }
var ...       { объявления скрытых глобальных переменных }
procedure ... { заголовки и тела общедоступных и скрытых процедур }
function ...  { заголовки и тела общедоступных и скрытых функций }
begin       { начало инициирующей части }
. . . . .   { здесь могут располагаться любые операторы }
end.        { конец  модуля }
```

На практике обычно отсутствуют какие-либо объявления, и в целом структура получается проще.

Замечание

В Delphi состав модуля немного расширен. Во-первых, инициирующая часть теперь начинается служебным словом `initialization`, во-вторых, добавилась еще одна заключительная часть с ключевым словом `finalization`, которая выполняется при завершении программы. Например, если в инициирующей части открывается какой-то нужный служебный файл, то в заключительной он должен закрыться (что означает действие открытия файла, будет пояснено в *гл. 17*). В целом назначение частей модуля не изменилось.

Следует знать:

- имя модуля служит для его связи с другими модулями и основной программой, поэтому *заголовок модуля опускать нельзя* (в отличие от заголовка программы);
- *имя модуля должно совпадать с именем того файла, в который помещается исходный текст модуля.* Если, например, имеем заголовок `unit triangle`, то исходный текст соответствующего модуля должен размещаться в файле Triangle.pas, *что очень важно*;
- в интерфейсной части приводятся только *общедоступные* объявления, т. е. доступные для использования в любой программе или модуле, к которым будет подключен данный модуль. Порядок появления различных объявлений и их количество могут быть произвольными;
- исполняемая часть содержит полные описания подпрограмм, объявленных в интерфейсной части. В ней могут располагаться также вспомогательные типы, константы, переменные, процедуры и функции, возможно и метки, если они используются в инициирующей части;
- все вспомогательные программные элементы, объявленные в исполняемой части, называются *скрытыми*, т. к. они доступны для использования только в данном модуле и невидимы для программы, использующей модуль;
- инициирующая часть завершает модуль. Она может отсутствовать вместе с начинающим ее словом `begin` или быть пустой — тогда за `begin` сразу следует признак конца модуля `end.` (с точкой);
- в инициирующей части могут размещаться исполняемые операторы. Эти операторы исполняются еще до начала выполнения основной программы и обычно используются для подготовки ее работы. Например, в них могут задаваться начальные значения переменных, открываться нужные файлы и т. п.

Замечание

В заголовке подпрограммы, объявленной в исполняемой части, можно опускать список формальных параметров и тип возвращаемого значения для функции, т. к. они уже описаны в интерфейсной части. Если заголовок подпрограммы все-таки приводится в полном виде, то он должен *полностью совпадать* с заголовком, объявленным в интерфейсной части.

Для подключения модуля к программе или к другому модулю используется предложение: `uses список модулей;`.

В программе оно располагается до начала других объявлений.

Например, `uses crt, graph, triangle;`.

12.4. Снова о глобальных, локальных и статических переменных

Обратимся снова к нелегкой теме — к области видимости и времени жизни переменных *(см. разд. 10.5)*.

Обратите внимание — глобальные переменные, описанные в *исполняемой части* модуля, являются на самом деле не глобальными, а *статическими переменными*. Они видимы только *во всех подпрограммах данного модуля* (это немало), но *недоступны основной программе*. Налицо локальная область видимости. Однако память под эти скрытые от основной программы переменные выделяется в сегменте данных до начала выполнения программы, и *время их жизни длится от начала до конца выполнения программы*. Такие переменные программисты используют довольно часто для обмена данными между отдельными подпрограммами модуля, но обращаться с ними нужно почти так же осторожно, как и с глобальными переменными основной программы.

Еще более осторожно следует обращаться с переменными, объявленными в интерфейсной части модуля, поскольку они — полноценные *глобальные переменные*, имеющие глобальную видимость и время жизни. Они располагаются в той же области памяти, где и глобальные переменные программы, и используются без всякого предварительного описания.

Замечание

Если и в модуле, и в программе, к которой подключен модуль, объявить переменные или другие объекты с *одинаковыми именами*, то соответствующий объект из модуля станет невидим программе (замаскирован). К счастью, из этой ситуации есть выход: для обращения к объекту из модуля можно воспользоваться составным именем `ИмяМодуля.ИмяОбъекта`. Например, если и в программе, и в модуле с именем `myunit`, который к ней подключен, есть глобальные переменные с одинаковым именем (допустим, `x`), то к переменной из модуля можно обратиться по имени `myunit.x`, а к переменной из программы — просто по имени `x`. Все-таки еще раз напомним, что всем глобальным переменным как в программе, так и в модулях следует давать осмысленные, достаточно длинные имена (`x` не годится) и держать их под постоянным контролем, чтобы свести возможные коллизии с именами к минимуму.

К сожалению (или к счастью), глобальные переменные основной программы нельзя использовать в модулях, которые к ней подключены. Модули задуманы как автономные библиотеки подпрограмм, так чтобы их можно бы-

ло компилировать отдельно и затем использовать в разных программах. Однако возможно сформировать отдельный модуль, содержащий набор глобальных определений (типы, переменные, константы), и подключить его к основной программе и ко всем модулям, которые в ней используются. Так часто поступают при создании больших проектов.

Несколько слов скажем о локальных переменных подпрограмм. Память для них выделяется в том же программном стеке, что и для локальных переменных основной программы, и по тому же самому принципу (т. е. они имеют автоматическое время жизни). Поэтому об их именах можно не беспокоиться особо. Опытные программисты советуют выбирать для локальных переменных короткие имена, с тем чтобы они отличались от глобальных и статических переменных.

12.5. Пример модуля

Разработаем модуль `triangle`, содержащий набор процедур и функций для выполнения расчета треугольников, используя известные из геометрии формулы. При этом предусмотрим два наиболее часто встречающихся способа задания треугольника: по трем сторонам и по координатам его вершин (листинг 12.1). Модуль позволяет выполнить следующие действия:

- вычисление сторон по координатам вершин треугольника;
- проверка существования треугольника;
- расчет площади и периметра;
- расчет радиусов вписанной и описанной окружности.

Такой модуль может оказаться полезным при решении геометрических задач, особенно если вы забыли какую-либо из формул.

Листинг 12.1. Модуль для расчета треугольников

```
unit triangle;
interface
{ вычисление сторон треугольника по координатам вершин }
procedure getabc (xa,ya,xb,yb,xc,yc:real; var a,b,c:real);
{ проверка существования данного треугольника }
function exist(a,b,c:real):boolean;
{ вычисление периметра }
function perimetr(a,b,c:real):real;
{ вычисление площади }
function square(a,b,c:real):real;
{ вычисление радиуса вписанной окружности }
```

```
function rv(a,b,c:real):real;
{ вычисление радиуса описанной окружности }
function ro(a,b,c:real):real;
{ исполняемая часть, использующая известные из геометрии формулы }
implementation
{ скрытая функция для вычисления расстояния между двумя точками }
function len(x1,y1,x2,y2:real):real;
begin
   len:=sqrt(sqr(x1-x2)+sqr(y1-y2));
end;
procedure getabc;
begin
   a:=len(xa,ya,xb,yb);
   b:=len(xb,yb,xc,yc);
   c:=len(xc,yc,xa,ya);
end;
function exist;
begin
   exist:=(a<b+c) and (b<a+c) and (c<a+b);
end;
function perimetr;
begin
   perimetr:=a+b+c;
end;
function square;
var p:real;
begin
   p:=(a+b+c)/2;              { определение p - полупериметра }
   square:=sqrt(p*(p-a)*(p-b)*(p-c));   { определение площади }
end;
function rv;
begin
   rv:=square(a,b,c)/perimetr(a,b,c)*2;end;
function ro;
begin
   ro:=a*b*c/4/square(a,b,c);
end;
begin
{ инициирующая часть отсутствует }
end.
```

Замечание

Для того чтобы сделать функцию len общедоступной, достаточно занести заголовок этой функции в интерфейсную часть.

В качестве примера использования модуля напишем программу (листинг 12.2), решающую следующую несложную задачу. Пусть имеется два участка треугольной формы, обнесенных забором. Каким-то образом известны координаты вершин этих треугольников. Требуется определить:

- у которого из участков длиннее забор (забор — строго по периметру);
- на каком из участков можно разместить больше растений (считаем, что их количество прямо пропорционально площади);
- на каком из них можно разместить идеально круглую клумбу наибольшего размера (оценим по радиусам вписанных окружностей).

Листинг 12.2. Программа расчета треугольников, использующая модуль

```
uses triangle;
var xa,ya,xb,yb,xc,yc,
    xd,yd,xe,ye,xf,yf,a,b,c,d,e,f: real;
begin
   writeln('Введите координаты вершин первого участка');
   readln(xa,ya,xb,yb,xc,yc);
   getabc(xa,ya,xb,yb,xc,yc,a,b,c);
   if not exist(a,b,c) then
      writeln('Такого участка не существует')
   { все вершины лежат на одной прямой }
                      else { первый участок существует }   begin
      writeln('Введите координаты вершин второго участка');
      readln(xd,yd,xe,ye,xf,yf);
      getabc(xd,yd,xe,ye,xf,yf,d,e,f);
      if not exist(d,e,f) then
         writeln('Такого участка не существует')
                         else { оба участка существуют }
         begin
           if perimetr(a,b,c)>perimetr(d,e,f) then write('У первого')
                                              else write('У второго');
           writeln(' участка забор длиннее');
           if square(a,b,c)>square(d,e,f) then write('На первом')
                                          else write('На втором');
```

```
            writeln(' участке поместится больше растений');
            if rv(a,b,c)>rv(d,e,f) then write('На первом')
                                   else write('На втором');
            writeln(' участке поместится самая большая круглая клумба');
          end;
   end; readln
end.
```

12.6. Раздельная компиляция модулей

При разработке крупных проектов в их состав входит большое количество модулей. Любая система программирования предоставляет средства управления раздельной компиляцией модулей.

12.6.1. Компиляция модулей в Turbo Pascal

Результатом компиляции модуля в системе Turbo Pascal является файл с тем же самым именем и расширением tpu (Turbo Pascal Unit). Его можно сохранить на диске, также как exe-файл.

Меню **Compile**, управляющее процессом компиляции, содержит следующие опции:

- **Compile** (клавиши <Alt>+<F9>);
- **Make** (клавиша <F9>);
- **Build**;
- **Destination** (**Memory**, **Disk**);
- **Primary file**.

Первые три опции — это режимы компиляции. При компиляции модуля или основной программы в режиме **Compile** все упоминавшиеся в нем модули должны быть предварительно откомпилированы. Если какой-либо файл tpu не обнаружен, то система ищет подобный файл с расширением pas, т. е. файл с исходным текстом модуля, и при обнаружении компилирует его.

В режиме **Make** система следит за возможными изменениями исходного текста модуля. Если в текст модуля были внесены какие-либо изменения, то система заново его компилирует и только затем приступает к компиляции основной программы. Кроме того, если были внесены изменения в интерфейсную часть модуля, то будут перекомпилированы и все другие модули, обращающиеся к нему.

В режиме **Build** автоматически компилируются все модули, независимо от времени их обновления. Это самый надежный, но и самый медленный режим подготовки модульной программы.

Опция **Destination** нужна для задания возможности сохранения файлов tpu (а также и exe) на диске. Для этой цели нужно установить опцию **Destination** в значение **Disk** (по умолчанию ее значение **Memory**). В среде Borland Pascal файлы exe и tpu автоматически сохраняются на диске, и данная опция отсутствует в меню **Compile**.

Наконец, последний пункт **Primary file** позволяет задать файл, который будет автоматически добавляться в начало исходного текста перед компиляцией. С помощью этой опции удобно отлаживать модули, подключая к ним исходную программу в качестве **Primary file**. При таком способе отладки не придется постоянно перемещаться между окнами основной программы и модуля.

12.6.2. Управление модулями в Delphi. Понятие проекта

В Delphi принцип модульности доведен до полного логического завершения. При визуальной разработке приложения Windows основная программа вообще формируется средой автоматически и содержит всего несколько стандартных строк. Вся логика работы программы содержится в модулях, которые в совокупности с основной программой и некоторыми служебными файлами образуют *проект (Project)*.

При разработке консольного приложения (т. е. приложения, которое исполняется в специальном окне, имитирующем текстовый режим операционной системы DOS) автоматически формируется только каркас основной программы, в который программист вписывает свой код, но при создании многомодульных консольных приложений их можно также удобно объединить в единый проект.

Система Delphi включает в себя стандартное средство управления проектом — менеджер проекта (Project Manager). Менеджер проекта доступен через специальное окно, которое вызывается при помощи команды меню **View/Project Manager**. Работая в окне менеджера проекта, можно легко добавлять к проекту новые модули и удалять ненужные, а также быстро просматривать и корректировать все модули.

При запуске проекта на выполнение автоматически компилируется основная программа и все модули, в которые были внесены изменения. При необходимости выполнить одну только компиляцию без выполнения программы можно воспользоваться командами меню **Project — Compile** или **Build**. При этом по команде **Build** заново компилируются все модули, входящие в проект, а по **Compile** — только те, в которые были внесены изменения.

Средства управления проектами имеются во всех современных системах разработки. Сравнив возможности по управлению проектами систем Turbo Pascal и Delphi, можно догадаться, что они полностью совпадают, с той оговоркой, что менеджер проектов Delphi обеспечивает больше удобств

разработчику. При начальном освоении техники создания многомодульных проектов (в любой среде) трудности замечались, но быстро преодолевались. К хорошему всегда привыкают быстро.

12.6.3. Распространение готовых модулей

Если модуль полностью отлажен и протестирован, то можно распространять его в виде tpu-файла (или dcu-файла), приложив к нему заголовок и интерфейсную часть исходного текста модуля в качестве инструкции по использованию с подробными комментариями. Однако исходный код обязательно должен храниться в надежном месте, т. к. из объектного кода его восстановить невозможно. Из исходного кода можно получить объектный в считанные секунды — компиляторы с языка Pascal считаются одними из самых быстрых.

Большинство стандартных модулей, содержащихся в системах на основе Pascal, доступны не только в откомпилированном виде, но и в виде исходных текстов (но не все, ведь должны же быть и секреты фирмы). Наличие исходных текстов очень удобно для программистов. Во-первых, они задают образцы стиля программирования и типовых приемов. Во-вторых, можно при необходимости внести изменения в исходные тексты и заново откомпилировать модули. Поэтому модуль с открытым исходным текстом ценится выше, чем объектный код даже с подробными комментариями и инструкцией по использованию.

12.7. Стандартные модули

Любая система программирования на основе языка Pascal содержит немалый набор стандартных модулей, которые входят в комплект поставки. Их состав постоянно обновляется даже при выпуске новых версий одной и той же системы, поэтому перечислить все имеющиеся модули не представляется возможным.

Однако модуль, содержащий основную библиотеку стандартных подпрограмм, неизменно имеет имя `system`. Без этого модуля не может быть выполнена ни одна программа.

Обратите внимание — лишь один модуль `system` ввиду его исключительной важности подключается к любой программе автоматически. В обычных условиях работы можно даже не подозревать о существовании этого модуля, однако в случае отсутствия или повреждения файла, содержащего модуль `system`, при запуске любой программы тут же появляется сообщение о его отсутствии. Например, Turbo Pascal выдает сообщение **File not found (SYSTEM.TPU)**.

В Delphi модуль `system` по-прежнему составляет основную библиотеку, однако большое количество стандартных функций размещено в модуле `sysutils`.

Этот модуль не подключается автоматически, но при создании нового приложения строка `uses sysutils;` автоматически формируется системой. Обычно при создании консольных приложений бывает достаточно одних только модулей `system` и `sysutils`. Основная библиотека Delphi, которая используется при визуальной разработке приложений Windows, называется VCL — Visual Component Library. Она полностью доступна в исходных текстах и размещена в большом количестве модулей, имена которых не будем перечислять.

При работе в среде Borland (Turbo) Pascal большое значение имеют стандартные модули, входящие в состав файла turbo.tpl, который является основной библиотекой. Собственно, для того, чтобы вся система программирования была работоспособной, достаточно иметь на диске всего два файла — turbo.exe (bp.exe) и turbo.tpl (они легко размещаются на одной дискете). Кроме модуля `system` в состав библиотеки входят:

- модуль `crt`. Он содержит процедуры и функции, обеспечивающие управление текстовым режимом работы экрана, а также управление клавиатурой и звуком. Подпрограммы, входящие в модуль, могут управлять перемещением курсора в произвольную позицию экрана, менять цвет фона экрана и выводимых символов, создавать окна, управлять звуком, а также обеспечивать чтение кодов нажимаемых клавиш;

  Замечание

  Стандартный модуль `crt`, входящий в состав библиотеки Turbo.tpl, не работает на компьютерах с тактовой частотой, превышающей 300 МГц — при попытке его подключения к любой программе возникает ошибка **Division by zero** (Деление на ноль). Ошибка в исходном коде инициирующей части модуля легко исправляется, поэтому имеются модифицированные варианты библиотеки Turbo.tpl, в которой модуль `crt` работает корректно. В версии Borland Pascal для Windows имеется модуль `wincrt`, который эмулирует текстовый режим в окне Windows. Этот модуль имеет практически тот же набор функций, что и модуль `crt`, но разработан специально для операционной системы Windows. В среде Turbo Pascal этот модуль не работает.

- модуль `dos`. В модуль входят процедуры и функции, организующие доступ ко всем средствам дисковой операционной системы DOS. Большинство из них правильно работает и под управлением различных версий Windows;
- модуль `printer`. Предоставляет простой способ для вывода информации на печатающее устройство. Выводить на печать информацию можно с помощью следующей программы:

```
uses printer;
begin
   writeln(LST,'Turbo Pascal');
end.
```

Замечание

Во многих случаях при использовании современных принтеров могут возникнуть трудности с кодировкой русских букв. Проблему можно решить при использовании соответствующего *драйвера* — специально написанной программы управления периферийным устройством (в том числе и принтером). Однако эта тема выходит за рамки данной книги.

- *Модуль* `graph` обычно представлен в виде отдельного файла graph.tpu. Он содержит обширный набор типов, констант, процедур и функций, служащих для управления графическим режимом работы экрана. С помощью подпрограмм, входящих в модуль, можно создавать разнообразные графические изображения;
- Еще один полезный модуль, появившийся в Turbo Pascal 7.0, — *модуль* `strings` — позволяет работать со строками текста неограниченной длины *(подробнее о строках см. гл. 14)*.

Учитывая, что в большинстве случаев изучение основ программирования начинается со среды Turbo Pascal, в *приложении* 2 нами представлено довольно обстоятельное описание модулей `crt` и `graph`, а также некоторых полезных подпрограмм модуля `dos`.

Модули `crt` и `graph` просты в изучении и предоставляют начинающим программистам большой набор несложных, но мощных средств управления устройствами компьютера: экраном, клавиатурой, динамиком. Использование модулей позволит превратить ваши учебные программы в полноценные программные продукты, обладающие удобным *пользовательским интерфейсом* и отличающиеся красивым оформлением.

12.8. Контрольные вопросы и задания

Вопросы

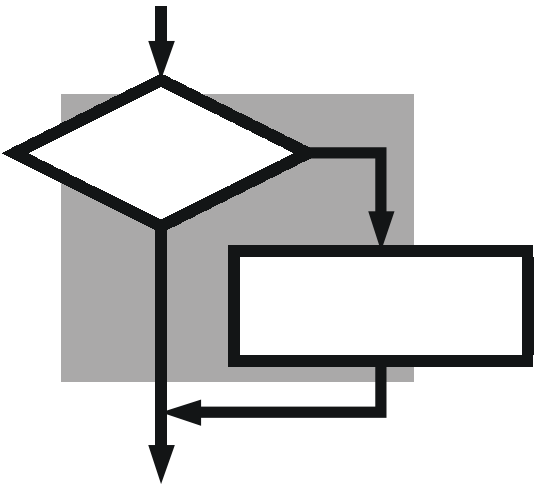
1. Что такое библиотечные модули? Чем вызвана необходимость их использования?
2. В чем состоит принцип модульности? Перечислите преимущества модульного программирования.
3. На каком этапе процедуры и функции, содержащиеся в модуле, подключаются к программе, которая их использует?
4. Перечислите названия и укажите назначение основных частей модуля.
5. Можно ли опускать заголовок модуля подобно заголовку программы? Почему?
6. Какие требования предъявляются к имени файла, в который помещен исходный текст модуля?

7. Как выполняется подключение модуля к программе?
8. Можно ли в программе использовать глобальные переменные, описанные в исполняемой части подключенного к ней модуля? А в интерфейсной части? Можно ли в модуле использовать глобальные переменные программы?

Задания

1. Доработайте модуль `triangle`, включив в него другие полезные подпрограммы для расчета треугольников. Все необходимые формулы найдете в заданиях к *гл. 7*.
2. Разработайте модуль `mathematics` (математика), включив в него подпрограммы для решения математических задач численными методами, приведенные в качестве примеров к *гл. 10* (листинги 10.9—10.11).
3. Разработайте модуль `sequences` (последовательности), включив в него подпрограммы формирования следующих числовых последовательностей:
   - простые числа;
   - сверхпростые числа;
   - числа Фибоначчи;
   - палиндромы;
   - числа, сумма цифр которых является параметром.

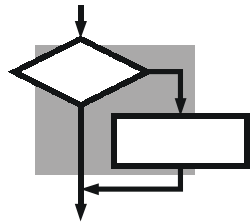
   Многие из этих последовательностей реализованы в качестве примеров в *гл. 9* и *10*.
4. Разработайте модуль `random_num` (случайные числа), включив в него следующие подпрограммы:
   - `rand(min,max)` — целое случайное число в заданном интервале;
   - `rand` — целое случайное число в интервале `[0, maxint]` (т. е. любое допустимое в языке целое);
   - `realrand(min,max)` — случайное действительное число в заданном интервале;
   - `normalrand(min,max)` — целое (или действительное) случайное число в заданном интервале, распределенное по нормальному закону (см. листинг 10.12 и пояснения к нему).



ЧАСТЬ IV

СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ ОБРАБОТКИ

Глава 13



Массивы

13.1. Общие сведения

Если работа программы связана с хранением и обработкой большого количества однотипных переменных, для их представления в программе можно использовать массивы.

Например, пусть программа пользователя выполняет некоторые действия над последовательностью целых чисел, насчитывающей сто элементов i_1, i_2, ..., i_{100}, которые требуется сохранить до конца ее работы. Вместо того чтобы описывать указанные переменные сто раз, можно один раз объявить целочисленную переменную *i*, состоящую из ста элементов, — массив.

Массив представляет собой совокупность данных (элементов) *одного типа* с *общим* для всех элементов *именем*.

Элементы массива *пронумерованы*, и обратиться к каждому из них можно по номеру (или по нескольким номерам — например, для элемента таблицы задается номер строки и столбца). Номера элементов массива иначе называются *индексами*, а сами элементы массива — *переменными с индексами (индексированными переменными)*.

Обратите внимание — данные в массивах сохраняются, как и в случае использования обычных неиндексированных переменных, только до конца работы программы. Для их долговременного хранения программа должна записать данные в файл *(см. гл. 17)*.

Характеристики массива:

- тип — *общий* тип всех элементов массива;
- размерность (ранг) — *количество индексов* массива;
- диапазон изменения индекса (индексов) — определяет *количество элементов в массиве*.

Вектор (одномерный массив) — это массив, в котором элементы нумеруются одним индексом.

Если в массиве хранится *таблица значений (матрица)*, то такой массив называется *двумерным*, а его элементы нумеруются *двумя* индексами — номером *строки и столбца* соответственно. Массивы еще большей размерности (трехмерные, четырехмерные и т. д.) на практике встречаются довольно редко.

Замечание

В памяти компьютера все элементы массива обязательно занимают одну непрерывную область (массив), отсюда и произошло это название. Двумерные массивы располагаются в памяти по строкам: сначала все элементы первой строки, затем — второй и т. д.

В качестве *номера (индекса)* элемента массива в общем случае используется выражение *порядкового типа*. Наиболее часто индекс — это целая константа или переменная типа `integer`, реже — типа `char` или `boolean`.

При обращении к элементу массива индекс указывается в *квадратных* скобках после имени массива. Например, `a[3]`, `b[1,2]`. Однако использование элементов массива в качестве обычных переменных не дает существенной выгоды. Массивы ценны тем, что их индексы сами могут быть переменными или выражениями, обеспечивая доступ не к одному элементу, а к *последовательности* элементов. Обработка массивов производится при изменении индексов элементов.

Например, следующие переменные удобно применять для просмотра в цикле элементов (или групп элементов) массива `a`:

- `a[i]` — для всех элементов;
- `a[2*i]` — для элементов, стоящих на четных местах;
- `a[2*i-1]` — для элементов, стоящих на нечетных местах.

13.2. Описание массивов

Элементы массива могут иметь любой тип, кроме файлового, индексы — порядковый тип, чаще всего диапазон. Сам массив — это *тип*, *определяемый пользователем* на основе типа его элементов и типа индексов. Поэтому имеются два основных способа описания массива:

- сначала описать тип массива в разделе описания типов, а затем описать переменную этого типа;
- описать массив непосредственно в разделе описания переменных. В этом случае используется *анонимный*, т. е. безымянный тип, понятие которого уже вводилось *(см. разд. 6.3.6)*.

Кроме того, можно описать типизированную константу типа массива. Обычную именованную константу типа массива описать, конечно, нельзя.

Рассмотрим эти способы немного подробней.

13.2.1. Описание массива в разделе *var*

Самый простой и короткий способ описания массива — это объявить переменную в разделе описания переменных `var` с использованием зарезервированного слова `array` (т. е. массив). В общем виде описание выглядит так:

- для одномерного массива:

```
var ИмяМассива: array[НижняяГраница.. ВерхняяГраница]
    of ТипЭлементов;
```

Например:

```
var a: array[1..100] of integer; { 100 элементов — целые числа }
    c: array[-3..4] of boolean;  { 8 элементов - логические значения }
    count: array ['a'..'z'] of integer;{ 26 элементов (всего 26 букв)
}
```

- для двумерного массива:

```
var ИмяМассива:array[НижняяГраницаИндекс1..ВерхняяГраницаИндекс1,
НижняяГраницаИндекс2..ВерхняяГраницаИндекс2] of ТипЭлементов;
```

Например, пусть в памяти компьютера расположена таблица чисел:

```
1   2   3   4
5   6   7   8
9  10  11  12
```

Описание этого двумерного массива y выглядит так:

```
var y: array[1..3,1..4] of integer;
```

13.2.2. Предварительное описание типа массива

Предварительное описание типа считается более строгим способом описания массива. Такое описание необходимо, например, при использовании имени массива в качестве параметра процедуры или функции. Дело в том, что при описании типа параметров по правилам не разрешено использование анонимного типа, а можно пользоваться только именованным типом, который описан заранее *(см. разд. 10.3)*. Мы еще вернемся к вопросу использования массивов как параметров процедур *(см. разд. 13.5)*.

Например, вот как выглядит описание одномерного массива:

```
type
   ИмяТипа = array[ НижняяГраница.. ВерхняяГраница ] of ТипЭлементов;
var
   ИмяМассива : ИмяТипа;
```

Описание массивов других размерностей выполняется аналогично, например:

```
const
   maxrow=10; maxcol=15;
type
   matr = array [1..maxrow, 1..maxcol] of real;
var
   matrix: matr;{matrix – имя переменной, matr –имя ее типа}
```

Приведем еще один пример описания массива, который расширит наши понятия об индексах. Опишем двумерный вещественный массив `supply` типа `energy` для хранения энергопотребления по дням недели в течение каждого часа дня:

```
type
    days=(sunday,monday,tuesday,wednesday,thursday,friday,saturday);
    hour=1..24;
    energy=array[days,hour] of real;
var supply: energy;
```

Тогда переменная `supply[friday,18]` будет обозначать количество электроэнергии, потребленной в пятницу в течение 18-го часа суток.

Здесь первый индекс массива имеет не тип диапазона, как обычно, а перечисляемый тип, что вполне согласуется с правилами, допускающими и другие порядковые типы. Возможно даже такое описание: `var a: array[byte] of integer;`. Массив `a` в этом случае будет состоять из 256 элементов, а индекс изменяться от 0 до 255 (диапазон типа `byte`).

13.2.3. Инициализация массивов начальными значениями

Массив можно описать как *типизированную константу* в разделе описания констант. Список значений элементов массива при этом заключается в круглые скобки.

Например:

```
const x: array[1..5] of integer=(1,3,5,7,9);
      y: array[1..2,1..3] of integer=((1,3,5),(2,4,6));
```

Во втором случае в памяти компьютера будет расположена таблица чисел:

```
1   3   5
2   4   6
```

В этих примерах не просто выделяется память под массив, а происходит заполнение ячеек заданными значениями по строкам.

В языке Turbo Pascal элементы такого массива можно *изменять* в ходе программы, как и любые другие типизированные константы. В Delphi такой массив можно использовать только как константу *(см. разд. 6.3.4)*, но инициализировать массив начальными значениями можно непосредственно при его описании как переменной

```
var x: array[1..5] of integer=(1,3,5,7,9); {только в Delphi!}
```

Описание массива как типизированной константы используется на практике для задания массивов с неизменными значениями элементов, например, массива из названий дней недели или количества дней в каждом месяце года.

При отладке программы, для того чтобы каждый раз не заполнять массив "вручную", также можно инициализировать его нужными начальными значениями.

13.2.4. Подробнее о выделении памяти

Описание массива требуется компилятору для выделения памяти под его элементы.

Например, для вектора `x` из 100 элементов (вещественные числа, тип `real`) в памяти будет выделено 100 ячеек по 6 или 8 байт — всего 600 (800) байт. Двумерный массив `y` из 20 строк и 30 столбцов, все 600 элементов которого — целые положительные числа, не превышающие 255, можно уложить в те же 600 байт, если воспользоваться экономным с точки зрения расхода памяти типом `byte` (600 ячеек по одному байту).

Обратите внимание — при выполнении программы вовсе не обязательно заполнять все ячейки данными, т. е. *реальное количество элементов в массиве может быть меньше, чем указано при описании, но ни в коем случае не должно быть больше.*

При использовании 16-разрядных компиляторов, например, Turbo Pascal, существует ограничение по памяти, отводимой под переменные одной программы — 64 Кбайт. Это ограничение существует и для сегмента данных, и для стека, что существенно ограничивает размеры как глобальных, так и локальных массивов. При попытке описать массив, превышающий по размерам сегмент данных, например,

```
var huge: array[1..105,1..105] of real;
```

выводится сообщение об ошибке **Structure too large** (Структура слишком большая).

При использовании 32-разрядных компиляторов практически нет никаких ограничений на размеры используемых массивов. Это великолепный подарок программисту.

Замечание

Более того, в последних версиях Delphi (начиная с четвертой) разрешено использовать динамически формируемые (открытые) массивы, размеры которых не задаются при описании, но устанавливаются по ходу программы при помощи специальной процедуры `setlength`. Немного подробнее о таких массивах будет рассказано в *гл. 18*, посвященной динамическим структурам данных. В данной главе разговор пойдет об обычных массивах.

А теперь несколько первых выводов. *Следует знать:*

- границы изменения индекса должны быть константами, причем очень удобно использовать *именованные константы*. Нельзя использовать переменные при описании границ массива, т. к. память под массив должна быть выделена *до начала выполнения программы*, а переменные получают значения *при выполнении программы*;
- нижняя граница индекса чаще всего равна 1, т. е. обычно элементы массива нумеруются начиная с единицы. Иногда нумерация начинается с нуля.

  Например, `var x: array[0..5]of integer;` — массив содержит 6 элементов, номера — с нулевого по пятый. Другие способы нумерации применяются редко;
- верхнюю границу индекса обычно определяют исходя из максимально возможного количества элементов в массиве. В большинстве реальных задач такие ограничения можно определить. Но если количество данных заранее предсказать невозможно, то в такой задаче лучше пользоваться не обычным массивом, а динамическими структурами, которые будут рассмотрены в *гл. 18*;
- поскольку в качестве индексов используются переменные и выражения, то возможна ситуация выхода индекса за границы массива. Например, если массив `a` описан как `var a: array [1..100] of integer`, то обращение к `a[i]` при `i`, равном 0 или 200, означает выход индекса за границу массива. У начинающих программистов эта ситуация встречается довольно часто;
- ошибка выхода индекса за границы выдается, только если включена директива компилятора `{$R+}`. Если эта директива отключена (а по умолчанию она как раз отключена), *выход за границы массива может испортить соседние с ним ячейки памяти*, что приведет к непредсказуемым ошибкам в программе. Чтобы не попадать в ситуацию, когда кажется, что ваш компьютер "сошел с ума", при отладке программ с массивами используйте директиву `{$R+}`. После того как программа отлажена, эту директиву лучше отключить, т. к. проверка индекса немного увеличивает время выполнения программы и размер exe-файла.

13.3. Действия над массивами

Над массивом как целым допускается только одна операция — массивы, идентичные по структуре, т. е. с одинаковыми типами индексов и элементов, могут быть операндами в операторе присваивания.

Например, если a и b являются именами массивов одного типа:

```
type mas = array[1..100] of integer;
type vec = array[1..100] of integer;
var a,b: mas; c: vec;
```

то разрешено присваивание: a:=b;. В этом случае массив a будет представлять собой точную копию массива b. Присваивание a:=c; запрещено и вызовет сообщение об ошибке **Type mismatch** (Несоответствие типов).

Здесь и далее слова "одного типа" подразумевают тип *с тем же именем*. Тип, имеющий такое же описание, но *другое имя*, не подходит.

Можно присваивать одной строке двумерного массива значения одномерного массива и наоборот. Например:

```
type
 TVec = array[1..10] of integer;
 TMatrix = array[1..10] of TVec;
var
 a: TMatrix;
 b: TVec;
begin
 b:=a[5];
end.
```

Эта операция используется довольно редко. Гораздо чаще приходится обрабатывать массив последовательно, элемент за элементом, используя циклы. Цикл for вводится в любой язык программирования специально для удобства обработки массивов, поэтому предпочтительно использовать именно этот цикл.

13.3.1. Заполнение массива данными

Инициализированные массивы уже содержат данные. В остальных случаях необходимо заполнить массив данными, прежде чем выполнять с ними какие-либо действия.

Заполнение массива можно выполнить следующими способами:

- вводом значений элементов с клавиатуры;

- присваиванием заданных или случайных значений;
- считывая значения элементов из файла *(см. гл. 17)*.

В любом случае для заполнения массива используется *цикл*. Наиболее удобен цикл `for`, причем для многомерных массивов применяются *вложенные циклы*.

Например, *"слепой"*, без использования комментариев ввод с клавиатуры выглядит так:

- векторы из 5-ти элементов:

```
for i:=1 to 5 do readln(a[i]);
```

- матрицы размером 3×2 с клавиатуры (всего потребуется ввести 6 чисел):

```
for i:=1 to 3 do
   for j:=1 to 2 do
      readln(a[i,j]);
```

На практике ввод элементов массива обычно сопровождается выводом соответствующих поясняющих текстов (например, см. листинг 13.1).

Ввод данных с клавиатуры является основным, но не единственным способом заполнения массивов. Довольно часто массив заполняется при помощи присваивания элементам определенных или случайных значений.

Например, фрагмент программы заполнения одномерного массива `x` из `n` элементов случайными числами в диапазоне от 0 до 99 включительно выглядит так:

```
randomize;   { инициализация датчика случайных чисел }
for i:=1 to n do x[i]:=random(100);
```

Заполнение массивов случайными числами часто используется при математическом моделировании и в программируемых играх для создания "случайной" ситуации.

Нередко приходится заполнять массив нулевыми значениями — *обнулять* его.

Например, обнуление элементов двумерного массива `a` осуществляет такой вложенный цикл:

```
for i:=1 to n do
   for j:=1 to m do
      a[i, j]:=0;
```

Можно предложить еще один, очень быстрый способ обнуления массива с использованием стандартной процедуры `fillchar`. Данная процедура предназначена для заполнения заданной области памяти побайтно заданным

числом или символом (вспомним, что любой массив занимает непрерывную область памяти).

Так, для обнуления массива a достаточно записать следующую инструкцию:

```
fillchar(a, sizeof(a),0);
```

13.3.2. Вывод элементов массива

Вывод значений элементов массива также выполняется в цикле for с использованием операторов write и writeln.

Например, запишем вывод вектора из 5-ти элементов:

- в столбец:
  ```
  for i:=1 to 5 do writeln(a[i]);
  ```
- в одну строку, через пробел-разделитель:
  ```
  for i:=1 to 5 do write(a[i],'  '); writeln;
  ```
- или с заданием формата, где под каждый элемент отводится 4 позиции:
  ```
  for i:=1 to 5 do write(a[i]:4); writeln;
  ```

Вывод матриц в стандартной форме записи — по строкам и столбцам — выполняется так:

```
for i:=1 to n do begin
   for j:=1 to m do write(a[i,j]:4);
   writeln;
end;
```

Замечание

Оператор writeln без параметров переводит курсор на следующую строку. Задание формата вывода помогает расположить матрицу на экране ровными столбцами.

13.3.3. Обработка одномерных массивов

Рассмотрим типовые задачи обработки на примере одномерных массивов. Для массивов других размерностей применяются аналогичные алгоритмы, которые несколько усложняются использованием вложенных циклов.

Условимся, что в векторе a содержится n элементов. Над n элементами вектора a выполним следующие действия.

1. Вычисление *суммы* элементов:
   ```
   s:=0;
   for i:=1 to n do s:=s+a[i]; { накопление суммы в s }
   ```

2. Вычисление *произведения* элементов:

```
s:=1;
for i:=1 to n do s:=s*a[i]; { накопление произведения в s }
```

3. Подсчет количества элементов, удовлетворяющих какому-либо условию. Например, подсчет количества четных чисел в целочисленном массиве:

```
k:=0;
for i:=1 to n do
   if a[i] mod 2=0 then k:=k+1;{увеличиваем на 1 счетчик четных}
                              {чисел, если число делится на 2}
```

4. Поиск элемента с заданным значением. Найти элемент — это значит выяснить его номер в массиве. Например, найдем номер первого из элементов массива a, имеющего нулевое значение. Если таких элементов нет, выведем соответствующее сообщение.

```
i:=0; { номер элементов массива }
repeat
   i:=i+1;
until(a[i]=0) { нашли } or (i=n) { массив кончился };
if a[i]=0 then writeln('Номер первого нулевого элемента = ',i)
          else writeln('Таких элементов нет');
```

Комментарий

Поскольку в данном примере требуется найти только одно, самое первое, значение, можно использовать цикл repeat. Заметим, что таким же образом можно было бы искать последний элемент, начиная поиск с конца массива и уменьшая на 1 значение переменной i. Если нужно знать номера всех элементов, имеющих заданное значение, то для их хранения придется использовать дополнительный массив, а цикл repeat заменить циклом for. В процессе поиска можно выводить искомые номера на экран непосредственно в цикле for. В этом случае необходимость использования дополнительного массива отпадает:

```
writeln('Номера элементов, имеющих нулевое значение:');
for i:=1 to n do
   if a[i]=0 { нашли 0 } then write(i,'  '); { вывели номер }
```

5. Поиск максимального элемента и его номера.

Переменная max хранит значение максимума, k — его номер в массиве:

```
max:=a[1]; k:=1;   { поиск начинаем с первого элемента }
for i:=2 to n do   { перебираем элементы, начиная со второго }
if a[i]>max then begin
```

```
    max:=a[i]; k:=i; { запоминаем значение и номер элемента, }
                     { который больше всех предыдущих}
end;
```

Аналогично, при смене знака a[i]<min, находится минимальный элемент min.

Комментарий

Если в массиве имеется несколько элементов, имеющих максимальное значение, то в переменной k будет сохранено значение первого по номеру из числа этих элементов. Если в условии записать a[i]>=max, то будет найден номер последнего из нескольких максимальных элементов. Если же нужно знать номера всех максимальных элементов, то для их хранения придется использовать дополнительный массив или выводить номера в цикле, как в примере с поиском заданного значения.

Если нужно определить только номер максимального элемента k, то можно обойтись и без переменной max. Получится совсем просто:

```
k:=1;   { поиск начинаем с первого элемента }
for i:=2 to n do   { перебираем элементы, начиная со второго }
if a[i]>a[k] then k:=i; { запоминаем только номер элемента, }
                        { который больше всех предыдущих }
```

6. Изменение значений элементов.

   Например, пусть в массиве a хранятся зарплаты n сотрудников. Тем сотрудникам, у которых зарплата меньше минимально возможной суммы, поднимем зарплату до этого минимального значения minzp.

```
{сначала задаем значение minzp}
for i:=1 to n do
   if a[i]<minzp then a[i]:=minzp;
```

Приведем два коротких примера программ, использующих рассмотренные алгоритмы.

Листинг 13.1 содержит программу, которая находит в массиве элементы, равные числу, заданному пользователем, подсчитывает их количество и выводит номер первого найденного элемента. Массив задается при помощи ввода с клавиатуры.

Листинг 13.1. Программа поиска заданного числа, количества вхождений и номера

```
const count=10;
var n, { число для поиска } a, { номер первого элемента }
```

```
    b, { количество элементов } i: integer;
    m : array [1..count] of integer;
begin
   writeln('Ввод исходного массива:');
   for i:=1 to count do begin
      write('элемент #',i,':   '); readln(m[i]);
   end;
   a:=0; b:=0;
   write ('Введите число для поиска — > '); readln(n);
   for i:=1 to count do      { поиск элемента, равного n }
      if m[i]=n then begin
         if b=0 then a:=i;{ запомним номер 1-го элемента, равного n }
         b:=b+1;          { увеличить число найденных элементов на 1 }
      end;
   if b=0 then writeln('Нет таких элементов в массиве')
   else begin
      writeln('Количество элементов, имеющих значение',n,' =', b:3);
      writeln('Первый элемент имеет номер', a:3);
   end; readln;
end.
```

Данная программа невелика по размерам, поэтому в ней не выделены отдельные процедуры, хотя ввод массива и его обработка явно представляют собой независимые задачи, причем процедура ввода массива могла бы использоваться и в других программах. Все-таки рекомендуем даже небольшие задания структурировать так, как рассказывалось в *гл. 11*, посвященной структурному программированию.

В качестве примера разработаем программу, которая формирует одномерный массив случайных чисел в диапазоне [−100,100], а затем выполняет поиск максимальных элементов в первой и второй половине массива. Для удобства разработки и отладки выделим процедуру формирования массива, заполненного случайными числами в заданном диапазоне, и функцию поиска максимального элемента в заданной части массива. Сравните листинги 13.1 и 13.2, и, возможно, вы сумеете оценить преимущества структурного подхода к разработке программ.

Листинг 13.2. Программа поиска максимальных элементов

```
const n=20;
type vector=array[1..n] of integer;
```

```
var m:vector;
procedure randomarray(var m:vector; n:integer; min,max:integer);
{ формирование массива m из n случайных чисел в диапазоне [min,max] }
var i:integer;
begin
   for i:=1 to n do
      m[i]:=random(max-min+1)+min;
end;
function getmax(m:vector; a,b:integer):integer;
{поиск максимального элемента в части массива m с номера a до номера b}
var i,max:integer;
begin
   max:=m[a];
   for i:=a+1 to b do
      if m[i]>max then max:=m[i];
   getmax:=max;
end;
begin
   randomize; randomarray(m,n,-100,100);
   write('Наибольший элемент в левой части массива равен ');
   writeln(getmax(m,1,n div 2),', а в правой- ',getmax(m,n div 2 + 1,n));
   readln
end.
```

Комментарий

Мы умышленно оформили процедуру и функцию *с параметрами* для того, чтобы придать им универсальный характер. *Обратите внимание* — в процедуре формирования массива имя массива является параметром-переменной, а в функции поиска максимума — параметром-значением.

13.3.4. Действия с двумерными массивами

Условимся, что массив a состоит из n строк и m столбцов. Рассмотрим для примера несколько типовых фрагментов.

- Суммирование элементов каждой строки.

  Результатом является массив с именем d, состоящий из n сумм элементов строк:

```
for i:=1 to n do begin
   s:=0;
```

```
   for j:=1 to m do s:=s+a[i,j];
   d[i]:=s;
end;
```

Комментарий

Аналогично вычисляется сумма в столбцах; для этого внешний цикл необходимо организовать по переменной `j` (номер столбца), а внутренний — по `i` (номер в строке).

- Поиск минимального элемента по всей матрице.

  Переменная `min` используется для хранения значения минимального элемента, `imin` — номер строки, `jmin` — номер столбца, где он находится:

```
min:=a[1,1]; imin:=1; jmin:=1;
for i:=1 to n do
   for j:=1 to m do
      if a[i,j]<min then begin
         min:=a[i,j]; imin:=i; jmin:=j;
      end;
```

- Умножение матрицы `a` на вектор x, в результате получается новый вектор `y`:

```
for i:=1 to n do begin
   s:=0;
   for j:=1 to m do
      s:=s+a[i,j]*x[j];
   y[i]:=s;
end;
```

В качестве примера приведем программу (листинг 13.3), которая реализует ввод с клавиатуры квадратного массива целых чисел по строкам и формирует два вектора. В первый записываются элементы исходного массива, расположенные на главной диагонали и выше, а во второй — элементы, лежащие ниже главной диагонали.

Замечание

Главную диагональ массива образуют элементы, у которых номер строки равен номеру столбца: a_{11}, a_{22}, … a_{nn}.

Листинг 13.3. Выделение элементов массива относительно главной диагонали

```
const row=3;      { количество строк }
      col=row;    { количество столбцов }
```

```
type vector=array[1..row*col] of integer;
var  a: array[1..row,1..col] of integer;
     b,c: vector;
     i,j,n,m: integer;
procedure print(x:vector; n: integer);
begin
   for i:=1 to n do write(x[i],' '); writeln
end;
begin
   writeln('Введите массив по строкам');
   for i:=1 to row do begin
      write('строка № ',i,'   ');
      for j:=1 to col do read(a[i,j]);
      readln;
   end;
   n:=0;   { количество элементов на главной диагонали и выше }
   m:=0;   { количество элементов ниже главной диагонали }
   for i:= 1 to row do
      for j:= 1 to col do
         if j>=i then   { элемент стоит на главной диагонали и выше }
            begin
               n:=n+1;      { найден еще один элемент }
               b[n]:=a[i,j];{ сохранить в массив b под номером n }
            end
         else           { элемент расположен ниже главной диагонали }
            begin
               m:=m+1;      { найден еще один элемент }
               c[m]:=a[i,j];{ сохранить в массив c под номером m }
            end;
   writeln('Элементы на главной диагонали и выше:'); print(b,n);
   writeln('Элементы ниже главной диагонали:'); print(c,m);
   readln;
end.
```

Комментарий

При вводе массива по строкам необходимо разделять числа в строке при помощи пробела, а ввод строки завершать нажатием клавиши <Enter>.

13.3.5. Перестановки элементов в массиве

При перестановках элементы массива меняются местами друг с другом. Хотя их значения в результате перестановок и не изменяются, но *изменяется порядок следования* элементов в массиве.

При выполнении перестановок обычно используется буферная переменная, которая служит для временного хранения одного из элементов.

Например, переставим местами первый и второй элементы массива a, используя для временного хранения переменную buf:

```
buf:=a[1]; a[1]:=a[2]; a[2]:=buf;
```

В случае перестановки строк или столбцов матрицы необходимо переставить местами все их элементы. При этом вовсе не обязательно использовать в качестве буфера обмена целый массив. Одной переменной вполне достаточно, т. к. перестановки всех элементов выполняются по очереди. Например, переставим местами все элементы первой и второй строки:

```
for j:=1 to m do begin
   buf:= a[1,j]; a[1,j]:=a[2,j]; a[2,j]:=buf;
end;
```

Хорошим примером перестановки элементов двумерного массива является *транспонирование матрицы*. При транспонировании элементы матрицы переставляются таким образом, что строки исходной матрицы становятся столбцами транспонированной матрицы. При этом элементы, расположенные на главной диагонали исходной и транспонированной матриц, одни и те же. Операция транспонирования сводится к обмену элементов матрицы, расположенных симметрично относительно главной диагонали.

Листинг 13.4 содержит программу, которая формирует матрицу случайных чисел и транспонирует ее.

Листинг 13.4. Программа транспонирования матрицы случайных чисел

```
const row=3; col=row;
var a: array[1..row,1..col] of integer;
    i,j,buf: integer;
procedure getmatrix; { формирование матрицы }
begin
   for i:= 1 to row do
      for j:= 1 to col do
         a[i,j]:=random(100);{ случайное значение элемента }
end;
```

```
procedure tmatrix; { транспонирование матрицы }
begin
   for i:=1 to row do      { просмотр всех строк матрицы }
   { просмотр элементов в строке, расположенных выше главной диагонали }
      for j:=i+1 to col do begin
   { обмен элементов, симметричных относительно главной диагонали }
         buf:=a[i,j]; a[i,j]:=a[j,i]; a[j,i]:=buf;
      end;
end;
procedure print;
begin
   for i:= 1 to row do begin
      for j:= 1 to col do write(a[i,j]:4);
      writeln;
   end;
end;
begin
   randomize;   { инициализация датчика случайных чисел }
   getmatrix;
   writeln('Исходная матрица случайных чисел:'); print;
   tmatrix;
   writeln('Результат транспонирования матрицы:'); print;
   readln
end.
```

13.3.6. Сортировка массива

Сортировка и поиск являются важнейшими понятиями информатики. *Сортировка* — это процесс упорядочивания набора данных одного типа по возрастанию или убыванию значения какого-либо признака.

При сортировке массива его элементы *меняются местами* таким образом, что их значения оказываются *упорядоченными или по возрастанию, или по убыванию*. Если в массиве есть одинаковые элементы, то говорят о сортировке по *неубыванию* или по *невозрастанию*. В большинстве случаев речь идет о сортировке одномерного массива (аналогия — построение учащихся по росту на занятиях физкультурой).

Следует знать:

- *сортировка массивов* — одно из наиболее важных действий над массивами в системах сбора и поиска информации, т. к. в отсортированных масси-

вах найти нужную информацию можно гораздо быстрее по сравнению с неотсортированными;

- существует множество различных алгоритмов сортировки, которые значительно отличаются друг от друга по *скорости* работы;
- *"быстрые"* способы сортировки массивов могут дать колоссальный выигрыш на *больших* массивах, содержащих тысячи элементов, однако для *небольших* массивов можно использовать самые *простые* способы сортировки.

Задача по сортировке массива в обычной постановке выглядит следующим образом: программа должна вывести неотсортированный массив целых чисел на экран, выполнить его сортировку по невозрастанию или неубыванию, а затем вывести отсортированный массив.

К сожалению, в рамках данной книги невозможно рассмотреть все способы сортировки. Мы рассмотрим три довольно распространенных метода: два из них просты, но работают медленно, третий — несколько сложнее для понимания, но работает значительно быстрее. Каждый из методов реализован в виде процедуры с параметрами, поэтому воспользоваться ими легко. В целях экономии места все три способа собраны в один листинг 13.5, после которого дается пояснение к каждому из способов. Исходный массив для разнообразия задан в виде типизированной константы, в Delphi — в виде инициализированной переменной (`const` заменить на `var`).

Листинг 13.5. Программа для демонстрации трех видов сортировки по невозрастанию

```
const count=20;
type vector= array [1..count] of integer;
var   m: vector =(9,11,12,3,19,1,5,17,10,18,3,19,17,9,12,20,20,19,2,5);
var i,j,buf: integer;
procedure print(m:vector; count:integer);
begin
   for i:=1 to count do write(' ', m[i]);
   writeln;
end;
procedure bublesort(var m:vector;count:integer);
begin
   for i:=2 to count do
      for j:=count downto i do
         if m[j-1]<m[j] then begin { перестановка элементов }
               buf:=m[j-1]; m[j-1]:=m[j]; m[j]:=buf;
         end;
end;
```

```
procedure linsort(var m:vector;count:integer);
begin
   for i:=1 to count-1 do
   {сравниваем поочередно первый элемент неотсортированной части массива}
   { со всеми остальными до конца }
      for j:=i+1 to count do
         if m[i]<m[j] then { меняем местами элементы } begin
            buf := m[i]; m[i] := m[j]; m[j] := buf;
         end;
end;
procedure quicksort(var m:vector;count:integer);
procedure quicks(first,last:integer);
var i,j,x,buf,l:integer;
begin
   i:=first;   { левая граница массива - первый элемент }
   j:=last;    { правая граница массива - последний элемент }
   x:=m[(first+last) div 2];  { определить, например, середину массива }
   repeat {разделим массив на две части — левая больше x, правая меньше}
      while m[i]>x do i:=i+1;
      while x>m[j] do j:=j-1;
      if i<=j then
         begin  { поменять местами элементы }
            buf:=m[i]; m[i]:=m[j]; m[j]:=buf;
            i:=i+1; j:=j-1;
         end;
   until i>j;
   {рекурсивные вызовы процедуры quicks для левой и правой части массива}
   if first<j then quicks(first,j);
   if i<last then quicks(i,last);
end;      { конец сортировки }
begin
   quicks(1,count);{первый вызов для всего массива}
end;
begin
   writeln('Исходный массив:');print(m,count);
   bublesort(m,count); { или linsort(m,count); или quicksort(m,count); }
   writeln('Отсортированный массив'); print(m,count);readln;
end.
```

Поясним суть каждого метода.

Сортировка методом "пузырька"

Один из самых популярных методов сортировки — "пузырьковый" — основан на том, что в процессе исполнения алгоритма более "легкие" элементы массива постепенно "всплывают". Особенностью данного метода является сравнение, а затем, если нужно, и перестановка *соседних* элементов. При таком подходе на первом шаге внешнего цикла самый большой элемент окажется самым первым, на втором шаге самый большой из оставшихся окажется вторым и т. д.

Все происходит очень просто и понятно, но, к сожалению, так медленно, что медленнее не бывает. Допустим, последний элемент был самым большим. Тогда для того, чтобы попасть на первое место, ему надо обменяться местами сначала с предпоследним, затем со всеми остальными без исключения. Вспомним аналогию с построением шеренги по росту: сколько времени она будет строиться, если местами меняются только соседи?

Процесс можно немного ускорить, если использовать в качестве внешнего цикла `repeat` с выходом по такому условию: каждый элемент, кроме первого, меньше своего левого соседа (для этой цели специально заводится переменная-признак). Но ускорение можно будет почувствовать только на частично отсортированных массивах. Предлагаем вам в качестве самостоятельного упражнения переработать процедуру `bublesort`, т. к. в реальных задачах встречаются ситуации, когда для получения отсортированного массива достаточно переставить всего несколько элементов.

Линейная сортировка (сортировка отбором)

Идея линейной сортировки по невозрастанию заключается в том, чтобы, последовательно просматривая весь массив, отыскать наибольший элемент и поменять его местами с первым элементом. Затем просматриваются элементы массива, начиная со второго, снова находится наибольший, который, в свою очередь, меняется местами со вторым и так далее (процедура `linsort` из листинга 13.5).

Этот метод требует меньшего числа перестановок элементов, чем "пузырек", но количество итераций (повторений цикла) у обоих методов одинаково и составляет $n(n-1)/2$, где n — количество элементов массива. В приведенном примере $n = 20$, поэтому количество итераций составляет 190.

Метод быстрой сортировки с разделением

Значительно быстрее работает алгоритм сортировки К. Хоора, который называют сортировкой с разделением или "быстрой сортировкой". В основу алгоритма положен метод последовательного дробления массива на части и обмен элементами между частями.

Сама идея, как и в предыдущих случаях, вполне понятна. Выбирается какой-либо из элементов массива (самый первый, средний или вообще слу-

чайный). После этого массив делится на две части: в левую помещаются все элементы, которые больше выбранного элемента, а в правую — все, которые меньше. В результате таких действий выбранный элемент окажется на своем месте, а массив разделится на две части, которые можно сортировать независимо друг от друга.

Каждый из двух полученных массивов снова разбивается на левую и правую части относительно произвольно выбранного элемента, и таким образом мы получаем уже четыре части, которые можно сортировать автономно. Этот процесс дроблений продолжается до тех пор, пока в каждой части не останется по одному элементу.

Такой алгоритм удобнее всего реализовать при помощи рекурсивного вызова процедуры обмена значений элементов *(см. разд. 10.6)*. Процедура `quicks` использует в качестве разделителя элемент, который находится в середине массива или части массива. После серии обменов элементов, в результате которых элемент-разделитель окажется на своем месте, процедура два раза рекурсивно вызывает сама себя (для левой и правой части).

Количество итераций, выполняемых при такой сортировке, зависит от расположения элементов в массиве и от того, насколько удачно выбран элемент-разделитель. Для приведенного примера количество итераций было специально подсчитано при помощи дополнительной переменной-счетчика (в листинге 13.5 ее нет) и составило 28 (сравните со 190). В общем случае количество итераций пропорционально $n\log_2 n$, где n — количество элементов массива.

13.3.7. Быстрый поиск в упорядоченных массивах

Поиск в неупорядоченном массиве заключается в последовательном переборе элементов массива и сравнении их значений с искомым значением (см. листинг 13.1). Разумеется, искомое значение может оказаться последним или вообще отсутствовать. При обработке крупных объемов информации (большом числе элементов) процесс поиска может затянуться. *Наиболее эффективные методы поиска работают только на упорядоченных наборах данных*, для чего их предварительно сортируют (аналогия — алфавитный порядок облегчает поиск в словаре или справочнике).

Одним из эффективных методов поиска в больших *отсортированных* массивах является *бинарный поиск, или поиск методом деления пополам*.

Идея метода — вместо просмотра подряд всех элементов массива делим его пополам. Так как массив отсортирован, то, сравнивая искомый элемент со значением среднего элемента массива, мы в состоянии сделать вывод, в какой половине он находится, и тем самым сузить область дальнейшего поиска. Затем делится пополам та часть массива, в которой находится элемент, и

так до тех пор, пока рассматриваемая часть массива не будет содержать всего один элемент.

Листинг 13.6 содержит программу, которая выполняет бинарный поиск заданного элемента в отсортированном массиве целых чисел.

Листинг 13.6. Программа бинарного поиска

```
const count=20;
      m: array[1..count] of byte =
         (20,20,19,19,19,18,17,17,12,12,11,10,9,9,5,5,3,3,2,1);
var n,i,first,last: byte;
    a: integer; found: boolean;
begin
   writeln('Исходный массив:');
   for i:=1 to count do write(' ',m[i]); writeln;
   write('Введите число для поиска: '); readln(n);
   a:=0; first:=1; last:=count;
   found:=false;                                  { элемент еще не найден }
   repeat                                                { повторять поиск }
      i:=(first+last) div 2;              { разделить массив на две части }
      if m[i]=n then found:=true
         else
            if m[i]>n then first:=i+1             { число в правой части }
               else last:=i-1;                     { число в левой части }
      a:=a+1;                          { увеличить счетчик числа итераций }
   { завершить, если найдется искомый  или будет просмотрен весь массив }
   until (found)or(first>last);
   if found then
      writeln('Искомый элемент ',n,' в массиве занимает ',i,'-ю позицию')
   else writeln('В массиве нет искомого элемента...',n);
   writeln('Поиск выполнен за ',a,' итераций'); readln;
end.
```

13.3.8. Удаление и вставка элементов в массив

Вспомним, что элементы массива располагаются в соседних ячейках памяти. Значит, без особых усилий при необходимости можно удалить только по-

следний элемент массива или добавить новый элемент в конец массива. Во всех остальных случаях придется при удалении *сдвигать* элементы массива, чтобы избавиться от "дырки", а при вставке, наоборот, *раздвигать* элементы для того, чтобы высвободить ячейку памяти для нового значения. В качестве примера рассмотрим удаление элемента.

Листинг 13.7 содержит программу, которая удаляет из массива все элементы, имеющие заданное значение. Оставшиеся после удаления ненужные ячейки в конце массива ("хвост массива") обнуляются.

Листинг 13.7. Программа ищет заданное значение и удаляет его из массива

```
const count=10;
      m: array [1..count] of byte=(33,43,2,33,90,78,60,33,33,21);
var x,   { элемент, который надо удалить }
    j,i,n: byte; { n — количество элементов }
begin
   writeln('Исходный массив:');
   for i:=1 to count do write(' ',m[i]); writeln;
   write('Введите удаляемый элемент: '); readln(x);
   n:=count; j:=0; i:=0;  { инициализация }
   repeat                 { просматривать все элементы массива }
      i:=i+1;
      while m[i]=x do
         begin         { пока очередной элемент массива равен x }
            j:=i;      { сдвинем оставшиеся элементы на 1 позицию влево }
            repeat
               m[j]:=m[j+1]; j:=j+1;
            until j>=n;
            n:=n-1;       { уменьшаем число элементов массива }
         end;
   until i>=n;
   for i:=n+1 to count do m[i]:=0;   { обнуление 'хвоста' }
   writeln('Полученный массив:');
   {если вывод 'хвоста' массива не нужен, то в цикле замените count на n}
   for i:=1 to count do write(' ',m[i]); writeln; readln;
end.
```

Комментарий

Повторяющуюся процедуру поиска в массиве удаляемого элемента мы записали в виде оператора повтора `repeat`, в котором циклически сравнивается значение очередного `i`-го элемента массива со значением переменной `x`. Если условие `a[i]=x` выполняется, то все элементы, расположенные правее `i`-го, сдвигаются влево на один. При этом `i`-й элемент заменяется `i+1`-м и т. д. Так как один элемент удален из массива, то оператором `n:=n–1` количество элементов массива уменьшается на 1.

13.3.9. Умножение матриц

Рассмотрим задачу, связанную с нахождением результата *произведения двумерных массивов*. Три продавца продают четыре вида товаров. Количество продаваемого товара сведено в таблицу (табл. 13.1).

Таблица 13.1. *Продавцы и товары*

Продавец	Товар			
	№ 1	№ 2	№ 3	№ 4
№1	5	2	0	10
№2	3	5	2	5
№3	20	0	0	0

В табл. 13.2 представлены цена каждого товара и комиссионные, получаемые от продажи.

Таблица 13.2. *Цены и комиссионные*

Товар	Цена	Комиссионные	Товар	Цена	Комиссионные
№ 1	1,5	0,2	№ 3	5	1
№ 2	2,8	0,4	№ 4	2	0.5

Вырученные от продажи деньги подсчитываются так, как показано в табл. 13.3.

Таблица 13.3. *Выручка от продажи*

Продавец	№ 1	$5 \cdot 1{,}5$	+	$2 \cdot 2{,}8$	+	$0 \cdot 5$	+	$10 \cdot 2$	=	33,1
	№ 2	$3 \cdot 1{,}5$	+	$5 \cdot 2{,}8$	+	$2 \cdot 5$	+	$5 \cdot 2$	=	38,5
	№ 3	$20 \cdot 1{,}5$	+	$0 \cdot 2{,}8$	+	$0 \cdot 5$	+	$0 \cdot 2$	=	30

А полученные комиссионные так, как показано в табл. 13.4.

Таблица 13.4. *Комиссионные*

Продавец	№ 1	5 · 0,2	+	2 · 0,4	+	0 · 1	+	10 · 0,5	=	6,8
	№ 2	3 · 0,2	+	5 · 0,4	+	2 · 1	+	5 · 0,5	=	7,1
	№ 3	20 · 0,2	+	0 · 0,4	+	0 · 1	+	0 · 0,5	=	4

Эти вычисления называются *умножением матриц* и традиционно записываются в форме, представленной в табл. 13.5.

Таблица 13.5. *Умножение матриц*

		1	2	3	4				1	2				1	2
A	1	5	2	0	10		B	1	1.5	0,2		C	1	33,1	6,8
A	2	3	5	2	5	×	B	2	2,8	0,4	=	C	2	38,5	7,1
A	3	20	0	0	0		B	3	5	1		C	3	30	4
							B	4	2	0.5					

Обратите внимание — число столбцов *A* должно быть таким же, как число строк *B*, а результат имеет столько строк, сколько их у *A*, и столько столбцов, сколько их у *B*.

Листинг 13.8 содержит программу, которая вводит исходные массивы "продавец-товар" и "цена-комиссионные", находит их произведение и выводит каждую из трех матриц на экран.

Листинг 13.8. Программа умножения матриц

```
const mmax=10;nmax=10  { максимальное количество строк и столбцов};
type matr = array[1..mmax,1..nmax] of real;
var m1,n1,m2,n2: integer;
    a,b,c: matr;
procedure input_array(m,n: integer; var mas: matr);
{ input_array – ввод двумерного массива mas из m строк и n столбцов }
var i,j: integer;
begin
   for i:=1 to m do
      for j:=1 to n do begin
         writeln('Введите элемент',i:2,',',j:2); readln(mas[i,j]);
      end;
end;{input_array}
```

```
procedure mas_mult(som1,som2: matr; var rez: matr; m1,n1,n2:  integer);
{ mas_mult – умножение двумерных массивов som1 и som2}
var i,j,k: integer;
begin
   for i:=1 to m1 do
      for j:=1 to n2 do begin
         rez[i,j]:=0.0;
         for k:=1 to n1 do rez[i,j]:=rez[i,j]+som1[i,k]*som2[k,j];
      end;
end;{mas_mult}
procedure out_array(m,n:integer; mas:matr);
{ out_array – вывод двумерного массива mas }
var i,j: integer;
begin
   writeln(m:2,n:2);
   for i:=1 to m do begin
      for j:=1 to n do write(mas[i,j]:7:3);
      writeln
   end;
end;{out_array}
begin
   writeln('Введите матрицу 1 сомножителя');
   repeat
      writeln('Введите кол-во строк <=',mmax:2); readln(m1);
      writeln('Введите кол-во столбцов <=',nmax:2); readln(n1);
   until (m1<=mmax) and (n1<=nmax);
   input_array(m1,n1,a);
   writeln('Введенная матрица 1 сомножителя');
   out_array(m1,n1,a); writeln;
   writeln('Введите матрицу 2 сомножителя');
   repeat
      writeln('Введите кол-во столбцов <=',nmax:2); readln(n2);
   until (n2<=nmax);
   m2:=n1; {число столбцов A равно числу строк B}
   input_array(m2,n2,b);
   writeln('введенная матрица 2 сомножителя');
   out_array(m2,n2,b); writeln;
   mas_mult(a,b, c, m1, n1, n2);
```

```
   writeln('Матрица - результат произведения');
   out_array(m1,n2,c); readln;
end.
```

13.4. Массивы как параметры подпрограмм

Осталось обсудить вопрос использования массивов в качестве параметров подпрограмм. Вопрос небольшой, но достаточно тонкий. В примерах этой главы массивы несколько раз использовались и в качестве параметров-значений, и в качестве параметров-переменных. Была отмечена только одна особенность — тип массива должен быть обязательно описан заранее в разделе типов, причем типы фактического и формального параметров должны быть одинаковыми.

Это, казалось бы, не очень серьезное ограничение, тем не менее, сильно связывает руки программисту, которому хотелось бы писать универсальные подпрограммы, работающие с массивами любых размеров. Но получается, что для вывода массива из 10-ти элементов надо писать одну процедуру, а для массива из 100 элементов — другую.

Чтобы снять ограничения размерности для параметров — одномерных массивов, можно использовать открытые массивы.

Открытый массив — это конструкция описания типа массива без указания типа индексов, например:

```
array of real;   или   array of integer;
```

Такое определение массива в Turbo Pascal возможно только при описании формальных параметров подпрограммы. В последних версиях Delphi появилась возможность использования открытых массивов в программе *(см. разд. 18.4)*.

Применяя открытые массивы, *следует знать*:

- индексы параметров, описанных как открытые массивы, всегда начинаются с нуля (а не с единицы);
- размерность открытых массивов (количество индексов) всегда равна 1 — за этим следит компилятор.

Реальное количество элементов фактического параметра массива можно определить двумя способами. Во-первых — передать его через дополнительный параметр, во-вторых — использовать следующие специальные функции:

- `high(ИмяМассива)` — для обычного массива возвращает верхнюю границу индекса массива, для открытого — максимальное значение индекса;
- `low(ИмяМассива)` — для обычного массива возвращает нижнюю границу индекса массива, для открытого — ноль.

Упомянем еще один способ передачи в подпрограмму массивов переменных размеров. Он состоит в использовании нетипизированного параметра *(см. разд. 10.7.4)* для передачи массива. В этом случае в теле подпрограммы описывается локальный массив больших размеров, который при помощи ключевого слова absolute совмещается в памяти с нетипизированным параметром. Такой способ хуже, чем открытые массивы, т. к. он требует больших затрат стековой памяти, однако он работает (в том числе и в Delphi). Возможно, читатель придумает для этого приема лучшее применение.

Проиллюстрируем использование открытых массивов и нетипизированных параметров на примере процедуры вывода массива переменной длины (листинг 13.9).

Листинг 13.9. Передача в процедуру массивов переменных размеров

```
procedure print1(a: array of integer; n:integer);
{ Способ 1 - задаем количество элементов в качестве параметра }
var   k:integer;
begin
   for k:=0 to n-1 do write (a[k]:4);
   writeln;
end;
procedure print2(a: array of integer);
{ Способ 2 - используем функцию high }
var   k:integer;
begin
   for k:=0 to high(a) do write (a[k]:4);
   writeln;
end;
procedure print3(var a; n:integer);
{ Способ 3 - используем нетипизированный параметр }
var   k:integer;
      x:array[1..10000] of integer absolute a;
begin
   for k:=1 to n do write (x[k]:4);
   writeln;
end;
const a: array[1..5] of integer=(1,2,3,4,5); {в Delphi вместо const– var}
      b: array[0..2] of integer=(6,7,8);
```

```
begin {программа выводит массивы разных размеров}
   print1(a,5); print2(a); print3(a,5); writeln;
   print1(b,3); print2(b); print3(b,3); readln;
End.
```

13.5. Примеры программ

В данной главе уже приводилось немало примеров программ, поэтому рассмотрим напоследок всего две задачи, которые, возможно, окажутся полезными для читателя.

13.5.1. Перевод числа в двоичную систему

Листинг 13.10 содержит программу, которая переводит числа из десятичной в двоичную систему счисления. Используется стандартный алгоритм, основанный на последовательном делении числа на 2, вычислении остатков от деления и последовательном выводе остатков в обратном порядке. Массив в этой задаче используется для хранения остатков — двоичных цифр.

Листинг 13.10. Программа перевода числа в двоичную систему счисления

```
var a: array[1..20] of 0..1; { массив для хранения двоичных цифр }
    n, { число, вводимое пользователем }
    k, { переменная-счетчик }
    kol  { количество двоичных цифр }: integer;
begin
   writeln('Введите число для перевода'); readln(n);
   kol:=0;          { пока цифр в массиве нет }
   while n>0 do   { делим n на 2 и находим остаток (0 или 1) }
      begin
         kol:=kol+1; a[kol]:=n mod 2; n:=n div 2;
      end;
   writeln('Двоичная запись данного числа:');
   { выводим элементы массива в обратном порядке }
   for k:=kol downto 1 do write(a[k]); readln;
end.
```

13.5.2. Решение системы линейных уравнений

Продолжаем тему численных методов решения математических задач.

Листинг 13.11 содержит программу, которая решает систему линейных уравнений методом Гаусса. Суть этого метода проще всего объяснить на примере.

Рассмотрим систему уравнений:

$$x + y - 2z - 3 = 0;$$
$$2x - 3y + z - 1 = 0;$$
$$3x + y - 12z + 1 = 0.$$

Для удобства запишем коэффициенты и свободные члены в виде матрицы:

$$\begin{matrix} 1 & 1 & -2 & -3 \\ 2 & -3 & 1 & -1 \\ 3 & 1 & -12 & 1 \end{matrix}$$

Вычтем из второго уравнения первое, умноженное на 2, а из третьего — первое, умноженное на 3:

$$\begin{matrix} 1 & 1 & -2 & -3 \\ 0 & -5 & 5 & 5 \\ 0 & -2 & -6 & 10 \end{matrix}$$

Вычтем из третьего уравнения второе, умноженное на 0.4 (или 2/5):

$$\begin{matrix} 1 & 1 & -2 & -3 \\ 0 & -5 & 5 & 5 \\ 0 & 0 & -8 & 8 \end{matrix}$$

Как видите, полученную систему можно легко решить, начиная с последнего уравнения: $z = 1$, $y = 2$, $x = 3$.

Листинг 13.11. Решение системы линейных уравнений методом Гаусса

```
const max=10; { максимальное число уравнений }
var a: array[1..max,1..max+1] of real; { массив коэффициентов }
    b: array[1..max+1] of real; { решение системы }
    n,k,i,e:integer;
{ вычитание j-й строки из i-й }
procedure difference(i,j:integer; var error:integer);
var k:integer;
begin
   if a[j,j]=0 then error:=1
   else begin
      { j-я строка перед вычитанием умножается на a[i,j]/a[j,j] }
      for k:=n+1 downto j do
         a[i,k]:=a[i,k]-a[j,k]*a[i,j]/a[j,j];
      { a[i,j]=0 }
```

```
      error:=0;
   end;
end;
procedure getx(i:integer); { получение корня из i-й строки }
var k:integer;
begin
   b[i]:=0;
   for k:=i+1 to n+1 do
      b[i]:=b[i]-b[k]*a[i,k];
   b[i]:=b[i]/a[i,i];
end;
begin
   writeln('Введите количество уравнений в системе'); readln(n);
   for k:=1 to n do begin
      write('Введите коэффициенты и свободный член ',k,'-го уравнения:');
      for i:=1 to n+1 do read(a[k,i]);
      readln;
   end;
   e:=0; { код ошибки }
   for k:=1 to n-1 do
      for i:=k+1 to n do
         if e=0 then difference(i,k,e);
   if e<>0 then
      writeln('Система не может быть решена без перестановки строк.')
   else
      if a[n,n]=0 then
         if a[n,n+1]=0 then writeln('Система имеет много решений.')
         else writeln('Система не имеет решений.')
      else begin { единственное решение }
         b[n+1]:=1;
         for k:=n downto 1 do getx(k);
         writeln('Система имеет единственное решение:');
         for k:=1 to n do write(b[k]:0:2,' ');
         writeln;
      end;
   readln
end.
```

13.6. Контрольные вопросы и задания

Вопросы

1. Что такое массив? В каких случаях его применение оправдано?
2. Будут ли сохраняться данные в массиве после окончания работы программы?
3. В чем состоит разница между вектором и матрицей?
4. Как расположены элементы массива, в т. ч. многомерного, в памяти компьютера?
5. Что такое индекс? Каким требованиям он должен удовлетворять?
6. В чем состоит разница между индексом и индексированной переменной?
7. В каких целях используется описание массива, каким образом оно выполняется и что в нем указывается? Приведите примеры.
8. Назовите общие и отличительные черты одномерных, двумерных и *n*-мерных массивов.
9. Может ли реальное количество элементов в массиве быть меньше, чем указано при описании? Больше? Почему?
10. Почему при описании границ массива предпочтительно употребление констант? Можно ли использовать переменные? Почему?
11. В каких целях используется описание массива как типизированной константы?
12. В каких случаях массивы могут участвовать в операторе присваивания без указания индекса?
13. Перечислите способы задания значений элементов массива. Приведите примеры.
14. Как выполняется перестановка элементов в массиве?
15. Что такое сортировка? Почему ей придается такое большое значение?
16. Какие методы сортировки массива вам известны?
17. В чем состоит идея линейной сортировки? Пузырьковой? Быстрой сортировки с разделением? Поясните с использованием листингов программ.
18. Как выполняется быстрый поиск в массиве? В каких случаях целесообразно использовать линейный, а в каких — бинарный поиск?
19. Как выполняются удаление и вставка элементов массива?
20. В каком случае матрицы можно перемножать друг с другом? Изменится ли результат при перестановке матриц-сомножителей местами?

Задания

Составьте программу для выполнения заданных действий над одномерным массивом — вектором A из N элементов. В программе необходимо предусмотреть заполнение вектора A любым удобным способом.

1. В заданном векторе выполните нормировку по максимальному элементу. Для этого необходимо поделить значения всех элементов вектора на значение его максимального элемента.
2. В заданном векторе необходимо определить количество элементов, значения которых больше среднего арифметического всех его элементов.
3. В заданном векторе есть хотя бы один ноль. Вычислить произведение элементов вектора до первого нуля.
4. В заданном векторе необходимо умножить все его элементы, имеющие четные номера, на значение его максимального элемента.
5. В заданном векторе необходимо вычислить сумму положительных элементов и произведение отрицательных.
6. В заданном векторе необходимо выполнить его сжатие, удалив все элементы, предшествующие минимальному значению.
7. В заданном векторе необходимо переставить местами первый и минимальный элементы, а также последний и максимальный элементы.
8. В заданном векторе необходимо поменять местами первый элемент с последним, второй с предпоследним и т. д.
9. В заданном векторе необходимо провести поиск среди пар его элементов и найти те пары, разность между элементами которых есть величина, наибольшая для данного вектора.
10. В заданном векторе необходимо проверить, упорядочен ли он по возрастанию или по убыванию. Вставьте в вектор число, введенное с клавиатуры, так, чтобы не нарушить упорядоченность.
11. В заданном векторе необходимо найти его оригинальные (неповторяющиеся) элементы и сформировать из них вектор B.
12. В заданном векторе необходимо найти его минимальный и максимальный элементы. Выполните сортировку элементов вектора, стоящих между минимальным и максимальным элементами: по возрастанию, если минимальный элемент стоит в векторе левее максимального; по убыванию, если максимальный элемент стоит в векторе левее минимального.
13. В заданном векторе необходимо вычислить сумму и произведение его элементов, порядковые номера которых являются: а) числами Фибоначчи; б) простыми числами.

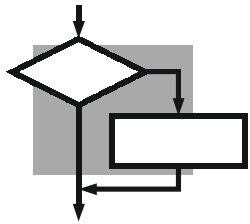
14. Задан вектор A из N элементов. Сформируйте из него матрицу B, содержащую по M элементов в строке. Недостающие элементы в последней строке, если такие будут, заполните нулями.
15. Задайте два одномерных вектора A и B из M и N элементов соответственно. Необходимо сформировать вектор C, включив в него элементы, присутствующие в A и B одновременно. Вектор C не должен содержать одинаковых элементов.
16. Задайте два упорядоченных одномерных вектора A и B из M и N элементов соответственно. Необходимо слить их в один вектор C, не нарушив при этом его упорядоченности.

Составьте программу для выполнения заданных действий над двумерным массивом вещественных чисел — матрицей $A(N, M)$. В программе необходимо предусмотреть любой удобный способ заполнения массива A, вывод исходной и, при необходимости, преобразованной матриц.

1. В заданной матрице необходимо вычислить сумму и число положительных элементов каждого столбца. Выведите результаты в виде двух строк.
2. В заданной матрице необходимо найти максимальный элемент на главной диагонали матрицы и присвоить нулевые значения другим элементам строки и столбца, на пересечении которых он стоит.
3. В заданной матрице необходимо поменять местами ее элементы симметрично побочной диагонали.
4. В заданной матрице необходимо найти наибольший и наименьший элементы и поменять их местами.
5. В заданной матрице необходимо определить максимальный и минимальный элементы. Переставьте местами строки, содержащие их.
6. В заданной матрице необходимо найти строку с наибольшей и наименьшей суммой элементов. Выведите на печать найденные строки и суммы их элементов.
7. В заданной матрице необходимо найти минимальный элемент на ее главной диагонали и поменять местами строку, в которой он находится, со строкой, номер которой вводится с клавиатуры.
8. В заданной матрице необходимо выполнить циклическую перестановку ее столбцов, при которой i-й столбец становится $i + 1$-м, а последний столбец становится первым.
9. В заданной матрице необходимо исключить строку и столбец, на пересечении которых расположен максимальный элемент ее главной диагонали.
10. В заданной матрице необходимо упорядочить по возрастанию элементы каждой строки и найти величину и местоположение максимального элемента матрицы.

11. В заданной матрице необходимо выполнить сортировку той части каждого столбца, которая находится под главной диагональю и на ней.
12. В заданной матрице необходимо выполнить сортировку нечетных строк по возрастанию, а четных — по убыванию.
13. В заданной матрице необходимо проверить, образуют ли ее элементы *магический квадрат*. В магическом квадрате суммы чисел по всем вертикалям, всем горизонталям и двум диагоналям одинаковы.
14. В заданной матрице необходимо: для каждого столбца найти число положительных и отрицательных элементов. Выполнить сортировку столбца по возрастанию, если количество положительных элементов превосходит количество отрицательных. Выполнить сортировку столбца по убыванию, если количество отрицательных элементов превосходит количество положительных. Оставить столбец без изменения, если число положительных и отрицательных элементов совпадает.
15. В заданной матрице целых чисел необходимо для каждого значения выполнить подсчет количества элементов, принимающих это значение. Подсчет проводите лишь для тех значений, которые представлены в матрице.

Глава 14



Обработка строк текста

14.1. Типы данных *char* и *string*

В Pascal имеется два типа данных для работы с текстами:

- `char` — литерный или символьный тип;
- `string` — строковый тип или просто строка.

14.1.1. Символьный тип

Значением переменных символьного типа `char` является один символ. Каждому символу соответствует *код символа* — целое число в диапазоне от 0 до 255. Из этого следует, что символьный тип является порядковым. В некоторых языках программирования (C, Java) тип `char` относят к целым типам, что разумно, т. к. в памяти компьютера нет символов — есть только их числовые коды. Это первый момент, который важно уяснить — все действия по обработке символов, в конечном счете, сводятся к действиям над *целыми числами*, расположенными *строго по порядку*.

Над данными символьного типа определены следующие *операции отношения*: `=`, `<>`, `<`, `>`, `<=`, `>=`, вырабатывающие результат логического типа.

Обратите внимание — при выполнении всех операций сравнения коды символов сравниваются как обычные целые числа. При этом получается, что любая заглавная буква всегда меньше соответствующей ей строчной, т. к. в кодовой таблице сначала располагаются все заглавные буквы, а затем строчные. Точно так же любая буква латинского алфавита всегда меньше любой буквы русского алфавита, даже если их изображения на экране практически неразличимы (A латинская, A русская).

Для данных символьного типа определены следующие стандартные функции:

- `chr(x)` — возвращает значение символа по его коду;
- `ord(ch)` — возвращает код заданного символа `ch`;
- `pred(ch)` — возвращает предыдущий символ;
- `succ(ch)` — возвращает следующий символ;

- `upcase(ch)` — преобразует строчную букву в заглавную. Обрабатывает буквы только латинского алфавита.

Например:

- `ord('A')=65`
- `chr(128)='Б'`
- `pred('Б')='А'`
- `succ('Г')='Д'`
- `upcase('n')='N'`

14.1.2. Строковый тип

Строка — это последовательность символов. Максимальное количество символов в строке *(длина строки)* может изменяться от 1 до 255. Переменную строкового типа можно определить через описание типа в разделе определения типов или непосредственно в разделе объявления переменных.

```
type
     ИмяТипа = string [максимальная длина строки];
var
    Идентификатор, ... : ИмяТипа;
```

Строковую переменную можно задать и без предварительного объявления типа:

```
var
    Идентификатор, ... : string[макс. длина строки];   или
var
    Идентификатор, ... : string;
```

Если максимальная длина строки не указывается, то она равна 255 байт.

Строковые данные могут использоваться в программе также в качестве констант. Строковая константа иначе называется литеральной константой или литералом. Иногда употребляется термин "текстовая константа" — это тоже правильно. При любом названии содержание неизменно — это последовательность произвольных символов, заключенная в апострофы (символ ' — не путать с ").

Например:

```
const stradres = 'ул. Мира, 35';      { строковая константа (литерал) }
      sadr: string[12] = ' ул. Мира' { типизированная константа }
type  str125 = string [125];
var   str1 :str125;                   { описание с заданием типа }
      st1 : string;                   { по умолчанию длина строки = 255 }
```

```
st2,st3 : string[50];
strnazv : string[280];          { ошибка, длина больше 255 }
```

В дальнейшем во всех примерах для удобства будем начинать имена всех переменных строкового типа с буквы s.

Обратите внимание — если внутри строковой константы требуется использовать апостроф, то он удваивается. Например:

```
writeln('Литерал имеет вид: ''ПроизвольныйТекст''');
```

На экране будет: `Литерал имеет вид: 'ПроизвольныйТекст'`.

Строка в языке Pascal трактуется одновременно и как скалярное (простое) значение, и как массив символов. Поэтому некоторые операции могут выполняться над строкой целиком (например, ввод и вывод), но одновременно можно обрабатывать и каждый символ по отдельности. В этом отличие строки от массива. Способ внутреннего представления строки тоже немного отличается от внутреннего представления массива.

Для строки из n символов в памяти отводится $n + 1$ байт: n байт — для хранения символов строки и один дополнительный байт — для значения текущей длины строки. Этот дополнительный байт имеет номер 0, соответственно первый символ строки имеет номер 1, второй — номер 2 и т. д.

Например, переменная `sadr` типа `string[12]` хранится в памяти таким образом (табл. 14.1).

Таблица 14.1. *Распределение памяти для хранения строковой переменной*

Номер байта	0	1	2	3	4	5	6	7	8	9	10	11	12
Его значение	8	У	л	.		М	и	р	а				

Всего строка занимает 13 байт, из них последние 4 байта оказались незанятыми, т. к. реальная длина строки составляет 8 символов, и это значение хранится в нулевом байте строки. Незанятые байты составляют "хвост" строки, в котором может быть любой "мусор", однако программа не будет обращаться к "хвосту", т. к. реальный размер строки ей известен.

Замечание

Из этого представления становится понятным и ограничение на максимальную длину строки — 255 символов, т. к. для хранения длины строки отведен всего один байт (максимальное двоичное число, которое можно уместить в один байт, — восемь подряд идущих единиц, что соответствует десятичному числу 255).

Следует отметить, что в версии Turbo Pascal 7.0 и в Delphi используется и другой способ внутреннего представления строки, который позаимствован

из языка C. Строка представляется в виде последовательности символов неограниченной длины, которая завершается символом с кодом 0 (ноль). Символ с кодом ноль — это служебный символ, который не может встретиться в качестве элемента строки, поэтому он играет роль ограничителя строки. В результате лишний байт все равно присутствует в строке, но не в начале, а в самом конце. Такая строка называется "строкой с завершающим нулем" (null-terminated string) или "строкой в стиле C". Для тех строк, которые используются в стандарте Pascal, соответственно, используется термин "строка в стиле Pascal".

Для работы со строками с завершающим нулем имеется специальный тип `pchar`, а также набор процедур и функций, которые в Turbo Pascal содержатся в модуле `strings`, а в Delphi — в стандартном модуле `sysutils`. Работать с типом `pchar` менее удобно, чем со стандартным типом `string`, но именно такой способ хранения строк используется в операционных системах семейств Windows и UNIX. Поэтому данный тип иногда требуется для того, чтобы из программы на Pascal обратиться к какой-либо стандартной функции операционной системы.

Еще один (уже третий!) способ используется в Delphi для внутреннего представления строк типа `ansistring`. Этот способ мы рассмотрим в *гл. 18*, посвященной динамическим структурам данных *(см. разд. 18.4)*. В данной главе речь пойдет только о строках в стиле Pascal.

К любому символу в строке можно обратиться как к элементу одномерного массива

```
array [0 .. n] of char
```

по номеру (индексу) данного символа в строке. Индекс определяется выражением целочисленного типа, которое записывается в квадратных скобках, как для массива.

Обратите внимание — в отличие от массивов переменные строкового типа могут участвовать как операнды в инструкциях ввода/вывода.

Например, `readln(st1); writeln(st2);`

При вводе строки количество символов в ней определяется автоматически, при этом автоматически заполняется нулевой байт. Для получения длины строки в Pascal имеется функция `length`, которая возвращает значение нулевого байта строки *(см. разд. 14.3)*.

Разумеется, никто не запрещает вводить и выводить строки по отдельным символам, как массивы, используя любой из операторов цикла. Однако такая необходимость возникает редко. Посимвольный ввод строки стоит реализовывать только в том случае, если необходимо совместить ввод с какой-то обработкой символов. Посимвольный вывод также удобно использовать для какой-либо нестандартной формы вывода.

Приведем пример, в котором исходная строка вводится целиком, а выводится по отдельным символам (листинг 14.1). В данном случае программа сначала выводит строку в "перевернутом" виде (например, вводим *РОЗА,* а выводится *АЗОР*). Затем та же строка выводится в натуральном виде, но каждое слово печатается с новой строки. Считаем, что слова разделены пробелами, причем необязательно одиночными. Обратите внимание на условие, которым проверяется начало нового слова. Разбивка строки на слова — это типовое действие со строками.

Листинг 14.1. Вывод строки в перевернутом виде и по отдельным словам

```
var s:string;
    i:integer;
begin
   write('Введите строку');readln(s);
   for i:=length(s) downto 1 do write(s[i]);
   writeln;
   for i:=1 to length(s)-1 do
      if (s[i])=' ') and (s[i+1]<>' ') then writeln
                                       else write(s[i]);
   readln;
end.
```

Приведем еще один пример, в котором выполняется обработка строки как массива символов. Программа из листинга 14.2 проверяет правильность расстановки скобок в выражении (можно считать ее маленькой частичкой компилятора, т. к. до преобразования любого выражения в исполнимый код всегда проверяется правильность расстановки скобок). В *гл. 18* (см. листинг 18.3) приводится пример программы для решения аналогичной задачи в более сложной постановке. А в конце данной главы (листинг 14.9) будет приведен пример вычисления арифметического выражения с полным анализом возможных ошибок в его записи.

Листинг 14.2. Проверка баланса скобок в скобочном выражении

```
var k,i:integer;{ k — номер символа, i — для определения баланса скобок }
    s:string;
begin
   writeln('Введите скобочное выражение:'); readln(s);
   k:=1; i:=0;
   while (k<=length(s)) and (i>=0) do
```

```
   begin
   { если i<0, то выражение неправильно: ')' предшествует '(' }
      if s[k]='(' then i:=i+1;
      if s[k]=')' then i:=i-1;
      k:=k+1;
   end;
   if i=0 then writeln('Скобки расставлены правильно.')
   { если i>0, то не хватает закрывающихся скобок }
   else writeln('Скобки расставлены неправильно.');
   readln
end.
```

14.2. Операции над строками

Над строковыми данными допустимы *операция сцепления* (конкатенации) и *операции отношения*. Используя строковые переменные, константы, функции и операцию сцепления, можно строить *строковые выражения*.

14.2.1. Операция сцепления *(+)*

Эта операция применяется для соединения нескольких строк.

Например:

выражение — `'Turbo'+' Pascal'+'7.0'`

результат — `'Turbo Pascal 7.0'`

Обратите внимание — операция конкатенации, в отличие от операции сложения в математике, не подчиняется известному правилу: "от перестановки слагаемых сумма не меняется", т. к. в этой операции каждый из операндов должен находиться строго на своем месте.

Замечание

Для того чтобы предоставить программистам возможность использовать один и тот же знак операции + для работы с данными разных типов, разработчикам компиляторов с языка Pascal пришлось изрядно потрудиться. При обработке текста программы компилятор анализирует типы операндов и формирует для операции + *различные* машинные коды для целых, вещественных и строковых типов. Такое явление в программировании принято называть *полиморфизмом*. Запомните этот термин — он пригодится при изучении последней главы, посвященной объектно-ориентированному программированию.

Для занесения в строковую переменную результата строкового выражения используется оператор присваивания.

Например, `str1 := 'Группа учащихся'; str2 := str1 + ' школы-лицея';`

Приведем пример использования операции конкатенации (листинг 14.3). Допустим, необходимо дополнить исходную строку справа символом '*' до заданной длины. Такое действие иногда выполняют при формировании важных денежных документов, с тем чтобы невозможно было ничего вписать в пустые места (скажем, после распечатывания документа).

Листинг 14.3. Дополнение строки звездочками

```
var s:string;
procedure dopstr(var s:string; const n:integer);
begin
   while length(s)<n do s:=s+'*';
end;
begin
   writeln('Введите строку:');readln(s);
   dopstr(s,80); writeln(s); readln
end.
```

Комментарий

Если в тексте процедуры поменять `s+'*'` на `'*'+s`, то строка будет дополняться пробелами не справа, а слева. Было бы разумно добавить в процедуру `dopstr` еще два параметра: символ, которым следует дополнять строку, и способ дополнения строки (слева, справа, слева и справа равномерно).

Обратите внимание — все строковые типы считаются *совместимыми по присваиванию*, т. е. можно присваивать одно другому значения строк различного размера. К сожалению, размеры строк при этом не контролируются. Если значение переменной после выполнения оператора присваивания превышает по длине максимальный размер, указанный при объявлении, *все лишние символы справа отбрасываются*, что служит источником трудно находимых ошибок.

Например:

```
const s1:string[13]='Turbo Pascal';
var   s2:string[5];
begin
   s2:=s1;  ...
```

Переменная `s2` при этом получит значение `'Turbo'`, т. к. присваиваемое значение `'Turbo Pascal'` не разместить в тех пяти байтах, которые отведены под символы переменной `s2`. Фактически операция присваивания будет выполнена неверно, хотя никакого сообщения об ошибке не будет выдано.

Замечание

При отладке задач со строками целесообразно помещать в программу директиву {$R+}, которая проверяет выход за пределы диапазона изменения индекса. Однако в приведенном выше случае и это не поможет. Остается надеяться только на собственную внимательность!

При передаче строки в качестве параметра процедуры совместимость строк разной длины возможна только при отключенной директиве компилятора строгой проверки типов строк {$V-}. По умолчанию она включена — {$V+}, поэтому строковые параметры (и формальный, и фактический) должны быть одного размера. Рекомендуем использовать данную директиву, но с осторожностью, т. к. существует риск потери символов справа в том случае, если не хватит места в строке, которая передается в качестве фактического параметра.

14.2.2. Операции отношения

Операции =, <>, >, <, >=, <= выполняют сравнение двух строковых операндов и используются в основном при проверке условий. Сравнение строк производится слева направо до первого несовпадающего символа, и та строка считается большей, в которой код первого несовпадающего символа больше. Такой способ сравнения строк называют *лексикографическим*. В жизни для обозначения лексикографического порядка используют другой термин — *алфавитный*. Строки считаются равными, если они совпадают по длине и содержат одни и те же символы.

Например:

выражение:	результат:
`'Иванов'='Иванов И.И.'`	`False`
`'Иванов'<'Иванов И.И.'`	`True`
`'program'>'PROGRAM'`	`True`

Операции отношения над строковыми данными широко используются при обработке массивов строк. Наиболее важными действиями над такими массивами являются:

- сортировка массива в лексикографическом порядке;
- быстрый поиск данных в отсортированном массиве;
- слияние двух отсортированных массивов.

При решении данных задач используются те же самые алгоритмы, что и для числовых массивов *(см. разд. 13.2.3)*, достаточно только заменить тип элементов массива на тип `string`. Например, при организации электронного телефонного справочника огромный массив абонентов обычно сортируется,

и за счет этого можно в считанные секунды найти телефон абонента по его фамилии.

В данном разделе мы приведем в качестве примера алгоритм слияния двух отсортированных массивов строк (листинг 14.4). Допустим, что происходит слияние двух предприятий и требуется объединить списки сотрудников таким образом, чтобы не нарушить алфавитного (лексикографического) порядка.

Листинг 14.4. Слияние двух отсортированных массивов строк

```
const m=4; n=3;     a:array [1..m] of
string=('Абрамов','Григорьев','Иванов','Сидоров');     b:array [1..n] of
string=('Васильев','Петров','Яковлев');var  c:array [1..m+n] of string;
k,i,j:integer; {индексы трех массивов}begin   i:=1; j:=1;
   for k:=1 to m+n do
      { если индекс i вышел за пределы массива a или b[j]<a[i] }
      if (i>m) or (b[j]<a[i]) then
      begin
         c[k]:=b[j]; j:=j+1;
      end
      else
      begin
         c[k]:=a[i]; i:=i+1;
      end;
   writeln('Массив c:');
   for k:=1 to m+n do  write(c[k],' '); readln
end.
```

14.3. Строковые процедуры и функции

Любые действия по обработке строк, вплоть до самых сложных, можно запрограммировать, используя алгоритмы обработки одномерных массивов. Однако в целях упрощения программирования обработки текстов Pascal, как и все другие языки программирования, содержит хорошо продуманный набор процедур и функций для выполнения типовых действий со строками.

Разумеется, программист может реализовать свои собственные подпрограммы, работая со строкой как с массивом символов и не забывая вовремя изменять нулевой байт. Однако рекомендуем во всех случаях, где это возможно, использовать стандартные процедуры и функции. Правильность их работы проверена многолетним опытом. Строковые процедуры и функции располагаются в модуле `system`, поэтому *для их применения дополнительные модули подключать не требуется.*

Все стандартные подпрограммы разделим на несколько групп.

14.3.1. Процедуры удаления и вставки символов

Используются две процедуры, которые корректно удаляют и вставляют символы в строку. При удалении символов оставшаяся часть строки подтягивается к началу, чтобы занять образовавшуюся после удаления "дырку". При вставке, наоборот, строка раздвигается, чтобы вместить вставляемые символы.

Процедуры удаления и вставки вызываются следующим образом:

- `delete(st,poz,n)` — удаление `n` символов строки `st`, начиная с позиции `poz`. Если значение `poz` больше, чем размер строки, ничего не удаляется.

  Например:

  ```
  st:='река Волга'; delete(st,1,5);   { в результате st='Волга' }
  ```

  Следующий фрагмент программы удаляет все пробелы из начала строки `st` (пробелы в начале строки называются *ведущими* пробелами):

  ```
  while st[1]=' ' do delete(st,1,1);
  ```

  Аналогичный фрагмент можно написать для удаления пробелов из конца строки (*завершающих* пробелов):

  ```
  while st[length(st)]=' ' do delete(st,length(st),1);
  ```

  Удаление ведущих и завершающих пробелов — типовое действие при обработке строк. Цикл `while` позволяет записать этот фрагмент в компактном виде, при этом ни один символ не удалится, если в строке нет ведущих пробелов.

- `insert(str1,str2,poz)` — вставка строки `str1` в строку `str2`, начиная с позиции `poz`. Не перепутайте: первый параметр — что вставляем, второй — куда.

  Например:

  ```
  s1 := 'Я разрабатываю программы'; s2 := 'хорошие ';
  insert(s2,s1,16);
  { в результате s1='Я разрабатываю хорошие программы' }
  ```

  При вставке в начало и конец строки действие процедуры `insert` аналогично выполнению операции конкатенации. Так, в примере со вставкой звездочек в конец строки (см. листинг 9.3) можно было бы заменить `s:=s+'*'` на `insert('*',s,length(s)+1)`.

Обратите внимание — при вставке символов нужно обязательно контролировать длину полученной строки, чтобы не потерять последние символы. При такой потере никакого сообщения об ошибке не выдается даже с включенной директивой `{$R+}`.

14.3.2. Функции для работы со строками

Имеется несколько полезных функций:

- `length(st)` — вычисляет текущую длину в символах строки `st`. Результат имеет целочисленный тип. Например, `n:=length('123456789'); { n=9 }`

  Обратите внимание — функция `length` возвращает выражение `ord(s[0])`, т. к. сам элемент `s[0]` имеет тип `char`. Некоторые программисты предпочитают не использовать функцию `length`, обращаясь непосредственно к нулевому байту.

- `copy(st,poz,n)` — выделяет из строки `st` подстроку длиной `n` символов, начиная с позиции `poz`. Если `poz` больше длины строки, то результатом будет пустая строка.

  Например:

```
s1:='Turbo Pascal';
s2:=copy(s1,1,5);
s3:=copy(s1,7,3);
{ в результате s2='Turbo'; s3='Pas' }
```

- `concat(str1,str2,...,strn)` — выполняет сцепление строк `str1`, `str2`, `strn` в том порядке, в каком они указаны в списке параметров.

  Например:

```
s:=concat('AA', 'XX', 'Y');   { в результате s='AAXXY'; }
```

  Функция `concat` выполняет те же действия, что и операция конкатенации. Например, для приведенного случая то же самое можно было записать так: `s:='AA'+'XX'+'Y';`

- `pos(str1,str2)` — обнаруживает первое появление в строке `str2` подстроки `str1`. Результат имеет целочисленный тип и равен номеру той позиции, в которой находится первый символ подстроки `str1`. Если в `str2` подстроки `str1` не найдено, результат равен `0`.

  Например:

```
s1:= 'Turbo Pascal';
n1:=pos('Паскаль',s1); n2:=pos('паскаль',s1);
{в результате n1=7; n2=0 (паскаль и Паскаль - это разные строки)}
```

  Следующий фрагмент удаляет из строки первое слово вместе со следующим за ним пробелом. При этом предусматривается удаление ведущих пробелов:

```
while st[1]=' ' do delete(st,1,1); delete(s1,1,pos(' ',s1));
```

14.3.3. Процедуры преобразования типов

Pascal позволяет преобразовывать данные числового типа в строку символов, а также преобразовывать строку в число, если она содержит последовательность символов, удовлетворяющую правилам записи чисел. Для этой цели имеются две процедуры: `str` и `val`.

- Процедуру `str` удобно использовать для вставки числовых данных в какой-либо текст, т. к. операция конкатенации и процедура `insert` могут работать только со строковыми данными.

  Вызов этой процедуры имеет вид:

  `str(number,st)` — преобразование числового значения величины `number` в строку `st`. После `number` может записываться формат, аналогичный формату вывода. Если в формате указано недостаточное для вывода количество разрядов, поле вывода расширяется автоматически до нужной длины.

  Например:

```
var S1,S2,s3,s4 : string; num1:integer;num2:real;
...
num1:=5; num2:=5.78;
str(num1,S1);str(num1:3,s2); str(num2,s3);str(num2:3:1,s4);
{ в результате s1='5'; S2='5';s3='5.780000000000E+00';s4='5.8'; }
```

- Соответственно, процедура `val` часто используется, чтобы преобразовать введенную с клавиатуры или прочитанную из файла строку в числовые данные.

  `val(st,number,code)` — преобразует значение `st` в величину целочисленного или вещественного типа и помещает результат в `numder`. `Code` — целочисленная переменная. Если во время операции преобразования ошибки не обнаружено, то значение `code` равно нулю, а если ошибка обнаружена (строковое значение не переводится в цифровое), `code` будет содержать номер позиции первого ошибочного символа, а значение `number` не определено.

  Например:

```
s1:= '5.78'; s2:= '5,78';
val(s1,num1,cod1);
val(s2,num2,cod2);
{ в результате cod1=0, cod2=2 - второй символ ошибочный }
```

Обратите внимание — использование функции `val` позволяет избежать неприятной ошибки выполнения **Invalid numeric format**, возникающей при неправильном вводе числовых данных. Приходится прибегать к такой хитрости — вводить вместо числовой переменной строковую (в ней разрешены любые символы). После этого введенная строка преобразуется в число и, если преобразование невозможно, об этом выводится сообщение на русском

языке. Чаще всего в таких случаях не прерывают программу, а просят пользователя повторить ввод. Здесь очень удобно использовать цикл repeat, т. к. в нем гарантируется выполнение тела цикла хотя бы один раз.

Например:

```
var s:string; n,error:integer;
begin
   repeat
      write('Введите число '); { просим ввести число, }
      readln(s);               { а вводим строку }
      val(s,n,error);          { преобразуем строку в число }
      if error>0 then writeln('Неверный символ № ',error)
   until error=0;
   { продолжение программы }
```

Еще один способ решения той же самой проблемы ошибок ввода числовых данных, основанный на использовании функции ioresult в цикле repeat, рассматривался ранее *(см. разд. 9.1)*. Все же более универсальным вариантом представляется использование вспомогательной строки, т. к. в этом случае можно не просто проверить правильность ввода, но исправить также некоторые ошибки при вводе.

14.4. Примеры программ

14.4.1. Вставка, удаление и замена фрагментов текста

Эти действия постоянно выполняются при редактировании текста различными текстовыми процессорами. Использование процедур и функций для работы со строками превращает программирование подобных задач обработки текстов в довольно увлекательное занятие. В качестве примеров рассмотрим две небольшие задачи.

Первая из них (листинг 14.5) позволит сократить размер текста без искажения его содержания. Удалим из текста все незначащие пробелы (ведущие, завершающие и лишние между словами). Кроме того, заменим устойчивые сочетания типа "так как", "то есть" их аббревиатурами "т. к.", "т. е.". Сокращение размеров текста особенно важно, например, при пересылке писем по электронной почте.

Листинг 14.5. Программа, сокращающая размер теста без искажения содержания

```
{$V-}{ отключили проверку полной совместимости типов }
var s:string[80];{ формальный и фактический параметры разных типов }
```

```
procedure delspace(var s:string);
begin
   { пробелы удаляются до тех пор, пока функция Pos выдает
   ненулевой результат при поиске двойного пробела в строке }
   while pos('  ',s)>0 do delete(s,pos('  ',s),1);
   if s[1]=' ' then delete(s,1,1); { удаляем пробел из начала строки }
   if s[length(s)]=' ' then delete(s,length(s),1); { и из конца }
end;
procedure itd_itp(var s:string);
const a:array[1..4] of string[15]=
     ('и так далее', 'и тому подобное','так как', 'то есть');
      a1:array[1..4] of string[5]=('и т.д', 'и т.п', 'т.к', 'т.е');
var k,p:integer;
begin
   for k:=1 to 4 do begin { сокращаем каждую из 4 фраз }
      while pos(a[k],s)>0 do begin { пока встречается фраза }
         p:=pos(a[k],s); { определяем номер первого символа фразы }
         delete(s,p,length(a[k]));{ удаляем фразу }
        { если после фразы стояла точка, то не добавляем последнюю '.') }
         if s[p]='.' then insert(a1[k],s,p) else insert(a1[k]+'.',s,p);
      end;
   end;
end;
begin
   writeln('Введите строку текста:'); readln(s);
   delspace(s); itd_itp(s);
   writeln('Сокращенный вариант:'); writeln(s); readln
end.
```

Вторая программа (листинг 14.6) может несколько увеличить размер обрабатываемой строки текста, т. к. она проверяет обязательное наличие пробела после любого знака препинания. При отсутствии пробела она вставляет его в нужное место.

Листинг 14.6. Программа, контролирующая наличие пробелов после знаков препинания

```
{ все знаки препинания поместили в служебную строку }
const ch:string[6]='.,:;!?';
var s:string;
```

```
procedure insspace(var s:string);
var k,l:integer;
begin
   k:=1;
   repeat
      for l:=1 to 6 do
      { немного усложним условие, чтобы не забыть про многоточие или
      три восклицательных знака подряд }
      if (s[k]=ch[l]) and (s[k+1]<>ch[l]) then insert(' ',s,k+1);
      k:=k+1;
   until k=length(s);{ последний символ не рассматриваем, чтобы
   строка не оканчивалась пробелом }
end;
begin
   writeln('Введите строку текста:'); readln(s);
   insspace(s);
   writeln('Вставляем пробел после знаков препинания:');
   writeln(s); readln
end.
```

14.4.2. Преобразование строчных букв в заглавные

Это тоже полезное действие; оно часто применяется в системах хранения и поиска данных. Дело в том, что при сравнении строк обязательно учитывается регистр, поэтому такие варианты записи одного слова, как `компьютер`, `Компьютер` и `КОМПЬЮТЕР`, будут восприняты как три разных слова, а в отсортированном списке они будут располагаться даже не рядом друг с другом. Учитывая данное обстоятельство, обычно при вводе информации выполняют преобразование букв слова к буквам какого-то одного регистра, чаще к заглавным.

С английскими буквами все просто. Достаточно знания одной функции `upcase`, которая преобразует к верхнему регистру один символ. Но к сожалению, эта функция не работает с русскими буквами. Придется писать свою функцию, работающую и с английскими, и с русскими буквами.

Будем использовать две вспомогательные строки: строку из всех заглавных букв русского алфавита и строку всех строчных букв. Заметим, что здесь обязательным является применение типизированных констант (или переменных), т. к. обычная строковая константа не может интерпретироваться

как массив символов, поскольку обычные константы не занимают память (листинг 14.7).

Листинг 14.7. Преобразование русских букв в заглавные

```
var s:string;
function upstring(s:string):string;
const small:string='абвгдежзийклмнопрстуфхцчшщъыьэюя';
        big:string='АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ';
var i,n:integer;
begin
   for i:=1 to length(s) do begin
      s[i]:= upcase(s[i]); { на случай латинских букв }
      n:=pos(s[i],small);{ находим номер символа в строке строчных букв }
      if n>0 then s[i]:=big[n];{ заглавная буква с таким же номером }
   end;
   upstring:=s;
end;
begin
   write('Введите строку');readln(s);
   writeln(upstring(s)); readln;
end.
```

Безусловно, обработка текстов на естественном языке (русском, английском) — важная область применения строковых переменных. Но вернемся к другим текстам, а именно, к текстам программ, записанных на языке высокого уровня. Очевидно, что при трансляции программы решается много сложных задач по обработке текста. В качестве введения в эту тему рассмотрим два примера обработки фрагментов программы — удаление комментариев и вычисление арифметических выражений.

14.4.3. Удаление комментариев из строки программы

Как известно, комментарии компилятор не обрабатывает. Разработаем программу удаления комментариев из строки программы. Будем считать, что комментарии заключены только в фигурные скобки, не рассматривая вариант (*...*), обработка которого существенно увеличит размеры текста программы. И еще одно упрощение — считаем, что все комментарии начинаются и заканчиваются внутри одной строки. В действительности именно так и есть — транслятор обрабатывает всю программу посимвольно, считывая

текст из файла как из одной большой строки. Но мы еще не научились посимвольно читать текст из файла *(см. гл. 17)*. Зато мы умеем вводить строки с клавиатуры.

"Ну, тогда все просто", — скажет читатель. Достаточно одной процедуры `delete`, внутри которой находятся две функции `pos` для определения позиции открывающей и закрывающей фигурной скобки. Конечно, задача несложная, но такой примитивный вариант программы явно не пройдет из-за некоторых нюансов:

- в строке может быть не один комментарий;
- нельзя удалять из текста директивы компилятора, которые также заключены в фигурные скобки, но в качестве первого символа содержат знак `$`;
- если фигурные скобки находятся внутри строковой константы, заключенной в апострофы, то они являются ее частью и не должны рассматриваться как признак комментария.

Для решения таких задач обычно реализуется автоматный распознаватель. В книге уже был приведен пример использования такого способа для перевода числа, записанного римскими цифрами, в десятичную систему (см. листинг 9.20). Суть такова: в процессе выполнения программа проходит через несколько состояний, и переход из одного состояния в другое происходит при определенных условиях.

В нашем примере каждое следующее состояние зависит от предыдущего состояния и от того символа, который сейчас обрабатывается (обработку выполняем посимвольно).

Выделим четыре возможных состояния:

- `out` — программа находится вне комментариев и текстовых констант;
- `lit` (от слова литерал) — программа находится внутри текстовой константы;
- `comdir` — программа находится внутри комментариев или внутри директивы компилятору (если после `'{'` стоит `'$'`, то мы имеем дело с директивой, а не с комментарием);
- `com` — программа находится внутри комментариев.

Начальное состояние равно `out`.

Схематически переходы из одного состояния в другое изображены на рис. 14.1. Над линиями со стрелками записан символ, при обработке которого выполняется переход из одного состояния в другое. Прямоугольник обозначает действие удаления комментариев (а не состояние).

Программа (листинг 4.8) реализует алгоритм перехода между состояниями. Как правило, программы для реализации автоматных распознавателей содержат оператор выбора `case`.

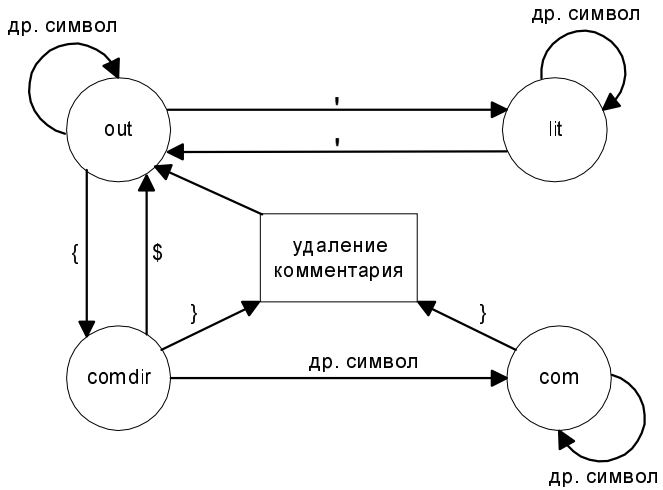


Рис. 14.1. Диаграмма переходов для программы удаления комментариев

14.8. Удаление комментариев из строки программы

```
type states=(out,lit,comdir,com); {перечисляемый тип для состояний}
var s:string;
state:states; {текущее состояние}
i,a:integer;
begin
   writeln('Введите строку программы'); readln(s);
   i:=1; state:=out; {начальное состояние}
   while i<=length(s) do begin
      case state of
         out: case s[i] of
           '''': state:=lit; {апостроф в текстовой константе удваивается}
           '{': state:=comdir;
         end; {внутреннего case}
         lit: if s[i]='''' then state:=out; {конец текстовой константы}
         comdir: case s[i] of
           '$': state:=out; {Директива компилятору}
           '}': begin       {Пустой комментарий}
                  delete(s,i-1,2); state:=out; i:=i-2;{его тоже удаляем}
                end;
           else begin
```

```
                    state:=com; a:=i-1; {a - начало комментария}
                end;
         end;
        com: if s[i]='}' then begin
           delete(s,a,i-a+1); i:=a-1; {удаляем непустой комментарий}
           state:=out;
        end;
      end; {case}
      i:=i+1; {переход к следующему символу}
   end; {цикла}
   writeln('Та же строка без комментариев:');
   writeln(s); readln;
end.
```

14.4.4. Вычисление арифметического выражения

Продолжим тему обработки арифметических выражений на этот раз более серьезным примером, чем проверка баланса скобок (см. листинг 14.2). Рискнем вычислить произвольное арифметическое выражение, правда, несколько упрощенного вида. Будем считать, что оно содержит только целочисленные константы, знаки операций и круглые скобки в любых количествах.

Изучение различных методов вычисления выражений выходит за рамки книги. Более подробную информацию, доступную начинающим, можно найти, например, в [21]. Предлагаемый нами алгоритм не требует никаких специальных знаний, кроме знания формальных способов записи конструкций языка. Эта тема рассматривалась в *гл. 2 (см. разд. 2.1.3)*. Запишем формальное определение выражения, используя расширенную форму Бэкуса—Наура (РБНФ):

```
выражение = слагаемое { ('+'|'-') слагаемое }
слагаемое = множитель { ('*'|'/') множитель }
множитель = ЧИСЛО | '(' выражение ')' | [('+'|'-')] множитель
```

Более наглядно то же самое изображено на рис. 14.2 в виде синтаксической диаграммы.

Попробуем реализовать приведенные формальные определения "в лоб", оформив вычисление выражения, слагаемого и множителя в виде отдельных функций (функции `expression`, `slag`, `mnoz`). Из формул РБНФ понятно, что expression должна вызывать `slag`, `slag` — `mnog`, а `mnog` — рекурсивно вызывать саму себя и `expression`. Налицо явная и косвенная рекурсия *(см. разд. 10.6, 10.7.1)*. Для реализации косвенной рекурсии необходимо выполнить опережающее описание процедуры `expression`.

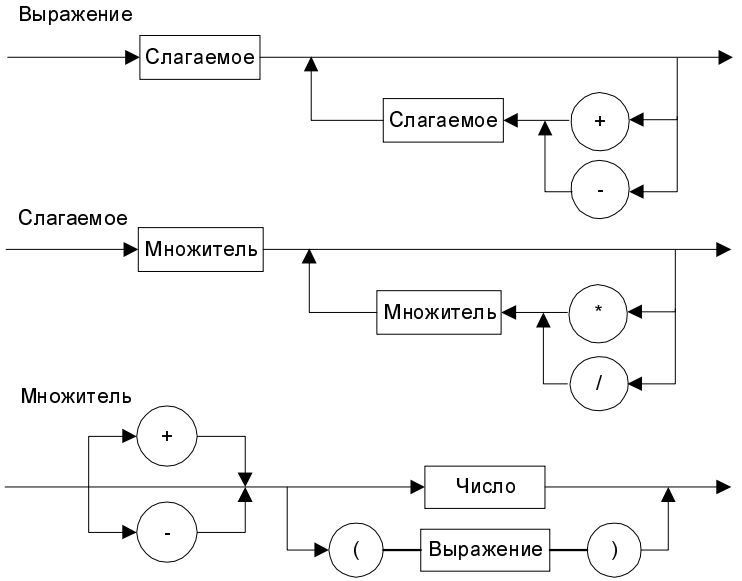


Рис.14.2. Синтаксическая диаграмма арифметического выражения

Листинг 14.9 вводит строку текста, представляющую собой выражение, содержащее целые числа, знаки арифметических операций и любое количество правильно расставленных скобок. Если выражение записано синтаксически правильно и в нем нет деления на ноль, то вычисляется его значение.

Листинг 14.9. Вычисление арифметического выражения

```
{$N+} {требуется для работы с типом double только в Turbo Pascal}
var s: string;
{возвращает следующий значимый символ, i - текущая позиция}
function nextsymbol(const s:string; var i: integer):char;
   {Пропуск пробелов}
   procedure skipspaces(const s:string; var i: integer);
   begin
    while (i<=length(s)) and (s[i]=' ') do inc(i);
   end;
begin
 skipspaces(s,i);
 if i>length(s) then nextsymbol := chr(0) else nextsymbol := s[i];
 inc(i);
end;
```

```
procedure error;
begin
 writeln('Синтаксическая ошибка'); readln; halt;
end;
{целое число}
function intnumber(const s:string; var i:integer):double;
{ выделение целого числа }
var
 res: double; f,tmp: integer;
begin
 f := i;
 while s[i] in ['0'..'9'] do inc(i);
 val(copy(s,f,i-f),res,tmp);
 intnumber := res;
end;
function expression(const s:string; var i:integer):double;forward;
{множитель}
function mnoz(const s:string; var i:integer):double;
var
 ch: char;
begin
 ch := nextsymbol(s,i);
 case ch of
  '0'..'9' :  begin dec(i); mnoz :=intnumber(s,i); end;
  '(' : begin
          mnoz := expression(s,i);
          ch:=nextsymbol(s,i);
          if ch<>')' then error;
        end;
  '+': mnoz := mnoz(s,i);
  '-': mnoz := - mnoz(s,i);
  else error;
 end;
end;
{слагаемое}
function slag(const s:String; var i:Integer):double;
var
 ch: char; res: double;
```

```
begin
 res := mnoz(s,i);
 ch := nextsymbol(s,i);
 while ch in ['*','/'] do
 begin
  if ch='*' then res:=res*mnoz(s,i) else res:=res/mnoz(s,i);
  ch := nextsymbol(s,i);
 end;
 dec(i);
 slag := res;
end;
{выражение}
function expression(const s:string; var i:integer):double;
var
 ch: char; res: double;
begin
 res := slag(s,i);
 ch := nextsymbol(s,i);
 while ch in ['+','-'] do
 begin
  if ch='+' then res:=res+slag(s,i) else res:=res-slag(s,i);
  ch := nextsymbol(s,i);
 end;
 dec(i);
 expression := res;
end;
function calc(const s:string):double;
var i: integer;
begin
 i:=1; calc := expression(s,i);
end;
begin
 writeln('Введите выражение: '); readln(s);
 writeln('Результат: ',calc(s):0:3); readln;
end.
```

Теперь уже нетрудно превратить представленный листинг в полноценный *интерпретатор арифметического выражения* (именно интерпретатор, по-

скольку он выполняет и обработку, и вычисление выражения). Предлагаем самостоятельно проделать следующие действия:

1. Добавить возможность обработки не только целых, но и действительных чисел.
2. Добавить наиболее распространенные стандартные функции (допустим, такой набор: синус, косинус, натуральный логарифм, экспонента и квадратный корень; неплохо добавить и функцию возведения в степень, например, с именем `power`).
3. Добавить возможность обработки переменных, значения которых передаются в интерпретатор как параметры. Для начала можно ввести одну переменную с именем `x`, на следующем шаге доработки имя переменной сделать параметром.

Если вы справитесь самостоятельно с этими задачами и оформите интерпретатор в виде самостоятельного модуля, то этот модуль можно будет использовать при решении математических задач, где требуется задавать вид функциональной зависимости. Например, можно значительно улучшить качество программ, приведенных в листингах 10.9, 10.10, 10.11.

14.5. Контрольные вопросы и задания

Вопросы

1. Какие типы данных используются в Pascal для обработки текста?
2. Почему символьный тип относят к порядковым?
3. Как отдельные символы представляются в памяти компьютера?
4. Поясните, что означает высказывание: любая буква латинского алфавита всегда меньше любой буквы русского алфавита?
5. Что такое строка?
6. Какова максимально возможная длина строки? Как определить текущую длину строки?
7. С какой целью для хранения строки в памяти компьютера выделяется дополнительный байт с номером ноль?
8. Почему в незанятых байтах строки может быть любой "мусор"?
9. Как можно обратиться к отдельным символам строки? организовать ее вывод?
10. Какие операции допустимы над строковыми данными? Приведите примеры.
11. Что такое конкатенация строк? Приведите примеры.

12. Что произойдет, если значение строковой переменной после выполнения оператора присваивания превысит по длине максимальный размер, указанный при ее объявлении? Почему?
13. Какой порядок называется лексикографическим?
14. На что программист должен обязательно обращать внимание при вставке символов? Почему?
15. Перечислите функции для работы со строками. Приведите примеры их использования.
16. Перечислите процедуры преобразования типов. Приведите примеры их использования.

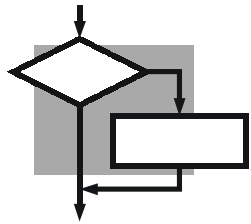
Задания

Составьте программу для обработки строки текста, введенной с клавиатуры. Все слова в строке разделены пробелами. Полученные результаты выведите на экран.

1. Выполните вывод строки текста, последовательно сокращая ее каждый раз на один символ до тех пор, пока в строке не останется последний символ.
2. В заданной строке текста удалите первое и последнее слово. Учтите, что в начале и в конце строки могут быть пробелы.
3. В заданной строке текста выведите самое короткое (длинное) слово. Учтите, что таких слов может быть несколько. Удалите (удвойте) эти слова.
4. В заданной строке текста удалите первое слово, начинающееся на заданную букву. Буква задается при помощи ввода.
5. В заданной строке текста необходимо выбрать все цифры и записать их в массив. Подсчитайте количество цифр.
6. В заданной строке текста определите наибольшее количество цифр, идущих в ней подряд.
7. В заданной строке текста определите число групп символов и число групп цифр.
8. В заданной строке текста определите слова, которые начинаются и заканчиваются на одну и ту же букву.
9. В заданной строке текста определите число различных букв.
10. Дана произвольная строка текста. Выясните, является ли она палиндромом, т. е. читается ли строка слева направо так же, как и справа налево, например: ПОТОП, НАГАН.
11. Дана произвольная строка текста. Выполните сортировку ее символов в порядке возрастания их номеров в таблице ASCII. Например, если введено: `'CBA'`, в результате должно быть получено `'ABC'`.

12. Заданы фамилия, имя и отчество человека, разделенные пробелами. Напечатайте его фамилию и инициалы, заканчивающиеся точками.
13. Для каждого слова текста укажите долю согласных в процентах. Определите слово, в котором эта величина минимальна.
14. Составьте программу, которая записывает прописью данное число, не превосходящее 1000.
15. Даны два слова. Составьте программу, определяющую, можно или нет из букв слова `A` составить слово `B`.
16. Составьте программу шифрования текстового сообщения. Можно использовать простейший способ шифрования, при котором шифровальщик задает ключ шифровки — целое число, определяющее величину смещения букв русского алфавита. Например, при значении ключа, равном 3, в тексте буква `а` меняется на `г` и т. д. Составьте программу дешифрования текстового сообщения, зашифрованного вашей программой.

Глава 15



Множества

Множества имеют большое значение в математике, поэтому не удивительно, что в языке Pascal имеется такой тип данных.

15.1. Понятие множества

Множество — это набор элементов одинакового типа, которые рассматриваются как единое целое. Элементы множества не пронумерованы, следовательно, *нельзя обратиться к отдельному элементу множества по его индексу*. Поэтому множества используются в тех задачах, где порядок следования элементов данных не имеет значения (например, множество гласных или согласных букв, множество ходов шахматной фигуры из определенного положения и т. д.).

Тип элементов множества называется *базовым типом* множества. *Область значений типа множества* — набор всевозможных подмножеств, составленных из элементов базового типа.

В языке Pascal имеются ограничения на базовый тип. Это может быть только порядковый тип, количество значений которого не превышает 256. Из простых типов к таким относятся `char`, `byte`, `boolean`. Разрешается использовать перечисляемый тип и диапазон (если он включает не больше 256 элементов).

Эти существенные ограничения не позволяют использовать множества в серьезных задачах обработки данных. Все же для ряда задач применение множеств может обеспечить серьезные преимущества по сравнению с использованием других структур данных — массивов или строк.

Количество элементов множества иногда называют *мощностью* данного множества.

При задании значений элементов множества используются квадратные скобки.

Например: `[1,2,3,4], ['a','b','c'], ['a' .. 'z']`

Если множество не имеет элементов, оно называется *пустым* и обозначается `[]`. Пустое множество включено в любое другое.

Для объявления множественного типа используется словосочетание `set of` (множество из). Формат объявления множественных типов следующий:

```
type
    ИмяТипа = set of ТипЭлементовМножества;
var
    ИмяПеременной, ... : ИмяТипа;
```

Можно описать переменные множественного типа и без предварительного объявления типа:

```
var ИмяПеременной, ... : set of Тип;
```

Можно объявить константы множественного типа:

```
const ИмяКонстанты=[ЗначениеМножества];
```

а также типизированные константы:

```
const ИмяКонстанты:ТипМножества=[ЗначениеМножества];
```

Например:

```
const number = [1,4,7,9];
type simply = set of 'a'..'h';
var pr : simply;
    letter : set of char; {без предварительного описания в разделе типов}
```

В данном примере в множество `pr` могут входить значения символов латинского алфавита от `'a' до 'h'`; в множество `letter` — значения любых символов. Попытка присвоить переменной другие значения вызовет ошибку выполнения.

Замечание

В памяти множества представлены особым образом. Каждому значению базового типа множества в памяти отводится 1 бит (не байт!). Следовательно, максимальный размер ячейки памяти, отводимой под множество, составляет 32 байта. Поскольку все значения порядкового типа расположены строго по порядку, 1 в соответствующем бите означает наличие данного значения в множественной переменной, а 0 — отсутствие.

Исходя из особенностей внутреннего представления множеств, можно сделать два основных вывода:

- *в множестве не может быть одинаковых элементов*, что согласуется и с нашими математическими понятиями;
- все операции над множествами выполняются значительно эффективней, чем над другими структурами данных.

15.2. Операции над множествами

При работе с множествами допускается использование следующих операций:

- отношения (=, <>, >=, <=);
- объединения множеств (+);
- пересечения множеств (*);
- разности множеств (-);
- проверки принадлежности элемента множеству (in).

Рассмотрим каждую из операций в отдельности.

- *Операция "равно"* (=). Два множества A и B считаются равными, если они состоят из одних и тех же элементов. Порядок следования элементов в сравниваемых множествах значения не имеет (табл. 15.1).

Таблица 15.1. *Примеры операции "равно"*

Значение A	Значение B	Выражение	Результат
[1,2,3,4]	[4,3,2,1]	A=B	True
['a','b','c']	['c','a']	A=B	False
[1..4]	[1,2,3,4]	A=B	True

- *Операция "не равно"* (<>). Два множества A и B считаются неравными, если они отличаются по количеству элементов или по значению хотя бы одного элемента (табл. 15.2).

Таблица 15.2. *Примеры операции "не равно"*

Значение A	Значение B	Выражение	Результат
[1,2,3]	[3,1,2,4]	A<>B	True
['a' .. 'z']	['b' .. 'z']	A<>B	True
[1..4]	[1,2,3,4]	A<>B	False

- *Операция "больше или равно"* (>=). Эта операция используется для определения принадлежности одного множества другому. Результат операции A>=B равен true, если все элементы множества B содержатся в множестве A. В противном случае результат равен false (табл. 15.3).

Таблица 15.3. *Примеры операции "больше или равно"*

Значение A	Значение B	Выражение	Результат
[1,2,3,4]	[2,3,4]	A>=B	True
['a' .. 'z']	['b' .. 't']	A>=B	True
['z','x','c']	['c,'x']	A>=B	True

- *Операция "меньше или равно"* (<=). Операция используется аналогично предыдущей операции, но результат выражения A<=B равен true, если все элементы множества A содержатся в множестве B. В противном случае результат равен false (табл. 15.4).

Таблица 15.4. *Примеры операции "меньше или равно"*

Значение A	Значение B	Выражение	Результат
[1,2,3]	[1,2,3,4]	A<=B	True
['d' .. 'h']	['z' .. 'a']	A<=B	True
['a','v']	['a','n','v']	A<=B	True

- *Операция* in. Эта операция используется для проверки принадлежности какого-либо значения указанному множеству. Она обычно применяется в условных операторах (табл. 15.5).

Таблица 15.5. *Примеры операции* in

Значение A	Выражение	Результат
2	A in [1,2,3]	True
'v'	A in ['a' .. 'n']	False
x1	A in [x0,x1,x2,x3]	True

Операция in позволяет эффективно и наглядно производить сложные проверки условий, заменяя иногда десятки других операций. Например, сложное условие if (a=1) or (a=2) or (a=3) or (a=4) or (a=5) then ... можно заменить более коротким выражением if a in [1 .. 5] then ....

Часто операцию in пытаются записать с отрицанием: x not in m. Такая запись является ошибочной, правильная инструкция имеет вид: not(x in m).

- *Объединение множеств* (+). Объединением двух множеств является третье множество, содержащее элементы обоих множеств (табл. 15.6).

Таблица 15.6. *Примеры операции объединения множеств*

Значение A	Значение B	Выражение	Результат
[1,2,3]	[1,4,5]	A+B	[1,2,3,4,5]
['a' .. 'd']	['e' .. 'z']	A+B	['a' .. 'z']
[ ]	[ ]	A+B	[ ]

- *Пересечение множеств* (*). Пересечением двух множеств является третье множество, которое содержит элементы, входящие одновременно в оба множества (табл. 15.7).

Таблица 15.7. *Примеры операции пересечения множеств*

Значение A	Значение B	Выражение	Результат
[1,2,3]	[1,4,2,5]	A*B	[1,2]
['a' .. 'z']	['b' .. 'r']	A*B	['b' .. 'r']
[1,3,5]	[ ]	A+B	[ ]

- *Разность множеств* (-). Разностью двух множеств является третье множество, которое содержит элементы первого множества, не входящие во второе множество (табл. 15.8).

Таблица 15.8. *Примеры операции разности множеств*

Значение A	Значение B	Выражение	Результат
[1,2,3,4]	[3,4,1]	A-B	[2]
['a' .. 'z']	['d' .. 'z']	A-B	['a' .. 'c']
[x1,x2,x3,x4]	[x4,x1]	A-B	[x2,x3]

Листинг 15.1 содержит небольшую программу, демонстрирующую использование операций над множествами. Множества чисел заполнены следующим образом: D1 — четными числами 2, 4, 6, 8; множество D2 — числами 0, 1, 2, 3, 5; множество D3 — нечетными числами 1, 3, 5, 7, 9. После этого над множествами выполнены операции объединения, разности, пересечения.

Листинг 15.1. Операции над множествами

```
type digits=set of 0 .. 9;
var d1,d2,d3,d : digits;
```

```
begin
   d1:=[2,4,6,8]; { заполнение множеств }
   d2:=[0 .. 3,5];
   d3:=[1,3,5,7,9];
   d:=d1+d2;      { объединение множеств d1 и d2 }
   d:=d+d3;       { объединение множеств d и d3 }
   d:=d-d2;       { разность множеств d и d2 }
   d:=d*d1;       { пересечение множеств d и d1 }
end.
```

Так как в Pascal отсутствуют средства ввода/вывода элементов множества, то действие программы можно проверить, исполняя ее по шагам и наблюдая текущие значения переменных d1, d2, d3, d в окне просмотра *(см. приложение 3)*.

Тем не менее, не трудно написать процедуру для вывода элементов множества. Например, процедура для вывода множества символов может иметь следующий вид:

```
type charset=set of char;
procedure writeset(a:charset);
var c:char;
begin
   for c:=chr(0) to chr(255) do
      if c in a then write(c,' ');
   writeln;
end;
```

Обратите внимание — значения элементов множества с помощью этой процедуры всегда будут выводиться в упорядоченном виде. Это не удивительно, т. к. в памяти они находятся в упорядоченном виде.

Рассмотрим следующий пример, демонстрирующий проверку принадлежности элемента множеству. Пусть требуется определить количество гласных букв в предложении, введенном с клавиатуры. Программа для решения этой задачи приводится в листинге 15.2.

Листинг 15.2. Подсчет количества гласных букв в предложении

```
const
   glasn=['а','е','и','о','у','ы','э','ю','я',
          'А','Е','И','О','У','Ы','Э','Ю','Я'];
var s:string; p,i:integer;
```

```
begin
   write('Введите строку текста: '); readln(s);
   p:=0;
   for i:=1 to length(s) do
      if s[i] in glasn then p:=p+1;
   writeln('В строке ',p, ' гласных букв'); readln;
end.
```

Комментарий

В программе используется константа `glasn`, представляющая множество гласных букв. Проверка принадлежности символов предложения множеству гласных букв записывается операцией `in`. Разумеется, для решения этой задачи можно было бы записать вспомогательную строку из гласных букв и использовать функцию `pos` для поиска буквы в этой строке. Однако вариант с множеством предпочтительней, т. к. текст получается нагляднее, и, кроме того, проверка на принадлежность множеству выполняется намного быстрей, чем поиск символа в строке.

15.3. Примеры программ

15.3.1. Формирование случайных неповторяющихся чисел

Пусть требуется сформировать последовательность натуральных чисел от 1 до *n*, расположенных в случайном порядке без повторения значений. Такая задача может встретиться, например, при программировании игр (при формировании случайной игровой ситуации). Существуют различные способы решения данной задачи. Интересный вариант решения возникает с использованием множества. Это логично, поскольку множество не содержит одинаковых элементов по определению.

Листинг 15.3. Множество случайных неповторяющихся чисел

```
var a:set of byte;
    k,x,n:byte;
begin
   randomize; a:=[]; k:=1;
   write('Введите количество чисел: '); readln(n);
   while k<=n do begin
      x:=random(n)+1;{ определяем случайное x от 1 до n }
      if not (x in a) then begin { нет такого числа в множестве? }
```

```
        write(x,' '); a:=a+[x]; { добавляем x в множество a } k:=k+1;
      end;
   end; readln;
end.
```

Приведенный вариант обладает одним существенным недостатком — количество сформированных случайных чисел не может быть больше 255. Более длинные последовательности формируются с использованием массивов.

Ввиду важности задачи приведем здесь эффективный вариант ее решения, который основан на использовании массива. Если применить примерно тот же алгоритм, что и для множеств, то решение получится неэффективным, т. к. проверка принадлежности элемента массиву осуществляется значительно медленнее, чем проверка на принадлежность множеству. Особенно много "лишних" действий выполняется при определении последних элементов массива.

Все же существует очень быстрый вариант решения этой задачи. Идея его состоит в том, что массив сначала заполняется последовательными числами, а затем каждый его элемент меняется значением с каким-нибудь другим элементом (его номер определяется случайным образом) этого же массива (листинг 15.4).

Листинг 15.4. Массив из случайных неповторяющихся чисел

```
var a:array[1..10000] of integer;
k,l,i,n:integer;
ok:boolean;
begin
   randomize;
   write('Введите количество элементов массива: '); readln(n);
   for k:=1 to n do
      a[k]:=k; { заполняем массив последовательными числами от 1 до n }
   for k:=1 to n do
      begin
         l:=random(k)+1; { определяем номер второго элемента }
         i:=a[k]; a[k]:=a[l]; a[l]:=i; { обмен значениями a[k] и a[l] }
      end;
   writeln('Сформированный массив:');
   for k:=1 to n do write(a[k],' ');
   readln;
end.
```

15.4. Контрольные вопросы и задания

Вопросы

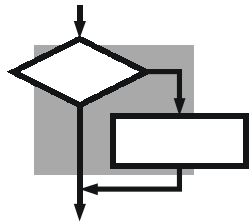
1. Что такое множество?
2. Можно ли обратиться к отдельному элементу множества по индексу? Почему?
3. В каких задачах используются множества? Их преимущества. Приведите примеры.
4. Что такое базовый тип множества? Как он задается?
5. Каким требованиям должны удовлетворять все элементы множества?
6. Какое множество называется пустым, как оно обозначается?
7. Как задается описание множественного типа?
8. Каким образом множества представлены в памяти?
9. Какие операции допустимы над множествами? Каков тип результатов выражений с применением операций над множествами?
10. Какие множества считаются равными, неравными? Имеет ли значение для сравниваемых множеств порядок следования элементов?
11. Для чего применяются операции "больше или равно", "меньше или равно"? В чем их отличие?
12. Для чего применяется операция `in`? Назовите особенности ее применения.
13. Что называется объединением множеств?
14. Что называется пересечением множеств?
15. Что называется разностью множеств?

Задания

1. Даны два множества `M` и `N`, состоящие из 10 целых чисел в диапазоне 11...00. Выделите из данных множеств подмножества `M1` чисел, делящихся на 3 без остатка, и `N1` чисел, делящихся на 2 без остатка, соответственно. Множества `M` и `N` опишите как типизированные константы. Выведите на экран мощность и значения элементов множества `MN = M1 * N1`.
2. Разработайте учебную программу для проверки знаний алфавита языка Pascal. Программа должна формировать запрос на ввод очередного символа, проверять, принадлежит ли он алфавиту языка, и предотвращать, при необходимости, повторный ввод символа, а также выводить соответствующие комментарии с оценкой. Например, если все символы введены верно — оценка "отлично", не более одной ошибки — "хорошо" и т. п.

3. Разработайте игровую программу для тренировки памяти. В ее основу положите следующее правило игры. Необходимо ввести как можно больше чисел при соблюдении следующих условий: числа должны быть из диапазона 0...255; запрещается последовательно вводить два числа, абсолютная разность между которыми меньше 7 (например, 5 и 6, 7 и 11). Программа должна заканчивать свою работу после обнаружения первой ошибки игрока. На печать выводите количество правильно введенных чисел.
4. Разработайте игровую программу. В ее основу положите следующее правило игры. В игре участвуют два человека. Первый последовательно вводит 10 символов русского алфавита. Второй пытается угадать то, что ввел его соперник, и вводит свои 10 символов. Программа считает и выводит на экран число угаданных символов. Затем игроки меняются ролями.
5. Дан одномерный массив положительных вещественных чисел. Его необходимо последовательно преобразовать согласно следующему алгоритму. Сначала обнуляется минимальный элемент массива, затем максимальный из оставшихся, далее минимальный из оставшихся и т. д. Выведите на экран сначала последний оставшийся ненулевой элемент, потом множество индексов элементов, которые были минимальными, а затем множество индексов элементов, которые были максимальными. В программе в целях экономии памяти не следует использовать вспомогательные массивы.
6. Создайте программу для ввода букв латинского алфавита. Если введенный символ не принадлежит к их числу, замените его на знак '?'. Выполните замену всех введенных заглавных букв латинского алфавита строчными буквами. Следите за тем, чтобы буквы не повторялись. Введенная повторно буква заменяется символом '*'. Выведите скорректированный результат на экран.
7. Задан ориентированный граф, содержащий четыре вершины. Хранение такого графа в компьютере можно осуществлять в следующей форме. Все вершины графа пронумерованы, каждой вершине поставлено в соответствие множество всех вершин, в которые из данной ведет ориентированная дуга. Создайте программу для ввода подобного графа. Программа должна определять, является ли граф сильносвязным, т. е. таким, у которого для любых двух вершин существуют пути как в прямом, так и в обратном направлении.
8. Создайте программу для ввода последовательности вещественных чисел в следующем формате: `номер_в_последовательности - значение`. Если значение с таким номером уже введено, то необходимо выдать соответствующий запрос на подтверждение операции перезаписи и в случае положительного ответа изменить старое значение, записав "поверх" его новое. Выведите множество введенных заново элементов, их номера и множество измененных элементов.

Глава 16



Записи

При решении задач обработки большого количества данных используются массивы. Но при работе с массивами основное ограничение заключается в том, что все элементы массива должны быть одного типа. Иногда для решения задач, в которых возникает необходимость хранить и обрабатывать комбинации данных различных типов, используются отдельные массивы для каждого типа данных, а установление соответствия между ними выполняется при помощи индексов. Но есть способ лучше.

Для *работы с комбинациями данных разных типов* в Pascal применяется комбинированный тип данных — *запись*. Запись представляет собой наиболее общий структурированный тип данных, обладающий максимальной гибкостью в силу того, что она может быть образована из неоднотипных элементов. Обычно записи служат для описания реальных объектов, при этом каждый элемент записи представляет собой одну из характеристик объекта. Например, запись о книге может содержать такие ее характеристики, как название, год издания и т. д.

16.1. Работа с записями. Примеры

16.1.1. Определение типа "запись". Правила работы с записями

Запись — это структурированный тип данных, состоящий из фиксированного числа компонентов одного или нескольких типов, называемых *полями* записи. В отличие от массива, компоненты (поля) записи могут быть разного типа. Чтобы можно было ссылаться на тот или иной компонент записи, *каждое ее поле имеет свое имя* (а не номер, как элемент массива). Записи можно объявить следующим способом.

Сначала объявляется тип записи:

```
type
   ИмяТипа = record
      ИмяПоля1: ТипПоля1;
```

```
      ИмяПоля2: ТипПоля2;
...
      ИмяПоляN: ТипПоляN
   end;
```

Затем объявляются переменные соответствующего типа.

```
var
   ИмяПеременной: ИмяТипа;
```

Например:

```
type avto=record
         number: integer;    { номер автомобиля }
         marka: string[20]; { марка автомобиля }
         fio: string[40];    { фамилия, инициалы владельца }
         address: string[60]{ адрес владельца }
      end;
var m,v: avto;
```

При выделении памяти для записи под каждое поле отводится столько байт, сколько требуется для переменной соответствующего типа. Наверное, понятно, что все поля располагаются в памяти рядом одно за другим (как элементы массива), таким образом, суммарный размер ячейки памяти для записи равен сумме размеров ячеек для ее полей. Это не трудно проверить, используя функцию `sizeof`. Например, можно записать в программе: `writeln(avto)` или `writeln(m)`. В любом случае на экран будет выведено число 125, если программа запускалась в среде Turbo Pascal, или 127 при использовании любого из 32-разрядных компиляторов (двухбайтовый или четырехбайтовый `integer`).

Можно задать в программе типизированную константу типа запись, определив значения каждого из полей. Например, для типа `avto` можно объявить константу:

```
const car:avto=(number:1000; marka:'Волга';
fio:'Иванов И.И.'; address:'ул. Горького, д.1,кв.3');
```

Значения полей записи могут использоваться в выражениях. *Обращение к значению поля* осуществляется с помощью конструкции, содержащей имя переменной и имя поля, разделенные точкой. Такая комбинация называется *составным именем*. Например, для того чтобы получить доступ к полям записи `avto`, надо записать `m.number`, `m.marka`, `m.fio`, `m.address`.

Составное имя можно использовать везде, где допустимо применение типа его поля. Для присваивания полям значений используется оператор присваивания.

Например:

```
m.number:=1964;
m.marka:='Audi - 100';
m.fio:= 'Федорова Н.В.';
m.address:='ул. Красина 53 к.1 - 73';
```

Составные имена можно использовать в инструкциях ввода/вывода:

```
readln(m.number,m.marka,m.fio,m.address);
write(m.number:4,m.marka:10,m.fio:13,m.address:23);
```

Обратите внимание — нельзя использовать в инструкциях ввода/вывода запись целиком (как и в случае массива). Например: `writeln(m)` — ошибочная инструкция.

Исключением является работа с типизированными файлами, в которые можно вывести запись или массив целиком (или прочитать сразу всю запись или массив). О работе с типизированными файлами записей будет подробно рассказано в следующей главе *(см. разд. 17.5.2)*.

Однако допускается применение оператора присваивания к записям в целом, если они имеют один и тот же тип:

```
v:=m;
```

После выполнения этого оператора значения полей записи `v` станут равны значениям соответствующих полей записи `m`.

В ряде задач удобно пользоваться массивами из записей. Их можно описать, например, следующим образом:

```
type person = record
         fio: string[20];
         age: 1..99;
         prof: string[30];
      end;
var list:array[1..50] of person;
```

Обращение к полям записи имеет несколько громоздкий вид. Значительно упрощает написание использование имеющегося в языке Pascal оператора `with`:

```
with ПеременнаяТипаЗапись do Оператор; { обычно составной оператор }
```

Один раз указав имя переменной типа запись в операторе `with`, можно работать с именами полей, как с обычными переменными.

Например:

```
with m do
   begin
      number:=1964;
```

```
      marka:='Audi - 100';
      fio:='Федорова Н.В.';
      address:='ул. Красина 53 к.1 - 73';
end;
```

16.1.2. Пример использования массива записей

Рассмотрим пример использования массива записей (листинг 16.1). Составим программу, которая организует ввод сведений об учащихся (имя, фамилия, возраст, школа, класс), заносит их в массив записей, а затем выводит сведения об учащихся по запрошенным номеру записи и номеру класса.

Для решения задачи введем тип записи `pupil` (ученик) для описания сведений об учащихся, а затем объявим массив `a`, состоящий из 10 записей типа `pupil`.

Листинг 16.1. Пример использования массива записей

```
type record_type = record      { описание типа записи }
       name: string[10];       { имя }
       surname: string[20];    { фамилия }
       school: integer;        { школа }
       class: byte;            { класс }
     end;
var school:array[1..10]of record_type;
    n: 1 .. 10;
    c:byte;
procedure input_data;
{ процедура ввода данных - используем составные имена }
begin
   writeln('Введите данные №', n,' :');
   write('Ваше имя ?'); readln(school[n].name);
   write(' Фамилия ?'); readln(school[n].surname);
   write(' Школа ?'); readln(school[n].school);
   write(' и класс ?'); readln(school[n].class);
   writeln;
end;
procedure write_data;
{ вывод на экран содержимого текущей записи - используем оператор with }
begin
   with school[n] do begin
```

```
      writeln('Имя :',name);
      writeln('Фамилия: ',surname);
      writeln('Школа: ',school);
      writeln('Класс: ',class);
   end;
end;
begin { основная программа }
   for n:=1 to 10 do input_data;
   writeln;
   writeln(' Вывод данных о 5 ученике :');
   writeln; n:=5; write_data;
   writeln(' Вывод данных по номеру класса :');writeln;
   writeln(' Задайте номер класса :'); readln(c);
   for n:=1 to 10 do if school[n].class=c then write_data;
   readln;
end.
```

Записи часто используются для работы с датами (день, месяц, год) или с временными отрезками (часы, минуты, секунды). В большинстве современных языков программирования предусмотрены специальные типы данных для работы с датой и временем (так, в Delphi имеется тип `tdatetime`), однако в Turbo Pascal все действия с датой и временем приходится программировать вручную.

Записи часто используются при работе с динамическими структурами данных и для организации файлов записей на дисках. Обо всем этом речь пойдет в следующих главах. Применение записей может улучшить исходный текст программы, если в ней используются переменные, которые можно объединить в группы по какому-либо признаку. Например, разумно использовать записи для описания координат точки на плоскости (в пространстве) или для описания комплексных чисел (см. листинг 16.2).

```
type complex=record
       re:real; { действительная часть }
       im:real; { мнимая часть }
     end;
var a,b,c: complex; { a,b,c - переменные типа complex }
begin
   a.re:=6.8;
   a.im:=1.6;
   ...
```

В данном случае можно было бы использовать две обычные переменные для хранения действительной и мнимой частей, однако использование записи делает текст программы более выразительным и понятным.

16.1.3. Модуль для выполнения операций с комплексными числами

Продолжаем тему решения математических задач. Разработаем модуль comarith, содержащий набор процедур для выполнения расчетов с комплексными числами, используя известные формулы из курса высшей математики. Модуль позволяет выполнить следующие действия:

- формирование комплексного числа;
- формирование числа, комплексно-сопряженного данному числу;
- сложение комплексных чисел;
- вычитание комплексных чисел;
- умножение комплексных чисел;
- деление комплексных чисел;
- вычисление модуля комплексного числа;
- извлечение квадратного корня из комплексного числа;
- вычисление произведения вещественного и комплексного чисел.

Такой модуль (листинг 16.2) может оказаться очень полезным при решении задач, встречающихся как в курсе высшей математики, так и в целом ряде технических дисциплин, прежде всего связанных с радиотехникой и энергетикой.

Листинг 16.2. Модуль для выполнения операций с комплексными числами

```
unit comarith;
{ интерфейсная часть }
interface
{ для описания комплексного числа используется тип complex }
type complex = record
   r:real; { действительная часть }
   i:real; { мнимая часть }
end;
procedure cmplx(re,im: real; var c: complex);
   { формирование комплексного числа: c=re+i*im }
procedure conjg(var z,c: complex);
   { формирование числа c - комплексно-сопряженного числу z }
```

```
procedure cadd(var a,b,c: complex);
   { сложение комплексных чисел (add): c=a+b }
procedure csub(var a,b,c: complex);
   { вычитание комплексных чисел (subtraction): c=a-b }
procedure cmul(var a,b,c: complex);
   { умножение комплексных чисел (multiplication): c=a*b }
procedure cdiv(var a,b,c: complex);
   { деление комплексных чисел (division): c=a/b }
procedure cabs (var z: complex; var cmod: real);
   { вычисление модуля комплексного числа c=|z| }
procedure csqrt(var z,c: complex);
   { вычисление квадратного корня из комплексного числа: c=sqrt(z) }
procedure rcmul(var x: real; var a,c: complex);
   { вычисление произведения вещественного и комплексного чисел: c=x*a }
{ исполняемая часть }
implementation
procedure cmplx(re,im: real; var c: complex);
begin
   c.r:=re;
   c.i:=im;
end;
procedure conjg(var z,c: complex);
begin
   c.r:= z.r;
   c.i:= -z.i;
end;
procedure cadd(var a,b,c: complex);
begin
   c.r:=a.r+b.r;
   c.i:=a.i+b.i;
end;
procedure csub(var a,b,c: complex);
begin
   c.r:=a.r-b.r;
   c.i:=a.i-b.i;
end;
procedure cmul(var a,b,c: complex);
begin
   c.r:= a.r*b.r - a.i*b.i;
```

```
   c.i:= a.i*b.r + a.r*b.i;
end;
procedure cdiv(var a,b,c: complex);
   var r: real;
begin
   r:= sqr(b.r) + sqr(b.i);
   if (r < 1E-10) then
      begin
         writeln('Комплексный делитель равен 0');
         exit
      end;
   c.r:= (a.r*b.r + a.i*b.i)/r;
   c.i:= (b.r*a.i - a.r*b.i)/r;
end;
procedure cabs(var z: complex; var cmod: real);
begin
   cmod:= sqrt(sqr(z.r)+ sqr(z.i))
end;
procedure csqrt(var z,c: complex);
   var cmod,are,aim: real;
begin
   cmod:= sqrt(sqr(z.r) + sqr(z.i));
   are:=sqrt((cmod+ z.r)/2);
   aim:=sqrt((cmod- z.r)/2);
   if (z.i>0.0)then
      if (z.r>0.0)then
         begin
            c.r:=are;
            c.i:=aim
         end
      else
         begin
            c.r:=-are;
            c.i:=-aim
         end
   else
      if (z.r>0.0)then
         begin
            c.r:=are;
```

```
            c.i:=-aim
         end
      else
         begin
         c.r:=-are;
         c.i:=aim
      end
end;
procedure rcmul(var x: real; var a,c: complex);
begin
   c.r:=x*a.r;
   c.i:=x*a.i;
end;
end.
```

16.2. Записи с вариантами. Пример

Возможности применения записей, имеющих строго определенную структуру, несколько ограничены. Однако в языке Pascal можно описать тип записи, содержащий несколько вариантов структуры. Записи этого типа называются *записями с вариантами*; они являются средством объединения похожих, но не идентичных по форме записей.

Запись с вариантами состоит из *фиксированной* и *вариантной* частей.

Фиксированная часть задается так же, как и при описании простых записей. Вариантная часть формируется с помощью оператора `case ... of` и может состоять из нескольких вариантов. Оператор `case` задает особое поле записи — поле *признака*, которое определяет, какой из вариантов в данный момент будет активизирован. Значением признака в каждый текущий момент выполнения программы должна быть одна из расположенных в операторе `case` констант. Константа, служащая признаком, задает вариант записи и называется *константой выбора*.

В любой записи может быть только одна вариантная часть, и, если она есть, она должна располагаться за всеми фиксированными полями. Имена полей должны быть уникальными в пределах той записи, где они объявлены. Однако если записи содержат поля-записи, т. е. вложены одна в другую, имена могут повторяться на разных уровнях вложенности. Формат записи с вариантами таков:

```
type
   ИмяТипа=record
   фиксированная часть
```

```
case ПолеПризнака : ИмяТипа of
   КонстантаВыбора1 :  (Поле : Тип,…);
   КонстантаВыбораN :  (Поле : Тип,…);
end;
```

Компоненты каждого варианта (идентификаторы полей и их типы) заключаются в круглые скобки. У части case нет отдельного end, как этого следовало бы ожидать по аналогии с обычным оператором case. Одно слово end заканчивает всю конструкцию записи с вариантами. Необходимо отметить, что количество полей каждого из вариантов не ограничено.

При использовании записей с вариантами необходимо придерживаться следующих правил:

- все имена полей должны отличаться друг от друга, по крайней мере, одним символом, даже если они встречаются в разных вариантах;
- запись может иметь только одну вариантную часть, причем вариантная часть должна размещаться в конце записи;
- если вариантная часть, соответствующая какой-либо константе выбора, является пустой, то она записывается следующим образом:

```
КонстантаВыбораN();
```

Интересно познакомиться с внутренним представлением записей с вариантами. С фиксированной частью записи, конечно, нет проблем. Но для вариантной части проблема заключается в том, что разные варианты могут потребовать и разных затрат памяти. Причем при выделении памяти еще неизвестно, какой из вариантов структуры записи реально будет использоваться (а возможно, что и все варианты будут задействованы). Поэтому до выделения памяти определяется размер наиболее длинной вариантной части, он и задает размер ячейки памяти.

Вообще, записи с вариантами — это серьезная проблема для разработчиков компиляторов и большое искушение для программистов. Смотрите, что получается — одна и та же ячейка памяти, которая выделяется для вариантной части записи, может в процессе выполнения программы заполняться данными разных типов (разные варианты, конечно, могут иметь разные типы). При этом компилятор не всегда сможет выполнить строгую проверку типов при выполнении различных операций. Поэтому, говоря, что Pascal является строго типизированным языком, иногда прибавляют "почти" [20].

Замечание

В *гл. 18* мы познакомимся с еще более сильными возможностями нарушить строгую типизацию языка. Но при этом никогда не забываем, что сильнодействующее лекарство врач выписывает только тогда, когда все обычные средства

уже перепробованы и не помогают. Традиционное же использование записей с вариантами (без всяких программистских "трюков") вполне безопасно, и к нему можно прибегать во всех случаях, когда это необходимо.

В качестве примера использования записей с вариантами разработаем программу, позволяющую пополнять электронную картотеку публикаций, а именно — после запроса о виде публикации производить ввод параметров указанной публикации (листинг 16.3). Пусть в картотеке хранятся сведения о публикациях трех видов: книги, журналы и рефераты.

Рассмотрим структуру каждого вида публикаций.

- Книга (`book`).
  - Автор (`author`).
  - Название (`title`).
  - Год (`year`).
  - Издательство (`publishinghouse`).
  - Страницы (`pages`).
- Статья (`article`).
  - Автор (`author`).
  - Название (`title`).
  - Год (`year`).
  - Журнал (`journal`).
  - Номер журнала (`numberofjournal`).
  - Начальная страница (`begpagej`).
  - Конечная страница (`endpagej`).
- Реферат (`paper`).
  - Автор (`author`).
  - Название (`title`).
  - Год (`year`).
  - Номер реферативного журнала (`numberofpaper`).
  - Начальная страница (`begpagerj`).
  - Конечная страница (`endpagerj`).

В предложенной структуре три параметра — Автор, Заголовок и Год — повторяются для каждого вида публикаций, поэтому их можно вынести в фиксированную часть записи, остальные же — в вариантную часть.

Листинг 16.3. Пример использования записей с вариантами

```
type
   kind = (book,article,paper);
   publication=record
   { фиксированная часть записи }
      autor,title : string;
      year : word;
      kindp : kind;
   { вариантная часть записи }
      case kind of
         book : (publishinghouse: string;
                          pages: word);
         article : (journal     : string;
         numberofjournal        : byte;
         begpagej, endpagej     : word);
         paper : (numberofpaper : byte;
         begpagerj, endpagerj   : word);
   end;
var pub: publication; j: word;
begin
   writeln('Выход - недопустимый вид публикации');
   repeat
       write('Вид публикации (0 .. 2)='); readln(j);
       if j in [0..2] then
         with pub do begin
            kindp:= kind(j);
            write('Автор :'); readln(autor);
            write('Название :');readln(title);
            write('Год :');readln(year);
            case kindp of
                book : begin
                   write('Издательство :'); readln(publishinghouse);
                   write('Объем, стр. :'); readln(pages);
                end;
                article : begin
                   write('Название журнала :'); readln(journal);
                   write('Номер :');readln(numberofjournal);
                   write('Страницы :'); readln(begpagej, endpagej);
                end;
```

```
                paper : begin
                   write('Номер реф. журнала :');readln(numberofpaper);
                   write('Страницы :'); readln(begpagerj, endpagerj);
                end;
            end;  { case .. of }
        end;      { with pub .. }
  until not (j in [0..2]);{повторять до недопустимого вида публикации }
end.
```

16.3. Контрольные вопросы и задания

Вопросы

1. В чем состоит основное преимущество записи, обуславливающее ее широкое применение?
2. Почему запись называют комбинированным типом данных?
3. Чем запись отличается от массива?
4. Как определяется тип записи? Что называется полем записи?
5. Какие требования предъявляются к идентификаторам поля в записи?
6. Чем определяется объем памяти, требуемый для размещения записи?
7. Что такое составное имя поля записи? Из каких частей оно состоит и как записывается?
8. Можно ли использовать в инструкциях ввода/вывода запись целиком? в операторах присваивания? Почему?
9. Как описываются массивы записей? Приведите пример.
10. Зачем при обращении к полю записи используется предложение `with`?
11. Как вы понимаете вложение записей? Приведите примеры.
12. С какой целью применяются записи с вариантами? Из каких частей состоит запись с вариантами?
13. Как записываются компоненты каждого варианта записи? Приведите примеры.
14. Какие правила следует соблюдать при использовании записей с вариантами?

Задания

1. Опишите запись с именем типа `Systema`, содержащую информацию о планетах солнечной системы: номер планеты по удалению от Солнца

(тип integer); название планеты (тип string); объем (real); диаметр (real); удаленность от Земли (real). Переменную, определяющую запись, назовите Planeta.

2. Опишите запись с именем типа Data, содержащую информацию о средней температуре в хранилище за 30 дней: номер месяца (тип integer); температура (тип real). Переменную, определяющую запись, назовите Zamer. Без помощи with присвойте начальное значение: месяц — июль и температура для первого дня — 9,5.

3. Составьте программу, которая описывает таблицу химических элементов, отображая текущую информацию: название, символическое обозначение, массу атома, заряд атомного ядра, перечень основных химических свойств. Программа должна выполнять вывод данных о химическом элементе по указанному символическому обозначению, находить элемент с самой большой массой, с самым маленьким зарядом ядра.

4. Составьте программу, которая описывает массив записей жильцов дома, отображая в нем следующую информацию о каждом: номер квартиры, фамилия, имя, возраст, для лиц старше 18 лет в зависимости от рода занятий (учеба, работа, пенсия) — запись места учебы, места работы и трудового стажа, для пенсионеров — год выхода на пенсию. Программа должна обеспечивать ввод данных, поиск квартиры с максимальным числом жильцов, поиск самого юного и самого пожилого жильца, поиск студентов, пенсионеров.

5. Опишите, используя структуру записи, вступительные экзамены, на которых абитуриенты сдавали три экзамена, а для поступления надо было набрать 12 баллов. Составьте программу, считывающую с клавиатуры результаты всех вступительных экзаменов и выводящую на экран следующую информацию: список абитуриентов, сдавших все три экзамена на 5; список абитуриентов, потерпевших неудачу на экзаменах; список абитуриентов, зачисленных в институт.

6. Опишите, используя структуру записи, школьный журнал. Предусмотрите в записи поля для хранения информации о фамилии учащегося, предмете, оценке. Составьте программу, считывающую с клавиатуры данные об успеваемости учащихся класса и выводящую на экран сведения об отличниках класса, о средней успеваемости учащихся класса.

7. Опишите, используя структуру записи, класс или студенческую группу (фамилия, инициалы, дата рождения, месяц рождения, год рождения). Составьте программу, считывающую с клавиатуры данные об учащихся класса и выводящую на экран данные о днях рождения учащихся по месяцам.

8. В справочной службе аэропорта хранится расписание вылета самолетов на следующие сутки. Для каждого рейса указаны его номер, тип самолета, пункт назначения, время вылета. Определите все номера рейсов, типы самолетов и время их вылета для заданного пункта назначения.

9. У администратора железнодорожных касс хранится информация о свободных местах в поездах по всем направлениям на ближайшую неделю. Данная информация представлена в следующем виде: дата выезда, конечный пункт назначения, время отправления, число свободных купейных мест, число свободных плацкартных мест. Оргкомитет конференции обращается к администратору с просьбой зарезервировать 50 купейных мест до Москвы на субботу. При этом время отправления поезда должно быть не позднее 10 часов вечера. Выведите на экран время отправления или сообщение о невозможности выполнить заказ в полном объеме.

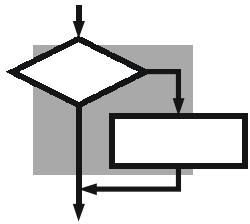
10. В радиоателье хранятся квитанции о сданной в ремонт радиоаппаратуре. Каждая квитанция содержит следующую информацию: наименование группы изделий (телевизор, радиоприемник и т. п.), марка изделия, дата приемки в ремонт, состояние готовности заказа (выполнен, не выполнен). Для директора ателье необходимо выдать информацию о состоянии заказов на текущие сутки по группам изделий.

11. В библиотеке имеется список книг. Каждая запись этого списка содержит фамилии авторов, название книги, год издания. Определите, имеются ли в данном списке книги, в названии которых встречается некоторое ключевое слово (например, `Pascal`). Если такие книги представлены, то выведите на экран фамилии авторов, название и год издания этих книг. Предусмотрите ввод ключевого слова с клавиатуры.

12. В магазине имеется список поступивших в продажу автомобилей. Каждая запись этого списка содержит марку автомобиля и его параметры: стоимость, расход бензина на 100 км, надежность (число лет безотказной работы), комфортность (отличная, хорошая, удовлетворительная). Покупатель в свою очередь имеет ряд требований по каждому из этих параметров. Эти требования задаются в виде некоторого интервала (например, стоимость — 3...6 тыс. долл.; расход бензина — 410 л на 100 км). Выведите на печать перечень автомобилей, удовлетворяющих требованиям покупателя. Предусмотрите ввод требований покупателя с клавиатуры.

13. В центре занятости населения ведется список вакантных рабочих мест на предприятиях города. Каждая запись такого списка содержит следующую информацию: наименование организации, местоположение организации (расстояние в км от центра города), наименование должности, требуемая квалификация (разряд или образование), требуемый стаж работы по специальности, размер заработной платы за месяц, наличие социальной страховки (да или нет), продолжительность ежегодного оплачиваемого отпуска. Безработный вводит информацию о своей квалификации и требованиях (например, размер заработной платы за месяц). Создайте программу, которая бы выводила для каждого клиента список рабочих мест в соответствии с его требованиями.

Глава 17



Файлы

Мы уже знаем, что эффективным способом хранения и обработки большого количества однотипных переменных в программе являются массивы *(см. гл. 13)*. Однако у них есть один недостаток: массивы не пригодны для долговременного хранения информации. В этом программисту могут помочь файлы (от английского слова *file* — подшивка, картотека).

Файл — это набор данных, хранящихся во внешней памяти компьютера (на жестком диске, дискете, компакт-диске и т. п.) под заданным именем.

Замечание

Любой источник или приемник информации в компьютере (клавиатура, экран, принтер и т. п.) может рассматриваться как файл *(см. разд. 17.1)*.

Особенности файлов рассмотрены ниже.

- Каждому файлу при создании *указывается имя*, по которому обрабатывающая его программа сможет *отличить* один файл от другого. Следовательно, *одна* программа может работать *одновременно с несколькими* файлами.
- Файл содержит элементы только *одного типа* (или тип его компонентов не оговаривается). *Исключение* — файловый тип недопустим. Например, можно создать файл записей или файл строк, но нельзя организовать "файл файлов".
- *Длина файла* — это число его элементов (компонентов). При создании файла длина файла *не задается заранее* и ограничивается только емкостью устройств внешней памяти. Файл, не содержащий элементов, называется *пустым*, его длина — ноль.

17.1. Некоторые сведения о файловой системе

В любой операционной системе информация на диске хранится в виде файлов, а система хранения информации называется *файловой системой*. В на-

стоящее время имеется несколько различных файловых систем, которые существенно отличаются друг от друга по внутреннему способу организации диска. Однако правила именования файлов, каталогов и дисков практически одинаковы во всех файловых системах Windows (FAT32, NTFS и др.). В свое время эти правила были разработаны для операционной системы DOS (файловая система FAT16).

Числа 16 и 32 уже знакомы читателю — это минимальные единицы обработки информации в DOS и Windows, выраженные в битах. Применительно к файловым системам эти значения соответствуют размеру элемента таблицы размещения файлов (File Allocation Table — FAT), которая является основой файловой системы FAT. 16-разрядные компиляторы, такие как Turbo Pascal, разрабатывались для операционной системы DOS, 32-разрядные — для семейств Window и UNIX. Не будем сейчас касаться файловых систем семейства UNIX, но кратко напомним основные требования к именам файлов, каталогов и дисков, принятые в DOS и Windows. Те, кто хорошо знаком с этими требованиями, могут пропустить следующий раздел.

17.1.1. Имя и расширение файла

Имя файла состоит из двух частей: собственно имени и его расширения.

Имя может содержать от 1 до 8 символов в DOS, в Windows — до 255 символов. *Запомним* — в Turbo Pascal мы именуем файлы, используя только короткие имена, избегая русских букв (они иногда обрабатываются некорректно) и ни в коем случае не используя пробелов. В таких системах, как Delphi, используем любые осмысленные имена, причем пробел разрешен наравне с остальными символами.

Расширение отделяется от имени точкой, за которой следует от 1 до 3 символов, составляющих его. Например, Example.pas или Example.com. Расширение характеризует *тип файла*, т. е. показывает, какая информация содержится в этом файле (программа, текст, рисунок, мелодия и т. д.). В принципе, можно дать файлу любое расширение или вообще не давать расширения, но рекомендуется пользоваться набором *стандартных расширений*, зарегистрированных в операционной системе. Это позволяет не только программисту, но и операционной системе правильно обрабатывать файлы, зная, какой программой они были созданы.

17.1.2. Каталоги

Для регистрации имен файлов в файловой системе компьютера используются каталоги (directory). В операционной системе Windows каталоги имеют еще одно название — папки (folders). *Каталог* — это поименованная область на диске. Имя каталога подчиняется тем же правилам, что и имя файла. Рас-

ширение каталогу обычно не назначается, хотя это и не запрещено. На диске может быть множество каталогов. Каждый каталог, в свою очередь, может содержать файлы и *вложенные каталоги* более низкого уровня *(подкаталоги)*, а сам он выступает по отношению к ним как *родительский* каталог *(надкаталог)*.

Каждый диск обязательно имеет свой *главный (корневой) каталог*, в котором зарегистрированы файлы пользователя и подкаталоги первого уровня, а в них — другие файлы пользователя и подкаталоги второго уровня, и т. д. Корневой каталог нельзя удалить. Он не имеет имени и обозначается знаком "\". Например, C:\ — это корневой каталог диска C:. Все прочие каталоги диска вложены в корневой. Так формируется *древовидная*, или *иерархическая* структура каталогов.

Каталог, с которым в настоящий момент времени работает пользователь, называется *текущим*.

До начала работы жесткого диска компьютера (или обычной дискеты) над ним выполняется операция *форматирования*. При этом поверхность диска разбивается на дорожки и секторы, а также создается служебная область и пустой *корневой* каталог. Другие каталоги и файлы создаются в процессе работы и располагаются на диске в произвольном порядке.

Внутренняя структура каталогов в различных файловых системах различается, но основные сведения, которые в них хранятся, неизменны. Каждый каталог содержит сведения обо всех своих файлах и вложенных каталогах, а также информацию о вышестоящем каталоге (вместо имени используется обозначение ..) и себе самом (вместо имени используется одна точка). Кроме имени и расширения файла или каталога для них хранятся такие *характеристики*:

- размер в байтах (только для файлов);
- дата и время создания файла или каталога (для файлов хранится и дата последнего изменения);
- специальные признаки, или *атрибуты* файла: "только для чтения", "скрытый", "системный" и "архивный". Каталог помечается специальным признаком — *directory*, поэтому операционная система всегда отличит его от файла;
- физический адрес файла или каталога на диске.

Информация о каждом конкретном файле или каталоге хранится только в одном каталоге.

Замечание

Пространство на диске выделяется любому файлу порциями — *кластерами*. Когда говорят, что файл *находится в каталоге*, нужно иметь в виду, что сам

файл содержится *на диске*, начиная с номера начального кластера. В каталоге хранятся только сведения о файле, в том числе и его адрес (номер начального кластера).

Для того чтобы программа смогла найти нужный файл, необходимо знать путь или *маршрут*, "ведущий" к файлу. *Путь* — это перечень имен подкаталогов, отделенных друг от друга обратной косой чертой, которые нужно последовательно пройти для того, чтобы отыскать нужный файл. Обычно путь начинается с имени диска, которое заканчивается символом двоеточия. Последним в пути указывается имя файла:

Диск:\ИмяПодкаталога1\...\ИмяПодкаталогаN\ИмяФайла

Путь к файлу называют также *полным именем* файла. Максимально допустимая длина пути — 79 символов.

Например, c:\turbo\pas\example.pas.

В этом примере файл программы example.pas содержится в каталоге pas, который зарегистрирован в каталоге turbo. Каталог turbo находится в корневом каталоге диска C:.

Для именования дисков используются латинские буквы от A до Z. Жесткий диск ("винчестер") — это C:. Если для удобства пользователя большой "винчестер" разбивают на части *(логические диски)*, то для их именования используются следующие в алфавитном порядке латинские буквы (D:, E:, и т. д.). Дисковод для гибких дисков (дискет) именуется A:. Если существует второй дисковод, то его имя — B:, в противном случае имя B: не используется. Если буква и следующее за ней двоеточие отсутствуют, то подразумевается используемый в настоящее время (назначенный по умолчанию или текущий) диск. Если путь к файлу начинается с обратной косой черты, то поиск файла начинается с корневого каталога диска, в противном случае — с текущего.

Например, \turbo\pas\table.txt или table.txt.

Иногда при записи пути используют .. (две точки подряд) для обозначения вышестоящего каталога, например, ..\table.txt обозначает, что файл table.txt находится в каталоге, вышестоящем по отношению к текущему. Вспомним, что в любом каталоге (кроме корневого) хранится информация о вышестоящем каталоге, имя которого обозначается двумя точками, поэтому переход в вышестоящий каталог выполняется быстро.

В заключение упомянем еще один непривычный, но допустимый способ записи пути, который начинается с одной точки, например, так: .\table.txt. Такая запись пути эквивалентна table.txt и обозначает, что файл находится в текущем каталоге. Разумеется, подобный способ записи пути может потребоваться только в исключительных случаях, когда требуется явно обозначить

текущий каталог. Тем не менее, возьмите на заметку обозначения текущего или вышестоящего каталогов в виде одной или двух точек.

17.1.3. Устройства

В составе компьютера принято выделять отдельные компоненты — *устройства*. Некоторые из них, предназначенные для ввода, вывода и хранения данных, называют *внешними*, или *периферийными*.

Многие языки высокого уровня рассматривают внешние устройства как файлы, с которыми можно работать точно так же, как с обычными файлами. Такое представление значительно упрощает жизнь программисту. Обращение к устройствам выполняется с помощью специально зарезервированных имен, которые запрещено использовать для именования обычных файлов — так называемых *имен логических устройств* компьютера: Это такие, например, имена, как:

- CON — наименование консоли. При помощи *консоли* выводимая информация пересылается на экран дисплея, а вводимая информация воспринимается с клавиатуры. При вводе с клавиатуры символы считываются из *буфера строки*, который передает их программе только после нажатия на клавишу <Enter>. При нажатии клавиш <Ctrl>+<Z> генерируется символ #26 (SUB) конца файла;
- PRN — наименование *принтера*. Если к компьютеру подключено несколько принтеров, доступ к ним осуществляется по логическим именам LPT1, LPT2 и LPT3. Обычно имена PRN и LPT1 — синонимы;

> **Замечание**
>
> Вообще говоря, LPT1, LPT2, LPT3 — это устройства, которые подключаются к параллельным портам 1, 2 и 3. *Портом* называют многоразрядный вход или выход. Чаще всего к порту LPT1 подключен именно принтер.

- COM1, COM2 и COM3 — устройства, подключенные к *последовательным портам*. Используются для связи с другими компьютерами. Возможны как прием, так и передача данных, однако в каждый момент времени может использоваться либо прием, либо передача. Обычно вместо COM1 можно использовать синоним AUX;
- NUL — *нулевое*, или *"пустое"* устройство. Чаще всего используется при отладке. Устройство работает следующим образом — при чтении из него программе сообщается о конце файла, а при выводе на него информация на самом деле никуда не выводится, но программе, которая выполняла вывод, сообщается, что он произошел успешно.

17.2. Описание файлового типа

17.2.1. Виды файлов. Файловая переменная

В Pascal имеются три вида файлов:

- текстовый файл (определяется типом `text`);
- типизованный файл (задается предложением `file of Тип`);
- нетипизованный файл (определяется типом `file`).

Для работы с файлами в программе необходимо определить *файловую переменную (файловый тип)* в разделе описаний программы.

Например:

```
type filetype=text;     { файловый тип }
var ftmp,f: filetype;   { файловые переменные }
```

или

```
var f1,lst: text; f2: file;
```

Файловые переменные, описанные в программе, называют *логическими файлами*. Все основные процедуры и функции, обеспечивающие ввод/вывод данных, работают только с логическими файлами *(см. разд. 17.3)*. Поэтому перед тем как использовать физический файл, его необходимо связать с логическим файлом (файловой переменной).

17.2.2. Указатель. Доступ к файлам

В любой момент времени программе доступен только один элемент файла, на который ссылается *указатель текущей позиции файла (указатель обработки)*. Он определяет то место в файле, откуда (куда) происходит чтение (запись) данных. При открытии или создании файла указатель помещается в его начало. Чтение или запись данных из файла вызывает автоматическое перемещение указателя. Указатель файла ведет себя подобно курсору, который, смещаясь при редактировании текста, все время показывает текущую позицию. При чтении данных из файла указатель рано или поздно достигнет его конца (если, конечно, цикл чтения данных не завершается с помощью какого-либо дополнительного условия: допустим, можно читать числовые данные до первого появившегося нуля или текстовые данные до первой точки).

По способу доступа к элементам различают файлы последовательного и прямого доступа *(см. разд. 17.4)*.

Файлом последовательного доступа называется файл, к элементам которого доступ выполняется в той же последовательности, в какой они записывались. Для поиска нужного элемента в этом случае необходимо перемещать

указатель обработки до тех пор, пока он не будет помещен на искомый элемент. Для таких файлов запрещено совмещение чтения и записи данных.

Файл прямого доступа — это файл, доступ к элементам которого осуществляется по их адресу (номеру). Таким образом, при поиске нужного элемента достаточно указать номер его позиции, что существенно ускоряет процесс. Для файла прямого доступа допустимо совмещение записи и считывания данных.

17.3. Общая схема работы с файлами

17.3.1. Последовательность действий

Вне зависимости от файлового типа любая программа работы с файлом должна выполнить следующие действия.

1. Определить переменные файлового типа (логические файлы).
2. Каждому из используемых физических файлов поставить в соответствие переменную файлового типа.
3. Открыть файл — т. е. сделать существующий файл доступным для ввода и/или вывода. В случае отсутствия файл должен быть создан.

Замечание

Для того чтобы ускорить обмен информацией с внешними устройствами, в файловой системе используется механизм буферизации. Все операции обмена данными между оперативной памятью и диском выполняются через файловый буфер. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока он весь не заполнится. Только после этого (или при использовании специальной команды сброса) происходит передача данных на внешнее устройство. Аналогично при чтении из файла данные вначале считываются в буфер.

4. Обработать файл. Все предыдущие этапы носили *подготовительный* характер. *Принцип обработки* файлов любых типов состоит в следующем: данные из файла сначала считываются в оперативную память компьютера, для чего в программе назначаются переменные *подходящих* типов. Вся дальнейшая обработка ведется над этими переменными. В случае необходимости результаты записываются в новый файл или дописываются в уже существующий. Чтение (ввод) данных или их запись (вывод) выполняются при помощи стандартных инструкций ввода/вывода. Особенности применения этих инструкций для файлов различных типов будут рассмотрены ниже.
5. Закрыть файловую переменную, т. е. сохранить данные на диске после окончания работы с файлом.

17.3.2. Общие процедуры и функции

Рассмотрим стандартные процедуры и функции языка Pascal, которые позволяют выполнить описанную последовательность действий, а также ряд других действий над файлами. Они применимы для файлов любых типов.

- Процедура `assign(ФайловаяПеременная, ИмяФайла)` — предшествует другим процедурам, т. к. ставит в соответствие физическому файлу на внешнем устройстве логический файл — файловую переменную (связывает их). `ИмяФайла` должно представлять собой выражение строкового типа. Дальнейшие операции с переменной `ФайловаяПеременная` будут выполняться над *физическим* файлом `ИмяФайла`. Процедуру `assign` недопустимо использовать для уже открытого файла. Прежде чем использовать файловую переменную повторно, необходимо закрыть файл с помощью процедуры `close`. После вызова `assign` связь файловой переменной с физическим файлом существует до тех пор, пока не будет выполнен другой `assign` для данной файловой переменной. Следовательно, файл можно повторно открыть без дополнительного использования процедуры `assign` даже после закрытия `close`.

- Процедура `reset(ФайловаяПеременная)` — открывает *существующий* файл на *чтение* (открывает *входной файл*) и ставит указатель на начало *первого* элемента файла. Если при чтении файла возникнет необходимость вернуть указатель в его начало, достаточно будет просто применить процедуру `reset` к этому файлу еще раз.

- Функция `ioresult` — возвращает статус последней выполненной операции ввода/вывода (успешно она прошла или нет). Пример использования данной функции для обработки ошибок ввода/вывода уже рассматривался *(см. разд. 9.1)*. При работе с файлами рекомендуется использовать ее при выполнении любого действия, которое может привести к ошибке. Например, при попытке открыть файл может оказаться, что файла с таким именем нет в текущем или заданном каталоге. Стандартный способ обработки такой ошибки — аварийная остановка программы с сообщением **File not found** (Файл не найден). Обработка функции `ioresult` позволит выполнить проверку существования файла на диске более гибко (см. листинг 17.2).

- Процедура `rewrite(ФайловаяПеременная)` — создает и открывает *новый (выходной)* файл для последующей *записи* данных. После ее успешного выполнения файл готов к записи в него *первого* элемента.

  Обратите внимание — использование `rewrite` требует особой аккуратности. Если физический файл с указанным именем уже существует, то он *удаляется*, и на его месте создается новый *пустой* файл *с тем же именем*. Для предотвращения потери информации на практике необходимо созда-

вать резервные копии файлов, над которыми могут производиться опасные действия (см. листинг 17.2).

- Процедура `close(ФайловаяПеременная)`. Используя процедуру `close`, программист должен закрыть файл после того, как завершена его обработка. В противном случае может произойти *потеря данных*. При закрытии физический файл обновляется и его автоматически завершает *символ конца файла*. Впоследствии `ФайловаяПеременная` может быть связана с другим (или вновь с тем же самым) физическим файлом.
- Процедура `rename(ФайловаяПеременная, ИмяФайла)` — используется для того, чтобы *переименовать* неоткрытый внешний файл любого типа. Новое имя задается строкой `ИмяФайла`.
- Процедура `erase(ФайловаяПеременная)` — *удаляет* неоткрытый внешний файл любого типа, задаваемый параметром `ФайловаяПеременная`.

  Обратите внимание — процедуры `rename` и `erase` нельзя использовать для открытых файлов. Если файл не существует, возникает ошибка выполнения программы.
- Логическая функция `eof(ФайловаяПеременная)` — выполняет проверку, не достигнут ли конец файла (End Of File) при чтении из него данных. Функция возвращает `true`, если при чтении обнаружен конец файла. Она позволяет задать условие выполнения цикла для чтения данных из файла. Если параметр `ФайловаяПеременная` отсутствует, подразумевается консоль (клавиатура). Иногда этот параметр забывают указать по ошибке. При этом компилятор считает функцию `eof` без параметра синтаксически правильной, но при выполнении программы анализируется не конец файла на диске, а символ конца файла, введенный с клавиатуры (такой символ ввести можно, например, при помощи комбинации <Ctrl>+<z>). Когда программа ожидает ввода с клавиатуры, у начинающего программиста создается ощущение, что она зависла по непонятным причинам. Не забывайте указывать параметр функции `eof`!

Замечание

В Delphi вместо процедур `assign` и `close` используются аналогичные процедуры `assignfile` и `closefile` с такими же параметрами. Для проверки существования файла на диске имеется стандартная функция `fileexists (ИмяФайла)`.

В качестве примера создадим программно новый текстовый файл с именем file01.txt, содержащий одну строку текста, которая записывается в него в виде текстовой константы. Для чтения используем строковую переменную `s`.

```
var f: text; s: string;
begin
   assign(f,'file01.txt'); rewrite(f);
```

```
   writeln(f,'Этот простой текстовый файл содержит строку текста');
   close(f);
   assign(f,'file01.txt'); reset(f);
   readln(f, s);
   close(f);
   writeln('Проверка ввода/вывода в файл:'); writeln(s);
end.
```

17.3.3. Процедуры для работы с каталогами

В Pascal существует несколько полезных процедур для работы с каталогами.

- `getdir` — определяет текущий каталог на заданном диске. Например: `getdir(0,s);` — в переменную `s` помещается текущий каталог на текущем диске.
- `mkdir` — создает каталог.

  Например:

```
{$i-} {отключили автоматическую проверку ошибок ввода/вывода}
   mkdir('prog_dir');{создаем каталог prog_dir в текущем каталоге}
   If ioresult <> 0 then {проверяем результат операции}
    writeln('Ошибка создания каталога');
{$i+} {не стоит надолго отключать автоматическую проверку — включили}
```

- `rmdir` — удаляет заданный каталог (пустой и не текущий).

  Например:

  `rmdir('prog_dir');` — удаляет только что созданный каталог. Это можно сделать, поскольку он еще пустой и не текущий.
- `chdir` — меняет текущий каталог.

  Например:

  `chdir('prog_dir');` — делаем каталог prog_dir текущим. Теперь процедура `rmdir('prog_dir');` не сможет его удалить.

Обратите внимание — все операции с каталогами опасны (кроме `getdir`), поэтому везде используйте функцию `ioresult`.

17.3.4. Процедуры и функции модуля *dos*

Для полноты картины приведем процедуры и функции модуля `dos` Turbo Pascal *(см. разд. 12.6)*, служащие для работы с файловой системой (табл. 17.1). Большинство из них входит в стандартный модуль Delphi `sysutils`, но у некоторых там могут быть изменены названия.

Таблица 17.1. *Процедуры и функции модуля dos для работы с файлами*

Процедура, функция	Описание
	Процедуры
`Fsplit`	Разделяет имя файла на путь, имя и расширение
`Getfattr`	Возвращает атрибуты файла
`Getftime`	Возвращает дату и время последней записи файла
`Setftime`	Назначает новую дату и время последней записи файла
`Setfattr`	Устанавливает атрибуты файла
	Функции
`Diskfree`	Возвращает число свободных байт на заданном диске
`Disksize`	Возвращает общий объем дисковой памяти на диске
`Fexpand`	Расширяет имя файла до полностью определенного
`Findfirst`	Ищет в заданном каталоге первый элемент, совпадающий с заданным именем файла и его атрибутами (`findnext` — следующий)
`Fsearch`	Ищет файл в списке каталогов

17.4. Текстовые файлы

Компонентами текстовых файлов являются символы, организованные в виде *последовательности строк произвольной длины*, что не позволяет считать их просто набором символов `char`.

Каждая строка в текстовом файле оканчивается *составным символом "конца строки"*, который является объединением двух символов: символа #13 (CR) — возврат каретки (carridge return) и символа #10 (LF) — перевод строки (line feed). Для составного символа вводят обозначение `eoln` (End Of LiNe). Символ `eoln` нельзя присвоить переменной типа `char`. Однако его можно распознать с помощью функции `eoln`.

Текстовый файл — это последовательность символов, сгруппированных в строки, заканчивающиеся специальным символом `eoln`.

В конце любого файла, в том числе и текстового, ставится символ #26 (SUB) — конец файла eof (End Of File).

Замечание

Текстовый файл можно подготовить и прочитать при помощи обычного текстового редактора, который не использует операции форматирования текста. На-

до только помнить, что в DOS и Windows используются разные кодировки для русских букв, поэтому при работе в Turbo Pascal нужно пользоваться редактором для DOS (встроенный редактор этой системы вполне годится). При работе с Delphi или другой 32-разрядной системой файл может быть подготовлен при помощи приложения Блокнот. Программа MS Word позволяет сохранить текстовые файлы с различной кодировкой (при этом теряется форматирование) с помощью команды **Файл | Сохранить как**. Далее следует выбрать **текст DOS (*.txt)** или **только текст (*.txt)**.

Объявление текстового файла в программе осуществляется заданием файловой переменной типа `text` (текст) в виде:

```
type ИмяТипа = text;
var ФайловаяПеременная : ИмяТипа;
```

или

```
var ФайловаяПеременная : text;
```

С файлами текстового типа можно работать только путем *последовательного продвижения* по файлу, прочитывая или записывая одну строку за другой, *начиная с первой*. Текстовые файлы являются файлами *последовательного доступа*, и этим они отличаются от типизированных файлов, в которых указатель можно поставить в любую позицию файла. Типизированные файлы — это файлы *произвольного доступа*.

17.4.1. Процедуры и функции для текстовых файлов

При обработке программой текстовых файлов могут использоваться некоторые дополнительные процедуры и функции, а ряд известных нам общих процедур и функций *(см. разд. 17.3)* имеют свои особенности.

Открытие текстового файла можно произвести тремя способами. Два из них являются стандартными: `reset` и `rewrite`. Процедура `reset` открывает текстовые файлы только для чтения. Процедура `rewrite` перезаписывает их. Новая процедура `append` позволяет *дополнять* файл новыми данными.

- Процедура `append(ФайловаяПеременная)`. Она открывает существующий файл для *дозаписи*. Указатель ставится не в начало, а в *конец файла*, куда и будут дописываться новые компоненты. `ФайловаяПеременная` текстового типа предварительно должна быть связана с внешним файлом с помощью процедуры `assign`. Процедура `append` применима *только* к текстовым файлам. Если файл ранее уже был открыт с помощью `reset` или `rewrite`, использование `append` приведет к закрытию этого файла и к повторному открытию, но уже для добавления элементов.

  После обращения к `append` текстовый файл доступен только *по записи*, и функция `eof` всегда принимает значение `true`. После вызова `reset` или

rewrite функция eof принимает значение true, только если файл пуст, иначе — false.

- Процедуры чтения:

```
read(ФайловаяПеременная, x1, x2,...,xN);
readln(ФайловаяПеременная);
readln(ФайловаяПеременная, x1, x2,...,xN);
```

где x1, x2,...,xN — список *ввода*, содержащий имена переменных *разных* типов (integer, real, char, string), значения которых процедура read считывает из текстового файла, *начиная чтение с элемента, на который установлен текущий указатель*. Файловая переменная имеет тип text.

Процедура readln выполняет те же действия, что и read, и дополнительно — переход к новой строке.

Вызов readln(ФайловаяПеременная) без параметров приводит к перемещению текущей позиции файла на начало следующей строки (если она имеется, в противном случае происходит переход к концу файла). Так можно пропускать строки.

При выборе между read и readln для выполнения чтения из текстового файла необходимо помнить, что после считывания последней переменной списка ввода процедура readln пропускает оставшуюся часть строки до eoln и обязательно переходит к следующей строке текстового файла, даже если из текущей строки прочитаны еще не все символы. Следующее обращение к очередной процедуре read или readln начнется с первого символа новой строки. В отличие от readln, процедура read не делает после считывания пропуск до следующей строки. Поэтому она непригодна для чтения последовательности строк, поскольку не дает возможности продвинуться дальше первой строки. Таким образом, для ввода следующих друг за другом строк текста необходимо использовать readln.

- Процедуры записи:

```
write(ФайловаяПеременная, y1, y2,...,yN);
writeln(ФайловаяПеременная);
writeln(ФайловаяПеременная, y1, y2,...,yN);
```

где y1, y2,...,yN — *список вывода*, содержащий выводимые выражения *разных* типов (integer, real, char, string, boolean), значения которых должны быть записаны в файл, *начиная с позиции текущего указателя*. Возможно использование формата вывода *(см. разд. 7.2.1)*. Файл должен быть открыт для вывода.

Процедура writeln выполняет те же действия, что и write, и дополнительно вставляет признак конца строки eoln.

Обратите внимание — особенность использования *текстовых* файлов состоит в том, что процедуры чтения `read (readln)` и записи `write (writeln)` разрешают работать со значениями *различных* типов. Последовательность символов, прочитанная из файла, *автоматически преобразуется* к значению типа той переменной, которая используется в списке ввода `read (readln)`. При записи в файл происходит *автоматическое преобразование* значений переменных списка вывода `write (writeln)` к символьному виду.

Например, при выполнении фрагмента программы

```
var f1: text; a,b,c: real;
...
readln(f1,a,b,c);
```

процедура `readln` выполняет чтение из очередной строки файла `f1` последовательности цифр, затем интерпретируемой как три вещественных числа, значения которых будут присвоены переменным `a`, `b`, `c`. Если вместо ожидаемой последовательности чисел в файле содержится любая другая последовательность символов, возникает ошибка ввода/вывода. Считывание значений символьного и строкового типов из текстового файла — более безопасная операция, чем чтение данных числового типа. Если длина строки при чтении превышает длину, максимально допустимую для строковой переменной, то она усекается.

- Функции проверки конца строки и файла:
  - кроме использования функции `eof`, принимающей значение `true`, если файл исчерпан *(см. разд. 17.3)*, при работе с текстовыми файлами необходимо уметь проверять также и конец строки. Для контроля используется функция `eoln(ФайловаяПеременная)`, принимающая значение `true`, если указатель текущей позиции находится на маркере конца строки (CR/LF), иначе — `false`. Если `eof` — `true`, то и `eoln` — `true`;
  - *функция* `seekeoln(ФайловаяПеременная)` аналогична функции `eoln`, но, в отличие от нее, пропускает пробелы и позиции табуляции перед проверкой на конец строки;
  - *функция* `seekeof(ФайловаяПеременная)` аналогична `eof`, но пропускает пробелы, позиции табуляции и маркеры конца строки перед проверкой на конец файла.

Обратите внимание — функции `seekeof` и `seekeoln` удобно использовать при чтении *числовых данных* из *текстового файла*, когда необходимо пропустить разделяющие числа пробелы или знаки табуляции (см. листинг 17.3).

17.4.2. Стандартные текстовые файловые переменные *input* и *output*

Любой программе доступны без описания *стандартные текстовые файловые переменные* input и output. Input (ввод) — это доступный только по *чтению* файл, используемый для ввода с клавиатуры. Output (вывод) — доступный только по *записи* файл, используемый для вывода на дисплей. Чтобы убедиться, что такие файловые переменные действительно существуют, вы можете выполнить такую программу:

```
begin writeln(output,'Привет'); readln(input); end.
```

Перед началом выполнения программы файлы input и output автоматически открываются, как если бы были выполнены следующие операторы:

```
assign(input,''); reset(input);
```

и

```
assign(output,''); rewrite(output);
```

После выполнения программы эти файлы автоматически закрываются.

Файлы input и output и все файлы, которым присвоено пустое имя, ссылаются на устройство "консоль". Конечно, имя con можно было бы записать и явно, но пустое имя короче.

Этими стандартными файлами можно с успехом пользоваться при работе с текстовыми файлами. Дело в том, что их можно связать и с любыми двумя текстовыми файлами на диске, и тогда процедуры read (readln) и write (writeln) будут работать не с консолью, как обычно, а с этими файлами на диске, которые при этом будут автоматически открываться и закрываться.

Например:

```
Var a,b: integer;
begin
   assign(input, 'input.txt'); {связали input с файлом input.txt }
   readln(a,b); {прочитали две переменные из файла, а не с клавиатуры}
   assign(output, 'output.txt'); { связали output с файлом output.txt }
   writeln(b,a); {записали переменные в файл, а не вывели на экран}
end.
```

Если файл input.txt действительно содержит в первой строке два целых числа, то после выполнения программы в текущем каталоге появится файл output.txt, содержащий эти числа в обратном порядке. И никаких файловых переменных описывать не потребовалось. Неплохо?

Этой возможностью обычно пользуются на олимпиадах по программированию, поскольку при автоматической проверке задач исходные данные передаются программе из файла, а программа обязана послать результаты в другой

файл, который и будет проверяться. При отладке программы, когда требуется работать с консолью, используют `assign(input,''); assign(output,'');`

Возможен и другой вариант — связать произвольную файловую переменную с консолью. Например:

```
var f : text; str: string;
begin
   assign(f, ''); reset(f); readln(str); close(f);
   assign(f, ''); rewrite(f); writeln(f, 'Вы ввели: ',str); close(f);
end.
```

Программа вводит данные с клавиатуры и выводит их на экран, хотя использует файловую переменную.

17.4.3. Задачи на обработку текстовых файлов

Одной из основных проблем при решении подобных задач является выбор типа буферной переменной, необходимой для обработки данных, считываемых из файла.

Для файлов с произвольной текстовой информацией можно рекомендовать следующие варианты выбора *типа буферной переменной*:

- массив или связанный список строк *(см. разд. 18.5)*, в который будет считан *сразу весь файл*;
- одна переменная строкового типа, в которую будут считываться *по очереди строки файла*;
- одна переменная символьного типа, в которую *по очереди будут считываться символы*;
- если файл содержит только числовые данные, разделенные пробелами и символами перевода строки, то разумнее использовать переменные числового типа или массивы чисел.

Варианты с массивами приводят к большим затратам оперативной памяти и применяются в тех редких случаях, когда максимальный размер файла известен заранее, а обработка требует присутствия в памяти сразу всех данных (например, при сортировке). Листинг 17.1 содержит программу, которая находит в существующем файле все строки, содержащие набор символов, вводимых с клавиатуры. Результат поиска выводится на экран и одновременно записывается в файл с именем analysis.txt.

Листинг 17.1. Поиск в текстовом файле

```
const sum: word=0;   { счетчик: сколько раз нашли }
var f1,f2: text; name,str,search: string[80];
```

```
begin
   write('Введите имя файла: '); readln(name);
   write('Введите строку поиска: '); readln(search);
   assign(f1,name);assign(f2,'analysis.txt');
   reset(f1); rewrite(f2); { открываем f1 и создаем f2 }
   writeln('Протокол поиска:'); writeln(f2,'Протокол поиска:');
   while not eof(f1) do { пока не конец файла f1, выполняем цикл }
      begin
         readln(f1,str);          { вводим строку str из файла f1 }
         if pos(search,str)>0 then
            begin                 { ищем search в str }
               inc(sum); writeln('Найдено (раз):',sum);
               writeln(str);      { вывод на экран }
               writeln(f2,str);   { вывод в файл }
            end;
      end;
   close(f1); close(f2); readln; { закрываем файлы }
end.
```

Недостаток этой программы состоит в том, что проверка существования физического файла в ней не выполняется. В случае отсутствия файла программа будет аварийно завершена. В листинге 17.2 представлена функция `fileexists`, служащая для проверки существования файла на диске (напоминаем, что в Delphi есть такая стандартная функция).

Сама программа добавляет текст, введенный с клавиатуры, в *начало* заданного файла. Открытие файла для дозаписи процедурой `append` используется только в том случае, если текст добавляется в конец файла, поэтому нам пришлось создать временный файл (это типовой прием), в который сначала помещается добавляемый текст, а затем текст из заданного файла.

Листинг 17.2. Добавление данных в начало текстового файла

```
var ftmp,f: text;   { файловые переменные }
    txtbuf: string;     { текстовый буфер }
    name: string;       { имя файла }
    ok: boolean; ch: char;
function fileexists(filename : string): boolean;
{ если файл существует в текущем каталоге, функция возвращает true, }
var f: file;
```

```
begin
   {$i-} assign(f, filename); reset(f); {$i+}
   fileexists:=(ioresult=0)and(filename<>'');close(f);
end;
begin
   { запись текста во временный файл на диск }
   assign(ftmp,'tmpfile.txt'); rewrite(ftmp);
   writeln('Вводите текст для записи. Окончание - <Enter>:');
   repeat
      write('->'); readln(txtbuf); writeln(ftmp, txtbuf);
   until txtbuf=''; { окончание текста — двойное нажатие <Enter> }
   close(ftmp); writeln('Ввод окончен');
   repeat    { проверка существования файла }
      write('Введите имя файла для добавления: '); readln(name);
      ok:=fileexists(name);
      if not ok then writeln('Файл ',name,' не найден');
   until ok;
   writeln('Файл ',name,' найден в текущем каталоге');
   assign(f,name); reset(f); { открыли файл f }
   append(ftmp); { текст из файла f добавляем в конец временного файла }
   while not eof(f) do
      begin
         { пустые строки читаются, но не записываются }
         readln(f, txtbuf);
         if txtbuf<>'' then writeln(ftmp,txtbuf);
      end;
   close(f); close(ftmp);
   writeln('Введенный текст добавлен в начало файла ',name);
   writeln('Сделайте выбор:');
   writeln('Удалить старый файл, без резервной копии - E/e (erase)');
   writeln('Удалить старый файл, создав bak-файл - B/b (backup)');
   writeln('Выйти - H/h (halt)');
   readln(ch);
   case ch of
        'E','e':
            begin
               { удалили старый файл и переименовали временный }
               erase(f); rename(ftmp,name);
            end;
```

```
        'B','b':
            begin
               txtbuf:=name; delete(txtbuf,length(txtbuf)-3,4);
               rename(f,txtbuf+'.bak'); {переименовали старый файл в bak}
               rename(ftmp,name);       {переименовали временный файл}
            end;
        'H','h':halt;
   end;
end.
```

Комментарий

Заметим, что можно было бы не закрывать файл tmpfile.txt, и тогда не потребовалось бы открывать его при помощи append. Но нам очень хотелось привести пример использования такого способа открытия файла.

Текстовые файлы удобны для *наглядного* хранения массива чисел.

Рассмотрим такую задачу. Известно, что многие насекомые ориентируются по солнцу. Предположим, что у нас есть квадратная поверхность, поделенная на клетки. В левом верхнем углу сидит "паучок", а солнце светит из противоположного угла. "Паучок" идет к солнцу. Из одной клетки в другую он может попасть лишь через отверстия в стенках клетки, т. е. не по диагонали. Меняя направления, "паучок" может пройти в противоположный угол квадрата различными путями. Записав число путей, ведущих в указанную клетку, мы получим таблицу, называемую *арифметическим квадратом*. Каждое записанное в ней число равно сумме двух чисел: находящегося непосредственно сверху и ближайшего слева. Программа (листинг 17.3), используя процедуры, *формирует арифметический квадрат* в виде массива целых чисел, *записывает* его в текстовый файл, *читает* массив из файла и *выводит* на экран.

Листинг 17.3. Запись арифметического квадрата в текстовый файл и чтение из файла

```
const max=10;
type massiv=array[1..max,1..max] of integer;
var m:integer; matrix1,matrix2:massiv;
procedure square(m:integer; var mas:massiv);
{ формирование арифметического квадрата - массива mas }
var i,j:integer;
begin
   for i:=1 to m do
```

```
      begin
         mas[i,1]:=1; mas[1,i]:=1;
      end;
   for i:=2 to m do
      for j:=2 to m do
         mas[i,j]:=mas[i-1,j]+mas[i,j-1];
end;
procedure inmatrfiletxt(m,n:integer; mas:massiv; priz:char; name:string);
{ вывод двумерного массива mas в текстовый файл name }
{ если priz='r' файл перезаписывается, иначе - дополняется }
var i,j:integer; file_text:text;
begin
   assign(file_text,name);
   if priz='r' then rewrite(file_text) else append(file_text);
   writeln(file_text,m:2,n:2); { m,n - кол-во строк и столбцов }
   for i:=1 to m do
      begin
         for j:=1 to n do write(file_text,mas[i,j]:6);
         writeln(file_text);
      end;
   close(file_text)
end;
procedure outmatrfiletxt(var m,n:integer; var mas:massiv; name:string);
{ ввод двумерного массива mas из текстового файла name }
var i,j:integer; file_text:text;
begin
   i:=1;j:=1;
   assign(file_text,name); reset(file_text);
   while not seekeoln(file_text) do read(file_text,m,n);
   readln(file_text);
   while not seekeof(file_text) do
      begin
         while not seekeoln(file_text) do
            begin
               read(file_text,mas[i,j]); j:=j+1;
            end;
         readln(file_text);
         i:=i+1; j:=1
      end;
```

```
   close(file_text)
end;
procedure outmas(m,n:integer; mas:massiv);
{ outmas - вывод двумерного массива mas на экран }
var i,j:integer;
begin
   writeln(m:2,n:2);
   for i:=1 to m do
      begin
         for j:=1 to n do write(mas[i,j]:7); writeln
      end;
end;
begin
   write('Введите длину стороны арифметического квадрата: ');readln(m);
   { формирование арифметического квадрата - массив matrix1 }
   square(m,matrix1);
   { запись арифметического квадрата в текстовый файл square.txt }
   inmatrfiletxt(m,m,matrix1,'r','square.txt');
   writeln('Арифметический квадрат ',m,' на ',m,':');
   { чтение арифметического квадрата из текстового файла square.txt }
   outmatrfiletxt(m,m,matrix2,'square.txt');
   { вывод арифметического квадрата на экран - массив matrix2 }
   outmas(m,m,matrix2);
end.
```

17.5. Типизированные файлы

Для хранения *однородной* информации, например, только числовых данных определенного типа, использование *текстового* файла является *невыгодным*. Компьютер непродуктивно тратит ресурсы на постоянное преобразование данных из символьного вида в двоичный и обратно (причем эти преобразования могут привести к ошибкам округления); часто требуется отслеживать признаки конца строки; доступ выполняется последовательно, а значит, медленно; объем файла велик.

Типизированный файл состоит из *последовательности элементов одного типа и длины*. Их число и, следовательно, размер файла не ограничиваются при задании файла. Каждый элемент файла имеет номер. Первый элемент считается нулевым. В каждый момент времени программе доступен только один — *текущий элемент*, на который установлен указатель файла. После выполнения операции чтения или записи для элемента N указатель автома-

тически смещается к следующему по порядку элементу $N + 1$, который, в свою очередь, становится доступным для чтения или записи.

Обратите внимание — т. к. все элементы файла имеют одинаковую длину, позиция *каждого* элемента легко вычисляется. Поэтому указатель может быть перемещен *на любой* элемент файла, обеспечивая тем самым *прямой доступ* к нему.

Замечание

Фактически типизированный файл — это массив на диске с неограниченным размером и сравнительно невысокой скоростью обработки.

В отличие от текстового файла типизированный файл не делится на строки. Данные в нем изначально записаны во внутреннем двоичном представлении компьютера — в том виде, в котором они хранятся в оперативной памяти. Вообще, все файлы, кроме файлов типа `text`, рассматриваются как двоичные. Непосредственный просмотр такого файла в текстовом редакторе ничего не даст пользователю. Для того чтобы разобраться в содержимом типизированного файла, необходимо знать формат хранения данных в нем и уметь обрабатывать его при помощи программы.

Объявление типизированных файлов в программе осуществляется заданием файловой переменной вида `file of Тип`, где `Тип` — любой тип, кроме файлов:

```
type ИмяТипа = file of Тип;
var ФайловаяПеременная : ИмяТипа;
```

или

```
var ФайловаяПеременная = file of Тип;
```

Обратите внимание — не следует смешивать типизированные файлы `file of char` и текстовые файлы `text`, хотя физический файл с текстом можно обрабатывать, используя любой из этих типов. Разумеется, обработка при этом будет выполняться по-разному.

17.5.1. Процедуры и функции для типизированных файлов

При обработке таких файлов могут использоваться некоторые дополнительные процедуры и функции, а ряд известных нам общих *(см. разд. 17.3)* имеют свои особенности.

Открытие типизированного файла можно произвести стандартными способами: используя `reset` и `rewrite`. Процедура `reset` применяется для открытия *существующего* файла. Процедура `rewrite` используется для создания *нового* файла и его открытия.

Следует знать:

- типизированные и нетипизированные файлы всегда допускают *одновременно как чтение, так и запись*, независимо от того, были ли они открыты с помощью `reset` или `rewrite`;
- для чтения и записи типизированного файла используются только процедуры `read` и `write`. Использование `readln` и `writeln` — запрещено. Поскольку типизированный файл содержит элементы *одного типа*, список ввода для `read` и список вывода для `write` также должен содержать переменные только *одного типа*. Если этих переменных в списке несколько, указатель будет смещаться после каждой операции обмена данными между переменными и дисковым файлом.

Рассмотрим процедуры и функции для типизированных файлов подробнее. Повторим, что для текстовых файлов их использовать нельзя.

- Функция `filepos(ФайловаяПеременная)` — возвращает целое число — текущую позицию в файле. Функцию нельзя использовать для текстовых файлов. Если текущей позицией является начало файла — его *первый* компонент, например, после выполнения `reset`, — то функция возвращает значение *ноль*. При переходе от одного элемента к другому возвращаемое значение увеличивается на единицу. Результат — `longint`.
- Функция `filesize(ФайловаяПеременная)` — возвращает число элементов файла, но не текущий размер файла в байтах. Если файл пуст, она возвращает 0. Результат — `longint`.
- Процедура `seek(ФайловаяПеременная, НомерПозиции)` — перемещает указатель файла с текущей позиции к позиции с указанным номером, не выполняя чтение или запись. Поскольку номер первого элемента файла равен 0, то оператор `seek(ФайловаяПеременная, filesizeФайловаяПеременная)` ставит указатель файла за его конец, что используется при добавлении данных к файлу. `НомерПозиции` имеет тип `longint`.
- Процедура `truncate(ФайловаяПеременная)` — усекает размер файла до его текущей позиции. Все элементы файла за текущей позицией удаляются, после чего текущая позиция становится концом файла.

17.5.2. Задачи на обработку типизированных файлов

Рассмотрим ряд задач, связанных с этим типом файлов, в том числе задачу на создание простейшей *базы данных*.

Процедуры записи в текстовый файл массива целых чисел и его чтения из файла уже были нами рассмотрены (см. листинг 17.3). Не повторяя про-

грамму, приведем процедуры записи массива вещественных чисел в типизированный файл и чтения массива из файла.

```
const max=10;
type massiv=array[1..max,1..max] of real;
var m:integer; matrix1,matrix2:massiv;
...
procedure outmatrixfiletyp(m,n:integer; mas:massiv; name:string);
{ вывод двумерного массива mas в типизированный файл name }
var
   i,j:integer; mreal,nreal:real; file_typ:file of real;
begin
   mreal:=m; nreal:=n;
   assign(file_typ,name); rewrite(file_typ);
   write(file_typ,mreal,nreal);
   for i:=1 to m do
      for j:=1 to n do
         write(file_typ,mas[i,j]);
   close(file_typ)
end;
procedure  inmatrixfiletyp(var m,n:integer; var mas:massiv; name:string);
{ ввод двумерного массива mas из типизированного файла name }
var i,j:integer; mreal,nreal:real; file_typ:file of real;
begin
   assign(file_typ,name); reset(file_typ);
   seek(file_typ,0); read(file_typ,mreal,nreal);
   m:=round(mreal); n:=round(nreal);
   seek(file_typ,2);
   for i:=1 to m do
      for j:=1 to n do
         read(file_typ,mas[i,j]);
   close(file_typ);
end;
```

Листинг 17.4 содержит программу, которая формирует типизированный файл из целых чисел, вводимых с клавиатуры. Их количество заранее неизвестно. Признаком конца ввода является число 0. Программа находит: сумму и произведение всех чисел из файла, разность между предпоследним и вторым по счету числами, наибольшее из чисел.

Листинг 17.4. Ввод и обработка типизированного файла целых чисел

```
type file_type=file of integer;
var  f: file_type; sum,mult,r,k1,k2,max: integer;
begin
   writeln('Введите элементы файла, окончание - 0');
   { запись элементов в файл }
   assign(f,'data.dat'); rewrite(f);
   repeat
     readln(r); if r<>0 then write(f,r);
   until r=0;
   { вычисление результатов }
   seek(f,0); sum:=0; mult:=1;
   read(f,r); max:=r; seek(f,filepos(f)-1);
   while not eof(f) do
      begin
         read(f,r); sum:=sum+r; mult:=mult*r;
         { поиск максимального элемента }
         if max<r then max:=r;
      end;
   seek(f,1); read(f,k1);            { чтение второго компонента }
   seek(f,filesize(f)-2); read(f,k2);{ чтение предпоследнего компонента }
   close(f);
   { вывод результатов }
   writeln('Сумма = ',sum,#10#13'Произведение = ',mult);
   writeln('Разность = ',k2-k1,#10#13'Максимум = ',max);
end.
```

Комментарий

Процедура `seek(f,0)` устанавливает указатель в начало открытого файла. В данном случае ее использование по сравнению с процедурой `reset` является предпочтительным. Процедура `seek(f,filepos(f)-1)` возвращает указатель, сместившийся после считывания `read(f,r)`, на предшествующий компонент, что позволяет начать вычисление суммы и произведения с первого элемента (нулевого компонента) файла.

Кроме сортировки массива можно выполнить сортировку типизированного файла, переставляя его элементы таким образом, чтобы они оказались упорядоченными по возрастанию или убыванию. Листинг 17.5 содержит программу, которая формирует типизированный файл из k случайных чи-

сел и сортирует его методом "пузырька" с использованием внешнего цикла `repeat`.

Листинг 17.5. Сортировка файла методом "пузырька"

```
var  f: file of integer; buf1,buf2: integer;
     n,k,kol: longint; ok: boolean;
begin
   assign(f,'bubble.dat'); rewrite(f); randomize;
   write('Введите количество чисел в файле: '); readln(k);
   for n:=1 to k do
      begin
         buf1:=random(100); write(f,buf1); write(' ',buf1);
      end;
   kol:=k-1; {сначала все элементы неотсортированы}
   repeat
     ok:=true; kol:=kol-1;{на каждом шаге количество неотсортированных
                           элементов уменьшается на единицу}
     for n:=0 to kol do
        begin
           seek(f,n);read(f,buf1);read(f,buf2);
           if buf1>buf2 then
           { перестановка местами соседних чисел }
              begin
                 seek(f,n);write(f,buf2);write(f,buf1);
                 ok:=false;
              end;
        end;
   until ok;  { условие выхода - нет перестановок }
   writeln; writeln('Файл bubble.dat отсортирован:');
   seek(f,0); { указатель - на первый компонент с номером 0 }
   for n:=1 to k do
      begin
         read(f,buf1); write(' ',buf1);
      end;
   close(f); readln;
end.
```

Содержимое типизированных файлов можно рассматривать как последовательность записей определенного типа *(см. гл. 16)*. Листинг 17.6 содержит программу, которая создает простейшую базу данных (хранилище данных) на основе типизированных файлов записей. Требуется создать файл записей с заданным именем, поместив в него сведения о студентах потока (номер группы, фамилия, имя и три отметки за семестр). На основе файла выяснить процент успеваемости на "4" и "5" (количество студентов без "3", отнесенное к общему числу учащихся). Требуется также создать файл записей с заданным именем, поместив в него сведения о плохо успевающих студентах потока (с оценками "2" и "3") — номер группы, фамилия, имя, средний балл семестра — и вывести его записи на экран.

Листинг 17.6. База данных на основе типизированных файлов записей

```
type studrec=record
               group  : byte;
               surname: string[20];
               name   : string[20];
{ массив оценок: за семестр ставятся 3 оценки (от 2 до 5 баллов) }
               oc     : array[1..3] of 2..5;
            end;
     foolsrec=record
               group  : byte;
               surname: string[20];
               name   : string[20];
{ среднее арифметическое элементов массива оценок }
               average: real;
            end;
    file_typ1 = file of studrec; file_typ2 = file of foolsrec;
var f1: file_typ1; f2: file_typ2;
    namefile1, namefile2: string[12];
    s: studrec;   { запись, содержащая сведения на одного студента }
    fs: foolsrec; { запись со сведениями на одного задолжника }
    i,nomrec: integer;
procedure read_data(var s: studrec);
{ ввод записи по студенту }
begin
   with s do
   begin
      writeln('Окончание ввода - 0, как n группы');
```

```
      write('N группы: '); readln(group);
      if group <> 0 then
        begin
          write('Фамилия: '); readln(surname);
          write('Имя:     '); readln(name);
          writeln('Оценки: '); for i:=1 to 3 do read(oc[i]);
        end;
   end;
end;
procedure copy_data(s: studrec; var fs: foolsrec);
{ копирование записи по студенту-задолжнику }
begin
   with fs do
   begin
     group:=s.group; surname:=s.surname; name:=s.name;
{ вычисление среднего арифметического элементов массива оценок }
     average:=(s.oc[1]+s.oc[2]+s.oc[3])/3;
   end;
end;
procedure write_list(var f: file_typ2);
{ вывод записей по плохо успевающим студентам на экран }
begin
   reset(f); seek(f,0);
   writeln('Плохо успевающие:');
   writeln('Группа','Фамилия':16,'Имя':8,'Средний балл':18);
   while not eof(f) do
      begin
         { чтение текущей записи из файла }
         read(f,fs);
         { вывод текущей записи на экран }
         with fs do writeln(group:4,surname:15,name:10,average:16:2);
      end;
end;
procedure create_file(var f: file_typ1; var n: integer);
{ ввод записей по всем студентам в файл }
begin
   n:=0; rewrite(f); read_data(s);
   while s.group <> 0 do
```

```
      begin
         { вывод текущей записи в файл }
         write(f,s);inc(n);
         { ввод записи по новому студенту }
         read_data(s);
      end;
   close(f);
end;
procedure write_data(var f1: file_typ1; var f2: file_typ2; n: integer);
{ вывод записей по плохо успевающим студентам в файл, }
{ подсчет % успевающих студентов }
var priz:char; k:integer;
begin
   reset(f1); rewrite(f2); k:=0;
   while not eof(f1) do
      begin
         read(f1,s);
         { признак учебы студента }
         priz:='n';
         for i:=1 to 3 do if (s.oc[i]>=2)and(s.oc[i]<=3) then priz:='y';
            if priz='y' then
               begin copy_data(s,fs); write(f2,fs) end
            else inc(k);
      end;
   close(f1); close(f2);
   clrscr; writeln('На "4"и"5" учатся ',k/n*100:3:0,'% студентов');
end;
begin
   write('Введите имя файла общей ведомости: '); readln(namefile1);
   assign(f1,namefile1);
   create_file(f1,nomrec);
   write('Введите имя файла ведомости задолжников: '); readln(namefile2);
   assign(f2,namefile2);
   write_data(f1,f2,nomrec);
   write_list(f2);
end.
```

Комментарий

Для каждого студента данные о номере его группы, фамилии, имени и отметках содержатся в полях записи `studrec`: переменных `group`, `surname`, `name` и массиве `os` соответственно. Для плохо успевающего студента данные о номере его группы, фамилии, имени и среднем балле содержатся в полях записи `foolsrec`: переменных `group`, `surname`, `name` и `average` соответственно. Создание типизированного файла базы данных на студентов потока выполняется процедурой `create_file`, а заполнение полей записи по отдельному студенту — процедурой `read_data`. Формирование типизированного файла базы данных на плохо успевающих студентов потока выполняется процедурой `write_data`, которая также подсчитывает процент обучающихся на "4" и "5". Процедура `copy_data` выполняет вспомогательную роль, копируя поля записи студента в соответствующие поля записи задолжника и вычисляя средний балл по оценкам семестра. Данные из файла задолжников выводятся процедурой `write_list`.

17.6. Нетипизированные файлы

При выполнении копирования или обработке баз данных приходится иметь дело с файлами, состоящими из компонентов одинакового размера, структура которых неизвестна или не имеет значения. На практике это приводит к тому, что любой файл, подготовленный как текстовый или типизированный, можно открыть и начать работу с ним как с *нетипизированным* набором данных прямого доступа.

Нетипизированные файлы используются, в основном, для *быстрого прямого* доступа к любому файлу на диске, *независимо от его типа и структуры*. При работе с нетипизированными файлами не нужно терять время на преобразование типов и поиск управляющих последовательностей, достаточно считать содержимое файла в определенную область памяти. *Объявление нетипизированных файлов* в программе выполняется путем задания файловой переменной типа `file`:

```
type ИмяТипа = file;
var ФайловаяПеременная : ИмяТипа;
```

или

```
var ФайловаяПеременная = file;
```

Для нетипизированных файлов в процедурах `reset` и `rewrite` допускается указывать дополнительный параметр, с тем чтобы задать размер записи, которая используется при передаче данных. По умолчанию длина записи равна 128 байт.

За исключением процедур `read` и `write` для всех нетипизированных файлов допускается использование любой стандартной процедуры, которая пригодна для типизированных файлов. Вместо процедур `read` и `write` здесь ис-

пользуются соответственно процедуры `blockread` (считывает из файла в переменную одну или более записей) и `blockwrite` (записывает одну или более записей из переменной в файл), которые позволяют пересылать данные с высокой скоростью.

Замечание

Для обеспечения максимальной скорости обмена данными следует задавать длину, которая была бы кратна длине физического сектора дискового носителя информации. Более того, как мы уже знаем, фактически пространство на диске выделяется файлу кластерами. Как правило, кластер может быть прочитан или записан за один оборот диска, поэтому наивысшую скорость обмена данными можно получить, если указать длину записи, равную размеру кластера.

17.7. Контрольные вопросы и задания

Вопросы

1. Что такое файл? Для каких целей используются файлы? Их достоинства?
2. Что такое диск, каталог, подкаталог, вложенный каталог, вышестоящий каталог, корневой каталог, текущий каталог? В чем состоит преимущество использования древовидной структуры каталогов? Приведите примеры.
3. Выполните сравнение файла и массива.
4. Назовите общие и отличительные черты текстовых, типизированных и нетипизированных файлов.
5. Назовите стандартные текстовые файловые переменные.
6. Для каких целей используется указатель позиции файла?
7. В чем состоит разница между файлами последовательного и прямого доступа?
8. Что такое файловый буфер? Для каких целей он используется?
9. Что общего у процедуры `reset` и `rewrite` и чем они отличаются? Приведите примеры.
10. Какие отличия существуют в использовании процедуры `reset` при открытии различных типов файлов? Приведите примеры.
11. Зачем применяется процедура `close`? Приведите примеры.
12. Какие файлы относятся к текстовым? Как представлена информация в текстовых файлах?
13. Дайте рекомендации по выбору типа буферной переменной для текстовых файлов.

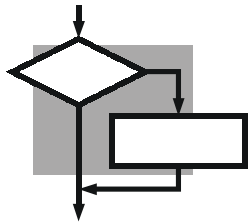
14. Какие файлы относятся к типизированным? Как представлена информация в типизированных файлах?
15. Выполните сравнение текстового и типизированного файла.
16. В чем заключается специфика нетипизированных файлов? Как выполняется их описание?

Задания

1. Заданы два текстовых файла x.txt и y.txt. Создайте третий файл z.txt, в который поместите сначала все строки файла x.txt, затем все строки файла y.txt. Подсчитайте число строк в полученном файле. Выведите на экран все самые длинные и самые короткие строки.
2. Задан текстовый файл с произвольным именем. Подсчитайте, сколько слов содержит этот файл. Все слова разделены пробелами, необязательно одиночными. Найдите все слова максимальной длины (или с максимальным количеством гласных букв).
3. Заданы два текстовых файла, состоящие из целых чисел, упорядоченных по возрастанию. Необходимо объединить их вместе таким образом, чтобы в полученном файле числа оказались также упорядоченными по возрастанию.
4. Задан текстовый файл, состоящий из строк с числами. Найдите в файле строку, числа в которой упорядочены по возрастанию, и выведите ее на экран. Если таких строк несколько, то выведите все.
5. Задан текстовый файл с произвольным именем. Создайте программу для выравнивания его строк. Программа должна найти в файле строку максимальной длины и удлинить все строки с меньшей длиной путем вставки дополнительных пробелов между словами. Полученный текст запишите в новый файл.
6. Задан символьный файл. Считая, что длина одного слова не больше 15, определите: сколько в файле имеется слов, состоящих из одного, двух, трех и т. д. символов; изобразите гистограмму (столбчатую диаграмму) длин всех слов.
7. Задан символьный файл. Считая, что каждое его предложение состоит не более чем из 20 слов, определите: сколько в файле имеется предложений, состоящих из одного, двух, трех и т. д. слов.
8. Задан символьный файл. Определите, сколько раз в заданном файле встречается каждая буква алфавита.
9. Задан текстовый файл, каждая строка которого может рассматриваться как запись целого числа в двоичной системе счисления. Преобразуйте двоичную запись числа в десятичную.

10. Задан типизированный файл целых чисел. На его основе создайте текстовый файл, в каждой строке которого необходимо записать число в десятичной системе счисления из исходного файла и соответствующее ему число в двоичной системе счисления.
11. Создайте типизированный файл вещественных чисел, упорядочите компоненты файла по возрастанию. Введите число с клавиатуры и поместите его в файл так, чтобы не нарушить упорядоченности (без создания дополнительного файла).
12. Создайте два типизированных файла вещественных чисел, упорядочите компоненты файлов по возрастанию. Получите новый файл слиянием двух исходных файлов, так чтобы не нарушить упорядоченности.
13. Создайте типизированный файл вещественных чисел. Задайте с клавиатуры номер компонента *N*, с которым необходимо удалить *K* элементов файла. Выведите компоненты исходного и усеченного файлов. Дополнительный файл для решения задачи не использовать.
14. Создайте типизированный файл вещественных чисел. Задайте с клавиатуры номер компонента *N*, после которого необходимо вставить *K* элементов файла. Выведите компоненты исходного и дополненного файлов.
15. Сведения о студенте включают фамилию, имя и перечень отметок, полученных им в сессию. Создайте отдельные типизированные файлы, собрав в них сведения об отличниках, успевающих и кандидатах на отчисление (это студенты, у которых есть хотя бы одна двойка).
16. Создайте типизированный файл, содержащий сведения об абитуриентах. Сведения о каждом абитуриенте — это фамилия, шифр специальности и оценки, полученные на трех вступительных экзаменах. Задайте с клавиатуры проходной балл для указанной специальности и выведите список абитуриентов, прошедших по конкурсу. Задайте с клавиатуры количество абитуриентов, которые могут быть зачислены на указанную специальность, выполните сортировку по убыванию среднего балла в файле, рассчитайте проходной балл и выведите список на экран. Замечание: в случае необходимости в запись могут быть добавлены дополнительные поля. Массивы не использовать.
17. Выполните обработку информации, используя два типизированных файла. Первый файл имеет структуру записи с полями: должность, оклад, отдел, и используется в качестве справочника. Второй файл имеет структуру записи с полями: фамилия, должность, дата приема, стаж, оклад, процент надбавки. Задайте с клавиатуры значения полей: фамилия, должность и дата приема на работу. Вычислите и заполните оставшиеся поля, используя первый файл и учитывая следующие правила: надбавка составляет 5%, если стаж работы менее 5 лет; 10%, если стаж работы от 5 до 10 лет; 15%, если стаж работы более 10 лет. Выведите информацию из файлов на экран.

Глава 18



Динамические структуры данных

Материал данной (и следующей) главы представляет известную трудность для начинающих, т. к. предполагает глубокое понимание основных принципов программирования и хорошо развитое логическое мышление. Надеемся, что к моменту работы над этими темами читатель уже достиг требуемого уровня и готов подняться еще на одну ступеньку в освоении профессионального программирования.

18.1. Еще раз о распределении памяти

Осталось совсем немного добавить к тому, что уже известно о распределении памяти для программы и ее данных. Окончательная картина распределения памяти в системе Turbo Pascal показана на рис. 18.1. При использовании 32-разрядных компиляторов возможны небольшие отличия от приведенной схемы, но сам принцип распределения памяти для программы и данных остался прежним.

Как известно, для хранения глобальных переменных, а также типизированных констант и статических переменных формируется область, которую называют *сегментом данных (областью данных)*. Для хранения локальных переменных и параметров процедур и функций используется другая область памяти, которая называется *стеком*. Размеры сегмента данных и стека не могут быть изменены в ходе выполнения программы.

В 16-разрядных системах программирования, таких как Turbo Pascal, размеры каждого сегмента ограничены 64 килобайтами. Превышение этих границ приводит к двум печально известным ошибкам **Structure too large** (при попытке объявить слишком большой глобальный массив) и **Stack overflow (**переполнение стека, возникает при использовании больших локальных массивов или при глубокой рекурсии).

Современные 32-разрядные системы позволяют использовать память компьютера гораздо рациональнее, снимая жесткое ограничение на размеры памяти под программные переменные. Однако это вовсе не означает, что

программист может расходовать память компьютера, как ему захочется. Наоборот, в современных многозадачных операционных системах ответственность за рациональное использование памяти, лежащая на программисте, еще более возрастает. А что может быть нерациональнее, чем объявление массива из десятков и сотен тысяч элементов (на всякий случай), если реальное количество элементов обычно оказывается намного меньше?

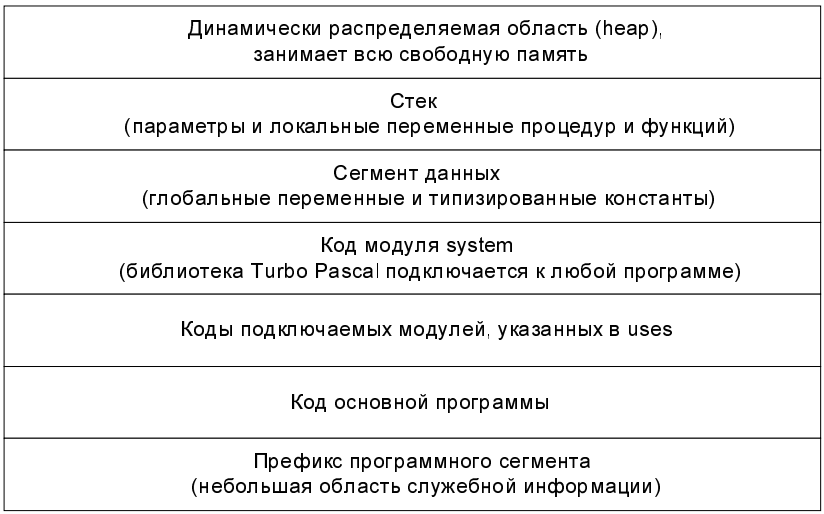


Рис. 18.1. Карта распределения памяти для исполнимого модуля программы на Turbo Pascal

Исходя из того, что реальное количество данных не всегда можно точно предсказать при разработке программы, большинство языков программирования предоставляют программисту замечательную возможность *создавать и уничтожать переменные по ходу выполнения программы.* Такие переменные, создаваемые и уничтожаемые непосредственно в программе, получили название *динамических.* Они не описываются заранее в разделе объявлений и не имеют своего собственного имени, а память под них выделяется (захватывается) в соответствии с инструкциями программы.

Заполнение выделенной памяти каким-либо значением в момент создания переменной вовсе не обязательно — это можно будет сделать позже. Точно так же можно и уничтожить динамическую переменную — просто освободить выделенную под нее память, не затирая содержимое ячейки.

Динамические переменные создаются не в сегменте данных и не в стеке, для них выделяется еще одна дополнительная область памяти, которая называется *динамически распределяемой областью*, или *кучей.* К сожалению, работать с переменными в куче можно только через указатели *(см. разд. 18.2).* Однако для

тех, кто освоит технику работы с указателями, в дальнейшем использование динамических переменных не представит особых сложностей.

Замечание

При работе в Turbo Pascal под кучу отводится вся оставшаяся доступная программисту память (в операционной системе MS-DOS это около 300 Кбайт). В некоторых случаях возникает необходимость высвободить часть этой памяти для других целей, допустим, из программы на Turbo Pascal требуется запустить другую программу или выполнить команду операционной системы *(см. приложение 3)*. В этом случае нужно использовать директиву компилятора

```
{$M РазмерСтека, Мин.РазмерКучи, Макс.РазмерКучи}
```

Используя эту директиву, можно заодно изменить и размеры стека, например: `{$M 65520, 0, 65536}` — увеличили стек до максимально возможного размера, а размер кучи уменьшили до 64 Кбайт.

Следует знать:

- динамические переменные являются переменными с *управляемым временем жизни*. Их время жизни длится от момента создания до момента уничтожения, и управляет этим процессом программа;
- по окончании работы программы вся выделенная для нее память, в том числе и куча, освобождается;
- тем не менее, старайтесь не забывать вовремя удалять все созданные при выполнении программы динамические переменные, если вы не хотите, чтобы ваша программа "кушала память" (так говорят о программах, которые в процессе работы захватывают все новые и новые области памяти).

18.2. Указатели. Описание указателей

Указатели — это особый тип данных. В переменных этого типа хранятся адреса других переменных, содержащих полезную для программы информацию. На первый взгляд может показаться, что использование указателей приводит к лишним затратам памяти и к усложнению программы, а также существенно усложняет и сам процесс программирования. В данной главе мы приведем такие примеры использования указателей, из которых станет ясно, что все дополнительные затраты на их хранение и обработку окупаются в полной мере. Работа с указателями предусмотрена не только в Pascal, но и в некоторых других языках программирования. Например, в языке C указатели используются практически в любой программе.

В Pascal роль указателей несколько скромнее, и тем не менее, начинающим программистам следует усвоить базовые принципы работы с указателями, чтобы глубже понять внутренний механизм обработки и выполнения любой компьютерной программы.

18.2.1. Указатели и адреса

Известно, что адресом переменной является адрес первого байта ячейки памяти, которая под нее отводится *(см. разд. 4.1.2)*. Для данных структурных типов (массивов и записей) их адресом считается адрес первого байта первого элемента.

В Turbo Pascal существует возможность прямого доступа к любому байту оперативной памяти по его адресу при помощи определенных в модуле `system` массивов `Mem`, `MemW` и `MemL`, которые позволяют записать информацию или прочитать ее непосредственно из ячеек памяти (один, два или четыре байта). Это очень опасные действия, поэтому они исключены в 32-разрядных системах программирования. Все же дадим краткие пояснения для тех, кто работает в среде Borland (Turbo) Pascal.

В качестве индекса в этих массивах используется адрес, записанный в виде, принятом в DOS: `Сегмент : Смещение относительно начала сегмента`. Такой странный способ записи адреса связан с тем, что в операционной системе DOS вся память разбита на сегменты, размеры которых не превышают 64 Кбайт. Для получения абсолютного адреса из пары `Сегмент : Смещение` система прибавляет к сегменту справа шестнадцатеричный ноль (это четыре нуля в двоичной системе), а затем складывает его со смещением. Таким способом можно адресовать 1 Мбайт памяти.

Например, начальный адрес видеобуфера запишется в виде `$B800:$000`, а обратиться к самому первому его байту можно так: `Mem[$B800:$0000]`, к первым двум байтам — `MemW[$B800:$0000]`, к первым четырем байтам — `MemL[$B800:$0000]`. Абсолютный адрес, соответствующий данной паре, — `$B8000`.

Еще один пример для любознательных — оператор `mem[0:$41C]:=mem[0:$41A];` можно применить для принудительной очистки буфера клавиатуры. Здесь адрес маркера конца буфера клавиатуры приравнивается к адресу его начала. Конечно, в данном случае лучше воспользоваться средствами модуля `crt` *(см. приложение 2)*.

Замечание

Для доступа к внешним устройствам компьютера используются предопределенные массивы `Port[Номер порта]` и `PortW[Номер порта]`.

Имеется еще один способ обращения к оперативной памяти — использование служебного слова `absolute` при описании переменной. В этом случае переменная будет располагаться именно по тому адресу в оперативной памяти, который указан после `absolute`. Разумеется, использование служебного слова `absolute` — столь же опасный способ, как и обращение к памяти через предопределенные массивы.

Однако absolute может использоваться и более безопасным способом, позволяя совмещать в памяти две переменные с разными именами. Пример использования такого совмещения приводился в листинге 13.9 (процедура print3).

В языке Pascal есть специальная операция получения указателя на переменную (или процедуру) — она обозначается как @. Имеется также эквивалентная ей функция addr.

Например, @x или addr(x) — адрес переменной x.

Имеется и обратная операция получения значения переменной по ее адресу, которая обозначается знаком ^. Например, p^ — переменная с адресом p.

В повседневной практике средства работы с адресами используются довольно редко. Основное назначение указателей состоит в том, чтобы обеспечить механизм использования в программе *динамических переменных*. Этот механизм мы и будем обсуждать подробно в следующих разделах.

18.2.2. Описание указателей

В Pascal имеются два различных вида указателей: типизированные и нетипизированные. *Типизированный указатель* — это указатель на переменную определенного типа, например, целого, строкового или типа массива. *Нетипизированный указатель* — это адрес первого байта области памяти, в которой может размещаться любая информация вне зависимости от ее типа.

Описание двух видов указателей выполняется по-разному:

```
var p1: ^integer; { указатель на переменную целого типа }
    p2: ^string;  { указатель на строку }
    p3: pointer;  { нетипизированный указатель }
```

Заметим, что тип pointer совместим со всеми типами указателей. В дальнейшем изложении для удобства имена всех указателей будем начинать с буквы p (pointer).

Каждый указатель размещается в сегменте данных или в стеке (если он объявлен в подпрограмме) и занимает там 4 байта. Это дополнительные "накладные расходы" памяти. Поэтому обычные переменные очень редко создают и уничтожают динамически, оставляя эту возможность для больших совокупностей данных.

Чем больше размер динамической переменной, тем меньше доля накладных расходов. Например, при хранении в динамической памяти массивов больших размеров лишние 4 байта, затраченные на указатель, несущественны.

18.3. Создание и удаление динамических переменных

В соответствии с двумя типами указателей существуют и две разные процедуры создания динамических переменных:

- `new(p);` — для типизированных указателей;
- `getmem(p, size);` — для нетипизированных указателей.

Параметр `size` задает размер памяти в байтах, которую необходимо выделить.

Замечание

В 16-разрядных компиляторах размер `size` не может превышать 64 Кбайт. Это существенное ограничение, которое не позволяет, например, сохранить в одной переменной образ всего графического экрана.

Например:

```
new(p1); new(p2); new(pmas);
getmem(p3,200);
```

При выполнении процедуры `new` в динамической памяти выделяется столько байтов, сколько требуется для хранения переменной заданного типа. При этом указателю, который является параметром-переменной, присваивается адрес первого байта выделенной памяти.

Для нетипизированных указателей в памяти выделяется ровно столько байтов, сколько указано во втором параметре процедуры `getmem` (в нашем примере — 200 байт), а указатель получает значение адреса первого байта выделенной области.

Таким образом, в приведенном примере указатели `p1`, `p2`, `p3`, `pmas` получают свои значения только после выполнения процедур выделения памяти.

Обратите внимание — динамические переменные не имеют собственного имени, и в процедурах выделения памяти задается не имя переменной, а имя указателя.

Обращаться к динамическим переменным нужно через их указатели, используя знак `^`. Эта операция называется *разыменованием* указателя.

Например:

```
p1^:=5; p2^:='любая строка текста'; pmas^[3]:=5.6;
```

До использования процедур `new` или `getmem` значения указателей считаются неопределенными, и работать с ними нельзя. Правда, можно присвоить им "пустое" значение `nil`, не указывающее ни на какое место в памяти (по аналогии с числовыми переменными можно назвать такое действие обнулением).

К сожалению, типовая ошибка начинающих заключается в том, что они пропускают (забывают) процедуры выделения памяти и обнуления указателя. Последствия такой ошибки состоят в том, что программа обращается к несуществующим адресам в памяти. Понятно, что ничего хорошего в этом нет.

При выделении динамической памяти полезными являются следующие функции:

- `memavail` — возвращает общий размер свободной памяти в байтах;
- `maxavail` — возвращает размер наибольшего непрерывного участка свободной памяти.

Например:

```
var p:^real;
...
if maxavail>sizeof(real) then new(p);
```

Существует несколько разных способов освобождения динамически выделенной памяти. Для типизированных указателей можно использовать процедуры:

- `dispose(p)` — освобождает память, на которую указывал `p`;
- `mark(p)` и `release(p)` — эти две процедуры могут использоваться только вместе и позволяют очистить сразу целую область памяти. При этом процедура `mark(p)` запоминает в указателе `p` адрес начала области динамической памяти, а процедура `release(p)` очищает всю память, начиная с адреса `p`. Процедура `mark` обычно помещается в начало программы, а `release`, наоборот, в конец.

В случае нетипизированных указателей можно использовать только одну процедуру:

`freemem(p,size)` — освобождает `size` байтов, начиная с адреса `p`.

После освобождения памяти указатели автоматически не обнуляются и фактически указывают на несуществующую переменную. Поэтому рекомендуем присвоить всем высвободившимся указателям значение `nil`.

Познакомившись с различными способами выделения и освобождения памяти, перейдем к практическим действиям. Уже понятно, что динамическое выделение памяти и указатели стоит использовать только при большом количестве данных, подлежащих обработке. Динамические переменные используются в основном в двух ситуациях:

- для работы с массивами больших (или переменных) размеров;
- для работы с особыми структурами переменных размеров, которые получили название *динамических структур данных*.

Поясним подробнее.

18.4. Динамически формируемые массивы и строки

Объявить указатель на массив можно, например, так:

```
type massiv=array[1..10000] of real;
     pmassiv=^massiv;
var p: pmassiv;
```

Обратите внимание — при таком объявлении память будет выделена *только под указатель* (всего 4 байта). Память под массив необходимо будет *выделить* в куче по ходу выполнения программы.

Если забыть об этом и начать работу с таким массивом, не захватив под него память, возникнут совершенно непредсказуемые, странные ошибки, т. к. значения, которыми мы будем заполнять массив, на самом деле запишутся в какие-то другие ячейки памяти, испортив другие переменные программы. К сожалению, компилятор не отлавливает такие ошибки, поскольку выделение памяти в этом случае выполняет не он.

В Turbo Pascal выделение памяти под динамический массив должен выполнить сам программист, используя процедуры для выделения памяти `new` или `getmem`. Например, для массива из приведенного примера это можно сделать так:

```
new(p);
```

или

```
getmem(p,n*sizeof(real));
```

где `n` — реальное количество элементов, которое к этому моменту уже должно быть известно. Первый способ используется для массивов больших размеров (в Turbo Pascal их размер все же ограничен 64 килобайтами), второй — для массивов переменных размеров. Работать с таким массивом можно почти так же, как с обычным массивом, используя выражение `p^` в качестве имени массива. В листинге 18.1 приводится пример работы с массивом переменного размера, динамически сформированным в куче.

Листинг 18.1. Работа с массивом в динамической памяти

```
type massiv=array[1..10000] of real;
     pmassiv=^massiv;
var  p:pmassiv; n,i:integer;
begin
   randomize;
   write('Введите количество элементов массива '); readln(n);
```

```
   getmem(p,n*sizeof(real)); { захватили в куче ровно n ячеек памяти }
   for i:=1 to n do p^[i]:=random; { заполняем массив в куче }
   for i:=1 to n do write(p^[i]:6:2); writeln; {выведем двумя способами,}
   for i:=n downto 1 do write(p^[i]:6:2);      { чтобы не сомневаться }
   readln; freemem(p,n*sizeof(real));   {не забыли освободить память}
end.
```

В Delphi, похоже, вообще отпала необходимость использования указателей для работы с массивами, поскольку появилась возможность использовать в программе открытые массивы, количество элементов которых не указывается при описании. Открытые массивы уже упоминались в связи с передачей массивов в подпрограмму в качестве параметра *(см. разд. 13.4)*. Там же были перечислены функции для работы с открытыми массивами `low`, `high`, возвращающие минимальное и максимальное значение индекса. В Delphi добавлена еще одна функция `length`, возвращающая длину массива (запомнить легко, т. к. аналогичная функция используется для строк).

Теперь можно пояснить, какое действие выполняет компилятор, встретив описание открытого массива, — он выделяет память под указатель на этот массив. Реальное выделение памяти под массив происходит при использовании процедуры

```
setlength(ИмяМассива, КоличествоЭлементов);
```

Освобождение памяти выполняет та же процедура:

```
setlength(ИмяМассива, 0);
```

В принципе, данная процедура может использоваться применительно к одному массиву много раз, а это значит, что размеры массива можно изменять по ходу работы программы. Неплохая возможность для экономии памяти.

В качестве примера — короткая программа (листинг 18.2).

Листинг 18.2. Демонстрация использования открытых массивов

```
var a: array of integer;
begin
   setlength(a,10); {захватили память под 10 элементов}
   writeln (low(a):3, high(a):3, length(a):4);
   setlength(a,100);  {захватили память под 100 элементов}
   writeln (low(a):3, high(a):3, length(a):4);
   setlength(a,0);  {освободили память}
   readln;
end.
```

При выполнении программы будет выведено:

```
0  9  10
0 99 100
```

Следует знать:

- нумерация индексов открытого массива всегда начинается с нуля, поэтому функция `low` всегда возвращает значение 0, а `high` — значение на единицу меньшее, чем `length`;
- работа с открытым массивом ничем не отличается от работы с обычным массивом, индекс которого начинается с нуля.

Кроме открытых массивов, в Delphi предлагается еще один тип динамически формируемых данных — *открытые строки*. Этот тип называется `ansistring`, поэтому иначе такие строки называют ansi-строками. Открытые строки интересны тем, что размер их практически неограничен, но работать с ними можно, используя те же самые стандартные процедуры и функции, что и для обычных строк в стиле Pascal.

Рассмотрим подробнее внутреннее представление типа `ansistring`. При описании для переменной такого типа отводится 4 байта — это указатель на строку, которому пока присвоено пустое значение `nil`. Все точно так же, как и для открытого массива.

При вводе строки или при присваивании ей какого-либо значения автоматически выделяется память, необходимая для хранения всех символов и завершающего строку нуля. Указатель на строку при этом получает значение, равное адресу первого символа строки.

Можно выделить память под строку и при помощи функции `setlength` точно так же, как и для динамически формируемых массивов.

Интересно, что для каждой такой строки ее длина все же хранится, хотя строка и завершается нулем. Для хранения длины строки в памяти отводится уже не 1 байт, как для строки в стиле Pascal, а 4 байта. Поэтому оператор `writeln(ord(s[0]))`, который прекрасно работает в Turbo Pascal, в Delphi даже не компилируется. Чуть ниже мы все-таки приведем пример небольшой программы, в которой сумеем вывести хранимое значение длины строки без использования стандартной функции `length`.

Важно понять еще одну особенность хранения строк `ansistring`. Когда мы присваиваем строке значение другой строки, например, так:

```
s2:=s1;
```

то никакого копирования строки в памяти не происходит, но адрес `s2` становится равным адресу `s1`. Теперь два указателя указывают на одну строку в памяти (иначе говорят, два указателя ссылаются на одну строку). Поэтому способ хранения строки типа `ansistring` иначе называют ссылочной моде-

лью строки. Для подсчета количества ссылок на строку в ней отводится 4 дополнительных байта. Таким образом, для каждой хранимой в памяти строки ansistring в памяти хранятся две дополнительных величины: длина строки и количество ссылок на эту строку.

Замечательной особенностью строк ansistring является тот факт, что каждую копию строки можно изменять независимо от других копий. При любом изменении копии происходит настоящее копирование с выделением памяти. Такой способ известен как копирование при записи. В этом случае значение счетчика ссылок автоматически уменьшается на единицу. Как только значение счетчика ссылок окажется равным нулю, память, занимаемая строкой, автоматически освобождается.

С учетом сказанного, внутреннее представление строки выглядит так, как показано на рис. 18.2. Представленная строка имеет значение "Текст", поэтому ее длина равна 5, а значение количества ссылок произвольно выбрано равным 1. "Хвост" строки, содержащий любые символы ("мусор"), может отсутствовать.

Рис. 18.2. Внутреннее представление строки типа ansistring

К сожалению, Delphi не предоставляет никакой стандартной функции для получения значения счетчика ссылок. Для доказательства правильности представления, изображенного на рис. 18.2, приведем следующую программу:

```
uses sysutils;
var s1,s2 :string; { тип string в Delphi эквивалентен ansistring }
    plen,pkol :pointer;{указатели на хранимые длину строки и счетчик
                        ссылок}
    len, kol  :integer; {значения длины строки и счетчика ссылок}
begin
s1:='строка текста'; {захватили память под s1 и поместили туда текст}
s2:=s1;{присвоили указателю s2 значение указателя s1, никакой памяти не
        захватили}
plen:=pointer((integer(pointer(s1))-4)); {от начала строки вычли 4 байта}
len:=integer(plen^);{нашли длину}
pkol:=pointer((integer(pointer(s1))-8));{от начала строки вычли 8 байт}
```

```
kol:=integer(pkol^);{нашли количество ссылок — оно будет равно 2}
writeln('длина строки ',len,', количество ссылок ',kol);
s2:='*'+s1+'*'; {теперь под строку s1 выделилась новая область памяти}
pkol:=pointer((integer(pointer(s1))-8)); {снова нашли указатель на
                                          количество ссылок}
kol:=integer(pkol^); {теперь количество ссылок равно 1}
writeln('теперь количество ссылок ',kol);
readln;
end.
```

Конечно, пришлось воспользоваться не самыми лучшими приемами программирования, поскольку нам требуется вывести на экран информацию, которая является служебной и не предназначена для вывода. Для того чтобы вычесть 4 или 8 байт из адреса начала строки, нам пришлось несколько раз выполнять автоопределенное преобразование то к типу `pointer`, то к типу `integer`. Иногда программистам приходится добывать ценную информацию и такими способами.

18.5. Структуры данных на основе указателей

Безусловно, массивы являются наиболее важным структурным типом данных. Они присутствуют практически во всех языках высокого уровня и позволяют эффективно реализовать многие алгоритмы обработки данных (эффективно — это значит быстро и с разумными затратами памяти).

Однако используя массивы, необходимо помнить, что при любом способе выделения памяти массив занимает в ней *непрерывную область*. В результате поисков решения проблемы быстрой обработки больших и быстро меняющихся объемов данных были разработаны и предложены к использованию *динамические структуры данных*.

Они характеризуются следующими особенностями:

- для отдельных элементов структуры память выделяется в тот момент, когда в них появляется необходимость (а не сразу непрерывной областью под весь массив);
- число элементов динамической структуры заранее не объявляется и может изменяться от нуля до некоторого значения, определяемого спецификой решаемой задачи или доступным объемом памяти;
- память, занимаемая структурой, *не представляет собой непрерывную область*, т. е. элементы структуры могут быть разбросаны в памяти хаотически, поскольку память под каждый элемент выделяется отдельно;

- логическая последовательность элементов задается в явном виде с помощью одного или нескольких указателей, хранящихся в самих элементах. Как правило, каждый элемент помимо своего значения хранит указатели на связанные с ним другие элементы (например, указатель на следующий элемент или на два соседних элемента).

Для того чтобы идея стала совсем понятной, проведем такую аналогию. При записи на прием к врачу каждый пациент получает свой порядковый номер в списке пациентов, и ему сообщается время приема. Если очередь к врачу движется строго по номерам, то никто не обязан запоминать своих соседей по очереди. Количество пациентов при таком способе организации приема известно заранее.

Другое дело — живая очередь (в магазине, в железнодорожной кассе). Количество человек в очереди постоянно меняется, люди приходят и уходят, но никакой неразберихи не возникает — каждый знает, за кем он стоит. Образуется связанная цепочка людей. Очевидно, кому-то из стоявших в очередях и пришла идея создания динамических структур данных.

Простая по своей сути идея оказалась не столь простой в реализации, поэтому дальнейший материал потребует повышенного внимания. В двух последних разделах данной главы мы рассмотрим две наиболее распространенные структуры на основе указателей — *связанные списки* и *двоичные поисковые деревья*.

18.6. Связанные списки

18.6.1. Общие сведения

Связанный список — это такая структура данных, элементами которой служат записи одного и того же формата, связанные друг с другом с помощью указателей, расположенных в самих элементах. Элементы списка часто называются его *узлами*.

Использование записей для представления элементов связанного списка вполне закономерно. Это единственный комбинированный тип данных, который допускает группирование данных различных типов *(см. разд. 16.1)*. Каждый элемент списка состоит из двух различных по значению частей: *содержательной* (информационной) и *указующей* (рис. 18.3). В минимальном варианте содержательная и указующая части занимают по одному полю записи, но могут представлять собой совокупность из более чем одного поля.

В содержательной части хранятся данные, ради которых и создается список.

Если указующая часть хранит адрес одного соседнего элемента списка, то такой список называется *односвязным*, или *однонаправленным*. Поле указателя последнего элемента содержит специальный признак `nil` (признак конца списка). Для каждого элемента (кроме последнего) имеется единственный

следующий элемент. Поэтому связанные списки относят к линейным структурам данных, как и массивы. Логическая структура односвязного списка представлена на рис. 18.4.

Элемент (узел) связанного списка - запись	
Содержательная часть	Указующая часть
Данные любого типа	Один или два указателя на соседний элемент (элементы)

Рис. 18.3. Элемент связанного списка

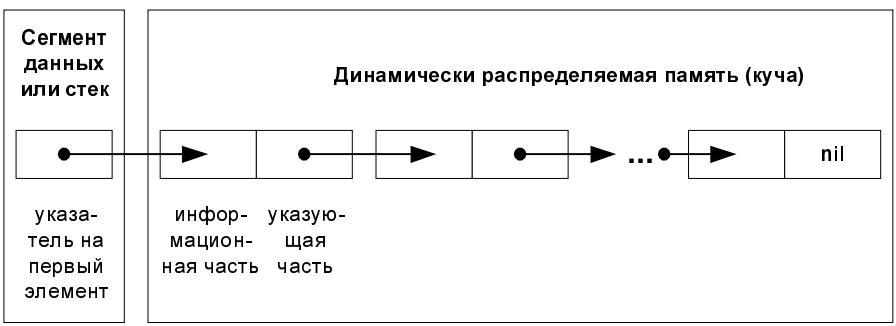


Рис. 18.4. Структура односвязного списка

В том случае, когда в указующей части хранятся адреса и предыдущего, и следующего элементов, список называется *двусвязным*, или *двунаправленным*. Двунаправленный список более универсален, т. к. по такой цепочке можно двигаться в двух направлениях: прямом и обратном. В двусвязном списке можно продвигаться от элемента к элементу по двум противоположным направлениям, поэтому в конце каждого пути в поле соответствующего указателя находится признак пустого указателя (`nil`).

Как композицию односвязного и двусвязного списков можно получить *кольцевой* список, который вообще не содержит пустых указателей. Кстати, буфер клавиатуры ПК реализован именно как кольцевой список.

С линейными списками можно выполнять те же действия, что и с одномерными массивами, поскольку назначение списков и массивов одно и то же — обработка данных в оперативной памяти. Перечислим типовые действия со списками:

- добавить новый узел непосредственно перед заданным узлом;
- удалить заданный узел;

- объединить два (или более) линейных списка в один список;
- разбить линейный список на два (или более) списка;
- сделать копию списка;
- выполнить сортировку узлов списка в возрастающем порядке по некоторым полям в узлах;
- найти в списке узел с заданным значением в некотором поле.

Очень часто встречаются линейные списки, в которых добавление и удаление производятся только в первом или последнем узлах. Назовем эти списки.

- *Стек* — линейный список, в котором все добавления и удаления (и обычно всякий доступ) производятся в одном конце списка.
- *Очередь* — линейный список, в котором все добавления производятся на одном конце списка, а все удаления — на его противопожном конце.
- *Дек* (очередь с двумя концами) — линейный список, в котором все добавления и удаления делаются на обоих концах списка.

Прежде чем рассматривать действия со связанными списками, введем обозначения переменных, которыми мы будем пользоваться при описании соответствующих алгоритмов и структур данных (табл. 18.1).

Таблица 18.1. *Обозначения переменных, используемых в листингах 18.3—18.5*

Элемент	Описание
`item`, `pitem`	Тип для одного элемента списка (запись) и указателя на него
`data`	Поле данных (информационная часть элементов списка)
`next`, `prev`	Указатели следующего, предыдущего элементов (указующая часть)
`head`, `top`	Указатели на первый и последний элемент списка
`P1`, `p2`	Рабочие указатели

Опишем тип одного элемента односвязного списка и указателя на этот элемент:

```
type pitem=^item;
     item=record
        data:...      { простой или определяемый пользователем тип }
        next:pitem;{ или prev:pitem; }
    end;
```

Обратите внимание на описание — это единственный разрешенный случай, когда тип используется до объявления (`item`). Очевидно, разработчики компилятора сделали исключение ввиду особой важности списковых структур.

18.6.2. Примеры программ

Перейдем к примерам. Наиболее просто реализуются действия со *стеком*, поэтому первый пример (листинг 18.3) демонстрирует использование стека. Все действия со стеком выполняются только на одном его конце, который называется *вершиной* стека. Стеки как структуры данных имеют широкое применение в системном программировании (например, при разработке компиляторов).

В частности, область памяти, в которую помещаются параметры и локальные переменные подпрограмм, имеет структуру стека *(см. разд. 10.5.1)*. Именно благодаря устройству стека возможны такие приемы программирования, как вложенные подпрограммы и рекурсия. Последняя выполняемая подпрограмма, данные которой помещены на вершину стека, всегда заканчивается первой, при этом ее данные первыми и удаляются из стека. Все данные внешней подпрограммы доступны во вложенной подпрограмме — они находятся в стеке под данными вложенной подпрограммы и поэтому могут быть удалены только после них.

В приведенном примере реализуется учебный стек, содержащий целые числа.

Листинг 18.3. Действия со стеком

```
type pitem=^item;
     item=record      { элемент стека }
        data:integer; { значение элемента }
        prev:pitem;   { адрес предыдущего элемента }
     end;
var top,p:pitem;
    n,k:integer;
procedure add(x:integer); { добавляет элемент на вершину стека }
begin
   new(p); { создаем произвольный элемент p }
   p^.data:=x; p^.prev:=top;
   top:=p; { устанавливаем p вершиной стека }
end;
procedure deltop; { удаляет узел с вершины стека }
begin
   if top<>nil then begin { если стек не пустой }
      p:=top^.prev; { запоминаем предшествующий вершине элемент }
      dispose(top); top:=p; { устанавливаем p вершиной стека }
   end;
end;
```

```
procedure writestack; { выводит стек на экран }
begin
   writeln('Содержимое стека (начиная с вершины): ');
   p:=top;
   while p<>nil do begin
      write(p^.data,' '); p:=p^.prev;
   end;
   writeln;
end;
begin     { начало программы }
   top:=nil;
   for k:=1 to 10 do add(k); { заполняем стек числами от 1 до 10 }
   writestack;
   writeln('Введите значение элемента для добавления в стек:');
   readln(n); add(n);
   writestack;
   writeln('Сколько элементов стека нужно удалить?'); readln(n);
   for k:=1 to n do deltop;
   writestack;
   readln
end.
```

Комментарий

В примере реализованы две основные операции со стеком: добавление и удаление элементов. Для решения задачи потребовалось всего два указателя типа pitem. Один из них (top) всегда указывает на вершину стека, второй (p) — это рабочий указатель, предназначенный для временного хранения адресов различных элементов. Обратите внимание на типовую процедуру вывода списка при помощи цикла while. Стандартное действие p:=p^.prev; означает переход к следующему элементу стека (для стека правильнее назвать этот элемент не следующим, а предыдущим, т. к. он был помещен в стек раньше). Поэтому элементы стека можно вывести только в порядке, обратном тому, в котором они вводились.

В следующем примере (листинг 18.4) при формировании списка указующие поля заполняются таким образом, что каждый элемент списка действительно указывает на следующий за ним элемент, т. е. на элемент, помещенный в список позже. Таким образом, при выводе списка его элементы появятся на экране именно в том порядке, в котором заполнялся список. Такой способ формирования списка используется при реализации очередей. Но в данном примере рассматривается общий случай списка и приводятся процедуры вставки и удаления элементов в произвольную позицию списка.

Листинг 18.4. Вставка и удаление элементов в произвольной позиции

```
type pitem=^item;
     item=record
        data:integer;
        next:pitem;
     end;
var head,p,p1:pitem;
    n,k,l:integer;
procedure add(x,i:integer);
var j:integer;
begin
   if (i>0) and (i<=n+1) then begin
      new(p); p^.data:=x;
      if i=1 then begin
         p^.next:=head; head:=p;
      end
      else begin
         p1:=head;
         for j:=2 to i-1 do { находим i-1-й элемент }
            p1:=p1^.next;
         p^.next:=p1^.next;
         p1^.next:=p;
      end;
      n:=n+1;
   end;
end;
procedure delitem(i:integer);
var k:integer;
begin
   if (i>=1) and (i<=n) and (head<>nil) then
      if i=1 then begin { если нужно удалить первый элемент }
         p:=head^.next; dispose(head); head:=p;
      end
      else begin
         p:=head;
         for k:=2 to i-1 do { находим i-1-й элемент }
            p:=p^.next;
         p1:=p^.next; { p1-i-й элемент }
```

```
         p^.next:=p1^.next; { p^.next ссылается на i+1-й элемент }
         dispose(p1);
      end;
end;
procedure writelist;
begin
   p1:=head;
   writeln('Содержимое списка');
   while p1<>nil do
   begin
      write(p1^.data,' '); p1:=p1^.next;
   end;
   writeln;
end;
begin
   n:=0; head:=nil;
   for k:=1 to 10 do add(k,k); { заполняем список числами от 1 до 10 }
   writequeue;
   write('Введите значение добавляемого элемента:'); readln(k);
   write('Введите позицию добавляемого элемента'); readln(l);
   add(k,l); writelist;
   write('Введите номер удаляемого элемента:'); readln(k);
   delitem(k); writelist;
   readln
end.
```

Комментарий

Проанализировав процедуры вставки и удаления элементов в произвольное место списка, можно прийти к выводу, что сама операция добавления и удаления выполняется на удивление просто — достаточно лишь переставить указатели. Гораздо больше времени уходит на то, чтобы найти элемент с заданным номером в списке. Список — это не массив, и обратиться по индексу к элементу нельзя. Остается только один вариант — двигаться по цепочке от одного элемента к другому, начиная от самого первого элемента.

Последний пример (листинг 18.5) демонстрирует применение списка для решения реальной задачи. Приведенная программа проверяет правильность расстановки скобок в текстовом файле любого размера (например, в тексте этой книги). При этом просматриваются все символы скобок: круглых, квадратных и фигурных. Учитывается, что различные скобки могут быть вложены одна в другую. Список из открывающих скобок строится по прин-

ципу стека. Это значит, что та скобка, которая была помещена в стек последней, будет извлечена из него первой, и легко проверить, соответствует ли закрывающая скобка своей открывающей. Все остальные символы файла программа игнорирует.

Листинг 18.5. Проверка правильности расстановки всех видов скобок в файле

```
type pstack=^stack;
     stack=record
        data:char;
        prev:pstack;
     end;
var f:text;
ptop,p:pstack;
correct:boolean;
c1,c2:integer;
begin
   assign(f,'skobki.txt'); reset(f);
   correct:=true;
   while (not seekeof(f)) and (correct) do
   begin
      new(p);
      read(f,p^.data); { читаем символ из файла }
      case p^.data of
         '(', '{', '[': begin { открывающая скобка }
            p^.prev:=ptop; ptop:=p;
         end;
         ')', '}', ']': { закрывающая скобка }
            if ptop=nil then correct:=false { выражение неверно }
            else begin
               case ptop^.data of
                  '(': c1:=1;
                  '{': c1:=2;
                  '[': c1:=3;
               end;
               case p^.data of
                  ')': c2:=1;
                  '}': c2:=2;
                  ']': c2:=3;
               end;
```

```
                if c1<>c2 then correct:=false { выражение неверно }
                else begin
                   p:=ptop^.prev; { извлекаем из стека последний элемент }
                   dispose(ptop);
                   ptop:=p
                end;
             end;
       end;
   end;
   if ptop<>nil then correct:=false;{ недостаточно закрывающих скобок }
   if correct then writeln('Скобки в файле расставлены верно.')
   else writeln('Скобки в файле расставлены неверно');
   readln
end.
```

18.6.3. Сравнение связанных списков и массивов

Подведем итоги. Итак, связанные списки и массивы — две основные линейные структуры в оперативной памяти, которые используются для обработки данных. Сравним их между собой, выделив главные моменты:

- организация данных в памяти в виде связанных списков обеспечивает более экономное использование памяти по сравнению с массивами;
- другим преимуществом связанных списков является удобство вставки и удаления элементов. Вспомним, что в массиве для этих целей приходилось раздвигать или сдвигать элементы *(см. разд. 13.3.8)*. В списке для удаления и вставки достаточно только поменять значения указующих полей соседних элементов;
- явным достоинством массивов является простота их использования по сравнению со списками (в этом вы, очевидно, уже убедились);
- еще более существенным преимуществом массива является высокая скорость доступа к элементу массива по его индексу. Исходя из того, что уже рассказано о списках, можно легко догадаться, что для получения доступа к последнему элементу односвязного списка необходимо последовательно обойти всю цепочку, начиная с самого первого элемента.

Из сказанного можно сделать *вывод* — при решении задач обработки данных во многих случаях выбор между массивом и списком сделать непросто. Необходимо тщательно взвесить все плюсы и минусы обоих вариантов, а далее действовать по обстоятельствам. Начинающим мы рекомендуем не

бояться указателей и динамических структур, т. к. программирование действий с ними — хороший способ развития логического мышления.

18.7. Деревья. Двоичные поисковые деревья

18.7.1. Общие сведения

Связанные списки — это *линейные* структуры данных, в которых каждый элемент, кроме первого и последнего, имеет ровно один предыдущий и ровно один следующий элемент. Деревья представляют собой *иерархические* структуры данных, в которых каждому элементу, кроме корневого, соответствует *один* родительский, но *несколько* (хотя возможно, и ни одного) дочерних элементов. Корневой элемент — это начало иерархии, ее самый первый элемент. На рис. 18.5 показаны связи между элементами списка и дерева.

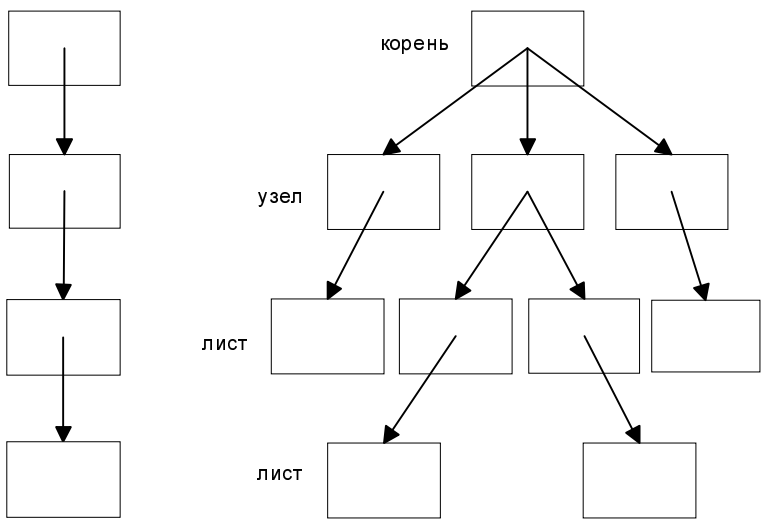


Рис. 18.5. Структура списка (слева) и дерева (справа)

В жизни иерархические структуры встречаются не менее часто, чем линейные. Допустим, в университете есть несколько факультетов, на каждом факультете — несколько специальностей, для каждой специальности имеется несколько студенческих групп и в каждой из них учится не один студент. При этом каждый студент учится только в одной группе, каждая группа принадлежит только одной специальности и т. д. Следовательно, каждого

студента можно отыскать, двигаясь от самого верхнего уровня, одним единственным путем. Этот вывод запомним.

Еще один пример — дерево каталогов любого диска, у которого существует только один корневой каталог, имеющий подчиненные (вложенные) каталоги, а те, в свою очередь, также имеют подчиненные каталоги и т. д. При этом каждый каталог имеет только один вышестоящий (родительский) каталог.

Из этих примеров, наверное, становится понятно, чем дерево отличается от списка. Перейдем к терминологии.

Элементы, составляющие дерево, иначе называются его *узлами*, а связи между элементами графически представляются в виде линий со стрелками — *ветвей*. Узел, в который не входит никакая ветвь, называется *корнем* дерева. Узлы, из которых не выходит ни одна ветвь, называются *листьями*. *Высотой* каждого узла называется количество узлов, которые надо пройти, двигаясь по стрелкам от корня до данного узла (помним, что такой путь всегда один, поэтому высота определяется однозначно). Высотой всего дерева называется максимальное значение высот узлов.

Каждый узел дерева, как и списка, состоит из части, содержащей данные, и из указателей, показывающих на следующие узлы (т. е. на ветви). Часть узла, содержащая данные, называется *ключом*.

18.7.2. Двоичные поисковые деревья. Индексы

Если каждый узел дерева имеет самое большее две ветви, такое дерево называется *двоичным*. Каждый элемент двоичного дерева, кроме листьев, имеет левого и правого *сына* (один из сыновей может отсутствовать). Сыновья данного узла, а также сыновья сыновей и т. д., называются потомками данного узла, а сам узел будет их *предком*. Пример двоичного дерева показан на рис. 18.6.

Двоичные деревья имеют большое значение в программировании. Они довольно легко описываются в виде связанной структуры, похожей на список, но не линейной, а иерархической. Элемент двоичного дерева можно описать, например, таким образом:

```
type pnode = ^node; {node в переводе с английского — узел}
     node = record
        key : integer; {key — ключ, может иметь любой тип, кроме
                        файлового}
        left, right : pnode; {левый и правый сыновья}
     end;
```

Двоичное дерево называют *упорядоченным*, если для каждого его узла, кроме листьев, выполняется такое правило: *все левые потомки* меньше своего предка (т. е. имеют меньший ключ), а *все правые потомки* его больше.

В силу такого свойства упорядоченности применение подобных деревьев для поиска исключительно удобно, поэтому их принято называть *деревьями поиска* или *поисковыми деревьями.*

Пример упорядоченного двоичного дерева приводится на рис. 18.6.

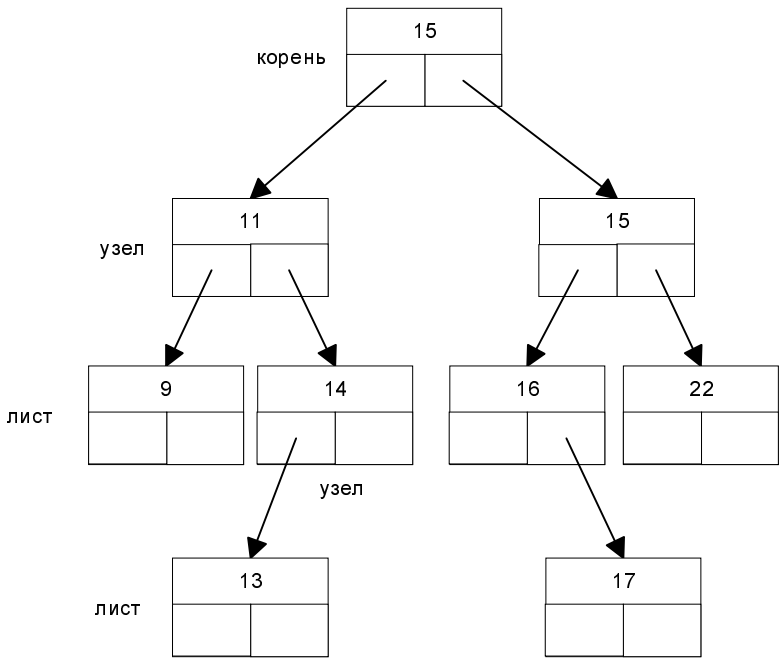


Рис. 18.6. Упорядоченное двоичное дерево

Представим себе, что данные, среди которых требуется выполнить поиск, хранятся в памяти не в виде отсортированного массива, а в виде упорядоченного двоичного дерева, подобного изображенному на рисунке. Тогда поиск любого значения нужно будет всегда начинать с корня, а затем спускаться вниз по дереву, выбирая то левого, то правого сына. Понятно, что такой поиск выполняется очень быстро, аналогично быстрому поиску в отсортированном массиве, но при этом не тратится лишнее время на вычисление индексов элементов.

Кроме того, имея двоичное поисковое дерево, нетрудно вывести его узлы в отсортированном виде. Для этого достаточно обойти все узлы в таком порядке: левый сын — узел — правый сын. Если сын — не лист, то его надо обойти в таком же порядке, и т. д. Попробуйте проделать это действие на бумаге для дерева, изображенного на рис. 18.6, и вы убедитесь, что значения выводятся в порядке возрастания. Если начать обход с правого сына, то получим значения, отсортированные по убыванию.

В силу этих обстоятельств двоичные поисковые деревья широко используются в базах данных в целях ускорения поиска и для сортировки данных. Например, пусть в базе данных хранятся такие сведения о сотрудниках фирмы: личный код, ф.и.о., телефон. Поиск как по коду, так и по ф.и.о. выполняется одинаково часто. Поэтому непонятно, по какому из полей должны быть отсортированы данные. А в реальных базах данных хранится еще большее количество полей данных, и по многим выполняется поиск. Как быть?

Все гениальное просто. Данные хранятся в файлах в неотсортированном виде, а для поиска информации создаются специальные вспомогательные структуры, называемые *индексами*.

Индексы представляют собой двоичные поисковые деревья особого вида, хранящиеся на диске. Обычно узлы дерева содержат пары: "значение поля — номер записи" в основном файле. При выполнении поиска сначала считывается в память нужное дерево-индекс, в нем выполняется быстрый поиск искомого значения, а затем по найденному номеру записи быстро находятся значения всех остальных полей основного файла.

18.7.3. Пример построения двоичного поискового дерева

Попробуем реализовать алгоритм построения самого простого двоичного поискового дерева, а также операции, которые выполняются с деревом. Поисковому дереву, которое мы сейчас построим, далеко до настоящих индексов, используемых в современных информационно-поисковых системах, но на простом примере легче понять принцип построения.

Предположим, что база данных представляет собой типизированный файл, состоящий из записей: личный код, ф.и.о., телефон, который находится в текущем каталоге под именем base.dat. Программа из листинга 18.6 сначала формирует в памяти двоичное поисковое дерево по полю "личный код", заодно проверяя данные на уникальность. Затем выполняется обход дерева с выводом данных в отсортированном виде и быстрый поиск в информации сначала по дереву, а затем в основном файле. Поскольку номер записи файла найден в дереве, мы используем процедуру `seek`, которая устанавливает

указатель типизированного файла на нужную запись. Все операции с деревом оформлены в виде рекурсивных процедур.

Листинг 18.6. Индекс на основе двоичного поискового дерева

```
type pnode=^node;
     node=record          { узел дерева }
        id:integer;       { поле, по которому создается индекс}
        number:integer;   { номер записи в базе }
        left,right:pnode; { указатели на левого и правого сына}
     end;
     people=record   { запись в файле }
        id:integer;
        surname:string[20];
        phone:string[10];
     end;
var root:pnode; { указатель на корень }
    id,n:integer;    ok:boolean;
    t:people; f:file of people;
{ проверка уникальности узла p^ и добавление его в дерево }
procedure addnode(top,p:pnode; var ok:boolean);
{ top - сначала корень дерева, а затем поддерева }
begin
   if p^.id=top^.id then ok:=false  { нарушение уникальности id }
   else
      if p^.id<top^.id then
         if top^.left=nil then top^.left:=p
         else addnode(top^.left,p,ok) { проход по левой ветви }
      else
         if top^.right=nil then top^.right:=p
         else addnode(top^.right,p,ok) { проход по правой ветви }
end;
{ создание дерева с корнем root }
procedure createindex(var ok:boolean; var root:pnode);
{ ok - признак уникальности, root - указатель на корень }
var n:integer; p:pnode;
begin
   n:=0; root:=nil; ok:=true;
   while not eof(f) and ok do begin
```

```
      read(f,t); new(p);
      n:=n+1; p^.number:=n; p^.id:=t.id;
      p^.left:=nil; p^.right:=nil; { пока p^ - это лист }
      if root=nil then root:=p  { создание корня }
      else addnode(root,p,ok);  { добавление в дерево }
   end;
end;
{ поиск номера записи в файле по личному коду человека }
function find(top:pnode; id:integer; var n:integer):boolean;
{ top - вершина поддерева, id - искомое значение,  n - номер записи,
  которую нашли }
begin
   if id<top^.id then
      if top^.left=nil then find:=false
      else find:=find(top^.left,id,n)
   else
      if id>top^.id then
         if top^.right=nil then find:=false
         else find:=find(top^.right,id,n)
      else begin
         n:=top^.number; find:=true;
      end;
end;
procedure writetree(top:pnode); { вывод в порядке возрастания }
begin
   if top^.left<>nil then writetree(top^.left);
   writeln(top^.id);
   if top^.right<>nil then writetree(top^.right);
end;
begin { начало программы }
   assign(f,'base.dat'); reset(f);
   createindex(ok,root); {формируем индекс — двоичное поисковое дерево}
   if not ok then writeln('Нарушение уникальности индекса в файле.')
   else
   begin
      writeln('Личные коды (в порядке возрастания):');
      writetree(root); { демонстрируем обход дерева }
      { демонстрируем поиск }
      writeln('Введите личный код человека'); readln(id);
```

```
      if find(root,id,n) then begin {нашли номер записи}
         seek(f,n-1); read(f,t); { остальные данные из файла }
         writeln('Фамилия: ',t.surname);
         writeln('Телефон: ',t.phone);
      end
      else writeln('Данных нет.');
   end;
end.
```

Комментарий

Сначала считывается первая запись из файла. Именно ее данные становятся корнем дерева. Далее при добавлении очередного узла программа движется по дереву, поворачивая налево или направо, в зависимости от поля id, пока не наткнется на нулевой указатель. Этот указатель привязывается к новому узлу, добавляя тем самым его в дерево. Процедура поиска узла в дереве работает аналогично. В конечном счете она натыкается либо на искомый узел, либо на нулевой указатель, если такого узла в дереве нет. Процедура writetree состоит из трех операторов. Первый из них выполняет обход по левой ветви узла top^, поскольку в ней содержатся числа (поля id), строго меньшие, чем в самом узле top^. После этого можно смело выполнять второй оператор, поскольку все меньшие числа были выведены первым оператором. В правой ветви узла top^ содержатся числа, большие, чем top^, их выводом на экран занимается третий оператор.

Остается добавить, что при таком алгоритме построения дерева в подавляющем большинстве случаев оно будет *несбалансированным*, т. е. левое и правое поддерево будут иметь разное количество узлов. Как ни странно, но самый несбалансированный вариант получится в том случае, если данные, по которым строится дерево, были уже упорядочены. Нетрудно догадаться, что в этом случае все узлы будут добавляться только в одну ветвь, т. е. вместо дерева получим список, а вместо быстрого поиска будем иметь линейный.

По этой причине для организации индексов в базах данных используются сбалансированные поисковые деревья. Алгоритм их построения значительно сложнее, чем реализованный нами в примере. Наиболее подробно эти вопросы изложены в [4].

18.8. Контрольные вопросы и задания

Вопросы

1. Почему динамические переменные называют переменными с управляемым временем жизни?

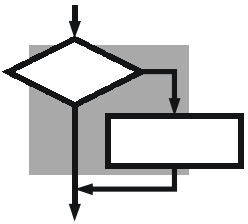
2. Что такое указатели? Какие значения они могут принимать? Какие операции возможны над указателями?
3. Чем отличаются типизированный и нетипизированный указатели?
4. Каково назначение указателя `nil`?
5. Для чего применяется и как задается операция получения указателя?
6. Что такое разыменование указателя? Каково правило разыменования?
7. Что называется кучей? В какой области памяти компьютера она располагается?
8. Зачем нужны динамически формируемые массивы?
9. В чем преимущества и недостатки списков по сравнению с массивами?
10. Какие частные случаи списков вы знаете?
11. Что представляют собой двоичные поисковые деревья?
12. Почему в базах данных для поиска используют только сбалансированные деревья?

Задания

1. Разработать программу вычисления суммы элементов массива, состоящего из n вещественных чисел. Массив должен быть размещен в памяти динамически, а значение n вводиться с клавиатуры.
2. Разработать программу перемножения двух матриц A и B размерности $n \times n$. Обе матрицы размещаются в оперативной памяти динамически, а значение n вводится по запросу с клавиатуры.
3. Дан текстовый файл. Разработать программу для просмотра файла на экране с возможностью пролистывания страниц в прямом и обратном порядке. Используйте двунаправленный список.
4. Разработать программу, определяющую симметричность произвольного текста любой длины. Симметричным назовем текст, который одинаково читается и справа налево, и слева направо (примеры симметричного текста — "потоп", "123321"). Эту задачу рекомендуется решать с помощью двух стеков. В первый стек следует поместить весь текст, затем во второй стек перенести его половину так, чтобы последний символ текста находился на дне стека. Далее путем поэлементного сравнения этих стеков получить ответ на вопрос о симметричности текста.
5. Разработать программу, которая из последовательности чисел, вводимых с клавиатуры, формирует двунаправленный список, записывая в него числа так, чтобы в результате они оказались упорядоченными. Распечатать список в прямом и обратном порядке (при этом числа должны быть выведены сначала по возрастанию, а затем по убыванию).

6. Ту же самую задачу решить с использованием двоичного поискового дерева.
7. Разработать программу слияния двух списков, содержащих возрастающую последовательность целых положительных чисел, в третий список так, чтобы его элементы располагались также в порядке возрастания.
8. Разработать программу, которая формирует частотный словарь для заданного текстового файла, подсчитывая число вхождений каждого из слов в файл. Слово — последовательность символов, за которой следует или пробел, или символ перевода строки. Используйте двоичное поисковое дерево.

Глава 19



Введение в объектно-ориентированное программирование

19.1. Абстрактные типы данных

19.1.1. Понятие АТД и структур данных

Массивы, связанные списки, двоичные поисковые деревья — все это примеры *структур данных*. Определим структуру данных как совокупность данных, для которой можно выполнить описание средствами конкретного языка программирования на основе его стандартных типов данных. Например, на языке Pascal описание структуры данных представляет собой группу описаний типов, переменных и, возможно, констант, которые объединены по смыслу и в совокупности описывают, допустим, связанный список или массив.

Структуры данных используются на этапе реализации (кодирования) алгоритма, когда он воплощается в программу на конкретном языке. Чтобы подчеркнуть это обстоятельство, иногда применяют термин "физические структуры данных", поскольку этап реализации также иногда называют этапом *физической реализации*.

Однако на начальных этапах разработки алгоритма детали физической реализации еще не ясны, да и не важны. Тем не менее, в общих чертах необходимо представлять себе способ организации данных. Для этих целей вводится понятие *абстрактного типа данных* (сокращенно АТД).

Абстрактный тип данных определяется как *математическая модель* некоторой совокупности данных, в рамках которой определены допустимые операции над ними. Для разработки алгоритма этого вполне достаточно, а при реализации алгоритма вместо АТД будет подобрана наиболее подходящая структура данных, и каждая операция из допустимого набора будет реализована в виде подпрограммы (рис. 19.1).

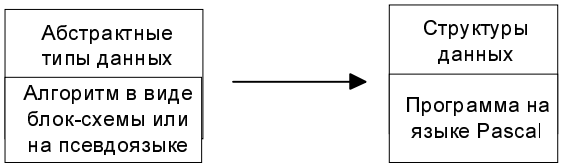


Рис. 19.1. Этапы разработки программы

Вообще-то понятие абстрактного типа данных неявно вводилось еще в предыдущей главе без упоминания этого термина. Вспомните рассуждение о том, чем линейные структуры отличаются от иерархических, или сравнение двух разных способов реализации линейных структур (массивы и связанные списки). Конечно, и линейные, и иерархические структуры — это и есть самые настоящие АТД. Их характерные особенности понятны, математическую модель при желании можно определить достаточно строго, так же строго можно определить и допустимый для них набор операций.

Обычно при обозначении этих АТД для краткости используют термины "списки" и "деревья". При этом не будем путать их с такими физическими структурами, как связанные списки или двоичные поисковые деревья. Мы уже понимаем, что и список, и дерево можно реализовать не только на основе указателей, но и на основе массивов, причем довольно разнообразными способами. Однако на начальных этапах разработки алгоритма это неважно. Допустим, для реализации линейного поиска мы всегда будем использовать список (неважно, как он реализован), а для реализации быстрого поиска дерево более предпочтительно.

19.1.2. Стандартные АТД

Любой программист может определить свои собственные АТД при разработке алгоритма, а затем их реализовать. Однако при решении многих задач не стоит "изобретать велосипед", поскольку имеется набор стандартных АТД, хорошо проработанных и даже реализованных в различных библиотеках.

Этот набор включает следующие АТД:

- списки или линейные структуры (разновидность — стеки, очереди);
- деревья или иерархические структуры (имеется много специальных видов);
- графы или сетевые структуры (разновидность — ориентированный граф);
- множества (имеется в виду математическое понятие множества, а не тип `set` языка Pascal).

Дадим формальное определение каждого типа АТД, а также приведем соответствующие им перечни операций и способов реализации. Более подробные сведения можно найти в [4], разбор многих типовых алгоритмов имеется в [16, 21, 22].

АТД "Список"

Список — последовательность элементов определенного типа a_1, a_2, ..., a_n, где $n \geq 0$. Количество элементов в списке называется *длиной* списка. Каждый элемент списка a_i имеет *позицию* i, $i = 1, 2, ..., n$. Элемент a_i *предшествует* элементу a_{i+1} для $i = 1,2, ..., n-1$ и a_i *следует* за a_{i-1} для $i = 2,3, ..., n$.

Операции над АТД "Список":

- вставка элемента в заданную позицию, при этом все последующие элементы сдвигаются вправо. Если в списке нет заданной позиции, то результат выполнения этого оператора не определен;
- поиск элемента в списке, при этом возвращается позиция заданного значения в списке. Если в списке искомое значение встречается несколько раз, то возвращается позиция первого от начала списка элемента, имеющего заданное значение (возможен вариант — задать номер вхождения). Если значения нет в списке, то возвращается обычно ноль (или `nil`);
- определение элемента, который стоит в заданной позиции. Результат не определен, если список пуст или нет заданной позиции;
- удаление элемента в заданной позиции. Результат не определен, если в списке нет заданной позиции или список пуст;
- получение следующего и предыдущего элемента для заданной позиции. Результат не определен в случае, если предыдущий элемент определяется для первого, а следующий — для последнего элемента, а также если в списке нет заданной позиции или он пуст;
- создание пустого списка или удаление всех элементов заданного списка;
- получение первого и последнего элемента в списке. Если список пустой, то возвращается обычно ноль (или `nil`);
- вывод всех элементов в порядке их расположения в списке.

Большинство этих операций было реализовано нами при изучении тем "Массивы" и "Связанные списки", а оставшиеся нетрудно реализовать самостоятельно. Еще раз подчеркнем, что АТД "Список" — это универсальное понятие, которое соединяет в себе *общие характерные черты массивов и связанных списков.*

АТД "Дерево"

Дерево — это совокупность элементов, называемых *узлами* (один из которых определен как *корень*), и отношений ("родительских"), образующих иерархическую структуру узлов. Формально дерево можно рекуррентно определить следующим образом.

- Один узел является деревом. Этот же узел также является корнем этого дерева.

- Пусть n — это узел, а T_1, T_2, ..., T_k — деревья с корнями n_1, n_2, ..., n_k соответственно. Можно построить новое дерево, сделав n родителем узлов n_1, n_2, ..., n_k . В этом дереве n будет корнем, а T_1, T_2, ..., T_k — *поддеревьями* этого корня. Узлы n_1, n_2, ..., n_k называются *сыновьями* узла n.

Основные операции над АТД "Дерево":

- получение родителя (`parent`) узла n в дереве T. Если n является корнем, который не имеет родителя, то в этом случае возвращается `nil`. Здесь `nil` указывает на то, что произошел выход за пределы дерева;
- получение самого левого сына узла n в дереве T. Если n является листом (и поэтому не имеет сына), то возвращается `nil`;
- получение самого правого брата узла n в дереве T или значение `nil`, если такового не существует. Для этого находится родитель узла n и все его сыновья, затем среди этих сыновей находится узел, расположенный непосредственно справа от узла n;
- получение узла, являющегося корнем дерева. Если T — пустое дерево, то возвращается `nil`;
- создание нового дерева или удаление всех узлов существующего дерева;
- обход дерева различными способами с выводом ключей узлов.

Основные способы реализации деревьев — массивы и связанные структуры. Самый простой способ — это представление дерева в виде одного массива, в котором хранятся номера родителей каждого узла (возможно: пары "номер — значение"). Еще один простой способ — определение массива, состоящего из указателей на списки сыновей каждого узла. Специальные виды деревьев имеют свою специфическую реализацию (например, реализация упорядоченного двоичного дерева подробно рассматривалась в *разд. 18.7.2*).

АТД "Граф"

Графы иначе называют *сетевыми структурами*. Их элементы связаны друг с другом произвольным образом, т. е. у каждого элемента может быть сколько угодно предыдущих и следующих элементов. Поэтому можно считать граф самой универсальной из всех возможных структур данных, а деревья и списки — частным случаем графов.

Наиболее распространенным видом графов являются ориентированные графы. *Ориентированный граф (орграф)* состоит из множества вершин и множества дуг. Вершины также называются *узлами,* а дуги — *ориентированными ребрами*.

Пример орграфа приведен на рис. 19.2.

Вершины орграфа можно использовать для представления объектов, а дуги — для задания отношений между объектами. Предположим, узлы могут обозначать населенные пункты, а дуги — пути между ними (классическая задача на графы).

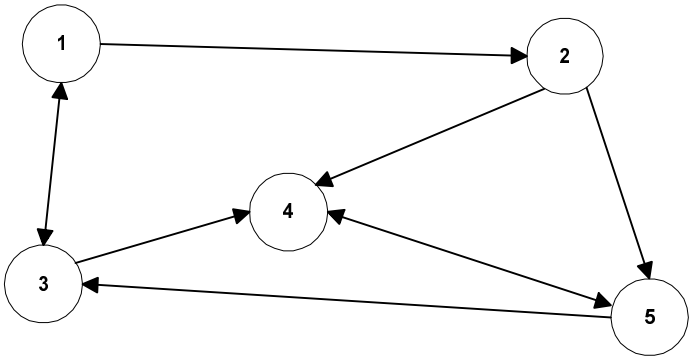


Рис. 19.2. Пример ориентированного графа

Путем в орграфе называется последовательность вершин $v_1, v_2, ..., v_n$, для которой существуют дуги $v_1 \rightarrow v_2$, ..., $v_2 \rightarrow v_3$, ..., $v_{n-1} \rightarrow v_n$. Этот путь начинается в вершине v_1 и, проходя через вершины v_2, v_3, ..., v_{n-1} заканчивается в вершине v_n. *Длина пути* — количество дуг, составляющих путь (в данном случае длина пути равна $n - 1$). На рисунке последовательность вершин 1, 2, 4 образует путь длины 2 от вершины 1 к вершине 4.

Очень часто каждой дуге графа ставят в соответствие какое-либо значение, называемое *весом*. Например, если узлы графа обозначают населенные пункты, то каждой дуге можно поставить в соответствие расстояние между пунктами.

Типовой задачей на графы является задача нахождения кратчайшего пути от одной вершины к другой. Имеется много типовых алгоритмов решения задач на графы, однако их рассмотрение выходит за рамки начального курса программирования.

Способы реализации графов:

- с помощью матрицы смежности;
- с помощью векторов смежности.

В *матрице смежности* элемент *i*-й строки и *j*-го столбца равен 1, если существует дуга из *i*-й вершины в *j*-ю. В противном случае он равен 0.

Для приведенного выше примера (рис. 19.2) матрица смежности будет выглядеть следующим образом:

0 1 1 0 0

0 0 0 1 1

1 0 0 1 0

0 0 0 0 1

0 0 1 1 0

При использовании *векторов смежности* для каждой вершины создается список смежных с ней вершин. В программе это выглядит как массив указателей на соответствующие вершинам *списки смежности*. В приведенном выше примере элементы вектора смежности будут указывать на следующие списки:

1. 2, 3
2. 4, 5
3. 1, 4
4. 5
5. 3, 4

Данный подход значительно экономит память и ускоряет выполнение программы, когда речь идет о графе с небольшим количеством дуг.

АТД " Множество"

Множество обычно изображается в виде последовательности его элементов, заключенной в фигурные скобки, например: {1, 4}. Порядок, в котором записаны элементы множества, несуществен, поэтому {4, 1} и {1, 4} — одно и то же множество.

Множество не является списком, поскольку его элементы не имеют позиции (порядкового номера). В множестве отсутствуют повторяющиеся элементы (так, {1, 4, 1} не является множеством). В силу этих обстоятельств, а также ввиду широкого применения множеств в задачах обработки данных, их выделяют в отдельный абстрактный тип.

Над данными АТД "Множество" определены следующие операции:

- *объединением* множеств *A* и *B* (обозначается $A \cup B$) называется множество, состоящее из элементов, принадлежащих хотя бы одному из множеств *A* и *B*;
- *пересечением* множеств *A* и *B* (обозначается $A \cap B$) называется множество, состоящее только из тех элементов, которые принадлежат и множеству *A*, и множеству *B*;
- *разностью* множеств *A* и *B* (обозначается $A \backslash B$) называется множество, состоящее только из тех элементов множества *A*, которые не принадлежат множеству *B*;
- *определение принадлежности элемента множеству* — наиболее часто используемая операция. Результатом является только логическое значение, поскольку номер позиции элемента в множестве не определен.

Представление множества:

Множество обычно реализуется с помощью упорядоченного массива, связанного списка или двоичного поискового дерева. Тип `set` языка Pascal используется редко ввиду крайней ограниченности размеров.

19.1.3. Принцип абстрагирования

Мы вплотную подошли к очень важному принципу современного программирования — принципу абстрагирования (или абстракции). Применение АТД позволяет отвлечься от всех несущественных деталей и сосредоточить внимание на главных особенностях используемых структур данных. Точно так же использование псевдоязыка или блок-схем алгоритмов позволяет отвлечься от конкретных операторов языка и сосредоточить внимание на особенностях алгоритма. Таким образом, принцип абстрагирования имеет две стороны — абстракция процессов (алгоритмы) и абстракция данных (АТД).

Этот принцип является основой идеологии объектно-ориентированного программирования (ООП).

19.2. Введение в ООП

19.2.1. Принцип сокрытия деталей. Идеи ООП

Перейдем от этапа разработки алгоритма к этапу его реализации. Как лучше всего реализовать АТД на языке Pascal?

Читатель, освоивший основы структурного и модульного программирования, быстро предложит следующую последовательность действий.

Шаг 1. Выбираем подходящую структуру данных и записываем ее определение на языке Pascal (константы, типы, переменные).

Шаг 2. Реализуем каждый оператор АТД в виде отдельной подпрограммы, при необходимости вводим служебные подпрограммы.

Шаг 3. Оформляем все это в виде библиотечного модуля *(см. разд. 12.3)*. В интерфейсную часть помещаем минимально необходимый набор определений, все детали реализации скрываем в исполняемой части модуля.

Это очень хорошее решение проблемы, т. к. оно позволяет распространить принцип абстрагирования и на этап реализации алгоритма. При этом абстрагирование выражается в *сокрытии всех несущественных деталей разработки*. При наличии осмысленных имен можно будет пользоваться данным модулем, совсем не думая о его реализации. В самом деле, используя стандартные процедуры и функции из модуля `System`, разве думаем мы о том, как они реализованы? Разумеется, нет, если они работают корректно, а они так и работают в подавляющем большинстве случаев.

Есть и другое очень важное преимущество *сокрытия деталей разработки — внутренний код подпрограмм можно изменять независимо от всех остальных программ и модулей*. Для сохранения межмодульных связей достаточно оставить неизменными только заголовки. Так обычно и поступают на практике.

Допустим, если у разработчиков системы программирования возникнет желание в корне изменить алгоритм какой-то из стандартных подпрограмм, то они даже не обязаны сообщать пользователям-программистам о внесенных изменениях (зачем раскрывать свои профессиональные секреты?) Программисты заметят лишь, что подпрограмма стала работать лучше (быстрее и/или надежнее), и этого вполне достаточно.

Дальнейшим этапом развития структурного и модульного программирования стало появление нового подхода к проектированию программных продуктов, который получил название объектно-ориентированного программирования. Идеи его не так уж сложны и имеют своей целью еще большее продвижение принципа абстрагирования (сокрытия деталей) в разработку программного обеспечения.

Перечислим эти идеи.

- При структурном подходе структуры данных и подпрограммы их обработки разрабатываются отдельно. В ООП данные и подпрограммы соединены вместе и в совокупности образуют новый тип данных, который называется *классом*. Данные класса надежно защищены, поскольку с ними можно работать только через подпрограммы класса. Если использовать аналогию с модулем, то это аналогично определению данных в исполняемой части модуля, закрытой для пользователя. Этот принцип называется *инкапсуляцией*.
- На основе одного класса можно создавать другие, производные, классы, не переписывая в них код базового класса, т. к. все элементы базового класса (данные и подпрограммы) достаются производному классу "по наследству". Разумеется, в производный класс можно добавить новые элементы, для этого он и создается. Такой принцип называется *наследованием*.
- В производный класс можно не только добавить новые элементы, но и заменить в нем коды (тела) каких-либо подпрограмм базового класса. Такой принцип называется *полиморфизмом*, поскольку в этом случае подпрограмма с одним и тем же заголовком в разных классах будет реализована по-разному (вспомним, что полиморфизм — это "много форм"). Вот задача для компилятора! А он и не будет ее решать, отложив проблему выбора нужной подпрограммы до этапа выполнения, когда уже будет точно известно, какая именно подпрограмма требуется. Это явление называют *поздним связыванием*.

Вот три основные идеи объектно-ориентированного программирования. Как обычно, эти идеи по сути не так уж и сложны, но научиться реализовывать их средствами языка значительно трудней. Для того чтобы понять материал данной главы, от читателя требуется полное понимание всего материала *гл. 10*, посвященной подпрограммам, а также *гл. 12 — "Модули"*. Рекомендуем перечитать их еще раз.

Сначала разберемся, как связаны между собой ООП и язык Pascal.

19.2.2. ООП и Pascal

Язык Pascal был создан задолго до массового распространения технологии ООП. Поэтому первые элементы ООП появились только в Turbo Pascal версии 5.5. Очевидно, в то время терминология ООП еще не устоялась, поэтому новый тип данных получил имя `object`, а не `class`. К сожалению, даже в Turbo Pascal 7.0 эта терминологическая неточность не исправлена, а поддержка ООП считается неполной. При разработке Delphi (Object Pascal) введены все элементы ООП, при этом там появился новый тип с именем `class`, а старый тип `object` оставлен для совместимости с Turbo Pascal.

Имеет смысл рассмотреть введение в ООП, используя язык Delphi, который признан специалистами как полноценный язык объектно-ориентированного программирования. В многочисленных учебниках по Delphi обычно подробно разбираются способы использования стандартных компонентов из библиотеки Delphi, при этом программирование и язык как таковой отходят на второй план. Наша книга посвящена вопросам программирования, поэтому для нас важно сосредоточить внимание на использовании конструкций языка для реализации идей ООП.

19.2.3. Определение класса

Описание класса выполняется как описание любого другого типа данных. В самом общем виде оно выглядит так:

```
type ИмяКласса=class
описание данных класса
заголовки подпрограмм
end;
```

Данные класса иначе называются *атрибутами* или *полями* (по аналогии с записями), а подпрограммы — *методами* класса.

Обратите внимание на имена самого класса и его элементов — они должны быть очень понятными и осмысленными. Имя класса в Delphi принято начинать с буквы `t` (от слова *type*, ведь класс — это тип).

Поскольку при описании класса, как в интерфейсной части модуля, записываются только заголовки подпрограмм, то далее каждый метод должен быть реализован. Полное описание каждого из методов выполняется в разделе описания процедур и функций:

```
procedure ИмяКласса.Имя процедуры (ФормальныеПараметры);
begin
тело метода — его код
end;
```

Аналогично можно описать метод-функцию.

Следует знать:

- в заголовке методов класса используется *составное имя*. Таким способом указывается, что это не обычная подпрограмма, а метод конкретного класса;
- в теле любого метода можно использовать поля класса так, как используются *глобальные переменные* в подпрограммах. Разумеется, при этом требуется соблюдать предельную осторожность;
- в качестве параметров в методы передаются какие-то дополнительные данные, причем обычно используются параметры-значения или параметры-константы.

Описание переменных типа класса выполняется, как обычно, в разделе описания переменных:

```
Var СписокПеременных: ИмяКласса;
```

Переменные типа класса иначе называются *объектами*. От этого слова и берет название термин "объектно-ориентированное программирование" (кстати, авторам встречалось и другое название — "программирование, ориентированное на классы").

Иными словами, объекты — это конкретные экземпляры заданного класса.

Вы, наверное, обратили внимание, что описание класса похоже на описание записи (только в записях нет методов, и вместо слова `class` используется `record`). Обращение к элементам класса также выполняется аналогично обращению к полям записи при помощи составного имени:

```
ИмяОбъекта.ИмяПоля или ИмяОбъекта.ИмяМетода(Фактические параметры);
```

Не перепутайте — при реализации методов мы используем в качестве первой части имени *имя класса*, а при обращении к методам в большинстве случаев — *имя объекта*.

К сожалению, при начальном изучении ООП в Delphi всегда существует опасность что-нибудь перепутать. Из приведенного выше ясного и понятного правила имеются два исключения.

Во-первых, обращение к полю при помощи составного имени, скорее всего, вызовет ошибку, поскольку, в соответствии с принципом инкапсуляции, принято скрывать поля и работать с ними только через методы. Некоторые методы тоже могут быть скрытыми.

Во-вторых, при вызове отдельных методов в качестве первой части составного имени указывается имя класса.

Разберемся с этим подробнее.

19.2.4. Область видимости элементов класса

Напомним, что область видимости переменной — это совокупность программных блоков, в которых можно пользоваться этой переменной. Это определение полностью применимо к полям. Аналогично, *видимостью метода* назовем совокупность программных блоков, в которых его можно использовать.

При описании класса явно указывается видимость его элементов. Для этой цели используются специальные ключевые слова языка. Эти слова имеют одинаковый смысл во всех языках программирования, поддерживающих ООП:

- `public` — так обозначаются общедоступные (открытые, публичные) элементы, которыми можно пользоваться везде: при реализации методов класса и в программе, использующей данный класс;
- `private` — скрытые (закрытые, приватные) элементы, которыми можно пользоваться только при реализации методов данного класса, в программе же, использующей данный класс, к ним нельзя обратиться при помощи составного имени;
- `protected` — защищенные элементы. Это некоторое расширение области `private` специально для поддержки принципа наследования. Защищенные элементы можно использовать *при реализации методов не только данного класса, но всех его производных классов*. Область видимости `protected` используется для сокрытия элементов даже чаще, чем `private`. Если на момент создания класса не планируется создавать на его основе производные классы, все равно неплохо оставить такую возможность на будущее;
- `published` — опубликованные элементы, которые поддерживаются в некоторых системах визуальной разработки программ — таких, например, как Delphi. Тема визуальной разработки программ выходит за рамки книги. Отметим лишь, что в Delphi элементы класса по умолчанию как раз являются опубликованными, поэтому все другие области видимости нужно указывать явно.

19.2.5. Пример описания класса

Пора пояснить сказанное на примере. Создадим несложный класс для работы с отрезком времени, содержащим часы, минуты и секунды. Можно считать этот класс реализацией для абстрактного типа данных "отрезок времени", в рамках которого мы определим такие операции:

- заполнение заданными значениями;
- вывод на экран;
- умножение или деление на целую величину;
- сложение или вычитание двух отрезков времени.

Разумеется, придуманный нами АТД не входит в список стандартных, поскольку он недостаточно универсален, да и реализация его тривиальна, однако для первого примера он вполне годится. Как все настоящие АТД, он имеет не один, а несколько возможных способов физической реализации. Например, можно хранить в памяти либо три поля (часы, минуты, секунды), либо всего одно (общее количество секунд). Второй вариант, пожалуй, удобнее, но для уяснения различных нюансов ООП нам лучше подходит первый.

Итак, разрабатываемый класс будет иметь три поля, данные в которых надо оберегать, поскольку значения минут и секунд должны находиться в интервале [0,59] и появление других значений недопустимо. На случай возможного создания производных классов будем использовать область видимости `protected`. В этой же области видимости опишем две служебных подпрограммы, которые помогут нам легко реализовать нужные операции — функцию для вычисления количества секунд в отрезке времени и процедуру, которая по заданному количеству секунд заполняет поля.

Итак, определяем тип:

```
type ttime=class
protected
hour, min, sec: byte;              {защищенные поля}
function total_sec:integer;        {возвращает полное число секунд}
procedure timefromsec(tsec:integer);   {переводит число секунд в часы,
                    минуты и секунды и присваивает эти значения полям}
public
procedure settime(h,m,s:byte);  {присваивает заданные значения полям,
          автоматически исправляя неверные значения минут и секунд}
procedure print;{выводит поля на экран}
procedure plus(first,second:ttime);{сложение first и second}
procedure minus(first,second:ttime);{ вычитание second из first }
procedure mult(first:ttime;second:byte){ умножение first на second };
procedure divide(first:ttime;second:byte){ деление first на second };
end;
{реализация всех методов — все просто, поэтому без комментариев}
function ttime.total_sec:integer;
begin
   total_sec:=hour*3600+min*60+sec;
end;
procedure ttime.timefromsec(tsec:integer);
begin
   hour:=tsec div 3600;
```

```
   min:=tsec mod 3600 div 60;
   sec:=tsec mod 3600 mod 60;
end;
procedure ttime.settime(h,m,s:byte);
begin
        hour:=h; min:=m; sec:=s;
        if sec>59 then begin
           min:=min+sec div 60; sec:=sec mod 60;
        end;
        if min>59 then begin
           hour:=hour+min div 60; min:=min mod 60;
        end;
end;
procedure ttime.print;
begin
        writeln(hour,':',min, ':',sec);
end;
procedure  ttime.plus(first,second:ttime);
begin
timefromsec(first.total_sec+second.total_sec);
end;
procedure  ttime.minus(first,second:ttime);
begin
if first.total_sec>second.total_sec then
timefromsec(first.total_sec-second.total_sec);
end;
procedure  ttime.mult(first:ttime;second:byte);
begin
timefromsec(first.total_sec*second);
end;
procedure  ttime.divide(first:ttime;second:byte);
begin
if second<>0 then
timefromsec(first.total_sec div second);
end;
```

Класс определен, и теперь его можно использовать. Но прежде чем продолжить разработку программы, придется сделать еще одно теоретическое отступление.

19.2.6. Создание и уничтожение объектов. Конструкторы и деструкторы

Особенность Delphi по сравнению с Turbo Pascal состоит в том, что объекты здесь можно создавать только динамически (т. е. по ходу выполнения программы захватывать для них память в куче). Но мы совсем недавно уже приводили простое описание:

```
Var СписокПеременных: ИмяКласса;
```

Разве при этом не выделяется память под поля объекта? Нет. Но ведь любое описание служит для выделения памяти! Совершенно верно, и в данном случае память выделяется *под указатель на объект* (всего 4 байта), хотя символ ^ не используется ни при описании объектов, ни в составных именах. С подобным "обманом" мы уже столкнулись один раз при изучении открытых массивов и открытых строк *(см. разд. 18.4)*.

Подобный способ описания динамически создаваемых объектов становится популярным — достаточно сказать, что в таких языках, как Java и C#, объекты также создаются только динамически, но работа с ними выполняется без использования указателей и операции разыменования. Будем к этому привыкать.

Следует знать:

- для выделения памяти под объект и инициализации полей объекта используется специальный метод, называемый *конструктором*;
- для удаления объекта из памяти используется другой метод, называемый *деструктором*.

Конструктор и деструктор — это особые методы, поэтому для их описания используются специальные ключевые слова `constructor` и `destructor`.

Например, для нашего класса можно было бы написать такой конструктор с тремя параметрами:

```
constructor ttime.create(h,m,s:byte);
begin
     hour:=h; min:=m; sec:=s;
     {далее выполнить автоматическую корректировку,
      как в методе settime}
end;
```

Обратите внимание — никаких специальных процедур выделения памяти (`new` или `getmem`) в конструктор помещать не надо, т. к. при вызове конструктора память под поля объекта выделяется автоматически.

Похоже, придется доработать описание класса, добавив к методам конструктор. Однако мы можем не делать этого, если выясним еще одну особенность языка Delphi.

Обратите внимание — все классы, создаваемые в Delphi, являются производными от стандартного класса `tobject`. А это значит, что, кроме своих собственных методов, они унаследовали все методы этого класса, в том числе конструктор с именем `create` без параметров. Поэтому можно не писать свой конструктор для каждого класса, а воспользоваться стандартным конструктором, при вызове которого все числовые поля заполнятся нулевыми значениями, а строковые — пустыми строками. В большинстве случаев этого достаточно.

Если все-таки вам захочется использовать собственный конструктор с параметрами, можно добавить его в свой класс так, чтобы не испортить стандартный конструктор `create`:

```
constructor create(h,m,s:byte); overload;
```

Ключевое слово `overload` — это указание компилятору о том, что в классе теперь есть два конструктора с одним и тем же именем и что при создании объектов нужно будет выбирать один из них. Такое явление называют *перегрузкой* подпрограмм, и его можно использовать не только применительно к конструктору.

Осталось выяснить, как обратиться к конструктору для создания объекта. Допустим, мы описали переменную `x` типа `ttime`. Может быть, написать так: `x.create;` или `x.create(1,10,30)`? Но ведь объекта `x` еще нет, только выделена память под указатель на него. Поэтому применяется другая конструкция:

```
x:=ttime.create;
```

или

```
x:=ttime.create(1,10,30);
```

Аналогичная история и с деструктором. В классе `tobject` имеется стандартный деструктор с именем `destroy`, а также процедура освобождения памяти `free`. Удобнее применять процедуру `free`, т. к. она перед вызовом проверяет существование объекта.

Теперь можно закончить программу. Для получения работоспособного приложения необходимо соединить приведенный ниже текст с описанием класса `ttime`, в которое добавлен конструктор с параметрами, причем не забыто волшебное слово `overload`.

```
var x,y,z:ttime;
begin
   x:=ttime.create(0,60,90); {используется наш конструктор с параметрами}
   y:=ttime.create; z:=ttime.create;            {стандартный конструктор}
   x.print; y.print;                          {проверяем, как заполнены поля}
   y.settime(0,30,30);y.print;          {теперь проверяем метод settime}
   z.plus(x,y); z.print;                          {проверяем все операции}
   z.minus(x,y); z.print;
```

```
   z.mult(x,5); z.print;
   z.divide(x,3); z.print;
   x.free; y.free; z.free; readln;                    {освобождаем память}
end.
```

Указатель *self*

Разберемся подробнее, как выделяется память при создании объектов. Память под поля, конечно, выделятся обычным образом в соответствии с их типом, а коды методов хранятся в памяти в одном экземпляре, т. к. они одинаковы для всех объектов.

Все это вполне логично, и возникает только один вопрос — как передается в методы информация о том, какой объект их вызвал? Если вернуться к тексту только что разработанной нами программы, становится понятно, что `x.print` выведет поля объекта `x`, а `y.print` — поля объекта `y`.

Ответ на поставленный вопрос очень прост — в каждый метод в качестве дополнительного параметра передается указатель на вызывающий его объект. Этот указатель в Delphi обозначается ключевым словом `self`. В описании методов его не нужно записывать специально, т. к. он добавляется к любому методу автоматически, но им можно пользоваться в теле любого метода. Например: `self.sec:=0;` равносильно `sec:=0;`.

19.2.7. Механизм наследования

С помощью механизма *наследования* (`inheritance`) можно определять одни классы на основании других, уже определенных. *Производный класс* (дочерний класс, подкласс) способен унаследовать от своего *базового класса* (родительского класса, суперкласса) все его методы и данные и иметь при этом свои специфические элементы.

Например, если реализовать группу классов для работы с файлами на языке Pascal, то можно предложить, например, такую схему наследования, какая представлена на рис. 19.3. Понятно, что суперкласс "Файл" при этом будет содержать только самые общие методы для работы с файлами, а все операции, специфичные для конкретных видов файлов, будут реализованы в наследниках.

Мы не будем доводить данный пример до стадии реализации, поскольку стандартные средства для работы с файлами в основном устраивают программистов. Самое главное для нас — понять, что принцип наследования позволяет выстроить стройную систему взаимосвязанных классов, отделяя общие черты, присущие целой группе классов, от специфических особенностей каждого конкретного класса.

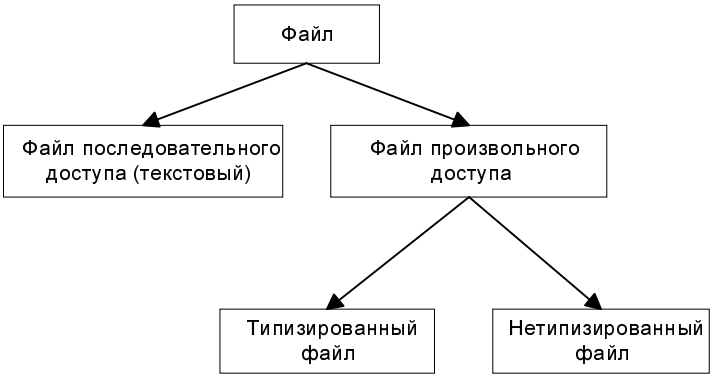


Рис. 19.3. Возможная иерархия классов для работы с файлами

Рассмотрим теперь, какие же средства языка Delphi поддерживают аппарат наследования. В минимальном варианте для его использования достаточно знать одну конструкцию:

```
type ИмяПроизводногоКласса=class(ИмяБазовогоКласса)
{Определение только новых полей и методов(!)}
end;
```

Если в нашем примере с отрезком времени в определении класса изменить `class` на `class(tobject)`, программа останется работоспособной. Дело в том, что базовый класс `tobject` подразумевается по умолчанию. Все другие классы при создании наследников требуется указывать явно.

19.2.8. Переопределение методов базового класса в потомках

Часто требуется не только добавлять новые элементы в производные классы, но и переопределять методы базового класса, которые в производном классе должны быть реализованы по-своему. В Delphi, как и во многих языках программирования, это можно сделать двумя способами:

- замещением;
- перекрытием.

Для *замещения метода* достаточно просто объявить одноименный метод в производном классе. При этом *на этапе компиляции* происходит анализ того, какой объект вызывает данный метод, и в зависимости от объекта подставляется нужный код. Такое действие называется *ранним связыванием*.

Для *перекрытия метода* необходимо выполнить два действия:

- в *базовом классе* к описанию данного метода должно быть добавлено слово `virtual`, т. е. метод в базовом классе должен быть виртуальным.

Например:

```
Procedure ИмяПроцедуры(ФормальныеПараметры); virtual;
```

- при описании одноименного метода *в производном классе* к нему добавляется слово `override` *(перекрытый)*, например:

```
Procedure ИмяПроцедуры(ФормальныеПараметры); override;
```

В этом случае вопрос о вызове того или иного метода не решается на этапе компиляции, а откладывается до этапа выполнения. Такое действие называют *поздним* связыванием, а принцип, использующийся при перекрытии методов, является принципом *полиморфизма*. Разумеется, при выполнении программы коды всех полиморфных методов находятся в оперативной памяти. Специальная *таблица виртуальных методов* хранит указатели на каждый из них.

При изучении этой темы обычно задают вопрос — а зачем вообще переопределять методы, не проще ли в каждом производном классе создать новый метод со своим уникальным именем? Приведем два аргумента в пользу переопределения.

Во-первых, не стоит вводить несколько новых имен там, где можно обойтись одним.

Во-вторых, использование одного и того же имени позволяет разрабатывать универсальные подпрограммы, которые будут правильно работать с объектами различных классов. Это позволит существенно сократить размер программного кода и повысить его надежность.

Для поддержки только первого из аргументов достаточно простого замещения, для поддержки сразу обоих необходимо использовать перекрытие (полиморфизм).

Обратите внимание — в любом случае в теле метода производного класса можно вызывать одноименный метод базового класса. В Delphi для этого имеется специальное ключевое слово `inherited`.

Как обычно, поясним сказанное на примере.

Приведем пример наследования с замещением и перекрытием методов.

Создадим производный класс на базе разработанного класса `ttime`, который будет содержать не только количество часов, минут и секунд, но еще и количество суток (допустим, 49 часов — это двое суток и один час). Назовем класс `tlongtime`.

Это не самый удачный пример для иллюстрации идеи наследования, поскольку обычно в производные классы добавляют новые специфические методы, в нашем же случае будет добавлено только новое поле. Зато идеи замещения и перекрытия методов демонстрируются достаточно наглядно.

Первоначально может показаться, что в новом классе придется переопределять все методы базового класса, поскольку добавилось лишнее поле, которое требует обработки. Но вскоре мы догадываемся, что все процедуры для выполнения операций можно оставить без изменения, если *перекрыть* всего две служебные подпрограммы, которые переводят отрезок в секунды и обратно. Вот где полиморфизм сослужит хорошую службу! Теперь процедуры выполнения операций с отрезком времени станут настолько универсальными, что их смогут одновременно использовать объекты двух разных классов.

Методы для вывода и задания значений времени не имеет смысла перекрывать, поскольку нами не планируется написание какого-то универсального кода на их основе. Достаточно просто заместить эти методы.

Сказано — сделано. Описание нового класса можно поместить в тот же файл, что и описание базового класса. Еще лучше — оформить определение двух классов в виде отдельного модуля или каждый класс оформить в виде отдельного модуля. Это дело техники. Приводим определение производного класса `tlongtime`.

```
{описание типа tlongtime}
type tlongtime=class(ttime)
protected
days: byte;
function total_sec:integer; override;    {в базовом классе используем}
procedure timefromsec(tsec:integer); override;     {слово virtual}
public
procedure settime(d,h,m,s:byte);
procedure print;
end;{обратите внимание — описание короче, чем для базового класса}
{реализация переопределяемых методов}
function tlongtime.total_sec:integer;
begin
   total_sec:=days*24*3600+hour*3600+min*60+sec;
end;
procedure tlongtime.timefromsec(tsec:integer);
var time:integer;
begin
   days:=tsec div (24*3600); time:=tsec mod (24*3600);
   inherited timefromsec(time); {вызываем метод базового класса}
end;
procedure tlongtime.settime(d,h,m,s:byte);
begin
   inherited settime(h,m,s); {вызываем метод базового класса}
```

```
   days:=d;
   if hour>23 then begin
      days:=days+hour div 24; hour:=hour mod 60;
   end;
end;
procedure tlongtime.print;
begin
   writeln(days,' суток ',hour,' часов ',min, ' минут ',sec, ' секунд ');
end;
```

Для того чтобы убедиться в том, что операции действительно стали полиморфными, т. е. выполняются правильно для объектов двух разных классов, напишем небольшую демонстрационную программу. В целях экономии места поместим в нее только процедуру plus, прося поверить на слово, что остальные также работают правильно.

```
var x,y,z:ttime; a,b,c:tlongtime;
begin
   x:=ttime.create;  y:=ttime.create; z:=ttime.create;
   x.settime(0,30,30);y.settime(0,40,40);
   z.plus(x,y); z.print;
   a:=tlongtime.create;  b:=tlongtime.create; c:=tlongtime.create;
   a.settime(2,23,30,30);b.settime(0,0,29,45);
   c.plus(a,b); c.print;   readln;
end.
```

19.2.9. Абстрактные классы

В приведенном примере один конкретный класс был создан на основе другого, такого же конкретного. Под словом "конкретный" мы понимаем класс, *от которого можно создавать объекты*, используя переменные соответствующего типа.

Интересная формулировка. Получается, что не от всех классов можно создавать объекты? Возникает сразу два вопроса: для чего нужны такие классы и как компилятор отличает абстрактный класс от конкретного?

Для тех, кто хорошо усвоил принципы наследования и полиморфизма, ответ на первый вопрос будет понятен. Абстрактные классы используются как базовые, на основе которых строятся производные, вполне конкретные классы. Имея базовый абстрактный класс, реализовать производные классы можно гораздо легче и быстрее, чем в случае разработки "с нуля".

Ответ на второй вопрос формален — абстрактным классом считается такой класс, который имеет хотя бы один абстрактный метод. В свою очередь, аб-

страктный метод не имеет реализации (реализация оставлена для классов-потомков). Допустим, в абстрактном классе "файл" нельзя реализовать такие методы, как чтение или запись информации, если не знать конкретного вида файла. Тем не менее, нам точно известно, что такая операция будет реализована во всех наследниках, поэтому стоит описать указанные методы класса как абстрактные. В свою очередь, операции удаления или переименования файла можно реализовать непосредственно в базовом классе.

Признаком абстрактного метода является ключевое слово `abstract`, которое добавляется к описанию виртуального метода, например:

```
Procedure ИмяПроцедуры(Формальные параметры); virtual; abstract;
```

Хотя бы одного такого описания в классе вполне достаточно, чтобы запретить компилятору создание объектов данного типа.

Хорошим примером абстрактного класса является стандартный класс `tobject`, являющийся общим предком всех стандартных классов библиотеки Delphi, а также предком всех классов, которые создаются с использованием ключевого слова `class`.

Замечание

Если все-таки вам захочется создать собственный класс начиная с нуля, воспользуйтесь типом `object` вместо типа `class`. Этот тип хоть и считается устаревшим, но сохраняется во всех версиях Delphi, и пользоваться им, пожалуй, легче, чем типом `class` (гораздо меньше всяких тонкостей). Правда, и возможностей у него также значительно меньше. Авторам же, разобравшись с новыми возможностями Delphi, не хочется возвращаться к старому типу `object`.

19.2.10. Свойства *(property)*

Рассмотрим еще одну интересную конструкцию Delphi, служащую для поддержки ООП, которая имеется далеко не во всех языках программирования. Мы имеем в виду специальные средства доступа к полям, называемые *свойствами*. Напомним, что, согласно принципу инкапсуляции, доступ к полям рекомендуется выполнять только через методы. *Свойства* позволяют *неявно* вызывать методы для доступа к полям. Другими словами, программист может работать с полями не только через методы, но и через свойства — вот прекрасное воплощение принципа инкапсуляции!

В Delphi свойства описываются наравне со всеми другими элементами класса, т. е. они являются такими же полноправными элементами класса, как поля и методы. Для многих полей определяются свойства, которые описываются с областью видимости:

- `public`, если экземпляры класса планируется создавать программно;
- `published`, если класс предназначен для использования на этапе визуальной разработки (такие классы в Delphi называются *компонентами*).

Для описания свойств используется следующая конструкция:

```
property ИмяСвойства: Тип read ЭлементКласса write ЭлементКласса;
```

После слов `read` и `write` записывается или имя поля, которому соответствует свойство, или имя метода для доступа к этому полю. При этом после `read` часто записывается имя поля, поскольку чтение информации из поля — безопасная операция и для нее редко пишется специальный метод. Напротив, после слова `write` в большинстве случаев стоит имя метода, используемого для занесения информации в данное поле, поскольку при занесении информации имеется риск испортить самое дорогое, что есть в объекте — его данные. Тип свойства в большинстве случаев совпадает с типом поля, за исключением случаев, когда поле неузнаваемо скрывается под свойством.

Обращаться к свойствам можно так же, как к полям. При этом в методах класса можно использовать для одних и тех же действий как поля, так и свойства, понимая, однако, что внутренний механизм будет различным. В программе, использующей класс, к свойству обращаются при помощи составного имени: `ИмяОбъекта.ИмяСвойства`.

Такая конструкция может стоять даже в левой части оператора присваивания. При этом вместо обычной операции присваивания будет выполняться метод для занесения информации в поле. Поскольку поля скрыты, а свойства открыты, получается, что обратиться к полю в программе можно только с помощью свойства, которое в свою очередь вызовет метод.

Небольшой пример. Приведем усеченный вариант класса `ttime`, в который добавлены свойства для доступа к часам, минутам и секундам. В этом случае каждому полю будут соответствовать два разных имени — имя поля и имя свойства. В Delphi принято обозначать их одинаково, но к имени поля добавлять первую букву `f` (от англ. *field* — поле). Поэтому поля класса будут называться `fhour`, `fmin` и `fsec`, а свойства — `hour`, `min` и `sec`. Чтобы не загромождать описание, уберем из класса все операции.

```
type ttime=class
protected
fhour, fmin, fsec: byte; {защищенные поля}
procedure correct(var a, b:byte);{служебная процедура для корректировки}
procedure setsec(s:byte);{метод для занесения секунд с корректировкой}
procedure setmin(m:byte);{ метод для занесения минут с корректировкой }
public
constructor create(h,m,s:byte); overload;
procedure settime(h,m,s:byte);
procedure print;
property sec: byte read fsec write setsec; {открытые свойства}
```

```
property min: byte read fmin write setmin;
property hour: byte read fhour write fhour;
end;
constructor ttime.create(h,m,s:byte);
begin
{при использовании свойств корректировка будет выполнена автоматически}
    hour:=h; min:=m; sec:=s;
 end;
procedure ttime.settime(h,m,s:byte);
begin
   hour:=h; min:=m; sec:=s;
end;
procedure ttime.correct(var a,b:byte);
begin
if a>59 then begin
   b:=b+a div 60; a:=a mod 60;
end;
end;
procedure  ttime.setsec(s:byte);
begin
fsec:=s; correct(fsec,fmin); correct(fmin,fhour);
end;
procedure  ttime.setmin(m:byte);
begin
fmin:=m; correct(fmin,fhour);
end;
procedure ttime.print;
begin
        writeln(hour,':',min, ':',sec);
end;
```

Как видим, использование свойств позволило упростить методы, поскольку вся автоматическая корректировка теперь запрятана так далеко, что ее видно только в скрытых служебных процедурах. Но она работает! Демонстрационная программа подтверждает это:

```
var x,y,z:ttime;
begin
   x:=ttime.create(0,60,90); x.print;
   y:=ttime.create; y.settime(0,60,90);y.print;
```

```
  z:=ttime.create; z.hour:=0; z.min:=60; z.sec:=90; z.print;
  x.min:=x.min+59; x.print; {даже такое работает правильно}
  readln;
end.
```

В приведенной программе значения времени задаются тремя различными способами с одинаковым результатом. Наверное, читатель понимает, что свойства могут найти и более серьезное применение, чем корректное задание значений отрезка времени.

К сожалению, ограниченный объем книги не позволяет нам остановиться на такой интересной разновидности свойств, как *указатели на обработчики событий*. Эта тема подробно разбирается в литературе по Delphi, например в [24].

19.2.11. Применение ООП

Разобраться с особенностями реализации ООП в любом языке программирования всегда нелегко для начинающих. Однако сейчас эта технология является ведущей, поэтому затраченное время и усилия не пропадут даром.

В настоящее время под общим термином "объектно-ориентированное программирование" понимается целая группа методов разработки программного обеспечения. Чтобы внести ясность в терминологию, перечислим основные направления в использовании объектно-ориентированного подхода:

- объектно-ориентированный анализ и проектирование используются на начальных стадиях разработки программы и состоят в выделении классов, которые необходимо будет реализовать на следующих стадиях, а также в установлении взаимосвязей между классами. Это тема отдельного курса;
- создание программистом собственных классов и объектно-ориентированных библиотек, содержащих иерархии классов, — это как раз и есть ООП в чистом виде, которым мы и занимались при изучении настоящего раздела;
- объектно-ориентированная разработка приложений (чаще всего визуальная) с использованием стандартных библиотек классов. Примером такой библиотеки является библиотека VCL (Visual Component Library), которая содержит около сотни полезных классов. Кроме того, имеется огромное количество различных специализированных библиотек классов для Delphi, распространяемых на компакт-дисках или через сеть Internet.

Визуальная разработка с использованием стандартных библиотек в настоящее время настолько популярна, что вопросы программирования часто отходят на второй план, уступая место вопросам грамотного использования стандартных классов. Однако приобрести квалификацию профессионально-

го программиста можно только через опыт собственных разработок. Знания, полученные при создании и применении собственных классов, помогут использовать стандартные классы более рационально.

19.3. Контрольные вопросы и задания

Вопросы

1. В чем состоит различие между АТД и структурами данных?
2. Какие основные действия выполняют со списками? деревьями?
3. Какие способы представления графа вы знаете? Какой способ предпочтительнее?
4. Как обычно реализуются множества?
5. Каковы преимущества сокрытия деталей разработки?
6. Перечислите основные принципы ООП.
7. Чем отличается область видимости `private` от области `protected`?
8. Можно ли в Delphi создавать объекты статически?
9. С какой целью используется указатель `self`?
10. В чем заключается механизм наследования классов?
11. Какие классы называются абстрактными?
12. Чем отличаются перекрытие и замещение методов в производных классах?
13. Зачем используются свойства?
14. Перечислите основные направления ООП.

Задания

Реализуйте следующие классы:

- класс для работы с датами;
- класс для работы с обыкновенными дробями;
- класс для работы со стеком;
- класс для работы с очередью;
- класс для работы с двусвязным списком;
- класс для работы с двоичным поисковым деревом;
- класс для работы с целыми числами неограниченных размеров. Для внутреннего представления чисел используйте массивы или списки.

Заключение

Читатель, проработавший все главы книги, разобравший многочисленные примеры и сумевший выполнить задания, имеющиеся в конце каждой главы, наверное, уже может не считать себя начинающим программистом, стоящим на самой начальной ступени обучения. Первые, самые трудные шаги позади, азы программирования освоены, определенный багаж типовых приемов и алгоритмов имеется.

Но и гордиться особыми достижениями пока рано, поскольку основной процесс обучения еще впереди и предстоит получить очень много знаний и умений, прежде чем можно будет говорить о профессионализме. Авторы верят, что полученные базовые знания помогут читателям в освоении новых языков и систем программирования, а знание типовых алгоритмов поможет в решении самых разнообразных задач — будь то разработка увлекательных игр и головоломок, студенческие расчеты или создание необычных графических изображений. В любом случае время, проведенное за компьютером, будет потрачено не зря. Ведь путешествие в увлекательный мир компьютерных чудес только начинается!

В заключение хотелось бы поблагодарить тех людей, которые оказали большую помощь авторам в их работе: заместителя главного редактора издательства "БХВ-Петербург" Людмилу Еремеевскую и всех сотрудников редакции за поддержку и доброжелательную критику рукописи, что, несомненно, способствовало ее улучшению. И еще несколько благодарностей индивидуального характера.

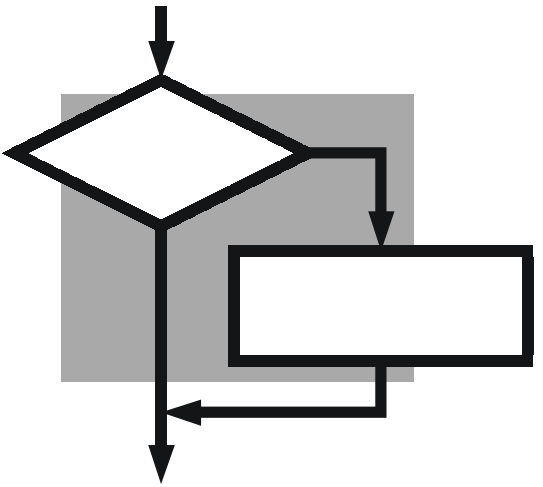
Рапаков Г. Г.

Хочется выразить глубокую признательность генеральному директору МООИ "Инфакто" Кириллу Сергеевичу Ярошу за предоставленную поддержку. Многолетняя работа связывает автора с деканом факультета радиотехники, электроники и связи Санкт-Петербургского государственного университета аэрокосмического приборостроения, профессором, лауреатом Государственной Премии СССР Дмитрием Васильевичем Тигиным, а также с заведующим кафедрой радиопередающих и телевизионных устройств СПб. ГУАП Борисом Семеновичем Тимофеевым и всем коллективом кафедры.

Сердечная признательность всем тем, без чьей помощи это издание никогда бы не состоялось: Дине Ивановне и Герману Георгиевичу Рапаковым, Людмиле Рапаковой, Наталье Невмывака (Федоровой) и многим другим.

Хочу поблагодарить своих детей-студентов Александра и Марину за активную помощь в подготовке книги. Александр отладил и протестировал многие из примеров книги (в двух системах — Turbo Pascal и Delphi), Марина проверила текст на соответствие правилам русского, кое-где и английского языка. Большое спасибо аспиранту Вологодского государственного технического университета Игорю Андрианову, опытному, талантливому программисту, с которым всегда можно обсудить любую проблему. Благодарю коллег-преподавателей нашего университета за их доброжелательное отношение и поддержку. Особая благодарность за понимание и заботу — моему мужу Виктору.

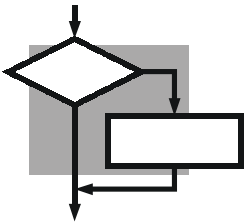
Ржеуцкая С. Ю.



ЧАСТЬ V

Приложения

Приложение 1



Компиляторы и системы программирования на основе Pascal

В данном приложении рассматриваются только 32-разрядные компиляторы и системы на их основе. Основная цель авторов — привлечь внимание читателей к новым возможностям использования старого, надежного и испытанного языка Pascal.

Система программирования Delphi (и ее версия Kylix для операционной системы Linux) уже многократно упоминалась на протяжении всей книги. Ее входной язык существенно дополнен по сравнению с Turbo (Borland) Pascal 7.0, причем расширения языка идут в русле с основными тенденциями развития языков программирования. Некоторые расширения не вошли в основные разделы книги, поэтому посвятим отдельный раздел приложения особенностям языка Delphi по сравнению с Turbo Pascal 7.0 (надеемся, что читатель уже освоил все нюансы этого языка).

Возможно, какие-то мелочи окажутся упущенными. Понятий, слишком сложных для начинающих, мы не касаемся намеренно. Но и тех языковых особенностей, которые перечислены ниже, уже достаточно, чтобы при переходе на новую среду программирования использовать новые возможности языка, получая при этом более качественные программы.

П1.1. Особенности языка Delphi

Комментарии

Начнем с самого простого и приятного — в Delphi появились однострочные комментарии (разумеется, в дополнение к тем комментариям, которые уже были). Их обозначение аналогично обозначению однострочных комментариев в языке C:

```
//Далее может располагаться однострочный комментарий.
```

Типы данных

Концепция типов данных в Delphi расширена за счет введения понятий *логического* и *физического* типов. При этом физические типы имеют строго определенное внутреннее представление, а логическому типу могут быть поставлены в соответствие различные физические типы. По своей сути логический тип близок к абстрактному типу данных *(см. разд. 19.1)*. Правда, для любого логического типа в Delphi всегда имеется физический тип по умолчанию (в отличие от абстрактного типа данных в обычном понимании). Для программиста это означает, что он может пользоваться логическими типами практически так же, как и физическими. Конечно, при желании всегда можно изменить внутреннее представление логического типа.

Рассмотрим основные поддерживаемые логические и физические типы.

Числовые типы. Основные числовые типы приведены в табл. П1.1 и П1.2.

Таблица П1.1. *Целые типы*

Тип	Диапазон	Размер ячейки памяти
`Shortint`	–128–127	1 байт
`Smallint`	–32 768–32 767	2 байта
`Longint`	–2 147 483 648–2 147 483 647	4 байта
`Int64`	-2^{63}– $2^{63} - 1$	8 байт
`Byte`	0–255	1 байт, беззнаковый
`Word`	0–65 535	2 байта, беззнаковый
`Longword`	0–4 294 967 295	4 байта, беззнаковый

Логический тип `integer` по умолчанию эквивалентен `longint`.

Таблица П1.2. *Действительные типы*

Тип	Диапазон	Кол-во значащих цифр	Размер ячейки памяти
`Real48`	2.9×10^{-39}–1.7×10^{38}	11–12	6 байт
`Single`	1.5×10^{-45}–3.4×10^{38}	7–8	4 байта
`Double`	5.0×10^{-324}–1.7×10^{308}	15–16	8 байт
`Extended`	3.6×10^{-4951}–1.1×10^{4932}	19–20	10 байт
`Comp`	$2^{63}+1 - 2^{63}-1$	19–20	8 байт
`Currency`	–922337203685477.5808–922337203685477.5807	19–20	8 байт

Логический действительный тип — `real`, который эквивалентен `double`.

Символы и строки. Язык Delphi поддерживает два физических символьных типа:

- тип `ansichar` — это символы в кодировке ANSI, которым соответствуют числовые коды в диапазоне от 0 до 255;
- тип `widechar` — это символы в кодировке Unicode, им соответствуют числовые коды от 0 до 65535, при этом под код каждого символа отводится 2 байта.

Логический символьный тип — `char`, который эквивалентен `ansichar`.

Язык Delphi поддерживает три строковых типа:

- тип `shortstring` представляет собой строки в стиле Pascal длиной от 0 до 255 символов *(см. главу 14)*;
- тип `ansistring` представляет собой динамически размещаемые в памяти (открытые) строки, длина которых ограничена только объемом свободной памяти *(см. разд. 18.4)*. Работать с данными строками можно точно так же, как и с обычными строками, т. е. нумерация символов начинается с единицы, и все строковые функции Turbo Pascal хорошо работают с этими длинными строками;
- тип `widestring` аналогичен типу `ansistring`, но каждый символ строки типа `widestring` является Unicode-символом.

Логический строковый тип `string` по умолчанию эквивалентен типу `ansistring`, но можно назначить ему и тип `shortstring` с помощью директивы компилятора `{$H-}`.

Помимо перечисленных типов, для представления строки можно использовать тип `pchar`. Строки, которые описываются таким образом, полностью эквивалентны строкам с завершающим нулем, использующимся в языке C, т. е. нумерация символов в них начинается с нуля, и для них нужно использовать функции, аналогичные функциям языка C. Например, для получения длины строки типа `ansistring` используется функция `length`, а для получения длины строки типа `pchar` — функция `strlen`, как в языке C. Строки с завершающим нулем в стиле C используются в Windows API (Application Programming Interface — библиотека стандартных функций операционной системы, доступная программисту), поэтому при обращении к функциям API из программы на Delphi иногда требуется преобразовывать тип `string` к типу `pchar` при помощи процедуры `strpcopy`.

Преобразование типов. Для преобразования числовых данных в строковые и обратно по-прежнему можно использовать процедуры `str` и `val` *(см. разд. 14.3)*, однако более удобно использовать специальные функции преобразования `inttostr` (`floattostr`, `currtostr`) и наоборот — `strtoint` (`strtofloat`, `strtocurr`). Например, `s:=inttostr(n);` — переменная `n` целого типа будет преобразована в строку `s`.

Типы для работы с датой и временем. В языке введено несколько новых типов, предназначенных для работы с датами и временем. Они определены следующим образом:

```
type
   tdatetime = double;
   tdate = tdatetime;
   ttimestamp = record
      time: integer; { время в миллисекундах от полуночи }
      date: integer; { единица + число дней с 01.01.0001 г.}
   end;
end;
```

Из определений понятно, что внутреннее представление даты и времени сводится к обычным числовым типам (целым или вещественным). Однако смысл введения этих новых типов все же есть, поскольку появилось несколько очень полезных и необходимых процедур и функций для работы с датами и временем:

- `function now: tdatetime` — возвращает текущую дату и время;
- `function date: tdatetime` — возвращает текущую дату;
- `function time: tdatetime` — возвращает текущее время;
- `function datetostr (d:tdatetime):string` — преобразует дату в строку символов;
- `function timetostr(t:tdatetime):string` — преобразует время в строку символов;
- `function datetimetostr(datetime:tdatetime):string` — преобразует дату/время в строку символов;
- `function datetimetotimestamp(datetime:tdatetime):ttimestamp` — конвертирует `tdatetime` в `ttimestamp`;
- `function timestamptodatetime(const ts:ttimestamp):tdatetime` — конвертирует `ttimestamp` в `tdatetime`;
- `procedure decodedate(date:tdatetime; var year,month,day:word)` — раскладывает дату `date` на год, месяц и день;
- `procedure decodetime(time:tdatetime; var hour,min,sec,ms:word)` — раскладывает время `time` на часы, минуты, секунды и миллисекунды;
- `function encodedate(year,month,day:word):tdatetime` — функция, обратная `decodedate`;
- `function encodetime(hour,min,sec,ms:word):tdatetime` — функция, обратная `decodetime`;
- `function formatdatetime(const frmt:string;dt:tdatetime):string` — преобразует `datetime` в строку с заданным форматом.

Этого внушительного набора функций вполне достаточно, чтобы полностью обеспечить любые вычисления, в которых участвуют дата или время, например, вычисление возраста человека по дате его рождения или подсчет банковских процентов за определенный срок.

Тип `variant` — особый тип языка Delphi. Значение этого типа наперед неизвестно, однако может быть определено через присваиваемое значение одним из следующих типов: все целые, вещественные, строковые, символьные и логические типы, за исключением `int64`.

Следующие примеры демонстрируют использование типа `variant` и механизм преобразования типов при смешивании его с другими типами.

```
var v1, v2, v3, v4, v5: variant; { описание variant-переменных }
    i: integer; d: double; s: string;
begin
   v1 := 1; { integer-значение }
   v2 := 1234.5678; { real-значение }
   v3 := 'Иванов'; { string-значение }
   v4 := '1000'; { string-значение }
   v5 := v1 + v2 + v4; { real-значение 2235.5678 }
   i := v1; { i = 1 (integer-значение) }
   d := v2; { d = 1234.5678 (real-значение) }
   s := v3; { s = ' Иванов' (string-значение) }
   i := v4; { i = 1000 (integer-значение) }
   s := v5; { s = '2235.5678' (string-значение) }
end.
```

Функция `vartype(const v: variant)` возвращает фактический тип переменной типа `variant` в виде целого числа (при этом каждому числовому значению поставлен в соответствие конкретный тип).

Структурные типы в Delphi используются те же, что и в Turbo Pascal, но работать с массивами стало гораздо удобнее за счет снятия двух существенных ограничений:

- размеры массивов ограничены только доступными размерами памяти;
- отпала необходимость всегда задавать размеры заранее, поскольку появились открытые массивы *(см. разд. 18.4)*.

Упакованные массивы и записи. Элементы упакованного массива хранятся в памяти максимально плотно. При записи он предварительно упаковывается с целью экономии памяти. При чтении, наоборот, распаковывается. Операции упаковки и распаковки требуют дополнительного времени. Поэтому использование упакованных массивов несколько замедляет работу програм-

мы. От обычного массива `array` описание упакованного массива отличается тем, что перед этим словом добавляется слово `packed`, например:

```
var x: packed array [1..1000] of integer;
```

Аналогично упакованным массивам появились и *упакованные записи* (`packed record`), работа с которыми выполняется точно так же, как и с обычными записями, но несколько медленнее.

Тип "множество" не претерпел изменений, к сожалению, он по-прежнему сильно ограничен в размерах (не более 256 элементов).

Файловые типы также остались прежними, немного изменились названия стандартных процедур `assign` (`assignfile`) и `close` (`closefile`). Список стандартных функций для работы с файлами существенно расширен, по сравнению с Turbo Pascal 7.0 добавлено много новых функций.

В Delphi введено еще 3 новых типа данных:

- `class` *(см. разд. 19.2)*;
- `class of Тип` — объектная ссылка (например, `class of tobject`);
- `interface(РодительскийИнтерфейс)` — тип "интерфейс", который используется при компонентно-ориентированной разработке приложений.

Изучение двух последних типов выходит за рамки книги.

Константы и переменные

Правила записи обычных именованных и неименованных констант не изменились. Описание типизированных констант также осталось прежним, но в Delphi они стали настоящими константами, значения которых *нельзя изменять*. Однако такие константы по-прежнему занимают память в отличие от обычных именованных констант, которые вписываются непосредственно в текст программы. Более того, локальные типизированные константы по-прежнему располагаются в области памяти глобальных переменных *(см. разд. 10.5)*.

Синтаксис объявления переменных несколько расширен, поскольку появилась возможность инициализировать их начальными значениями, например, так:

```
var x: integer=0;
```

Правда, таким способом нельзя инициализировать локальные переменные в стеке.

Стандартные функции

Список стандартных функций существенно расширен. Теперь они располагаются в трех модулях — `system`, `sysutils` и `math`, причем только `system` подключается автоматически, остальные нужно указывать в предложении

uses. Некоторые из полезных стандартных подпрограмм мы уже рассмотрели выше. Из наиболее необходимых отметим следующие:

- `function Power(Base, Exponent: extended): extended` — для возведения в степень, логарифмы по основанию 2 и основанию 10;
- `function Log10(X: extended): extended` — логарифм по основанию 10;
- `function Log2(X: extended): extended` — логарифм по основанию 2;
- полный набор всех тригонометрических функций;
- полный набор функций для выполнения типовых действий с массивами (таких как вычисление суммы, среднего, минимального или максимального элемента и ряд других).

Операторы языка

Основные операторы не претерпели изменений. Замечена лишь небольшая реконструкция цикла `for`, синтаксис которого также не изменился:

- после окончания цикла значение параметра становится на единицу больше конечного значения (в Turbo Pascal оно равнялось конечному значению);
- параметр цикла запрещено изменять в теле цикла (что всегда считалось плохим приемом программирования).

Обработка исключительных ситуаций

К их числу относятся сбои в работе аппаратуры, ошибки вычислений (например, деление на ноль), попытки присвоить значение, выходящее за пределы типа, и т. д. Для корректной обработки подобных ошибок в язык добавлено несколько новых операторов, использовать которые просто и удобно. Операторы для обработки исключительных ситуаций (ИС) имеются во всех современных языках программирования, поэтому познакомимся с ними поближе.

Блок `try – except – end` предоставляет возможность разместить в тексте программы код обработки исключительных ситуаций. Синтаксис таков:

```
try
{опасные операторы, способные генерировать ИС}
except
{операторы, обрабатывающие ИС}
end;
```

Блок `try – except – end` работает следующим образом. Выполнение начинается с операторов, расположенных в блоке `try – except`. Если в каком-либо операторе возникает ИС, то она *подавляется* (т. е. не происходит ава-

рийного завершения программы). Затем выполняются все операторы, расположенные в блоке except – end.

Например:

```
k:= 0; n:= 3;
try
   n:= n div k; {Деление на нуль. Оператор сгенерирует ИС}
   writeln(n);
except
   writeln('Ошибка деления на нуль');
end;
```

Результатом выполнения блока операторов будет появление на экране сообщения **Ошибка деления на нуль**, после чего программа продолжит работу с оператора, следующего за словом end.

Если бы оператор n:= n div k не был защищен блоком try – except – end, то возникшая при его выполнении ИС привела бы к нежелательному аварийному завершению программы.

Оператор on – do, используемый внутри блока except – end, позволяет не просто предотвратить прерывание программы, но и определить, какого именно вида была ИС.

Например:

```
var k,n:integer; s:string;
begin
...
k:= 0; s:='3'; n:= 3;
try
   n:=strtoint(s); {в данном примере этот оператор завершится успешно, но
                    потенциально он опасен}
   n:= n div k; {Деление на нуль. Оператор сгенерирует ИС}
   writeln(n);
except
   on e: EConvertError do writeln('Ошибка преобразования типов');
   on e: EdivByZero do writeln('Деление на нуль');
end;
```

В этом примере оба оператора внутри блока try – except потенциально опасны, поэтому проверяются две различных ИС. Объявленная в блоке except – end переменная e может иметь любое имя.

Блок try – finally – end также используется для предотвращения ИС, которые могут возникнуть при выполнении программы. В отличие от блока try – except – end, блок try – finally – end используется для освобождения ресурсов памяти, закрытия файлов и т. п. в случае возникновения ИС.

Синтаксис:

```
try
{операторы, способные генерировать ИС}
finally
{операторы освобождения ресурсов памяти }
end;
```

Отметим, что блок `finally – end` выполняется всегда вне зависимости от того, была или не была сгенерирована ИС.

Например:

```
try
getmem (p, 8000); {выделение памяти}
{Далее следуют операторы, возможно опасные}
finally
freemem (p, 8000); {освобождение памяти}
end;
```

В данном примере освобождение памяти будет выполнено в любом случае.

Оператор `raise`. Программист может программно сгенерировать исключительную ситуацию. При этом выполнение текущего программного блока (подпрограммы) прерывается и сгенерированная ИС передается в вышестоящий блок, где может быть обработана. Например:

```
if not FileExists('c:\autoexec.bat') then
 raise Exception.Create('File not found');
```

Подпрограммы и модули

В язык Delphi добавлено несколько новых возможностей работы с подпрограммами.

Использование переменной `result`. При определении функции в ее теле можно использовать переменную `result` вместо имени функции. Примеры будут приведены чуть ниже.

Перегрузка подпрограмм. Директива `overload` позволяет использовать одно имя для нескольких подпрограмм.

Например:

```
function divide(X, Y: real): real; overload;
begin
Result := X/Y; {вот и пример использования переменной result}
end;
function divide(X, Y: integer): integer; overload;
```

```
begin
Result := X div Y;
end;
```

В этом примере описаны две функции с одним именем `divide`. При обращении к функции с таким именем вызвана будет та функция, фактические параметры которой соответствуют формальным параметрам. Так, при обращении `divide(6.8, 3.2)` будет вызвана первая функция, т. к. ее формальные параметры имеют действительный тип, а при обращении `divide(6, 8)` будет вызвана вторая функция.

Директива `Overload` разрешена для подпрограмм, в которых могут различаться только типы параметров. Такое действие называется *перегрузкой подпрограмм*. Перегрузка подпрограмм давно имеется в языке C, теперь эта полезная возможность добавлена в язык Delphi.

Параметры по умолчанию. Эта возможность также добавлена по аналогии с языком C. Объявить значения параметров по умолчанию можно, например, так:

```
procedure p(x:integer; n:integer=10);{для n задано значение по умолчанию}
begin
текст процедуры
end;
```

К процедуре `p` можно обратиться двумя способами:

```
p(5,8);
```

или просто

```
p(4);
```

В последнем случае параметр `n` будет иметь значение по умолчанию, равное 10. Просто и удобно.

Некоторые изменения внесены в правила оформления модулей, введены новые названия частей модуля `initialization` и `finalization` *(см. разд. 12.3)*.

П1.2. Другие компиляторы и системы программирования на основе языка Pascal

Популярность Delphi, несомненно, обусловлена наличием мощнейшей библиотеки классов VCL (Visual Component Library) и удобной визуальной средой разработки. Большую роль играет и репутация фирмы Borland и лично А. Хейлсберга.

Однако система Delphi — коммерческий программный продукт, и стоимость лицензионной версии Delphi существенно выше, чем стоимость аналогич-

ных продуктов фирмы Microsoft. Поэтому имеет смысл обратить внимание и на другие системы на основе Pascal.

Многие из них являются бесплатными и распространяются свободно в исходных текстах (в основном на C/C++) в рамках проекта GNU (аббревиатура GNU раскрывается рекурсивно — GNU's Not UNIX, такое могут придумать только программисты). Подобно тому как права обычных компаний, производящих программное обеспечение, охраняются их знаком авторских прав, свободно распространяемые программные продукты защищены документом с заголовком "GNU General Public License". В этом документе говорится о правах, которыми располагает любой текущий владелец данного программного продукта, в частности, о невозможности создавать на его основе коммерческие программные продукты с закрытыми исходными текстами.

Предлагаем короткую информацию по наиболее распространенным 32-разрядным компиляторам и системам программирования на основе языка Pascal. Все они обеспечивают совместимость по входному языку с Turbo Pascal, расширения языка идут в направлении сближения с языками Delphi и C++. На сегодняшний момент, однако, ни в одном из известных компиляторов не обеспечивается полная совместимость с Delphi, но во многих есть возможности, имеющиеся в C++ и не реализованные в Delphi (например, перегрузка операторов).

Материал следующего раздела почерпнут, в основном, в сети Internet, поэтому не претендует ни на полноту, ни на абсолютную безошибочность, тем более, что постоянно появляются новые версии компиляторов и IDE (интегрированных систем разработки).

П1.2.1 Free Pascal

Домашняя страница: **http://www.freepascal.org**.

Поддерживаемые операционные системы: Win32, DOS, Linux, FreeBSD и OS/2. Кроме того, Amiga OS на базе процессора Motorola 680x0.

Язык

Обеспечивается совместимость с Turbo Pascal 7.0. Совместимость с Delphi пока не обеспечена, но многие расширения Delphi уже есть, так что в дальнейшем, возможно, будет обеспечена и полная совместимость.

Основными расширениями по сравнению с Turbo Pascal являются:

- поддержка строк `ansistring`;
- поддержка исключительных ситуаций;

- поддержка информации о типах, доступной во время выполнения программы. Эта информация имеет устойчивую аббревиатуру — RTTI (Run Time Type Information);
- перегрузка процедур, функций и операторов, поддержка функций с переменным числом параметров. Все эти возможности имеются в C/C++;
- возможность использовать C-подобные операторы, например, `a++` и `a+=b`;
- вместо исполняемого файла компилятор может просто сгенерировать программу на Ассемблере, которую потом можно обработать, используя любой из компиляторов с Ассемблера;
- возможность вызывать внешние процедуры, написанные на C;
- выборочная компоновка — из модулей в исполняемый файл включаются только те процедуры, которые действительно используются, что помогает существенно уменьшить размер получаемой программы;
- поддержка стандартного GNU-отладчика (который входит в комплект поставки).

Модули и библиотеки

Базовая библиотека включает в себя стандартные модули, реализующие управление памятью, работу с файловой системой, управление консольным вводом/выводом, работу со строками и датами, математические функции, обработку исключений и интерфейс к API поддерживаемых операционных систем. При этом многие модули очень похожи на стандартные модули фирмы Borland. Если сравнивать с Turbo Pascal, то мы видим модули `crt`, `dos`, `graph` (реализует работу с VGA-графикой), `strings`. Сравнивая с Borland Delphi, видим соответствующие модули `sysutils` и `classes`. В комплект поставки входит также модуль `objects`, позволяющий писать программы с использованием стандартных элементов интерфейса (кнопок, списков, меню и т. д.).

IDE и утилиты

Для Win32 и DOS существует консольная оболочка `fp`, построенная по образу и подобию Turbo Pascal 7.0. Она интегрирована со стандартным GNU-отладчиком.

Отладчик прилагается в дистрибутиве. К сожалению, GNU-отладчик не ориентирован на Free Pascal, зато Free Pascal ориентирован на него.

Справка

Справка Free Pascal распространяется с дистрибутивом в двух форматах: PDF и HTML, при желании можно скачать отдельно справку и в других форматах. И в том, и в другом виде она хорошо структурирована и прекрасно читается.

Лицензия

Free Pascal распространяется на условиях GNU General Public License (GNU GPL). Модули и библиотеки распространяются на условиях GNU Library General Public License (GNU LGPL), последняя версия данной лицензии называется GNU Lesser Public License.

На сайте разработчика имеется множество дистрибутивов под различные платформы.

П1.2.2. GNU Pascal

Домашняя страница: **http://www.gnu-pascal.de**.

Бесплатный кроссплатформенный 32/64-битовый компилятор, распространяемый под лицензией GNU.

Поддерживаемые операционные системы: DOS32, Win32, UNIX (все клоны UNIX-подобных систем).

Язык

Входной язык является реализацией в первую очередь стандарта языка. Обеспечивается полная поддержка стандарта Pascal, и большая часть стандарта Extended Pascal. В дополнение к стандартам содержит и многие расширения, внесенные фирмой Borland. Совместим с Turbo Pascal 7.0 по исходным текстам.

Ориентирован в первую очередь на UNIX-платформу, поэтому для работы с версиями DOS и Win32 нужно загрузить соответствующие расширения (ссылки на них есть на домашнем сайте проекта GNU Pascal).

Для загрузки доступна версия 2.0. Более новая версия доступна в виде исходных текстов на C, которые следует компилировать при помощи компилятора GNU C.

К сожалению, для данного компилятора не имеется интегрированной среды разработки, поэтому работать с ним не очень удобно. Но зато с его помощью можно создавать настоящие кроссплатформенные приложения, написанные на языке Pascal.

П1.2.3. TMT Pascal

Домашняя страница: **http://www.tmt.com**.

(русское зеркало **http://pascal.sources.ru/tmt/index.htm**).

Поддерживаемые операционные системы: Windows 95/98/ME/NT/2000/XP и IBM OS/2.

Язык

Совместим по языку с Turbo Pascal 7.0. К преимуществам данного компилятора можно отнести:

- поддержка технологий Intel MMX и AMD 3Dnow!;
- полностью 32-битовый код, отсутствие ошибки в модуле CRT;
- встроенная поддержка мыши;
- поддержка графического разрешения вплоть до 1600×1200 при 32-битовом цвете;
- С-подобные комментарии и операторы;
- перегрузка операторов.

Некоторые из стандартных процедур и функций Turbo Pascal претерпели некоторые изменения, все они отражены в удобной и содержательной справочной системе.

Модули и библиотеки

В состав TMT Pascal входят мощные библиотеки для поддержки технологий Silicon Graphic OpenGL и Microsoft DirectX 8.0. Эти программные интерфейсы (API) специально предназначены для разработки игр и других высокопроизводительных мультимедийных приложений. Они включают поддержку двумерной (2D) и трехмерной (3D) графики, музыки, звуковых эффектов и т д.

Широко известная графическая библиотека SciTech Multi-platform Graphics Library (MGL) полностью поддерживается в TMT Pascal.

Благодаря поддержке столь мощных мультимедийных библиотек основное назначение TMT Pascal — программирование игр и трехмерной графики.

IDE и утилиты

Вместе с компилятором поставляется удобная графическая оболочка для Windows.

Лицензия

Фирма-разработчик — TMT Development. На сайте разработчика доступны две версии: TMT Pascal v4.0 Multi-target Edition (платная версия) и TMT

Pascal Lite v3.90 MS-DOS Edition (бесплатная для не коммерческого использования).

П1.2.4. Virtual Pascal

Домашняя страница: **http://www.vpascal.com**.

Поддерживаемые ОС: Win32, OS/2, Linux.

Язык

Реализует синтаксис Borland Pascal 7.0 и некоторое подмножество Object Pascal, в частности, совместим с Delphi 2.0. В общем-то, является полноценной заменой Borland Pascal для 32-битовых приложений.

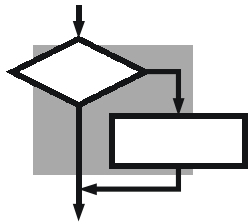
IDE

Имеет среду разработки, очень похожую на Borland Pascal 7.0, поскольку для ее создания использовалась библиотека Turbo Vision от Borland Pascal 7.0, адаптированная под 32-битовую платформу.

Лицензия

Ранее компилятор был коммерческим, но с версии 2.1 Virtual Pascal стал свободно распространяемым.

Приложение 2



Стандартные модули Turbo Pascal и дополнения к ним

В данном приложении содержится описание модулей `crt` и `graph`, а также некоторых функций модуля `dos`, которые полезны и при работе в Windows.

Кроме того, приводятся собственные модули, разработанные авторами, для дополнения модулей `crt` и `graph`, содержащие возможности, которые определенно заинтересуют читателей, работающих в среде Turbo Pascal.

Для того чтобы приведенный материал был понятен, необходимо сначала дать краткие пояснения по основным принципам формирования изображений на экране.

П2.1. Принципы формирования изображения

Дисплей (монитор) персонального компьютера включает в себя экран с электронно-лучевой трубкой (Cathode Rate Tube, CRT), а также комплекс оптических и электронных средств, обеспечивающих формирование изображения. В основе его работы есть много общего с телевизором. В результате соударения пучка электронов с поверхностью экрана, покрытой светящимся веществом (люминофором), образуется светящаяся точка — *пиксел* (от *Picture Element — PixEl*), причем интенсивность ее свечения зависит от энергии электронного пучка. Изменяя энергию пучка, можно получить различную яркость пиксела.

Электронный луч обегает экран слева направо и сверху вниз с определенной скоростью, последовательно формируя множество пикселов, которые, сливаясь друг с другом, воспринимаются глазом как единое целое. Но в отличие от изображения на экране телевизора, изображением на экране дисплея управляет программист.

Дело в том, что изображение, которое мы видим на экране, на самом деле формируется в специальной буферной памяти — *видеопамяти*. Схемы элек-

тронной развертки отображают построенное в видеопамяти изображение на экране. Программа может поместить нужное значение в видеопамять или прочитать из нее ранее установленное значение точно так, как это делается с обычной оперативной памятью.

Видеопамять вместе с электронными схемами управления дисплеем располагается на одной печатной плате, которая называется *дисплейным адаптером (видеокартой)*. Одной из наиболее важных характеристик видеокарты является размер размещенной на ней видеопамяти. От этого зависят такие важные параметры изображения, как количество пикселов по горизонтали и вертикали — *разрешающая способность*, а также количество цветов, которые могут одновременно отображаться на экране, — *палитра*.

Существуют два вида информации, появляющейся на экране: *текстовая*, т. е. состоящая из знаков алфавита, цифр и специальных символов, и *графическая* — чертежи, рисунки, графики. Хотя механизм формирования изображений — общий, однако существенные различия в использовании видеопамяти заставляют говорить о двух режимах работы дисплея — *текстовом* и *графическом*.

Учитывая существенные различия между текстовым и графическим режимом, процедуры и функции управления экраном для этих режимов сгруппированы в два отдельных модуля. Модуль `crt` содержит те из них, которые необходимы для управления экраном в текстовом режиме, а также процедуры и функции для управления клавиатурой и звуком. Модуль `graph` — графическая библиотека Turbo Pascal.

П2.2. Модуль *crt*

П2.2.1. Процедуры и функции управления экраном

При работе на персональном компьютере пользователь обычно имеет дело с операционной системой Windows и ее приложениями, поэтому более привычным кажется графический режим. Однако программы, написанные на Turbo Pascal, часто используют текстовый режим, особенно если они предназначены для делового применения. Сам по себе этот режим заслуживает внимания, поскольку использует видеопамять значительно экономнее, чем графический.

В текстовом режиме наименьшей единицей изображения является не отдельный пиксел (точка), а символ целиком, иногда говорят — *знакоместо*. Разумеется, каждый символ состоит из отдельных пикселов, но процесс прорисовки каждого символа берет на себя операционная система.

В видеопамяти изображение в текстовом режиме хранится следующим образом. Под каждое знакоместо экрана отводится по два байта. Первый байт

хранит *код* символа, который занимает данное знакоместо, а второй — так называемый *цветовой атрибут* символа. Байт цветового атрибута включает цвет символа (четыре младших бита), цвет фона (3 старших бита), а также специальный бит мерцания, установка которого в единицу позволяет получать мерцающее изображение. Оно используется, в основном, только для выдачи сообщений, которые должны привлечь внимание пользователя программы. Содержимое каждого из битов байта цветовых атрибутов показано на рис. П2.1.

Рис. П2.1. Состав байта цветового атрибута (текстовый режим)

Единица в соответствующем бите говорит о наличии данной составляющей цвета. Из рисунка понятно, что все цвета образуются смешением трех основных цветов: синего, зеленого и красного.

В текстовом режиме возможны 16 различных цветов для символов (различные сочетания четырех битов дают 16 различных комбинаций, от 0000 до 1111) Они обозначаются цифрами от 0 до 15 или константами, определенными в модуле `crt` (табл. П2.1). При этом последние восемь цветов отличаются от первых только повышенной яркостью (интенсивностью). Например, желтый цвет — это коричневый с повышенной яркостью. Можно установить только 8 различных цветов фона, т. к. под цвет фона отведено 3 бита.

Текстовый режим работы характеризуется двумя важными параметрами: максимальным числом символов в строке и количеством строк на экране. Стандартные значения этих параметров — *25 строк по 80 символов* в каждой строке. Однако можно установить и ряд других значений этих параметров с помощью стандартной процедуры `textmode` в виде:

```
textmode(mode);
```

где `mode` — константа целого типа `word` (табл. П2.2).

Например:

`textmode(BW40);` или `textmode(0);` — активизация черно-белого текстового режима 25 строк по 40 символов;

`textmode(259);` или `textmode(CO80+Font8x8);` — активизация цветного текстового режима 50 строк по 80 символов в строке.

По умолчанию (при отсутствии `textmode`) устанавливается режим `mode = 3`.

Таблица П2.1. *Константы цветов*

Цвет	Наименование константы	Значение константы
Черный	`Black`	0
Синий	`Blue`	1
Зеленый	`Green`	2
Бирюзовый	`Cyan`	3
Красный	`Red`	4
Малиновый	`Magenta`	5
Коричневый	`Brown`	6
Светло-серый	`LightGray`	7
Темно-серый	`DarkGray`	8
Светло-голубой	`LightBlue`	9
Светло-зеленый	`LightGreen`	10
Светло-бирюзовый	`LightCyan`	11
Светло-красный	`LightRed`	12
Светло-малиновый	`LightMagenta`	13
Желтый	`Yellow`	14
Белый	`White`	15

Таблица П2.2. *Текстовые режимы*

Наименование константы mode	Значение константы mode	Количество строк, символов в строке	Тип адаптера	Вид вывода
`BW40`	0	25×40	Цветной	Черно-белый
`CO40`	1	25×40	Цветной	Цветной
`BW80`	2	25×80	Цветной	Черно-белый
`CO80`	3	25×80	Цветной	Цветной
`Mono`	7	25×80	Моно	Черно-белый
`Font8x8`	+256	50 строк	Цветной	Цветной

Положение каждого знакоместа текстового экрана можно определить двумя координатами, при этом считается, что *левый верхний угол экрана имеет координаты (1,1)*. Координата *X* обозначает позицию символа *в строке*, а координата *Y* — номер *строки*, отсчитываемый по направлению от верха экрана к его низу.

При выводе на экран текстовой информации Turbo Pascal позволяет управлять атрибутами каждого символа. Для этого используются стандартные процедуры модуля `crt` (табл. П2.3).

Таблица П2.3. *Процедуры управления цветом*

Процедура	Назначение
`LowVideo`, `NormVideo`	Устанавливают режим нормальной яркости свечения выводимых на экран символов
`HighVideo`	Устанавливает режим наибольшей яркости свечения
`TextBackGround(Color)`	Устанавливает цвет фона (т. е. цвет области, которая окружает выводимый символ). `Color` — выражение целого типа в диапазоне от 0 до 7, соответствующее одной из первых восьми констант цветов, определенных в модуле `crt`
`TextColor(Color)`	Устанавливает цвет выводимых символов. Здесь `Color` — выражение целого типа, находящееся в диапазоне от 0 до 15, соответствующее одной из констант модуля `crt`

Очень удобно задавать одновременно все цветовые атрибуты символа, используя переменную `textattr`, определенную в модуле `crt`. Значения этой переменной лучше задавать в виде шестнадцатеричной константы, представляющей значение всего байта атрибутов. При этом первая шестнадцатеричная цифра соответствует цвету фона (сюда же включается и значение бита мерцания), вторая цифра соответствует цвету выводимых символов.

Например:

```
textattr:=$1E;{ желтые символы на синем фоне }
textattr:=7; { значение по умолчанию: белые символы на черном фоне }
```

В последнем случае черному цвету фона соответствует значение 0 (`black`) константы цвета, поэтому используется десятичное значение 7 вместо шестнадцатеричного $07.

Листинг П2.1 содержит простую программу, которая демонстрирует использование процедур установки цветов, очистки экрана, а также позиционирования курсора (табл. П2.4).

Листинг П2.1. Элементарные функции управления экраном

```
uses crt;
begin
   textcolor(lightcyan);  { или textcolor(13) или textcolor($D) }
   textbackground(green); { или textbackground(2) }
   { вместо всего этого можно было написать textattr:=$2D; }
   clrscr; { очистка экрана зеленым цветом }
   gotoxy(35,13);writeln('Здравствуйте!'); readln;
end.
```

П2.2.2. Работа с окнами

Окно — это ограниченная прямоугольная область экрана, выполняющая те же функции, что и полный экран. Для определения окна используется процедура

```
window(x1,y1,x2,y2);
```

которая определяет на экране новое активное текстовое окно. При определении окна `x1,y1` являются координатами верхнего левого угла окна, а `x2,y2` — координатами нижнего правого угла с учетом того, что левый верхний угол полного экрана имеет координаты (1, 1), а минимальный размер окна включает один столбец и одну строку.

Обратите внимание — при неправильном задании параметров `x1,y1` или `x2,y2` (например, когда `x1=85` или `y2=30`) процедура `window` игнорируется. При этом никакого сообщения об ошибке не выдается.

Следует знать:

- после вызова процедуры определения окна никаких видимых изменений на экране не происходит, поэтому ощутить действие процедуры можно только после выполнения следующих за ней команд управления экраном;
- после определения окна все координаты задаются относительно активного окна, начиная от его левого верхнего угла, а не полного экрана;
- весь вывод выполняется только в активное окно.

Следует учитывать, что если окно не определено в явном виде при помощи процедуры `window`, то окном является *полный экран*, что соответствует вызову процедуры `window(1,1,80,25);`. Такой вызов процедуры `window` может быть использован и для отмены действия активного окна.

Табл. П2.4 содержит основные процедуры и функции, позволяющие управлять выводом в активное окно (или полный экран, если окно не объявлено).

Таблица П2.4. *Процедуры и функции для управления выводом в активное окно*

Процедура или функция	Назначение
`ClrScr`	Очищает активное окно и устанавливает курсор в левый верхний угол
`ClrEol`	Очищает строку активного окна от текущей позиции курсора до конца строки без изменения позиции курсора
`GoToXY(x,y)`	Перемещает курсор в позицию с координатами *X*, *Y* в рамках активного окна
`WhereX`	Функция, которая возвращает *X*-координату текущей позиции курсора (относительно активного окна)
`WhereY`	Функция, которая возвращает *Y*-координату текущей позиции курсора (относительно активного окна)

Программа, приведенная в листинге П2.2, изображает на экране 7 полосок (окон), соответствующих разным цветам фона (без черного). Таким же способом можно изображать на экране узоры, состоящие из разноцветных прямоугольников.

Листинг П2.2. Демонстрация процедуры `window` и различных цветов фона

```
uses crt;
var k,x:integer;
begin
   textbackground(0);
   clrscr;
   x:=2;
   for k:=1 to 7 do begin
     textbackground(k);
     window(x,1,x+11,25);clrscr;
     x:=x+11;
   end; readln
end.
```

П2.2.3. Задержка при выполнении программы

Иногда работу компьютера приходится искусственно замедлять. Для этих целей используется еще одна процедура из модуля `crt`, которая реализует задержку в выполнении программы:

```
delay(time);
```

где `time` — время задержки в миллисекундах.

Эта процедура особенно часто используется при выводе различных изменяющихся изображений на экран. Кстати, именно она является виновницей печально знаменитой ошибки в модуле `crt`.

В примере (листинг П2.3) процедура `delay` замедляет процесс заполнения экрана по спирали разноцветными звездочками. Другие примеры использования процедуры `delay` будут представлены в программах оживления изображений ниже.

Листинг П2.3. Рисуется спираль из разноцветных звездочек

```
uses crt;
type direction=(right,down,left,up);
var c,x,y,h,k:integer;
procedure go(d:direction; n:integer);
begin
   for k:=1 to n do begin
      case d of
         right: x:=x+2;
         down:  y:=y+1;
         left:  x:=x-2;
         up:    y:=y-1;
      end;
      c:=random(15)+1; textcolor(c);
      gotoxy(x,y); write('*'); delay(30);
   end;
end;
begin
   clrscr; x:=39; y:=13; h:=1;
   gotoxy(x,y); write('*');
   repeat
      go(right,h); go(down,h); h:=h+1;
```

```
      go(left,h); go(up,h); h:=h+1;
   until h>24;
end.
```

П2.2.4. Управление клавиатурой

Стандартная клавиатура имеет три типа клавиш:

- символьные (буквы, цифры);
- управляющие (функциональные клавиши, клавиши перемещения курсора, вставка, удаление и т. д.);
- сдвига <Ctrl>, <Alt>, <Shift>, <NumLock>, <CapsLock> и др.

При нажатии какой-либо клавиши клавиатуры вырабатывается код, который называется *кодом сканирования (скан-кодом)*. Каждая клавиша имеет собственный уникальный код сканирования, который зависит только от местоположения клавиши на панели клавиатуры и не связан с изображенным на клавише символом. По сути дела, скан-код является порядковым номером клавиши (например, клавиша <Esc>, расположенная в верхнем левом углу, имеет скан-код 01).

Но программе нужен не порядковый номер нажатой клавиши, а соответствующий обозначению на этой клавише код символа — код ASCII. Этот код не зависит однозначно от скан-кода, т. к. одной и той же клавише могут соответствовать несколько значений ASCII-кода, в зависимости от состояния других клавиш. Например, клавиша с обозначением <1> используется еще и для ввода символа <!> (если она нажата вместе с клавишей <Shift>).

Все преобразования скан-кода в ASCII-код выполняются операционной системой, которая анализирует состояние клавиш сдвига, выполняет преобразование и помещает полученный ASCII-код в специальную область оперативной памяти, которая называется *буфером клавиатуры*. Оттуда его и может получить программа Turbo Pascal или другого языка программирования.

Замечание

Клавишам сдвига, разумеется, не соответствуют никакие ASCII-коды, поэтому получить их состояние в программе можно только путем анализа соответствующих байтов оперативной памяти (два байта с адресами $417 и $418). Скан-код только что нажатой клавиши можно получить, обратившись к порту с номером $60 (`port[$60]`).

Модуль `crt` содержит две функции для работы с буфером клавиатуры, которые предоставляют программисту больше возможностей управления клавиатурой, чем стандартные процедуры `read` и `readln`.

Логическая функция `keypressed` проверяет наличие символов в буфере клавиатуры и возвращает значение `true`, если на клавиатуре была нажата какая-либо клавиша (т. е. буфер не пуст), и `false` в противном случае.

Например, следующий пустой цикл может быть использован для реализации в программе паузы до нажатия любой клавиши:

```
repeat
until keypressed;
```

Обратите внимание — функция `keypressed` не удаляет введенный символ из буфера клавиатуры.

Это делает функция `readkey`, которая выполняет гораздо больше полезной работы.

Функция `readkey`, возвращающая значение типа `char`, считывает код символа из буфера клавиатуры и удаляет его из буфера ("слепой" ввод без эхо-отображения на экране). Если буфер был пуст, то возникает пауза до нажатия любой клавиши.

Например:

```
c:=readkey;{ переменная с получит значение символа нажатой клавиши }
readkey;   { вызов этой функции как процедуры разрешен и используется }
           { для создания паузы до нажатия любой клавиши }
```

Обратите внимание — при нажатии одной из управляющих клавиш (функциональные клавиши, клавиши управления курсором и т. п.) в буфер клавиатуры помещаются расширенные коды клавиш, состоящие из двух значений, причем первое всегда равно #0 (символ с кодом 0). При этом функция `readkey` возвращает сначала нулевой символ (#0), а при повторном вызове — вторую (расширенную) часть кода.

Листинг П2.4. Программа, определяющая коды, соответствующие нажатым клавишам

```
uses crt;
var ch: char;
begin
   repeat
      ch:=readkey;
      if ch=#0 then
         begin
            ch:=readkey; write('Спец. клавиша');
         end
      else
```

```
        begin
           write('char = ');
           if ch < #32 then write('^',chr(ord(ch)+64))
                       else write(ch);
        end;
        writeln(' ASCII = ',ord(ch));
   until ch=#27;
end.
```

С помощью этой программы можно узнать коды всех интересующих вас клавиш.

Замечание

Для реализации паузы до нажатия любой клавиши в программе корректнее использовать функцию `readkey`, чем цикл `repeat ... until keypressed`, поскольку при этом не остается "мусора" в буфере клавиатуры. Очистить буфер клавиатуры можно очень просто: `while keypressed do readkey;`.

Функция `readkey` широко используется в программах не только текстового, но и графического режима.

П2.2.5. Управление звуком

Для этих целей достаточно двух процедур:

- `sound(Hz)` — процедура, которая включает внутренний динамик. `Hz` задает частоту генерируемого динамиком сигнала в герцах. Звуковой сигнал звучит до тех пор, пока не будет выключен с помощью процедуры `nosound` (если вы забыли написать вызов этой процедуры в программе, то звук не выключится, даже если программа уже закончила работу);
- `nosound` — процедура, которая отключает внутренний динамик.

Например:

```
sound(5000); delay(100); nosound;
```

В листингах П2.5, П2.6 приведены тексты программ, демонстрирующих возможности этих процедур.

Листинг П2.5. Программа выводит гамму в первой, второй и третьей октаве

```
uses crt;
{ частоты, соответствующие нотам первой октавы,
для каждой следующей октавы частоты удваиваются }
const octava_1:array[1..7] of integer=
(262,294,330,349,392,440,493);
```

```
var n,octava,koef: integer;
begin
   koef:=1;
   for octava:=1 to 3 do
      begin
         for n:=1 to 7 do
            begin
               sound(octava_1[n]*koef);
               delay(300); nosound;
            end;
         koef:=koef*2;
      end;
   sound(octava_1[1]*koef);delay(300); nosound;
end.
```

Листинг П2.6. Программа, которая позволяет перемещать символ @ по экрану со звуком

```
uses crt;
const
{ коды клавиш-стрелок для управления движением символа }
   left =#75; right=#77; up =#72; down =#80;
   ch   ='@';       { символ для перемещения }
var c: char; curx, cury: byte; badkey: boolean;
procedure beep(badkey: boolean);
begin
   if badkey then   { сигнал ошибки }
      begin
         sound(500); delay(100);
         sound(440); delay(200);
         nosound;
      end
   else
      begin   { щелчок в 360 Гц длительностью 0.02 секунды при движении }
         sound(360); delay(20); nosound;
      end;
end;
begin
   clrscr; curx:=40; cury:=12;
```

```
   repeat
      gotoxy(curx,cury); write(ch);
      badkey:=false; c:=readkey;
      if c=#0 then   { расширенный код? } c:=readkey;
      case c of
         #27  : ;   { esc - ничего не делать }
         right: if curx < 80 then inc(curx);
         left : if curx > 1  then dec(curx);
         up   : if cury > 1  then dec(cury);
         down : if cury < 25 then inc(cury);
      else badkey:=true;
      end;
      beep(badkey);
   until c=#27
end.
```

П2.3. Модуль *graph*

П2.3.1. Общие сведения

В настоящее время графический режим является основным режимом работы видеоадаптера современного компьютера. Этот режим гораздо более требователен к видеопамяти, чем текстовый режим, но и преимущества его очевидны. Видеопамять в графике используется так — для каждого пиксела (а не знакоместа!) хранится его цвет. Таким образом, в этом режиме программист имеет уникальную возможность управлять каждым пикселом, что позволяет строить на экране любые изображения. Графическая библиотека Turbo Pascal проста и удобна в использовании, что делает разработку графических программ увлекательным занятием для программистов.

Модуль `graph` не входит в библиотеку Turbo.tpl, а находится в отдельном файле Graph.tpu. Если при попытке запуска программы возникла ошибка: **File Graph.tpu not found** (Файл Graph.tpu не найден), то возможно одно из двух: или этого файла нет на вашем компьютере, или он находится в недоступном каталоге. В последнем случае для установки каталога необходимо выполнить команду **Options | Directories | Unit Directories**, после чего ввести полный путь к каталогу, где находится Graph.tpu.

Для работы в графическом режиме необходимо наличие еще одного файла — так называемого графического драйвера. Это файл с расширением bgi (Borland Graphic Interface), который содержит небольшую программу-драйвер для управления работой дисплейного адаптера. Обычно это файл

egavga.bgi, рассчитанный на работу с дисплейным адаптером типа VGA, но вместо него может быть и файл с другим именем. При использовании графических шрифтов для вывода текста необходимо наличие файлов с расширением chr.

Переключение между текстовым и графическими видеорежимами

Среда Turbo Pascal использует текстовый режим работы адаптера, поэтому любая программа, использующая графические средства компьютера, должна выполнить переключение в графический режим.

Загрузка графического драйвера и инициализация графики осуществляются с помощью процедуры `initgraph`, которая вызывается следующим образом:

```
initgraph(Драйвер,Режим,ПутьКДрайверу);
```

Поясним назначение параметров процедуры.

- `Драйвер` — переменная типа `integer`, определяющая тип графического драйвера.
- `Режим` — переменная типа `integer`, определяющая режим работы адаптера. Режим определяет разрешающую способность и палитру цветов.
- `ПутьКДрайверу` — выражение типа `string`, содержащее путь к файлу драйвера; если в качестве параметра используется пустая строка или пробел, то предполагается, что драйвер находится в текущем каталоге. При этом файл egavga.bgi (или другой файл с расширением bgi) действительно должен находиться в том каталоге, к которому ведет указанный путь. В противном случае может появиться сообщение об ошибке **BGI Error: Graphics not initialized (Use UnitGraph)**. Такое же сообщение появляется при отсутствии процедуры `initgraph` в графической программе.

Если оба файла (модуль `graph` и графический драйвер) обнаружены, то процедура загружает графический драйвер в оперативную память и переводит адаптер в графический режим работы. Для указания типа драйвера в модуле предопределены различные константы. Дело в том, что для каждого типа графического адаптера требуется свой драйвер. Однако в настоящее время используется в основном адаптер типа VGA, поэтому большинство графических драйверов, имеющихся в составе системы Turbo Pascal, уже не находят применения. Фактически из всех констант типов драйверов используются только две:

```
const
   detect = 0; { режим автоопределения типа }
   vga = 9;    { адаптер vga }
```

Если параметр Драйвер равен detect (значение 0), то процедура initgraph автоматически загружает именно тот графический драйвер, который нужен для данного адаптера, при этом устанавливается режим, обеспечивающий наилучшую разрешающую способность и палитру. Обычный графический режим, который устанавливается при подключении драйвера egavga.bgi, задает разрешение 640×480 пикселов при палитре из 16 цветов.

Этот режим не может обеспечить такое же высокое качество изображения, как распространенные программы для Windows, однако при определенном старании можно получить очень неплохие графические программы. Если все-таки вам очень хочется иметь лучший графический режим, то имейте в виду, что модуль graph позволяет подключать не только стандартные, но и пользовательские драйверы, если они написаны в соответствии со стандартом. В настоящее время распространяется несколько драйверов, обеспечивающих лучшее разрешение и/или палитру цветов, чем обеспечивает egavga.bgi. Для их установки в модуле graph имеется функция RegisterBGIDriver.

Основные процедуры и функции, используемые при переключении видеорежимов, представлены в табл. П2.5.

Таблица П2.5. *Процедуры и функции для переключения видеорежимов*

Процедура или функция	Назначение
InitGraph(gd,gm,path)	Инициализация графического режима
GraphResult	Функция, которая возвращает целое число — код ошибки при переключении в графику. Для проверки успеха операции используется константа GrOK=0
GetGraphMode	Возвращает код текущего графического режима
SetGraphMode(Mode)	Устанавливает новый графический режим
RestoreCrtMode	Временное переключение в текстовый режим
CloseGraph	Выход из графического режима, восстанавливается тот режим, который был до инициализации графики
ClearDevice	Очистка экрана в графическом режиме

Система координат графического экрана

Любое изображение в графическом режиме строится с использованием системы координат, в которой каждый пиксел имеет две координаты (X и Y). Программист может раскрасить любым цветом каждый пиксел. Начало координат находится в левом верхнем углу так же, как и в текстовом режиме. Однако отсчет координат начинается с нуля, т. е. левый верхний угол экрана имеет координаты (0, 0). Горизонтальная координата экрана (X) увеличи-

вается слева направо, а вертикальная (Y) — сверху вниз. Таким образом, при разрешении экрана 640×480 максимальная координата X равна 639, максимальная координата Y — 479. В табл. П2.6 приведены основные функции для работы с координатами.

Таблица П2.6. *Основные функции для работы с координатами*

Функция	Назначение
`GetMaxX`	Возвращает максимальную координату X
`GetMaxY`	Возвращает максимальную координату Y
`GetX`	Возвращает текущую координату X
`GetY`	Возвращает текущую координату Y
`GetPixel(X,Y)`	Возвращает цвет точки с координатами (X, Y)

Текущий указатель

Текущий указатель в графическом режиме играет ту же роль, что и курсор в текстовом режиме, однако, в отличие от курсора, *он невидим*. Многие графические процедуры и функции используют текущий указатель — например, приведенные выше функции `GetX` и `GetX` возвращают координаты текущего указателя. В дальнейшем это понятие будет использоваться нами при изложении материала.

П2.3.2. Графические примитивы

Это элементарные геометрические фигуры, для изображения которых имеются процедуры в графической библиотеке. Модуль `graph` поддерживает большое количество примитивов. Процедуры для их изображения приводятся в табл. П2.7.

Таблица П2.7. *Процедуры для изображения графических примитивов*

Процедура	Назначение
`PutPixel(X,Y,Color)`	Выводит на экран точку с координатами (X,Y) и цветом `Color`; положение текущего указателя не изменяется
`Line(X1,Y1,X2,Y2)`	Проводит прямую линию из точки с координатами (X_1,Y_1) в точку с координатами (X_2,Y_2). Положение текущего указателя не изменяется
`MoveTo(X,Y)`	Перемещает текущий указатель в точку (X,Y)

Таблица П2.7 *(продолжение)*

Процедура	Назначение
`LineTo(X,Y)`	Проводит прямую линию из точки, где находится текущий указатель, в точку с координатами (X,Y). Текущий указатель при этом перемещается в точку (X,Y)
`LineRel(Dx,Dy)`	Проводит прямую линию из точки, где находится текущий указатель, в точку с приращением координат на *Dx* (по *X*) и на *Dy* (по *Y*). Текущий указатель перемещается в конец линии
`Rectangle(X1,Y1,X2,Y2)`	Рисует прямоугольник с координатами (X_1,Y_1) — верхний левый угол и (X_2,Y_2) — нижний правый угол
`Bar(X1,Y1,X2,Y2)`	Заштрихованный прямоугольник с координатами (X_1,Y_1) — верхний левый угол и (X_2,Y_2) — нижний правый угол; используется стандартный цвет и стиль заливки
`Bar3d(x1,y1,x2,y2,h, Top)`	Объемная (трехмерная) прямоугольная полоса толщиной *h*; логический параметр `Top`, принимающий значения `TopOn` или `TopOff`, указывает, нужно ли изображать верхнюю грань
`Circle(X,Y,Radius)`	Окружность с центром в точке (X,Y) и радиусом `Radius`
`Arc(X, Y, StAngle, EndAngle, Radius)`	Дуга окружности от угла `StAngle` до угла `EndAngle` с центром в точке (X,Y) и радиусом `Radius`; углы задаются в градусах по направлению против часовой стрелки
`Ellipse (X, Y, StAngle, EndAngle, Xradius, Yradius)`	Дуга эллипса с центром в точке (X,Y) и радиусом `Xradius` (по оси *X*), `Yradius` (по оси *Y*) от начального угла `StAngle` до конечного угла `EndAngle`. Значения `StAngle=0` и `EndAngle=360` приведут к вычерчиванию полного эллипса
`FillEllipse(X, Y, Xradius, Yradius)`	Эллипс, заштрихованный текущим цветом и типом штриховки
`PieSlice(X, Y, StAngle, EndAngle, Radius)`	Заштрихованный сектор круга, начальный и конечный угол в градусах
`Sector(X, Y, StAngle, EndAngle, Xradius, Yradius)`	Заштрихованный сектор эллипса, параметры те же, что у процедуры `Ellipse`
`DrawPoly(N,ArrayOfPoint)`	Ломаная линия, которая имеет *N* вершин, координаты которых заданы в массиве записей `ArrayOfPoint`. Пример приведен ниже
`FillPoly(N,ArrayOfPoint)`	Заштрихованная замкнутая фигура, параметры те же

Таблица П2.7 *(окончание)*

Процедура	Назначение
`FloodFilll(X,Y,Border_Color)`	Заливка произвольной замкнутой области с цветом границ `Border_Color`. (*X*,*Y*) — координаты любой внутренней точки. Заливка области выполняется установленным стилем и цветом *(см. табл. П2.8)*

Примеры простых программ с использованием графики

Теперь мы имеем достаточно информации, чтобы перейти к делу. Не будем торопить события и начнем с несложных программ, которые помогут освоить технику построения изображений с использованием графических примитивов (листинги П2.7—П2.10). Во всех примерах предполагаем, что графический драйвер находится в текущем каталоге. Вероятнее всего, на вашем компьютере он находится в каталоге BGI родительского каталога Turbo Pascal. Не забудьте указать правильный путь к графическому драйверу.

Листинг П2.7. Программа рисует "домик"

```
uses graph;
var gd,gm: integer;
begin
   gd:=detect;
   initgraph(gd,gm,'');
   if graphresult <> grok then begin
      writeln('Ошибка при запуске графики');
      readln; halt; { выход из программы }
   end;
   { разбили изображение на примитивы и определили все координаты }
   rectangle(280,180,360,300); bar(305,210,335,270);
   line(280,180,320,150); line(320,150,360,180);
   circle(320,165,8);
   readln; closegraph;
end.
```

Замечание

Во всех дальнейших примерах для экономии места мы не будем проверять функцию `GraphResult`.

Листинг П2.8. Применение процедуры DrawPoly — рисуется треугольник

```
uses graph;
const { используем стандартный тип pointtype, определенный в graph }
   triangle: array[1..4] of pointtype = { задаем координаты всех вершин }
   (x: 50; y: 100), (x: 100; y: 100),   { замкнутой ломаной линии }
   (x: 150; y: 150), (x: 50; y: 100));  { крайние точки совпадают }
var gd, gm: integer;
begin
   gd := detect; initgraph(gd, gm, '');
   drawpoly(4, triangle);
   readln; closegraph;
end.
```

В следующем примере приводится самый простой, но далеко не самый лучший способ "оживить" изображение — с использованием процедуры cleardevice для стирания картинки (в данном случае — "вагончика"). Процедура delay, которая используется для задержки картинки на экране, входит в состав модуля crt.

Листинг П2.9. Оживление изображения (вариант № 1)

```
uses crt, graph;
const t=20;d=5;
{ t - время задержки картинки в мсек, d — приращение координаты X, }
{ их значения подбираются, чтобы создать иллюзию плавного движения }
var x,gd,gm:integer;
procedure vagon;
begin
   bar(x,100,x+70,130);
   circle(x+15,140,10);
   circle(x+55,140,10);
end;
begin
   gd:=detect; initgraph(gd,gm,'');
   x:=0;
   repeat
      vagon;   { нарисовали }
      delay(t);{ задержали изображение на экране }
```

```
      cleardevice; { быстро очистили экран }
      x:=x+d;  { изменили координату }
   until x>639;
end.
```

Последний пример, приведенный в листинге П2.10, несколько серьезнее. Программа изображает на экране график функции. Для того чтобы можно было быстро настроить программу на любой вид функциональной зависимости, используем отдельную функцию для задания функциональной зависимости, в которой легко можно изменить одну строку. Сам график строим в виде отрезков прямых. Для многих функций график, построенный таким способом, получается качественней, чем при прорисовке по отдельным точкам, к тому же это прекрасная возможность продемонстрировать действие процедур `moveto` и `lineto`, которые работают с текущим графическим указателем. Использование масштабных коэффициентов придает универсальность процедуре построения графика. При этом считаем, что график занимает все экранное пространство (ровно 640 точек), вне зависимости от введенного интервала изменения аргумента.

Листинг П2.10. Программа для построения графиков функций

```
uses graph;
var gd,gm:integer;
function f(x:real):real; { задание функции }
begin
   f:=x*x; { здесь может быть любая функция }
end;
procedure drawgrafik(a,b:real); { a,b — начальное и конечное значения x }
var x,dx,max,min,koef:real;
    k,x0,y0:integer; { x0,y0 — положение осей координат }
begin
   dx:=(b-a)/640; { определили шаг изменения x (640 точек на графике) }
   x:=a; max:=f(a); min:=f(a);
   for k:=1 to 640 do { определяем область значений f(x) }
   begin
      if f(x)>max then max:=f(x);
      if f(x)<min then min:=f(x);
      x:=x+dx;
   end;
```

```
   koef:=479/(max-min); { коэффициент по оси y }
   x:=a; { начальное значение x }
   moveto(0,round(479-koef*(f(a)-min))); { начальное значение указателя }
   for k:=1 to 639 do { строим график }
   begin
      x:=x+dx;
      lineto(k,round(479-koef*(f(x)-min)));
   end;
   x0:=round(639*a/(a-b)); { a/(a-b)=(0-a)/(b-a) }
   line(x0,0,x0,479); { ось y }
   y0:=round(479-479*(min/(min-max)));
   line(0,y0,639,y0); { ось x }
end;
begin
   gd:=detect; initgraph(gd,gm,'');
   drawgrafik(-4,4);
   readln; closegraph
end.
```

П2.3.3. Установка цветов и стилей

Процедуры установки сами по себе не производят никаких видимых изменений на экране, но установленные цвета и стили используются процедурами для вывода всех графических примитивов (кроме точки). Поэтому процедуры установки цветов и стилей должны предшествовать процедурам для изображения примитивов. Текущие установки сохраняются до тех пор, пока не будут изменены другими процедурами установки. При инициализации графического режима текущие цвета — черный для фона и белый для линий и штриховок, текущие стили — сплошная линия и заливка. В табл. П2.8 приведены процедуры для установки цветов и стилей.

Таблица П2.8. *Установка цветов и стилей для фигур*

Процедура или функция	Назначение
`SetColor(Color)`	Устанавливает цвет выводимого изображения, задаваемый параметром `Color`
`SetBkColor(Color)`	Устанавливает цвет фона
`SetLineStyle(Style, Pattern, Thickness)`	Устанавливает следующие параметры линии — тип (стиль), 16-битовый образец (для нестандартного стиля), толщина

Таблица П2.8 *(окончание)*

Процедура или функция	Назначение
`SetFillStyle(Style, Color)`	Устанавливает тип и цвет штриховки
`SetFillPattern(Pattern, Color)`	Позволяет задавать пользовательские типы штриховок. Параметр `Pattern` определен как `array[1..8] of byte`
`GetColor, GetMaxColor, GetBkColor`	Функции, которые возвращают текущее и максимальное значение цвета для заданного графического режима и значение цвета фона

Названия и значения констант для цветов такие же, как в модуле `crt` (см. табл. П2.2).

Всего имеется 12 стандартных стилей для штриховок и 4 для линий. Они нумеруются с нуля. Программист имеет возможность задавать свои образцы для линий и штриховок, формируя шестнадцатеричные константы из двоичных цифр, каждая из которых соответствует одному пикселу (1 — пиксел светится, 0 — не светится). Толщина линии задается в пикселах и может принимать только два значения: 1 или 3 (обычная линия и утолщенная).

Приведем две небольшие программы (листинги П2.11, П2.12), демонстрирующие все цвета и стили.

Листинг П2.11. Программа, демонстрирующая цвета и стили штриховок

```
uses graph;
var gd,gm,x,y,color,style:integer;
begin
   initgraph(gd,gm,''); x:=0;
{ цвет и стиль 0 не будут видны на черном }
   for color:=1 to 15 do begin
      y:=0;
      for style:=1 to 11 do begin
         setfillstyle(style,color);
         bar(x,y,x+30,y+35);
         y:=y+44;
      end;
      x:=x+43;
   end; readln;
end.
```

Листинг П2.12. Программа, демонстрирующая все виды линий

```
uses graph;
var gd,gm,y,thickness,style:integer; { thickness - толщина }
procedure linedemo(thickness:word);
begin
   for style:=0 to 3 do begin
      setlinestyle(style,0,thickness);
      line(0,y,639,y);
      y:=y+44;
   end;
end;
begin
   initgraph(gd,gm,'');
   setcolor(red); y:=0;
   linedemo(1); linedemo(3); readln;
end.
```

П2.3.4. Окна в графическом режиме

Напомним, что окно — это прямоугольная область экрана, выполняющая все функции полного экрана (так же как и для текстового режима). После установки окна любое изображение будет строиться в относительных координатах текущего окна. Процедуры для работы с окнами приведены в табл. П2.9.

Таблица П2.9. *Работа с окнами*

Процедура	Назначение
`SetViewPort (X1, Y1, X2, Y2, Clip)`	Устанавливает окно с координатами (X_1, Y_1) — левый верхний угол и (X_2, Y_2) — правый нижний угол. Если параметр логического типа `Clip=True`, то изображение, не вмещающееся в окно, отсекается, в противном случае не отсекается
`ClearViewPort`	Очищает текущее окно

Обратите внимание — для заливки текущего окна фоновым цветом не следует применять процедуру `setbkcolor`, т. к. она заливает весь экран. Удобнее выполнять это действие с помощью процедур `setfillstyle` и `bar`.

Например:

```
setviewport(200,100,440,380,true);
setfillstyle(1,4);
bar(200,100,440,380);
```

П2.3.5. Вывод текста

Вывод текста в графическом режиме выполняется только средствами модуля graph.

Обратите внимание — процедуры модулей system и crt для вывода на экран не работают корректно в графическом режиме, хотя и не выдают никаких сообщений об ошибках при выполнении. Это объясняется различиями в принципах формирования изображения в текстовом и графическом режимах. Не используйте в графическом режиме процедуры write, writeln, clrscr, gotoxy и др. Процедуры для вывода текста приведены в табл. П2.10. Пример программы, выводящей текст, приведен в листинге П2.13.

Таблица П2.10. *Вывод текста*

Процедура	Назначение
OutText(Text)	Выводит на экран строку текста Text, начиная с текущей позиции указателя. Текущий указатель перемещается в конец строки
OutTextXY(X,Y,Text)	Выводит на экран строку Text, начиная с точки (*X*, *Y*). Положение текущего указателя не изменяется
SetTextStyle(Font, Direction, CharSize)	Устанавливается шрифт, направление текста (горизонтальное или вертикальное), размер символов (коэффициент увеличения исходного размера шрифта Font)
SetTextJustify (Horiz, Vert)	Способы выравнивания текста относительно заданной точки (*X*, *Y*) для процедуры OutTextXY
TextWidth(Stroka)	Функция возвращает ширину строки текста на экране в пикселах, используя установленный шрифт
TextHeight(Stroka)	Аналогичная функция, но возвращает высоту строки текста

При выводе текста необходимо помнить, что все шрифты, кроме DefaultFont (с номером 0), хранятся в отдельных файлах с расширением chr (табл. П2.11). Стандартные варианты этих файлов не содержат русских букв, однако в настоящее время распространяются переработанные файлы со шрифтами, поддерживающие русский алфавит. Более того, имеется возможность создания своих собственных файлов со шрифтами, которые регистрируются в програм-

ме при помощи функции InstallUserFont, в качестве параметра которой передается имя файла с новым шрифтом (без расширения chr).

Таблица П2.11. *Стандартные шрифты*

Обозначение шрифта	Значение (номер)	Файл
DefaultFont	0	Нет
TriplexFont	1	Tripp.chr
SmallFont	2	Litt.chr
SansSerifFont	3	Sans.chr
GothicFont	4	Goth.chr

Замечание

Для того чтобы вывести в графическом режиме переменные числового типа, можно воспользоваться стандартной процедурой str *(см. разд. 14.3)*, преобразующей числовое значение в строку текста, а затем вывести полученную строку при помощи процедур OutText или OutTextXY.

Листинг П2.13. Пример вывода текста

```
uses graph;
var gd,gm:integer;
begin
   gd:=detect;
   initgraph(gd,gm,'');
   settextstyle(DefaultFont,HorizDir,3);
   settextjustify(CenterText,CenterText);
   setcolor(red);
   outtextxy(320,240,'Текст в центре экрана'); readln
end.
```

П2.3.6. Сохранение и восстановление битовых образов изображений

Напомним, что изображение строится в некоторой области видеопамяти. Очевидно, должна существовать возможность копирования этой области видеопамяти в обычную оперативную память с целью сохранения образов экрана в определенные моменты времени. Затем такое сохраненное в памя-

ти изображение можно будет переместить обратно в видеопамять, чтобы снова увидеть его на экране.

В текстовом режиме для подобных манипуляций приходится обращаться напрямую к известным адресам видеопамяти (см. листинг П2.16). К счастью, модуль `graph` содержит средства сохранения и восстановления образов изображений (табл. П2.12). В табл. П2.13 приведены константы для параметра `BitBit`, используемого в одной из процедур, перечисленных в табл. П2.12.

Таблица П2.12. *Сохранение и восстановление образов изображений*

Процедура или функция	Назначение
`ImageSize (X1,Y1,X2,Y2)`	Функция, которая возвращает размер необходимой памяти (в байтах) для сохранения изображения прямоугольного участка экрана с координатами (X_1, Y_1) — левый верхний угол и (X_2, Y_2) — нижний правый угол
`GetImage(X1, Y1, X2, Y2, BitMap)`	Запоминает изображение прямоугольного участка экрана с координатами (X_1, Y_1) и (X_2, Y_2) в буфере, на который указывает `BitMap`. Подробнее об указателях и работе с ними рассказано ниже *(см. гл. 18)*
`PutImage(X,Y,BitMap; BitBit)`	Выводит на экран изображение, сохраненное в буфере памяти, на который указывает `BitMap`. (X, Y) — верхний левый угол прямоугольной области, куда выводится изображение; `BitBit` — режим наложения на существующее изображение при выводе (логическая операция, которая выполняется между существующим на экране изображением и новым изображением, которое восстанавливается на экране)

Таблица П2.13. *Имеющиеся константы для параметра* `BitBit`

Константа	Логическая операция при выводе
`NormalPut=0; CopyPut=0`	Обычный вывод изображения без выполнения каких-либо операций
`XORPut=1`	Операция "исключающее ИЛИ" (`XOR`)
`OrPut=2`	Операция ИЛИ (`OR`)
`AndPut=3`	Операция И (`AND`)
`NotPut=4`	Инвертирование изображения (`NOT`)

Особый интерес представляет использование процедуры `PutImage` в режиме `XORPut`, т. к. в этом режиме наложение двух светящихся пикселов друг на друга гасит изображение, и, следовательно, таким способом можно воспользоваться для стирания изображений при их оживлении.

Приведем пример использования процедур GetImage и PutImage для "оживления" изображений. В тексте программы (листинг П2.14) используется динамическое выделение памяти и работа с указателем типа pointer.

Листинг П2.14. Оживление изображения (вариант № 2)

```
uses graph,crt;
const t=100; d=15;{ время задержки и приращение координаты x подбираем }
var p:pointer; { указатель (адрес) области памяти для хранения картинки }
      gd,gm,x,size:integer;
procedure vagon;
begin
   bar(x,100,x+70,130);
   circle(x+15,140,10);Circle(x+55,140,10);
end;
begin
   gd:=detect; initgraph(gd,gm,''); x:=0;
   vagon; size:=imagesize(0,100,70,150);
{ выделяем область памяти размера Size }
   getmem(p,size);
{ записываем изображение в память по адресу p }
   getimage(0,100,70,150,p^);  x:=0;
   repeat
     delay(t); { задержали на экране }
      putimage(x,100,p^,xorput); { стираем }
      x:=x+d; { изменяем координату }
      putimage(x,100,p^,xorput); { рисуем }
    until x>639;
{ освобождаем память }
   freemem(p,size);
end.
```

П2.4. Модуль DOS

Данный модуль содержит обширный набор разнообразных средств, дополняющих Turbo Pascal теми возможностями, которые предоставляет операционная система. Несмотря на то, что этот модуль разрабатывался для операционной системы DOS, большинство составляющих его элементов нельзя

считать устаревшими, т. к. они поддерживаются и в семействе Windows. Разделим их на несколько групп:

- работа с системной датой и временем;
- обработка параметров командной строки;
- вызов внешних программ из программы на Turbo Pascal;
- дополнение к стандартным средствам обработки файлов;
- работа с прерываниями операционной системы;
- получение значений переменных окружения (`environment`).

В данном приложении будут рассмотрены первые три группы.

П2.4.1. Работа с системной датой и временем

Имеются четыре простых в использовании, но чрезвычайно мощных процедуры, которые обеспечивают доступ к системной дате и времени (табл. П2.14). При этом из программы на Turbo Pascal можно не только получить текущие значения даты и времени, установленные в операционной системе, но и изменить их. Разумеется, последнее действие крайне опасно, т. к. результат его повлияет на дальнейшую работу всего программного обеспечения компьютера.

Таблица П2.14. *Процедуры для работы с системной датой и временем*

Процедура	Назначение
`GetDate(year,month,day, dayofweek)`	Возвращает системную дату: год, месяц, день и номер дня недели (0 соответствует воскресенью, 1 — понедельнику и т. д.)
`SetDate(year,month,day)`	Устанавливает новую системную дату (пользоваться осторожно!)
`Get-Time(hour,minute,sec,sec100)`	Возвращает системное время: часы, минуты, секунды и сотые доли секунды
`Set-Time(hour,minute,sec,sec100)`	Устанавливает новое системное время (пользоваться осторожно!)

Все параметры процедур имеют тип `word` (целое без знака).

Замечание

Процедура `GetTime` может быть с успехом использована для подсчета *времени выполнения* программы или ее фрагмента. Для этого достаточно поместить ее в начало и конец тестируемого фрагмента, а затем подсчитать интервал между двумя показаниями системного времени.

П2.4.2. Функции для обработки параметров командной строки

Сначала поясним назначение данной группы функций (табл. П2.15). Как известно, программу на Turbo Pascal можно откомпилировать с записью exe-файла на диск, а затем запускать полученный исполнимый файл из операционной системы. Если запуск программы выполняется из *командной строки* (например, через пункт **Выполнить** главного меню Windows или через какой-нибудь *файловый менеджер* типа Far или Norton Commander), то после имени программы можно указывать один или более параметров, разделяя их пробелами.

Например:

`format a:` — при запуске программы Format.com ей передается в качестве параметра имя диска, который нужно отформатировать.

Таблица П2.15. *Функции для обработки параметров командной строки*

Функция	Назначение
`ParamCount`	Количество параметров в командной строке
`ParamStr(nomer)`	Значение параметра с заданным номером (нумерация с единицы)

Например, следующий фрагмент программы выведет на экран значения параметров командной строки.

```
uses dos;
var i:integer;
begin
   for i:=1 to paramcount do writeln(paramstr(i));
... { продолжение программы }
end.
```

Замечание

При отладке можно задавать параметры командной строки непосредственно в среде Turbo Pascal с помощью установки **Run | Parameters**. Вызов `ParamStr(0)` возвращает полный путь к исполняемому файлу, из которого запущена программа.

П2.4.3. Запуск внешних программ из программы на Turbo Pascal

Запуск внешних программ выполняется при помощи процедуры exec. Она позволяет загружать и выполнять любые файлы с расширениями exe и com

так, как это делает сама операционная система. Запускаемым программам можно передавать параметры командной строки.

Вызов процедуры exec имеет следующий вид:

```
exec(path,params);
```

Параметр path — полное имя загружаемого файла, params — параметры. Параметры path и params имеют строковый тип. Если внешняя программа запускается без параметров, то params — пустая строка (два подряд идущих апострофа).

При запуске внешней программы обязательно необходимо проверять успешность этого действия при помощи функции DOSError, которая в случае успеха возвращает значение 0. Другие возможные значения DOSError задают следующие причины возникновения ошибки: 2 — файл не найден; 3 — маршрут к файлу не найден; 8 — недостаточно памяти для загрузки программы.

Поясним причину ошибки с кодом 8. Тема распределения памяти подробно обсуждалась по ходу книги и особенно в *гл. 18 (см. разд. 18.1)*. Процедура exec при выполнении не нарушает распределения памяти для текущей программы на Turbo Pascal. Однако для загрузки и выполнения новой программы необходима свободная память. Естественно, что для загрузки любой программы в зависимости от размера ее кода может понадобиться немалое количество памяти. В случае ее нехватки программа выполняться не будет, а переменная DOSError примет значение 8.

В связи с этой проблемой существуют строгие требования к распределению памяти для программ, которые используют обращение к процедуре exec. Программа, в которой выполняется запуск внешней программы, обязательно должна содержать директиву компилятора {$M} для того, чтобы освободить память для запуска внешней программы. Так, директива компилятора {$M $4000,0,$8000} задает требование на 16 Кбайт стека и 32 Кбайт для динамически распределяемой области. Оставшуюся память можно использовать для загрузки другой программы.

Пример, приведенный в листинге П2.15, может служить образцом для запуска любой внешней программы, т. к. хорошо демонстрирует технику выполнения этого действия.

Листинг П2.15. Запуск внешней программы

```
{$M $2000,0,0} {8 Кбайт стека и 0 байт Heap}
uses crt, dos;
var path, cmdline: string;
begin
   writeln('Полное имя программы для выполнения: '); readln(path);
```

```
   writeln('Командная строка для программы:'); readln(cmdline);
   swapvectors; { восстановить "захваченные" векторы прерываний }
   exec(path,cmdline); { выполнить программу }
   swapvectors; { "захватить" векторы повторно }
   if doserror<>0 then { проверить возможные ошибки }
      writeln('Ошибка DOS с номером ',doserror)
   else
   begin
      writeln('Программа ',path,' завершена.');
      writeln('Код завершения программы ',doseditcode);
   end; readln
end.
```

П2.5. Дополнения к модулям

П2.5.1. Модуль *crtplus*

Данный модуль (листинг П2.16) содержит несколько полезных процедур и функций, которых так не хватает в модуле `crt`. Здесь используются некоторые средства и приемы программирования, которые не рассматривались в книге.

Листинг П2.16. Модуль, дополняющий `crt`

```
unit crtplus;
interface
uses crt,dos;    const Up       = #72;  Down = #80;  Left  = #75; Right =
#77;          Enter    = #13;  Esc  = #27;  PgUp  = #73; PgDn  = #81;
type  massiv=array[1..20] of string[40];{для функции menu}
   procedure savescreen; {Запоминает экран в буфере}
   procedure loadscreen; {Восстанавливает экран из буфера}
   {Рисует окно с рамкой. Параметры: координаты окна, включая рамку,
   цвет рамки и цвет фона}
   procedure framewindow(x1,y1,x2,y2,frcolor,fon: integer);
   {Выводит меню с запоминанием и восстановлением экрана.
    Параметры: массив пунктов, координаты верхнего левого угла,
    кол-во пунктов, цвет фона, маркера и текста с рамкой}
   function menu(list:massiv; x,y,num,fon,marker,text:integer):integer;
implementation
   var screen:array[1..2000] of word absolute $B800:0; {видеопамять}
      bufer :array[1..2000] of word;{буфер для запоминания видеопамяти}
```

```
   procedure savescreen;
   var k:word;
   begin
      for k:=1 to 2000 do bufer[k]:=screen[k];
   end;
   procedure loadscreen;
   var k:word;
   begin
     for k:=1 to 2000 do screen[k]:=bufer[k];
   end;
{скрытая функция для изменения размеров текстового курсора,
используется в функции menu для того, чтобы убрать, а потом
восстановить курсор. Обращаемся к прерыванию операционной системы}
   procedure setcursorsize(csize:word);     var regs:registers;     begin
        with regs do
        begin
          ah:=$01; ch:=hi(csize); cl:=lo(csize);{это регистры процессора}
          intr($10,regs){вызываем прерывание с номером 10}
        end
     end;
   procedure framewindow(x1,y1,x2,y2,frcolor,fon: integer);
   {x1,y1,x2,y2 - координаты окна, включая рамку, frcolor - цвет рамки }
   const frchar:array[1..6] of char ={ для рисования рамки }
   (#218,#196,#191,#179,#192,#217);{используем символы псевдографики}
   var x,y : integer; {x,y - для вывода символов рамки }
       w,h:integer;{ w - ширина рамки, h - высота рамки }
       lastattr:byte;{нельзя портить текущие цветовые установки,
       поэтому для их сохранения используем переменную lastmode}
   begin
      lastattr:=textattr; {запомнили текущие цветовые установки}
      window(x1,y1,x2,y2);
      textbackground(fon); clrscr;
      window(x1,y1,x2+1,y2);
      textcolor(frcolor);
      gotoxy(1,1); write(frchar[1]);
      w:=x2-x1+1;h:=y2-y1+1;{определили ширину и высоту рамки}
      for x:=1 to w-2 do write(frchar[2]);{символы верхней границы рамки}
      write(frchar[3]); ;
      for y:=2 to h-1 do begin {символы левой и правой границ}
```

```
      gotoxy(w,y); write(frchar[4]);
      gotoxy(1,y); write(frchar[4]);{уголки}
   end;
   gotoxy(1,h); write(frchar[5]);
   for x:=1 to w-2 do write(frchar[2]); { символы нижней границы }
   write(frchar[6]);
   window(x1+1,y1+1,x2-1,y2-1); gotoxy(1,1);
   textattr:=lastattr;
end;
function menu(list:massiv; x,y,num,fon,marker,text:integer):integer;
var max,k:integer; c:char; oldtext:word;
begin
  oldtext:=textattr;
  savescreen;  {Копируем экран в буфер}
  textcolor(text);
  max:=0;
  for k:=1 to num do {Ищем самый длинный пункт меню}
     if length(list[k])>max then max:=length(list[k]);
  for k:=1 to num do {Дополняем более короткие строки пробелами}
     while length(list[k])<max do
        list[k]:=list[k]+' ';
  framewindow(x,y,x+max+3,y+num+1,text,fon);
  textbackground(fon);
  SetCursorSize($0100); {Убираем курсор с экрана}
  gotoxy(1,1);
  textbackground(marker);
  write(' '+list[1]);
  textbackground(fon);
  for k:=2 to num do begin
     writeln; write(' '+list[k]);
  end;
  k:=1;
  repeat
     c:=readkey;
     case c of
        up,down:
           begin
              gotoxy(1,k);
              textbackground(fon); write(' '+list[k]);
```

```
                if c=up then
                   if k>1 then k:=k-1
                   else k:=num
                else
                   if k<num then k:=k+1
                   else k:=1;
                gotoxy(1,k);
                textbackground(marker); write(' '+list[k]);
             end;
          esc: menu:=0;
          enter: menu:=k;
       end;
    until (c=esc) or (c=enter);
    SetCursorSize($0C0E); {Восстанавливаем курсор}
    window(1,1,80,25);   {Восстанавливаем полный экран в качестве окна}
    loadscreen;          {Восстанавливаем прежний вид экрана}
    textattr:=oldtext;   {Восстанавливаем прежние цветовые установки}
 end;
end.
```

П2.5.2. Модуль для подключения мыши к программе на Turbo Pascal

К сожалению, в Turbo Pascal нет стандартных средств для работы с компьютерной мышкой, к которой все так привыкли. Данный модуль восполняет этот недостаток, обращаясь к прерыванию DOS с номером $33.

Листинг П2.17. Модуль для работы с мышью

```
unit mouse;
interface
   uses dos;
   {Инициализация драйвера мыши}
procedure initmouse(var success:boolean; {true - драйвер
            успешно загружен, false - ошибка инициализации}
num:integer); {Число кнопок мыши}
   {Включение отображения курсора на экране}
   procedure showcursor;
   {Выключение отображения курсора на экране}
   procedure hidecursor;
```

```
 {Определение текущего состояния мыши}
 procedure mousecondition
   (var x,y:integer; {Координаты курсора}
    var left, {Состояние левой кнопки
               (true - нажата, false - отпущена)}
        right:boolean);{Состояние правой кнопки}
 {Установка курсора в точку экрана с координатами x и y}
 procedure putcursor(x,y:integer);
 {Определение количества нажатий левой кнопки после
 последнего вызова данной процедуры}
 function leftnum
   (var x,y:integer):word; {Координаты курсора при последнем нажатии}
 {То же самое для правой кнопки}
 function rightnum(var x,y:integer):word;
 {То же самое для средней кнопки (если она есть)}
 function midlenum(var x,y:integer):word;
 {implementation}
 var reg:registers;
 procedure initmouse;
begin
   reg.ax:=0; intr($33,reg);
   if reg.ax=1 then success:=true
   else success:=false;
   num:=reg.bx;
end;
 procedure showcursor;
 begin
   reg.ax:=1; intr($33,reg);
end;
 procedure hidecursor;
 begin
   reg.ax:=2; intr($33,reg);
end;
 procedure mousecondition;
begin
   reg.ax:=3; intr($33,reg);
   x:=reg.cx; y:=reg.dx;
   if reg.bx and 1=0 then left:=false
   else left:=true;
```

```
    if reg.bx and 2=0 then right:=false
    else right:=true;
  end;
  procedure putcursor;
  begin
    with reg do begin
      ax:=4; cx:=x;  dx:=y;
    end;
    intr($33,reg);
  end;
  {Вспомогательная (скрытая) процедура}
  function getnum(k:integer; {Номер кнопки (0, 1 или 2)}
              var x,y:integer):word; {Координаты курсора
                                      при последнем нажатии}
  begin
    reg.ax:=5; reg.bx:=k;
    intr($33,reg);
    x:=reg.cx; y:=reg.dx;
    getnum:=reg.bx;
  end;
  function leftnum;
  begin
    leftnum:=getnum(0,x,y);
  end;
  function midlenum;
  begin
    midlenum:=getnum(2,x,y);
  end;
  function rightnum;
  begin
    rightnum:=getnum(1,x,y);
  end;
end.
```

П2.5.3. Дополнение к модулю *graph* — вывод рисунков в формате BMP

Модуль `bmp` (см. листинг П2.3) позволяет выводить на графический экран рисунки в формате BMP (16 и 256 цветов). Напомним, что для нормального

вывода 256-цветного рисунка (или 16-цветного при палитре 256 цветов) нужен нестандартный графический драйвер. Нам известны два таких драйвера — Vesa256.bgi и Svga256.bgi, их легко найти в Internet, воспользовавшись любым поисковым сервером. Для подключения драйвера к программе удобно воспользоваться функцией `InstallUserDriver`.

Например:

```
Var gd,gm :integer;
begin
 gd:=installuserdriver('svga256',nil); initgraph(gd,gm,'');
```

Листинг П2.18. Модуль для работы с bmp-файлами

```
unit BMP;{Поддерживаемые форматы: BMP 16  цветов (не RLE)BMP 256 цветов
         (не RLE)}
interface
const
  {Коды ошибок, возвращаемых функцией ShowBitMapImage}
  BmpOk           =  0;  {OK}
  BmpUnknownError = -1;  {Неизвестная ошибка}
  BmpFileNotFound = -2;  {Файл не найден}
  BmpNotSuppMode  = -3;  {Текущий графический режим не поддерживается
                          для данного формата файла}
  BmpNotSuppFmt   = -4;  {Формат файла не поддерживается}
  {Цветовые составляющие}
  RED = 0;  GREEN = 1;  BLUE = 2;
{Вывод bmp-файла BMPFileName в позицию (X,Y). Дополнительный параметр
FirstDacReg используется для того, чтобы можно было вывести одновременно
несколько 16-цветных рисунков с разной палитрой (при 256-цветном режиме)
так, чтобы у каждого была своя палитра. Если используется обычный 16-
цветный режим или 256-цветный рисунок, задавайте нулевое значение}
  function ShowBitMapImage(BMPFileName : String; X, Y : Word;
                           FirstDacReg: Integer) : Integer;
implementation
uses  Dos, Graph;
type
  TPaletteEntry = record {тип для палитры}
    B,G,R: Byte; {голубой, зеленый, красный}
    Flags: Byte;
  end;
  TBitmapFileHeader = record {структура заголовка bmp-файла}
    bfType: Word;
```

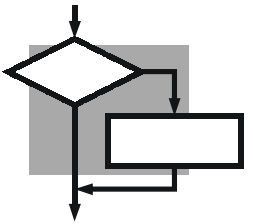
```
    bfSize: Longint;
    bfReserved1: Word;
    bfReserved2: Word;
    bfOffBits: Longint;
    biSize: Longint;
    biWidth, biHeight : Longint;
    biPlanes: Word;
    biBitCount: Word;
    biCompression: Longint;
    biSizeImage: Longint;
    biXPelsPerMeter, biYPelsPerMeter: Longint;
    biClrUsed, biClrImportant: Longint;
  end;
  TPalArray = array[0..255, RED..BLUE] of Byte;
var F : File; bfh : TBitmapFileHeader; {Заголовок bmp-файла}
  Pal : array[0..255] of TPaletteEntry; {Палитра}
function OpenFile(const BitmapName:String) : Boolean;
begin
  OpenFile := False; Assign(F, BitmapName);
  {$I-} Reset(F, 1); {$I+}
  if IoResult <> 0 then OpenFile:=False else OpenFile:=True;
end;
function CloseFile: Boolean;
begin
  {$I-} Close(F); {$I+}
  if IoResult <> 0 then CloseFile:=False else CloseFile:=True;
end;
function ReadBMPFileHead : Boolean;
begin
  ReadBMPFileHead := False;
  {$I-} BlockRead(F, bfh, SizeOf(bfh)); {$I+}
  if (IoResult <> 0) or (bfh.bfType <> $4D42) then Exit;
  ReadBMPFileHead := True;
end;
function ReadPalette(PalSize:Integer) : Boolean;
var C : Byte;
begin
  ReadPalette := False;
  {$I-} BlockRead(F, Pal, PalSize*4); {$I+}
```

```
  if IoResult <> 0 then Exit;
  ReadPalette := True;
end;
procedure SetPalette(PalSize: Integer; FirstDacReg: Integer);
var Palette : TPalArray; Reg : Registers; i : Byte;
begin
  if GetMaxColor>255 then exit;
  for i := 0 to PalSize-1 do begin
    Palette[i, RED]   := Pal[i].R SHR 2;
    Palette[i, GREEN] := Pal[i].G SHR 2;
    Palette[i, BLUE]  := Pal[i].B SHR 2;
  end;
  Reg.ah := $10; Reg.al := $12; { установка регистров палитры }
  Reg.bx := FirstDacReg;    { номер первого регистра }
  Reg.cx := PalSize;        { количество регистров }
  Reg.dx := Ofs(Palette);
  Reg.es := Seg(Palette);
  Intr($10, Reg);           { вызываем прерывание для установки палитры }
end;
{Вывод 16-цветного рисунка}
function ShowImage4(PalOffset:Integer; XStart,YStart:Word): Boolean;
var Px, C0, C1 : Byte;
  Lin4 : array[0..1023] of Byte;
  col: LongInt; Width, Height, xt, yt, W2 : Word;
begin
  ShowImage4 := False;
  Seek(F, bfh.bfOffBits);
  Width := bfh.biWidth; Height := bfh.biHeight;
  {Изменяем ширину, чтобы она стала кратна 8}
  while (Width mod 8)<>0 do inc(Width);
  W2 := (bfh.biWidth-1) div 2;
  for yt := Height-1 downto 0 do begin
    {$I-}  BlockRead(F, Lin4, Width div 2);  {$I+}
    if IOResult<>0 then exit;
     {выводим рисунок по отдельным пикселам}
     for xt := 0 to W2 do begin
       Px := Lin4[xt];  C0 := Px shr 4;
       Px := (Px shr 4) + (Px shl 4); C1 := Px shr 4;
       PutPixel(Xstart+xt*2, Ystart+yt, C0+PalOffset);
```

```
        PutPixel(Xstart+xt*2+1, Ystart+yt, C1+PalOffset);
      end
  end;
  ShowImage4 := True;
end;
{Вывод 256-цветного рисунка}
function ShowImage8(PalOffset:Integer; XStart,YStart:Word): Boolean;
type
  TLin8 = record
    X, Y : Word;
    Data : array[0..1023] of Byte;
  end;
var Lin8 : ^TLin8; i: Integer; l, col: Longint;
    Width, Height, xt, yt, SizeP  : Word;
begin
  ShowImage8 := False;
  Width := bfh.biWidth; Height := bfh.biHeight;
  {Изменяем ширину, чтобы она стала кратна 4}
  while (Width mod 4)<>0 do inc(Width);
  Seek(F, bfh.bfOffBits);
  SizeP := sizeof(TLin8); GetMem(Lin8, SizeP);
  Lin8^.X := bfh.biWidth-1; Lin8^.Y := 0;
  for yt := Height-1 downto 0 do begin
    {$I-}  BlockRead(F, Lin8^.Data, Width);  {$I+}
    if IOResult <> 0 then exit;
     {выводим рисунок по отдельным пикселам}
     for i:=0 to Width-1 do
       PutPixel(Xstart+i, Ystart+yt,Lin8^.Data[i]+PalOffset)
  end;
  FreeMem(Lin8, SizeP); ShowImage8 := True;
end;
function ShowBitMapImage(BMPFileName : String; X, Y : Word;
                         FirstDacReg:Integer) : Integer;
Var MaxC: LongInt; xt, yt : Word;
  begin
  MaxC := GetMaxColor;
  if (MaxC<>15)and(MaxC<>255) then begin
    ShowBitMapImage := BmpNotSuppMode;  exit;
  end;
```

```
  ShowBitMapImage := BmpFileNotFound;
  if not OpenFile(BMPFileName) then Exit;
  ShowBitMapImage := BmpNotSuppFmt;
  if not ReadBMPFileHead then Exit;
  ShowBitMapImage := BmpNotSuppMode;
  case bfh.biBitCount of
    4 : begin
          ReadPalette(16);
          SetPalette(16,FirstDacReg);
          ShowImage4(FirstDacReg,X,Y);
          ShowBitMapImage := BmpOk;
        end;
    8 : begin
          ReadPalette(256);
          SetPalette(256,FirstDacReg);
          ShowImage8(FirstDacReg,X,Y);
          ShowBitMapImage := BmpOk;
        end;
  end;
  CloseFile;
end;
end.
```

Приложение 3



Основы практической работы в интегрированной среде Turbo Pascal

П3.1. Работа в окне интегрированной среды, текстовый редактор Turbo Pascal

П3.1.1. Общие сведения

Разработка программы на Turbo Pascal предусматривает:

- набор и редактирование исходного текста программы;
- его компиляцию;
- запуск программы на выполнение;
- поиск ошибок.

Все эти операции выполняются при помощи *интегрированной среды разработки Turbo Pascal*. Среда называется интегрированной, потому что она *объединяет* соответствующие программы: *текстовый редактор, компилятор, отладчик и справочную систему*, обеспечивая комфортные условия работы программиста.

В комплект стандартной поставки Turbo Pascal 7.0 входят два варианта интегрированной среды для операционной системы DOS: Borland Pascal (файл Bp.exe) и Turbo Pascal (файл Turbo.exe). Внешние различия между ними — минимальны, однако среда Borland Pascal обладает бóльшими возможностями. В основном тексте книги, там, где это имеет значение, особенности Borland Pascal оговаривались отдельно. В данном приложении, говоря о характеристиках среды, мы будем иметь в виду Turbo Pascal.

Обратите внимание — размер программы в Turbo Pascal ограничен: редактор текстов и компилятор позволяют обрабатывать программы и библиотечные модули размером до 64 Кбайт.

П3.1.2. Начало работы

Интегрированная среда запускается следующей командой:

```
turbo [необязательное имя программы (файла)]
```

После этого на экране появляется окно, состоящее из нескольких частей: строки меню в верхней части экрана, рабочей области в центре и строки состояния внизу.

Строка меню предоставляет доступ к командам среды и может быть активной, если имеется возможность работы с командами, или неактивной, если редактируется текст программы. Меню и прочие компоненты среды являются невидимыми, если идет выполнение программы или просматриваются результаты ее работы, выведенные на экран. Для выбора строки меню и ее команд используются клавиша <F10>, левая кнопка мыши или совместное нажатие клавиши <Alt> и выделенной буквы в названии пункта, а также клавиши управления курсором. Каждому пункту основного меню соответствует группа команд, с которыми связаны подпункты меню. Если за командой меню следует знак многоточия (...), то выбор этой команды приведет к выводу диалогового окна. Если за командой следует значок >, то эта команда вызывает вложенное меню.

Строка статуса находится в нижней части экрана и содержит напоминание о назначении основных комбинаций клавиш.

Рабочая область в центре экрана предназначена для набора текста программы.

П3.1.3. Создание новой программы

После запуска интегрированной среды на экране должно появиться пустое активное окно редактирования. Если окно не пустое, то с помощью команды **File | New** следует открыть окно ввода *нового* текста. В верхней части окна редактирования в этом случае появится название, которое среда автоматически присвоит новому файлу — **NONAME00.PAS** ("безымянный" файл). После набора текста программы его надо изменить, иначе набранный текст может быть впоследствии замещен другим файлом с таким же стандартным именем.

Для предотвращения возможной аварийной потери данных, например, в случае перезагрузки компьютера, после набора очередных 10—15 строк текста рекомендуется *сохранить* файл программы, нажав на клавишу <F2>. При первой записи файла на диск компьютера система предложит задать его имя (не более 8 символов), при этом расширение pas добавится автоматически.

Перед первым сохранением рекомендуется выполнить команду **File | Change dir** для *смены* текущего каталога. Это позволит выбрать папку для хранения собственных файлов, отличную от стандартной (символ . .\ при поиске обозначает "подняться в вышестоящую папку").

При каждом сохранении файла его *предыдущая* версия остается на диске с расширением bak. Это позволит, в случае необходимости, восстановить "старый" текст программы, открыв указанный файл.

В том случае, если требуется создать *копию* рабочего файла, ее можно сохранить под другим именем по команде **File | Save as**.

П3.1.4. Набор и редактирование текста

Выполняется через пункт меню **Edit**.

Обратите внимание — внесение исправлений в текст программы можно ускорить, используя работу с *фрагментами текста* (копирование, перенос или удаление). Предварительно интересующий фрагмент выделяют мышью или при помощи клавиатуры.

Наиболее важными являются следующие функции редактора:

- **Undo** — отмена предыдущего действия, от которого вы решили отказаться;
- **Redo** — его восстановление при необходимости;
- **Show clipboard** — просмотр содержимого *буфера обмена* (вспомогательного файла Turbo Pascal, используемого для временного хранения перемещаемых или копируемых фрагментов текста);
- **Cut**, **Copy**, **Paste** — удаление, копирование в буфер и вставка текста из буфера;
- **Clear** — удаление выбранного текста (без помещения в буфер обмена).

Ускорить работу программиста при наборе текста программы и быстро обратиться к выбранным пунктам меню позволят клавиатурные комбинации — одновременные нажатия групп клавиш (табл. П3.1).

Таблица П3.1. *Комбинации клавиш для редактирования*

Клавиши	Команда меню	Функция
<Shift>+<^>, <v>,<>>,<<>	—	Помечает фрагмент текста
<Ctrl>+<Del>	**Edit | Clear**	Удаление выбранного текста
<Ctrl>+<Ins>	**Edit | Copy**	Копирование выбранного текста в буфер
<Shift>+<Del>	**Edit | Cut**	Перемещение выбранного текста в буфер
<Shift>+<Ins>	**Edit | Paste**	Запись текста из буфера в активное окно
<Ctrl>+<L>	**Search | Search Again**	Повторяет последнюю команду **Find** (Поиск)
<F2>	**File | Save**	Сохранение файла из активного окна редактора
<F3>	**File | Open**	Открытие файла

Если какие-либо команды меню в данный момент времени недоступны (их названия выводятся серым цветом), это значит, что использование команды в данной ситуации лишено смысла.

Запуск команды меню после ее выбора производится нажатием клавиши <Enter>. *Прервать* выполняемое действие можно клавишей <Esc>. Для того чтобы *прервать* выполнение программы или попытаться остановить ее в случае "зацикливания", используется одновременное нажатие клавиш <Ctrl>+ +<Break>.

При необходимости *завершить работу с файлом* его можно закрыть, выбрав команду меню **Window | Close** или нажав клавиши <Alt>+<F3>. Если с момента предыдущей записи на диск файл редактировался, последует приглашение сохранить изменения.

Для того чтобы вновь открыть существующий файл для продолжения работы с ним, необходимо выбрать **File | Open** или нажать <F3>. По умолчанию среда выведет шаблон *.pas и будет показывать только файлы Turbo Pascal (с расширением pas). Для просмотра всех файлов текущего каталога шаблон необходимо заменить на *.*.

П3.1.5. Работа с окнами

Редактор Turbo Pascal является *многооконным*. Это значит, что программист может одновременно работать с несколькими файлами программ. Каждый новый файл будет открываться в своем окне. Однако в любой момент времени вы может работать только с одним окном — *активным*. Именно к нему будут относиться выбранные команды меню. Допускается *межоконный перенос* фрагментов текста программ через буфер обмена. В табл. П3.2 приведены комбинации клавиш для управления окнами.

Таблица П3.2. *Комбинации клавиш для управления окнами*

Клавиши	Команда меню	Функция
<Alt>+<цифра>	—	Переход к окну с заданным номером
<Alt>+<0>	**Window | List...**	Список открытых окон
<Alt>+<F3>	**Window | Close**	Закрытие активного окна
<Alt>+<F5>	**Debug | User screen**	Показ экрана пользователя с результатами работы программы
<Alt>+<X>	**File | Exit**	Выход из Turbo Pascal
<F6>	**Window | Next**	Переход к следующему окну в списке
<Shift>+<F6>	**Window | Previous**	Переход к предыдущему окну

Таблица П3.2 *(окончание)*

Клавиши	Команда меню	Функция
<F5>	**Window | Zoom**	Активное окно на весь экран или обратно
<Ctrl>+<F5>	**Window | Size move**	Изменения размеров или перемещение активного окна

Обратите внимание — в случае возникновения затруднительных ситуаций можно попытаться воспользоваться клавишами <Esc> или <Ctrl>+<Break>. Для выхода из системы используется команда **File | Exit** (<Alt>+<X>).

П3.2. Справочная система

Основные возможности редактора, использование клавиатуры и специальных комбинаций клавиш подробно описаны в справочной системе Turbo Pascal.

Справочная система является *контекстно-зависимой*. Это означает, что выводимая при обращении к ней информация зависит от того, где в момент обращения находится курсор. Иначе говоря, если программисту потребуется получить справку по пункту меню, он должен выбрать этот пункт, но вместо нажатия <Enter> использовать <F1>. При нажатии клавиш <Ctrl>+<F1> выводится справочная информация, относящаяся к объекту программы в окне редактирования, на который в данный момент времени указывает курсор.

Значительным достоинством справочной системы является возможность скопировать приведенные там примеры в окно редактирования для практического изучения.

Самый быстрый способ вызова справочной системы — использование специальных клавиш или их комбинаций (табл. П3.3).

Таблица П3.3. *Комбинации клавиш справочной системы*

Клавиши	Команда меню	Функция
<F1>	**Help | Contents**	Открывает экран со справочной информацией
<F1>,<F1>	**Help | Using help**	Вызов справки по справочной системе
<Shift>+<F1>	**Help | Index**	Вызов оглавления справочной системы
<Alt>+<F1>	**Help | Previous Topic**	Показать предыдущий экран справки
<Ctrl>+<F1>	**Help | Topic Search**	Вызов контекстной информации по языку Pascal (например, по имени стандартной функции)

П3.3. Компиляция программы, поиск и устранение ошибок

После завершения набора программы и ее записи на диск можно приступать к компиляции — клавиша <F9> или <Alt>+<F9>. В процессе своей работы компилятор может обнаружить синтаксические ошибки, связанные с неправильным употреблением различных элементов и конструкций языка в тексте программы. Их причинами является недостаточное знание правил языка и опечатки.

Сообщение об ошибке выводится в верхней части экрана и выделяется красным цветом. Курсор при этом обычно устанавливается в той строке программы, где обнаружена ошибка. Для исправления ошибки программист должен проанализировать сообщение, определить ее подлинную причину и внести изменение в текст программы. После этого программа вновь компилируется или сразу запускается на выполнение, и так далее, до тех пор, пока автор не получит устраивающий его результат.

Замечание

Одна из наиболее распространенных ошибок, которую делают новички по невнимательности, — забывают поставить точку с запятой в конце оператора. В этом случае курсор помещается на первый символ следующего оператора, который, возможно, располагается в следующей строке, а на экран выводится сообщение об ошибке **Error 85: ";" expected** (Ошибка 85: Пропущен ";").

П3.4. Запуск программы на выполнение, просмотр результатов, отладка

После исправления всех синтаксических ошибок программа может быть запущена на выполнение (<Ctrl>+<F9>). Если в программе не предусмотрена приостановка в ходе ее работы для просмотра результатов (при помощи, например, оператора `readln;`), выведенные на экран данные перекрываются окном среды. Временно убрать это окно и просмотреть результаты можно, нажав <Alt>+<F5>.

Нет необходимости постоянно иметь под рукой Turbo Pascal для того, чтобы выполнять написанные на нем программы. После того как программа полностью отлажена, ее можно записать на диск в виде исполняемого файла с расширением exe. Такой файл может быть запущен из операционной системы. Для создания исполняемого файла перед запуском программы на выполнение необходимо выбрать команду меню **Compile | Destination** и, нажав <Enter>, заменить **Memory** на **Disk**.

Часто оказывается, что синтаксически правильная программа завершает свое выполнение аварийно, с сообщением об ошибке выполнения. Кроме того, программа может по непонятной причине зациклиться. Логические

(семантические) ошибки наиболее трудны для исправления. В этом случае прибегают к помощи *отладчика*. Выполняется *трассировка* программы (выполнение по шагам) и отслеживание значений переменных в специальном окне на разных шагах выполнения алгоритма (табл. П3.4).

Таблица П3.4. *Комбинации клавиш отладки*

Клавиши	Команда меню	Функция
<F4>	**Run | Go to cursor**	Выполнение программы до позиции курсора
<F7>	**Run | Trace into**	Выполнение программы по шагам
<F8>	**Run | Step over**	Выполнение программы по шагам без захода в процедуры (процедура выполняется за один шаг)
<Ctrl>+<F2>	**Run | Program reset**	Завершает выполнение программы
<Ctrl>+<F4>	**Debug | Evaluate/modify...**	Просмотр значений переменных по окончании программы. Курсор ставят на имя переменной и нажимают <Ctrl>+<F4>
<Ctrl>+<F7>	**Debug | Add watch...**	Просмотр значения переменной, на которую помещен курсор, в окне просмотра (при выполнении программы)

П3.4.1. Пример работы с отладчик ом

Пусть при выполнении программы

```
var a,b: integer; c: real;
begin
   readln(a, b);
   c:=b/a;
   writeln('b/a = ',c);
end.
```

были введены числа 0 и 1. В этом случае произойдет завершение программы по ошибке с кодом 200, а курсор укажет на оператор `c:=b/a;`. Рассмотрим на этом примере, как можно пользоваться отладчиком:

1. Выбрать в подменю **Run** команду **Trace into** (клавиша <F7>). Если необходима перетрансляция программы, система сделает это автоматически. В результате запуска команды **Trace into** инициируется режим трассировки. Первая строка `begin` выделится подсветкой. Подсвеченная строка — это текущая строка выполнения, т. е. строка, которая будет выполняться следующей.

2. Второе нажатие <F7> приводит к переходу к следующей строке и помечает следующий оператор `readln`, и т. д. Таким образом, каждое нажатие <F7> производит очередной шаг выполнения программы.

   При вхождении в режим трассировки экран среды состоит из двух окон: **Edit** (окно редактора) и **Watches** (окно просмотра). Программист находится в **Edit**. По мере продвижения по программе отладчик будет время от времени переключать экран **Edit** на экран **Output** (экран выполнения), например, когда программа потребует задать два числа для ввода, а затем снова возвращаться в **Edit** и ждать следующей команды.

3. Для просмотра текущих значений переменных программы (в окне **Watches**) надо произвести следующие действия:

   - нажать <Ctrl>+<F7>. В середине экрана появится рамка **Add Watch**. То, что будет изображено в этой рамке, зависит от места положения курсора при нажатии <Ctrl>+<F7>. В рамке автоматически появляется имя переменной, на которую указывает курсор;
   - набрать имя `a` и нажать <Enter>: добавить переменную `a` в окно **Watches**. В результате в окне **Watches**, в нижней части экрана, появится строка:

     ```
     a: 0
     ```

   - установить курсор в программе на переменную `c` и нажать <Ctrl>+<F7> и <Enter>. В окне будем иметь:

     ```
     c: 0.0
     a: 0
     ```

   - нажать <Ctrl>+<F7>, набрать `b*a` и нажать <Enter>. В окно **Watches** добавляется значение `b*a`:

     ```
     b*a: 0
     c: 0.0
     a: 0
     ```

   При этом размеры окна увеличиваются. По мере продвижения по программе в окне **Watches** будут отображаться текущие значения просматриваемых переменных и выражений.

4. Когда пошаговое выполнение программы в результате нажатия <F7> дойдет до оператора `c:=b/a;`, появится фраза **Error 200: Division by zero** (Ошибка 200: Деление на ноль).

5. Для завершения режима отладки надо нажать клавиши <Ctrl>+<F2> (команда **Run | Program reset**).

Обратите внимание — для пошагового исполнения программы кроме клавиши <F7>, используется еще клавиша <F8>, которая выполняет операторы вызова процедур и функций как единое целое.

П3.5. Перечень ошибок

В ходе своей работы программист неизбежно сталкивается с ошибками *(см. разд. 3.3)*. Обнаружение ошибок компиляции сопровождается выдачей краткого комментария, позволяющего сравнительно быстро понять причину и исправить синтаксическую ошибку. В отличие от них, ошибочные операции в процессе выполнения программы обычно вызывают сообщение, содержащее числовой код ошибки и адрес ячейки памяти. Для того чтобы облегчить работу с ошибками выполнения, программисту необходимо знать описания их кодов (табл. П3.5).

Таблица П3.5. *Коды ошибок выполнения*

Код	Описание
1	Обращение к несуществующей функции MS-DOS
2	Указанный файл не найден при обращении к нему процедур `Reset`, `Append`, `Rename`, `Erase`
3	Указанный путь не найден при попытке его использования процедурами `Reset`, `Rewrite`, `Append`, `Rename`, `Erase`, `ChDir`, `MkDir`, `RmDir`
4	Число открытых файлов превысило значение, заданное переменной `FILES` в стартовом файле Config.sys. Источник сообщения: процедуры `Reset`, `Rewrite`, `Append`
5	Доступ к файлу запрещен. Причина ошибки может заключаться в попытке открыть файл, имеющий атрибут `Read Only` ("только для чтения"), как для записи, так и для чтения. Источник ошибки: процедуры `Reset`, `Append`, `Rewrite`, `Rename`, `Erase`, `MkDir`, `RmDir`, `Read`, `BlockRead`, `Write`, `BlockWrite`
6	Ошибка обработки файла
12	Неправильный код доступа к файлу. Возникает, если в процедурах `Reset` или `Append` при работе с типизированными и нетипизированными файлами значение параметра `FileMode` задано неправильно
15	Неправильный номер устройства в процедуре `GetDir`
16	Нельзя удалить текущий каталог. Процедура `RmDir` не может удалить текущий каталог
17	При обращении к процедуре `Rename` сделана попытка присвоить имя, отвечающее файлу на другом устройстве
18	При обращении к `FindFirst` или `FindNext` оказалось, что соответствующих файлов нет
100	Попытка чтения из типизированного файла после того, как достигнут признак конца файла
101	Ошибка записи на диск (на диске не хватает места)
102	Файл не ассоциирован с файловой переменной

Таблица П3.5 *(окончание)*

Код	Описание
103	Файл не был открыт при обращении к нему процедур `BlockRead`, `BlockWrite`, `Eof`, `Close`, `FilePos`, `FileSize`, `Flush`, `Read`, `Write`, `Seek`
104	Файл не был открыт для чтения при обращении к нему процедур `Eof`, `EoLn`, `Read`, `ReadLn`, `SeekEof`, `SeekEoln`
105	Файл не был открыт для записи при обращении к нему процедур `Write` и `WriteLn`
106	Неправильный числовой формат (при вводе встретился недопустимый в записи числа символ)
150	Диск защищен от записи
151	Ошибка драйвера устройства
152	Устройство не готово
154	Ошибка контроля четности данных
156	Ошибка поиска данных на диске
157	Неизвестный тип носителя информации
158	Не найден сектор
159	В принтере нет бумаги
160	Ошибка записи на устройство
161	Ошибка чтения с устройства
162	Сбой оборудования
200	Деление на ноль (некорректная арифметическая операция)
201	Нарушение диапазона допустимых значений переменной или индекса массива
202	Переполнение стека
203	Переполнение динамически распределяемой области памяти
204	Неправильная операция с указателем
205	Переполнение при выполнении операции с плавающей точкой
206	Исчезновение порядка при выполнении операции с плавающей точкой
207	Ошибка при выполнении операции с плавающей точкой. Возникает при попытке вычислить значение функций `sqrt(x)` или `ln(x)` из отрицательного значения в случае, когда преобразование вещественного значения в целое не попадает в допустимый интервал значений для типа `longint` и т. д.
210	Объект не инициализирован. Возникает при обращении к виртуальному методу до инициализации объекта конструктора
215	Арифметическое переполнение
216	Ошибка доступа к памяти

Список литературы

1. Абрамов С. А. Начала информатики / С. А. Абрамов, Е. В. Зима — М.: Наука, Гл. ред. физ.-мат. лит., 1989. — 256 с.
2. Алексеев В. Е. Вычислительная техника и программирование. Практикум по программированию: Практ. пособие / В. Е. Алексеев, А. С. Ваулин, Г. Б. Петрова. — Под ред. А. В. Петрова. — М.: Высш. шк., 1991. — 400 с.: ил.
3. Алкок Д. Язык Паскаль в иллюстрациях: Пер. с англ. — М.: Мир, 1991. — 192 с.: ил.
4. Ахо А. Структуры данных и алгоритмы / А. Ахо, Д. Хопкрофт, Д. Ульман: Пер. с англ. — М.: Издательский дом "Вильямс", 2000. — 384 с.: ил.
5. Бородич Ю. С. Паскаль для персональных компьютеров: Справ. пособие / Ю. С. Бородич, А. Н. Вальвачев, А. И. Кузьмич — Мн.: Высш. шк.: БФ ГИТМП "НИКА", 1991. — 365 с.: ил.
6. Вирт Н. Алгоритмы + структуры данных = программы / Н. Вирт: Пер. с англ. — М.: Мир, 1985. — 406 с.
7. Вирт Н. Алгоритмы и структуры данных / Н. Вирт: Пер. с англ. — М.: Мир, 1989. — 360 с.: ил.
8. Дагене В. А. Сто задач по программированию: Кн. Для учащихся: Пер. с лит. / В. А. Дагене, Г. К. Григас, К. Ф. Аугутис. — М.: Просвещение, 1993. — 255 с.: ил.
9. Йенсен К. Руководство для пользователя и описание языка Паскаль / К. Йенсен, Н. Вирт: Пер. с англ. — М.: Финансы и статистика, 1982. — 150 с.
10. Керниган Б. Практика программирования / Б. Керниган, Р. Пайк; Пер. с англ. — СПб.: Невский Диалект, 2001. — 381 с.: ил.
11. Кнут Д. Искусство программирования. Том 3: Поиск и сортировка / Д. Кнут; Пер. с англ. — М.: Издательский дом "Вильямс", 2000. — 584 с.: ил.
12. Культин Н. Б. Программирование в Turbo Pascal 7. 0 и Delphi. 2-е изд., перераб. и доп. / Н. Б. Культин — СПб.: БХВ — Санкт-Петербург, 1999. — 416 с.: ил.
13. Культин Н. Б. Turbo Pascal в задачах и примерах / Н. Б. Культин. — СПб.: БХВ — Санкт-Петербург, 2000. — 256 с.: ил.
14. Мизрохи С. В. TURBO PASCAL и объектно-ориентированное программирование / С. В. Мизрохи. — М.: Финансы и статистика, 1992. — 192 с.: ил.

15. Немнюгин С. А. Turbo Pascal: практикум / С. А. Немнюгин. — СПб: Питер, 2001. — 256 с.: ил.

16. Окулов С. Программирование в алгоритмах / С. Окулов. — М.: "Бином", 2002. — 344 с.

17. Офицеров Д. В. Программирование на персональных ЭВМ: Практикум: Учеб. пособие / Д. В. Офицеров, А. Б. Долгий, В. А. Старых. — Под общ. ред. Д. В. Офицерова. — Мн.: Высш. шк., 1993. — 256 с.

18. Перминов О. Н. Программирование на языке Паскаль / О. Н. Перминов. — М.: Радио и связь, 1988. — 224 с.: ил.

19. Попов В. Б. Turbo Pascal для школьников. Версия 7.0 / В. Б. Попов. — М.: Финансы и статистика, 1996. — 446 с.: ил.

20. Себеста Р. Основные концепции языков программирования, 5-е изд. / Р. Себеста: Пер. с англ. — М.: Издательский дом "Вильямс", 2001. — 672 с.: ил.

21. Ставровский А. Турбо Паскаль 7.0. Учебник / А. Б. Ставровский — К.: Издательская группа BHV, 2000. — 400 с.

22. Стивенс Р. Delphi. Готовые алгоритмы / Р. Стивенс: Пер. с англ. — М.: ДМК Пресс, 2001. — 384 с.: ил.

23. Фаронов В. В. Программирование на персональных ЭВМ в среде Турбо Паскаль. 2-е изд. / В. В. Фаронов. — М.: Изд-во МГТУ, 1992. — 448 с.

24. Фаронов В. В. Delphi: Программирование на языке высокого уровня / В. В. Фаронов. — СПб.: Питер, 2003. — 640 с.

Предметный указатель

A

Ada 19
Algol 17

B

Basic 17

C

C++ 18
C 18

F, P

Fortran 17
Pascal 17

А

Абстрактные типы данных 369
Автоматические переменные 169
Автоматный распознаватель 149
Алгоритм 7
Алгоритмизация 7
Алфавит языка 64
Анонимный тип 222
Арифметические:
  выражения 85
  операции 85
  функции 91
Арифметическое:
  И (and) 88
  отрицание (not) 89
Ассемблер 15

Б

Байт 40
Бесконечные суммы 141
Библиотечные модули 203
Бинарный поиск 241
Бит 40
Блок-схема 9
Буфер:
  клавиатуры 81, 422
  файла 312

В

Ввод данных 81
Вектор 221
Ветвления 13
Видеопамять 414
Вложенные циклы 137
Вложенный оператор if 109
Возведение в степень 97
Встроенные процедуры и функции 158
Вывод данных 82
Выражения 20

Г

Глобальные объекты 169
Графический драйвер 426

Д

Двумерный массив 222
Дерево 360
Диалекты языка программирования 24
Динамическая память:
  куча 340
Динамические переменные 340
Директивы:
  {$I−} 127
  {$Q+} 96
  {$R+} 96

(окончание рубрики см. на с. 468)

Директива *(окончание)*:
компилятора 69
Доступ к памяти и портам 342

Ж

Жизненный цикл программы 27

З

Заголовок 69
Заказной программный продукт 25
Записи с вариантами 299
Запись 291
Зарезервированные:
константы 73
слова 65
Зацикливание 124

И

Инверсия 89
Индекс 221
Индексированные переменные 221
Инициализация переменных 73
Интерпретатор 30
Интерпретация 30
Исключительные ситуации 33, 405
Исполнительная часть 70

К

Каталог 307
Код:
ASCII 422
сканирования 422
Кодирование 28
Кодовая таблица 56
Компилятор 30
Компиляция 30
Компоновщик 32
Константа 41
Косвенная рекурсия 176
Кроссплатформенность 31

Л

Лексемы 19
Линейная:
программа 13
сортировка 240
Листинг 8
Литерал 257
Логические:
выражения 105
устройства 310
файлы 311
Логический (булевский) тип 56
Логическое сложение (or) 88
Локальные объекты 168

М

Массивы 221
Метасимволы 22
Метаязык 21
Метод быстрой сортировки
с разделением 240
Метод нисходящего проектирования
программ 194
Методы структурного
программирования 193
Множество 281
Модуль:
printer 215
system 214

Н

Нетерминалы 22
Нетипизированный параметр 179

О

Обработка ошибок ввода/вывода 127
Обратное суммирование 143
Обход 13
Объектный код 32, 203
Операнды 85
Операторные скобки 107
Операторы 20
case 112
for 132
goto 114
read и readln 80
repeat 124
while 129
write и writeln 82
вызова процедуры 158
присваивания 84

Операции:
  div и mod 87
  отношения 104
Опережающее описание 176
Описательная часть 70
Открытые:
  массивы 247
  строки 348
Отладка 28
Отладчик 34
Ошибки:
  ввода/вывода 82
  выполнения 33
  синтаксические 15

П

Палитра 415
Перегрузка подпрограмм 408
Перезагрузка компьютера 124
Переменная 41
Пиксел 414
Побитовые операции 88
Полный перебор 146
Преобразование типа 94
Приоритет операций 90
Проверка на кратность 88
Программа 25
Простые операторы 76
Процедурные типы и переменные 177
Процедуры 158
  append 317
  assign 313
  break 135
  close 314
  continue 135
  dec 85
  delay 421
  delete 265
  dispose 345
  erase 314
  exec 442
  freemem 345
  getmem 344
  inc 85
  initgraph 427
  insert 265
  mark release 345
  new 344
  readreadln 318
  rename 314
  reset 313
  rewrite 313
  seek 328
  sound nosound 424
  str 267
  truncate 328
  val 267
  window 419
  writewriteln 318
Процедуры и функции пользователя 158
Псевдоязык 9
Пустой оператор 115
Путь 309

Р

Раздел:
  констант const 72
  меток label 71
  модулей uses 71
  операторов 76
  переменных var 74
  процедур и функций 75
  типов данных type 74
Разделители 68
Разделы программы 70
Раздельная компиляция 204
Разрешающая способность 415
Рекурсия 174

С

Связанный список 351
Сдвиг:
  влево (k shl n) 89
  вправо (shr) 89
Семантика 20
Семантические ошибки 33
Символы 19
Символьный (литерный) тип 56
Символьный тип 256
  стандартные функции 256
Синтаксис 20
Синтаксические:
  диаграммы 22
  ошибки 33
Система:
  программирования 34
  счисления 56
Системная дата и время 441

Скалярные типы 48
Следование 13
Сложение по модулю 289
Сложные операторы 76
Сортировка 237
  методом "пузырька" 240
Составной оператор 107
Составные символы 64
Специальные (служебные) символы 57
Специальные символы 64
Ссылочная модель строки 349
Стандартные:
  идентификаторы 66
  типы 48
Статические переменные 171
Строка 257
Строковый тип 56, 256
Структура программы 70
Структурированные типы 49
Структурное кодирование 195
Структуры данных 369
Сцепление строк (конкатенация) 261
Счетчик 132

Т

Текстовый:
  видеорежим 415
  файл 316
Терминальные символы (терминалы) 22
Тестирование 28
Тип 42
Типизированные константы 72
Типизированный файл 327
Тиражный программный продукт 25
Транслятор 15
Транспонирование матрицы 236
Трассировка программы 34

У

Указатели 341
Указатель позиции файла 311
Умножение матриц 245
Унарные и бинарные операции 86
Условный оператор if 108

Ф

Файл 306
  имя и расширение 307
  последовательного доступа 311
  прямого доступа 312
Файловая переменная 311
Факториал 142
Фибоначчи (ряд числа) 141
Формат вывода 83
Формулы 97
Формы Бэкуса—Наура 22
Функции:
  trunc и round 94
  concat 266
  copy 266
  eof 314
  eoln 151
  filepos 328
  filesize 328
  frac 92
  hi 92
  int 92
  IOResult 127
  keypressed 423
  length 266
  lo 93
  maxavail 345
  memavail 345
  ord 92
  pos 266
  pred 92
  random 92
  readkey 423
  seekeof 319
  sizeof 93
  succ 92
  swap 93
  пользователя 165

Х

Ханойские башни 188
Характеристики программного продукта 26

Ц

Цикл 13, 123

Я

Язык программирования 14
  высокого уровня 15
  низкого уровня 16