Г. Г. Рапаков, С. Ю. Ржеуцкая

bhv®

Turbo Pascal для студентов и школьников

Алгоритмизация и программирование

Основные конструкции языка

Примеры программ

ОСНОВЫ ИНФОРМАТИКИ

Георгий Рапаков
Светлана Ржеуцкая

Turbo Pascal для студентов и школьников

Санкт-Петербург
«БХВ-Петербург»
2002

УДК 681.3.06(075.3+075.8)
ББК 32.973.26-018.1
Р23

Рапаков Г. Г., Ржеуцкая С. Ю.

Р23 Turbo Pascal для студентов и школьников. — СПб.: БХВ-Петербург, 2002. — 352 с.: ил.

ISBN 5-94157-240-9

Книга является обобщением многолетнего опыта авторов по обучению студентов и школьников старших классов основам программирования. Тематику ее можно определить как "Конспект начинающего программиста" или "Популярный учебник". Изложение материала построено таким образом, чтобы читатель сумел понять основные принципы разработки компьютерных программ, не увязнув в многочисленных тонкостях языка Turbo Pascal. В книге содержатся краткие сведения об алгоритмизации и программировании, операторах, процедурах, функциях и модулях, приводятся описания основных конструкций языка, иллюстрированные большим количеством примеров, рассмотрены основные приемы программирования и практической работы в интегрированной среде Turbo Pascal.

Для школьников старших классов и студентов

УДК 681.3.06(075.3+075.8)
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Людмила Еремеевская*
Зав. редакцией	*Анна Кузьмина*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.08.02.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 28,38.
Тираж 5000 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 5-94157-240-9

Содержание

Введение

Несмотря на появление новых технологий *Turbo Pascal* (Турбо Паскаль), во многом задуманный как язык для обучения, и на сегодняшний день остается одним из самых удобных средств для изучения программирования. Это определяет его популярность среди широкой аудитории начинающих программистов: школьников и студентов.

Целью издания, которое держит в руках читатель, является оказание помощи в изучении основ программирования.

В книге содержатся краткие сведения об алгоритмизации и программировании; приводятся описания основных конструкций Turbo Pascal, иллюстрированные большим количеством примеров; рассмотрены основные приемы программирования и практической работы в интегрированной среде Turbo Pascal.

Книга, сочетающая свойства справочника и конспекта, рассчитана на школьников старших классов и студентов. Две части книги отвечают первому и второму семестрам (полугодиям) обучения студентов (школьников) программированию. *Тематику книги* можно определить как "Конспект начинающего программиста" или "Популярный учебник".

Объективную оценку книги может 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 собственный опыт авторов и почерпнутый ими из многочисленных публикаций;
- отсутствия "наукообразности", с одной стороны, и "заигрывания с читателем", с другой: *деловой подход к получению максимума сведений в минимальные сроки*;

- *привлечения* внимания читателя к наиболее важным материалам в каждой теме;
- *большого количества подобранных примеров*, облегчающих обучение, не утомляющих читателя и стимулирующих у него интерес к материалу;
- наличия *комментариев* к текстам программ.

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

ЧАСТЬ I

Основы программирования на Turbo Pascal: путь от простейших задач до разработки собственных подпрограмм

ГЛАВА 1

Основы алгоритмизации

1.1. Алгоритмизация и требования к алгоритму

1.1.1. Алгоритм и алгоритмизация

Процессор электронно-вычислительной машины (ЭВМ) или персонального компьютера (ПК) — это ее "мозг", который умеет выполнять лишь простейшие команды. Для решения сложных задач обработки информации программист должен составить *алгоритм* — подробное описание последовательности арифметических и логических действий, расположенных в строгом логическом порядке и позволяющих 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.

Требования, предъявляемые к алгоритму:

- *однозначность* — предлагаемые действия должны быть "понятны" компьютеру, а порядок исполнения этих действий должен быть единственно возможным, любая неопределенность или двусмысленность недопустимы;
- *массовость* — пригодность алгоритма для решения не только данной задачи, а 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;
- *эффективность* — для успешного решения задачи должны использоваться ограниченные ресурсы конкретного компьютера (время работы процессора, объем оперативной памяти, быстродействие жесткого диска и др.).

1.2. Блок-схемы алгоритмов

1.2.1. Способы записи алгоритма

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 программы по имеющемуся алгоритму — другой. Следовательно, необходимо иметь такие способы записи алгоритмов, которые легко воспринимаются человеком, но являются достаточно строгими, чтобы их можно было впоследствии перевести на язык компьютера.

Существуют различные варианты записи алгоритмов. К основным относятся описательный и графический способы. *Описательным* называется алгоритм, составленный на естественном, в частности, математическом языке. *Графический* способ отличает компактная и наглядная форма записи в виде специальных графических знаков с указанием связи между ними.

1.2.2. Блок-схемы

При разработке программ рекомендуется использовать графический способ записи алгоритма в виде блок-схемы. *Блок-схема* — это графическое изображение алгоритма в виде плоских геометрических фигур (блоков), соединенных линиями. Внутри блока записывается действие, которое нужно выполнить, или условие, которое необходимо проверить. Блок-схема — стандартный способ записи алгоритма, существует государственный стандарт (ГОСТ), содержащий перечень правил построения блок-схем.

Основные блоки, которые используются при составлении графического алгоритма, изображены на рис. 1.1, где *а* — начало (конец) алгоритма; *б* — блок ввода/вывода; *в* — операционный блок; *г* — логический (условный) блок; *д* — цикл с параметром (для параметра цикла `i` указывается его начальное и конечное значение, шаг равен единице). Конструкция "цикл с параметром" отдельно ГОСТом не регламентируется, однако приведенное обозначение (шестиугольник) разрешено использовать для различных целей, поэтому такое изображение цикла на блок-схеме часто встречается в литературе по программированию. Цикл с параметром будет подробно рассматриваться далее *(см. разд. 3.9.3)*. В ГОСТ 19.701-90 (ИСО 5807-85), дата введения 01.01.1992 г., приводится еще один возможный вариант обозначений для циклов. Однако он представляется неудачным (рис. 1.1, *е—ж*).

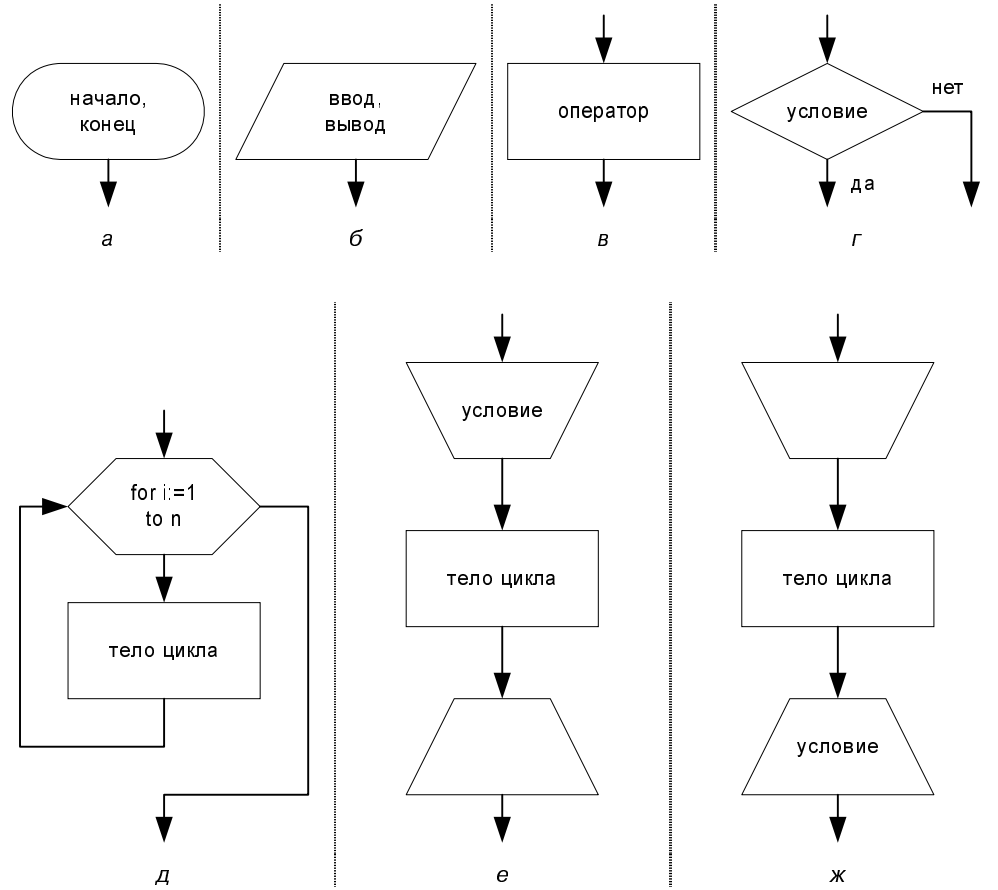

Рис. 1.1. Условные обозначения на схемах алгоритмов

1.2.3. 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 предусмотрен 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*) представляет собой многократно повторяющуюся последовательность шагов алгоритма.

1.2.4. Пример блок-схемы алгоритма

Рассмотрим пример блок-схемы алгоритма игры "Угадай число".

Условие игры: игрок должен угадать число, "задуманное" компьютером — случайное число в диапазоне от 0 до 1000. Алгоритм игры представлен на рис. 1.2. Игра начинается с того, что компьютер задумывает случайное

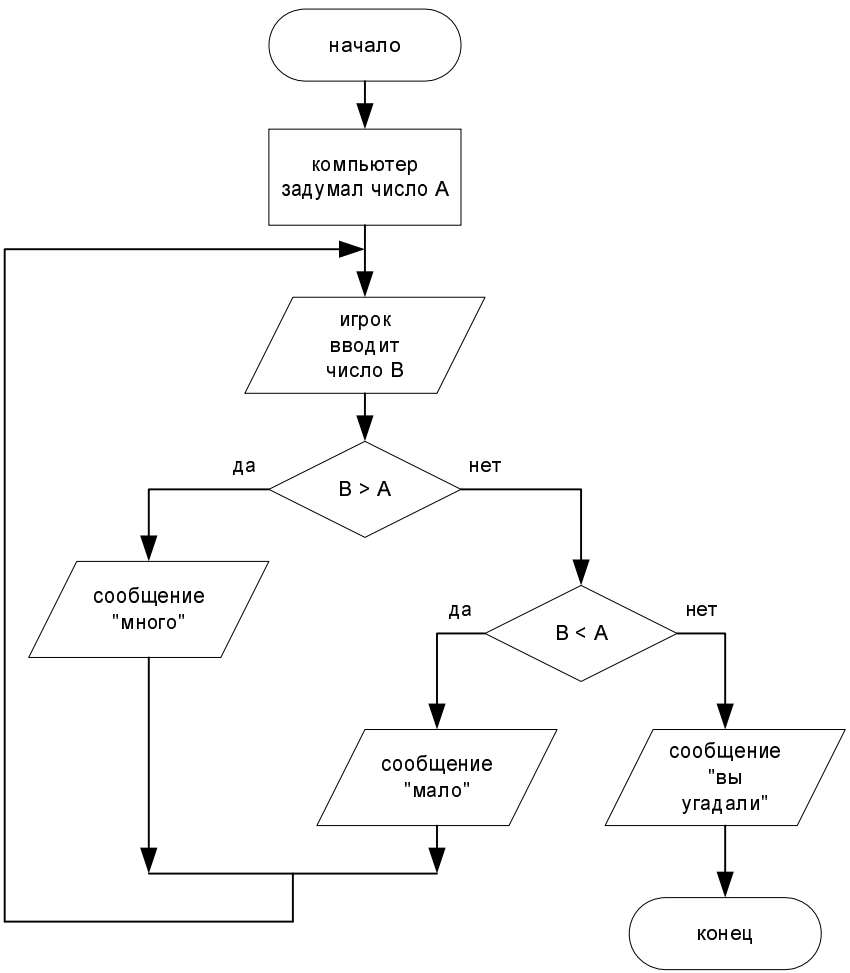

Рис. 1.2. Блок-схема алгоритма игры "Угадай число"

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 сообщение "много" и просит игрока ввести следующее число, еще раз испытав удачу. 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. Программа, соответствующая блок-схеме, приведена в листинге 3.12.

1.3. Этапы разработки программы

1.3.1. Язык программирования. Программа

Команды, выполняемые процессором ПК, являются электрическими сигналами, которые можно представить в виде последовательностей нулей и единиц. Каждой команде соответствует свое число. Таким образом, процессор имеет дело с *машинным кодом*. Написать программу на нем может только очень опытный программист, хорошо знающий *архитектуру процессора* (его устройство) и *систему команд* (набор допустимых инструкций). Большинство программ создаются при помощи "посредников", в качестве которых выступают *языки программирования высокого уровня*.

Совокупность средств и правил представления алгоритма в виде, пригодном для выполнения вычислительной машиной, называется *языком программирования*.

Программа — это запись (реализация) алгоритма на языке программирования.

1.3.2. Этапы разработки

В процессе создания любой программы можно выделить несколько этапов.

- *Постановка задачи* — выполняется специалистом в предметной области на естественном языке (русском, английском и т. д.). Необходимо определить цель задачи, ее содержание и общий подход к решению. Возможно, что задача решается точно *(аналитически)*, и без компьютера можно обойтись. Уже на этапе постановки надо учитывать эффективность алгоритма решения задачи на ЭВМ, ограничения, накладываемые *аппаратным и программным обеспечением* (АО и ПО).
- *Анализ задачи и моделирование* — определяются исходные данные и результат, выявляются ограничения на их значения, выполняется формализованное описание задачи и построение (выбор) математической модели, пригодной для решения на компьютере.

- *Разработка или выбор алгоритма решения задачи* — выполняется на основе ее математического описания. Многие задачи можно решить различными способами. Программист должен выбрать оптимальное решение. Неточности в постановке, анализе задачи или разработке алгоритма могут привести к *скрытой ошибке* — программист получит неверный результат, считая его правильным.
- *Проектирование общей структуры программы* — формируется модель решения с последующей детализацией и разбивкой на подпрограммы, определяется "архитектура" программы, способ хранения информации (набор переменных, массивов и т. п.).
- *Кодирование* — запись алгоритма на языке программирования. Современные системы программирования, подобные *Delphi*, позволяют ускорить процесс разработки программы, автоматически создавая часть ее текста, однако творческая работа по-прежнему лежит на программисте. Для успешной реализации целей проекта программисту необходимо использовать *методы структурного программирования (см. разд. 6.1)*.
- *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 правильный результат. 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. Использование специальных программ — *отладчиков*, которые позволяют выполнять программу по отдельным шагам, просматривая при этом значения переменных, значительно упрощает этот этап *(см. приложение 1)*.
- *Анализ результатов* — если программа выполняет моделирование какого-либо известного процесса, следует сопоставить результаты вычислений с результатами наблюдений. В случае существенного расхождения необходимо изменить модель.
- *Публикация* результатов работы, *передача заказчику* для эксплуатации.
- *Сопровождение программы* — включает консультации представителей заказчика по работе с программой и обучение персонала. Недостатки и ошибки, замеченные в процессе эксплуатации, должны устраняться.

1.4. Ошибки

1.4.1. Компилятор. Синтаксис и семантика

Особое значение для программиста имеет предупреждение и исправление ошибок в алгоритме и программе решения задачи. Прежде чем выполнить программу, ее текст необходимо ввести в компьютер. Для ввода и изменения (редактирования) текстов используется специальная программа — *текстовый редактор.*

Текст набранной программы, для того чтобы быть "понятым" компьютером, должен быть переведен на язык машинных кодов. Такой перевод называется *компиляцией* и выполняется специальной программой — *компилятором.* Компилятор анализирует программу и определяет, содержит ли она ошибки. В случае их обнаружения вся работа останавливается. Если же правила языка программирования не нарушены, то формируется модуль на машинном языке, который затем и исполняется *(см. приложение 1).*

В отличие от естественных языков, таких как русский, английский и др., язык программирования имеет очень ограниченное количество "слов", понятных компилятору, и строгие правила записи команд. Совокупность этих требований образует *синтаксис* языка программирования, а смысл команд и других конструкций языка — его *семантику.*

1.4.2. Типы ошибок

Программирование является творческим процессом, поэтому ошибки неизбежно встречаются даже у опытных программистов. Различают следующие типы ошибок: синтаксические ошибки (ошибки компиляции), ошибки выполнения и ошибки в алгоритме программы (семантические).

- *Синтаксические ошибки* возникают при нарушении правил языка (в нашем случае — языка Turbo Pascal), их обнаруживает компилятор, который не может из-за ошибки "понять" назначение команды.
- *Ошибки выполнения* не нарушают синтаксис языка. Однако они приводят к ошибочным операциям в процессе выполнения программы, например попытке деления на ноль или извлечения квадратного корня из отрицательного числа. Перечень Turbo Pascal об ошибках содержит более 200 сообщений *(см. приложение 1).*
- *Ошибки в алгоритме* 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, считает его верным, поскольку никаких сообщений об ошибках не было. Семантические ошибки должен обнаруживать сам программист. В поиске и исправлении ошибок ему может оказать существенную помощь *интегрированная среда разработки Turbo Pascal* и ее встроенный отладчик *(см. приложение 1).*

ГЛАВА 2

Программа на Turbo Pascal

2.1. Начальные сведения

2.1.1. Пример учебной программы

Создадим первую учебную программу вычисления суммы двух целых чисел. Сценарий взаимодействия человека и компьютера при решении данной задачи 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 2.1).

Листинг 2.1. 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 }
```

Прочитав текст программы, обратите внимание на ее структуру *(см. разд. 2.2)*.

1. В данной программе использованы следующие зарезервированные слова языка Turbo Pascal — слова, за которыми закреплено строго определенное значение:

   - `program` — заголовок программы, определяющий ее название. Эту строку в программе можно было бы опустить, т. к. она имеет чисто декоративное значение;
   - `var` — начало объявления переменных (сокращение от английского слова *variable*, переменная);
   - `integer` — указание, что переменные `a`, `b`, `sum` — целые числа, т. е. могут принимать целочисленные значения, например 2, 3, 0, 287, 21, 32 и др. на интервале [−32 768, 32767];
   - `write('Текст')` — инструкция компьютеру о выводе на экран сообщения `'Текст'`. Обратите внимание, что текст справа и слева ограничен символом `'` — апострофом;
   - `readln(a)` — инструкция компьютеру о считывании значения переменной `a` с клавиатуры.

2. Для вычисления суммы чисел `a` и `b` в программе использована запись инструкции присваивания суммы чисел `a` и `b` переменной `sum`. Присваивание записывается с помощью пары символов `:=` в виде `sum := a + b`.

3. Программа представляет собой последовательность символов, для удобства восприятия разбитых на строки. Никакие знаки переноса не используются. Начиная строку с нескольких пробелов, можно добиться, чтобы некоторые зарезервированные слова располагались одно под другим. Это облегчает составление и понимание программы. Каждая инструкция программы завершается *разделителем* `;` (точкой с запятой), в конце программы ставится `.` (точка). Пояснения к программе, не влияющие на исполнение, записываются в фигурных скобках: `{ это комментарий }`. Все комментарии в данном примере можно убрать, при этом программа останется работоспособной.

2.1.2. Базовые понятия

Характеристики программы. Данные. Результаты

Программа реализует алгоритм решения задачи. Основные *характеристики программы* следующие:

- точность полученного результата;
- время выполнения;
- объем требуемой памяти.

Функционирование любой программы связано с обработкой *данных*. Данные, предназначенные для обработки, называются *исходными* и задаются обычно в начале выполнения программы. Программа по ходу выполнения может запрашивать недостающие исходные данные. Основной способ задания исходных данных — ввод с клавиатуры (см. листинг 2.1). Выбор какого-либо пункта меню, щелчок мышью на определенной кнопке на экране — также способы ввода исходных данных. Иногда программа может считывать исходные данные из файлов на диске *(см. гл. 12)*.

В процессе выполнения программы исходные данные преобразуются в *результаты*. Результаты выводятся на экран или печатающее устройство — принтер в текстовом или графическом виде, а также могут быть записаны в файлы на диске.

Константы

Каждый элемент данных, используемый в программе, является константой или переменной. *Константами* называются элементы данных, значения которых в процессе выполнения программы не изменяются. В языке Turbo Pascal используются константы следующих видов: числовые, логические (булевские), символьные и строковые. Числовые константы предназначены для представления числовых данных (целых и вещественных). Булевские константы используются для представления данных, имеющих смысл логических высказываний (да-нет, истина-ложь, 1—0). Символьные и строковые константы — это отдельные символы и их последовательности.

Переменные

Переменные, в отличие от констант, могут менять свои значения при выполнении программы. В программировании переменную можно трактовать как *одну или несколько ячеек оперативной памяти компьютера, которым присвоено определенное имя (идентификатор)*. Имя в данном случае выступает как посредник, позволяющий отвлечься от "неудобного" для использования адреса конкретной ячейки (или ячеек) с интересующим нас содержимым, но сохранить возможность "простого" обращения к нему. *Содержимое этих ячеек может меняться, но имя переменной остается неизменным*. Каждое *новое*

значение, записанное в ячейку памяти, *"затирает"* предыдущее значение, поэтому *в любой момент времени* переменная имеет только одно, *текущее*, значение. Обычно переменные используются для хранения исходных данных, результатов программы, а также промежуточных данных, которые образуются по ходу выполнения алгоритма.

В математике значение переменной в рамках определенной задачи неизменно. Именно поэтому высказывание `a := a + 1` математик сочтет неверным. Тем не менее, для программиста это абсолютно правильная конструкция, которая задает вычисление суммы содержимого ячейки `a` и числовой константы `1` и занесение полученного результата в ту же ячейку `a`. После выполнения этого действия старое значение переменной `a` будет безвозвратно потеряно, т. к. одна ячейка памяти не может вместить сразу несколько значений. *Это очень важный момент в программировании.*

Turbo Pascal позволяет давать имена не только переменным, но и константам, и это считается хорошим стилем программирования. В рассмотренном выше примере (см. листинг 2.1) простейшая программа содержала только имена переменных, реальные программы обычно содержат раздел определения констант *(см. разд. 2.2.4)*.

Описание данных. Типы

Именование констант и переменных в программировании очень похоже на использование символических выражений в алгебре, однако, для того чтобы компилятор смог их обрабатывать, нужно снабдить его некоторой дополнительной информацией — *выполнить описание*. В этой информации сообщается о *типе* каждой именованной величины. Идея типов берет свое начало в математике и логике и призвана предотвращать двусмысленные и ошибочные конструкции языка программирования.

Человек, решающий какую-нибудь задачу "вручную", обладает интуитивной способностью быстро разобраться в типах данных и тех операциях, которые для каждого типа справедливы; известно, например, что нельзя извлечь квадратный корень из слова или написать число с заглавной буквы. Одна из причин, позволяющих легко провести такое распознавание, состоит в том, что слова, числа и другие обозначения выглядят по-разному. Однако для компьютера все типы данных сводятся, в конечном счете, к последовательности *битов, образующих байты* — содержимому ячеек памяти, поэтому различие в типах следует делать явным.

Таким образом, Turbo Pascal, как и другие так называемые языки высокого уровня, позволяет отвлечься от представления данных в виде последовательности двоичных разрядов, наилучшего с точки зрения компьютера. При написании программы программист может использовать понятия, соответствующие терминам решаемой задачи: целое и вещественное число, матрица (массив), запись, файл и т. д. Это существенно упрощает решение. Есте-

ственно, что в конце концов все они отображаются на конкретное битовое представление.

Замечание

В словосочетании "язык высокого уровня" термин "высокий" не следует воспринимать буквально. Аналогично, для ассемблера — языка программирования низкого уровня — слово "низкий" не значит "плохой". Просто во втором случае имеется в виду, что инструкции языка близки к машинному коду и учитывают систему команд процессора.

Инструкции. Операторы

Алгоритм решения любой задачи состоит из отдельных, довольно мелких шагов. В программе для каждого шага алгоритма записывается отдельная инструкция (команда). Отдельные инструкции записываются также для организации ветвлений и циклов. Таким образом, программа состоит из отдельных инструкций, или команд. Эти инструкции в программировании принято называть *операторами*. Программа состоит из операторов подобно тому, как здание строится из отдельных кирпичиков. Часто в литературе по программированию программу определяют как последовательность операторов.

Операторы могут объединяться в более крупные конструкции — *составные операторы, процедуры и функции*. Такие конструкции состоят из нескольких элементарных операторов, однако в программе могут использоваться как один оператор. Продолжая аналогию со строительством, можно сказать, что используются как отдельные кирпичики (элементарные операторы), так и строительные блоки.

Процедуры и функции универсального назначения могут располагаться в особым образом оформленных файлах — *библиотечных модулях*. Все эти конструкции будут подробно разбираться ниже.

2.1.3. Типы данных

Тип. Классификация типов

Тип определяет множество значений, которые могут принимать объекты программы (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, сцепления, или конкатенации* текста (обозначается через `+`).

Все типы данных, используемые в Turbo Pascal, можно разделить на две большие группы: скалярные (простые) и структурированные (составные). *Скалярные типы* в свою очередь подразделяются на стандартные и пользовательские (перечисляемый и интервальный). *Стандартные типы* предлагаются программисту разработчиками Turbo Pascal. К ним относятся: целочисленные, вещественные, символьный (литерный), логический (булевский) и указатели. *Структурированные типы* имеют в своей основе скалярные типы данных. К структурированным относятся: строки, массивы, множества, записи и файлы.

Целочисленные типы, символьный, логический и пользовательские типы данных (перечисляемый и интервальный) образуют группу так называемых *порядковых типов*, имеющих большое значение.

Тип данных очень важен при выделении памяти под переменные, поскольку каждому типу соответствует строго определенный размер ячейки памяти. В любом случае этот размер ограничен, следовательно, *все типы данных имеют ограниченный диапазон значений* (см. табл. 2.1—2.3). Этот факт не согласуется с нашими математическими представлениями о числовых множествах. Тем не менее, с ним приходится считаться.

Стандартные типы

Целые и *вещественные* типы предназначены для представления числовых данных. В математике рассматривается *бесконечное* множество целых чисел. Целый тип в языке Turbo Pascal — это *интервал* целых чисел (табл. 2.1). Операции над целыми числами (см. табл. 3.1) определены лишь тогда, когда исходные данные *(операнды)* и результат лежат в этом интервале. Иначе возникает ситуация, называемая *переполнением*. За исключением переполнения все операции над аргументами целого типа выполняются *точно (см. разд. 3.3.1)*.

Таблица 2.1. *Целочисленные типы данных*

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

В математике вещественные числа — это *бесконечное непрерывное* множество чисел. В вычислительных машинах вещественные числа представляются *конечным* множеством значений (табл. 2.2).

Например, внутреннее представление типа `real` может дать $2^{48} =$ $= 281\,474\,976\,710\,656$ (более чем 10^{14}) возможных комбинаций значащих разрядов в отведенных для него 6 байтах, или 48 битах. Это очень большое число, но все же оно не сопоставимо с множеством вещественных чисел.

Таблица 2.2. *Вещественные типы данных*

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

Логический (булевский) тип имеет всего два значения: `true` (да — истина, 1) и `false` (нет — ложь, 0), причем данные значения упорядочены, т. е. в операциях сравнения `true > false` (табл. 2.3).

Символьный (литерный) и строковый типы представляют данные, являющиеся *символами* и их последовательностями — *строками* (см. табл. 2.3). В памяти компьютера символы хранятся в виде их *числовых кодов*. 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 2.3. *Символьный и логический (булевский) типы данных*

Тип	Диапазон возможных значений	Память, байт
`char` (символьный, литерный)	Символы кодовой таблицы	1
`boolean` (булевский)	`true`, `false`	1

Переменные, описываемые любым из типов `byte`, `shortint`, `integer`, `word`, `longint`, принимают только целые значения. Типы `byte`, `word` — беззнаковые.

Переменные, описываемые любым из типов `single`, `real`, `double`, `extended`, `comp` принимают только вещественные значения — положительные и отрицательные.

Наиболее часто в простейших программах используются типы `integer` и `real`.

Замечание

При разработке 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 компилятора `{$N+}` или установив переключатель [×] для соответствующего пункта меню интегрированной среды Turbo Pascal: **Options | Compiler | 8087/80287**. Более подробно директивы компилятора рассмотрены ниже *(см. разд. 2.1.4)*.

Тип `comp`, являясь вещественным (приблизительно от $-9{,}2\times10^{18}$ до $9{,}2\times10^{18}$), 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.

Данные целых типов могут быть представлены как в десятичной, так и в шестнадцатеричной *системах счисления*.

Замечание

Признаком записи *шестнадцатеричной* константы является знак доллара — $, после которого следуют шестнадцатеричные цифры (допустимый диапазон — от $00000000 до $FFFFFFFF, при адресации абсолютных переменных — от $0000 до $FFFF). 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 ($123=1\times10^2+2\times10^1+3\times10^0$) — целое десятичное число, а $7B ($7B=7\times16^1+B\times16^0$) — число, равное ему, в шестнадцатеричной форме записи.

Бит. Байт

Вся информация (числа, логические значения, символы) хранится в памяти компьютера в *двоичной* форме в виде последовательности *битов* (от *binary digit*, двоичная цифра). Каждый бит может принимать значение одной двоичной цифры — *ноль* или *единица*. Восемь битов объединены в *байт*. Мак-

симальное 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 их 256).

Так как переменные разных типов могут принимать различные значения, то для их хранения нужен соответствующий им объем памяти (ячейки разных размеров). Память под переменные выделяется в байтах (целым числом).

Например, значением символьной переменной (типа `char`) может быть любой из 256 символов (столько разных символов в кодовой таблице). Поэтому для хранения переменной такого типа достаточно одного байта (8-разрядное слово). Значением переменной типа `integer` является обязательно *целое* число в диапазоне от −32768 до 32 767 (65 535 значений). Для хранения переменной этого типа требуется два байта (16-разрядное слово). Несимметричность диапазона значений относительно нуля вызвана тем, что при традиционной кодировке целых чисел в слове из n битов можно записать числа в диапазоне от -2^{n-1} до $2^{n-1}-1$. Для беззнакового числа типа `word` диапазон $[0..\ 2^{16}-1]$ соответствует значениям [0; 65535]. Очевидно, что чем больше диапазон значений типа, тем больше байтов нужно для хранения переменной. Так, для типа `longint` диапазон $[-2^{31};\ 2^{31}-1]$ соответствует значениям [−2 147 483 648; 2 147 483 647].

Формы записи вещественных чисел

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

Запись вещественного числа *в экспоненциальной форме* (в форме с *мантиссой* и *порядком*) использует степень десяти (например: 25×10^{-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) — экспоненциальная.

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.

Любое вещественное число хранится в памяти компьютера в экспоненциальной форме: отдельно — мантисса и отдельно — порядок. При этом под мантиссу и порядок отводится строго определенное количество двоичных разрядов. Выбор такого представления имеет несколько следствий:

- существуют очень маленькие значения (машинное ε — *"машинное эпсилон"*), которые не могут быть представлены (см. листинг 3.24). Попытки их использования обычно приводят к возникновению ошибок;
- каждое вещественное число будет иметь приблизительно одинаковое количество *значащих цифр* в его представлении (см. листинг 3.19). 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.

Вещественные числа представлены приближенно, следовательно, арифметические действия (см. табл. 3.1) над ними также выполняются *приближенно.*

Из изложенного следует несколько простых правил:

- вещественные числа нежелательно проверять на строгое равенство;
- необходимо проявлять осмотрительность при преобразовании вещественных чисел в целые и избегать вычитания почти равных чисел, т. к. могут возникнуть ошибки из-за потери многих значащих цифр;
- для уменьшения влияния ошибки округления при выполнении арифметических операций с вещественными числами необходимо иметь в виду следующее. Если складывается много чисел, то их нужно *разбить 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.* По аналогии с предыдущим получаются оценки для других арифметических операций и соответствующие практические рекомендации (см. листинг 3.29).

Вещественные числа в шестнадцатеричной системе счисления записывать нельзя.

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 А, `';'` — точку с запятой, `' '` — пробел, `#32` или `#$20` являются также символом 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) — смещение курсора на одну позицию назад;
- `#09` (HT) — горизонтальная табуляция: смещение курсора в позицию кратную 8, плюс 1 (9, 17, 25 и т. д.);
- `#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.

Кроме `#` в тексте программы может использоваться также `^` (знак тильды или каре), который вместе с символом, следующим за ним, воспринимается как управляющий. *Управляющие символы* могут группироваться в строке без разделителей между ними и использоваться вместе со строковыми константами. Так записываются только символы с кодами от 0 до 31 *(см. приложение 2)*.

Например, для выдачи предупреждения об ошибке с подачей звукового сигнала можно использовать команду `writeln('ошибка ввода данных',^G^G^G);`. Управляющий символ `^G` воспроизводит звук длительностью 0,25 секунды и частотой 800 Гц.

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. В отличие от имен пользователя *(см. разд. 2.1.4)*, *строчные и прописные буквы в составе строки различаются*. Под *длиной строки* понимают общее число символов в ней, включая символы пробела. 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 далее *(см. гл. 9)*.

Порядковые типы

Следующие типы данных — целые, символьный и логический имеют ограниченное количество значений, идущих по порядку, поэтому эти типы принято называть *порядковыми типами*. Общим для них является то, что в компьютере они представляются *целым* числом. Вещественные типы, как уже указывалось выше, тоже принимают конечное число значений. Оно определяется форматом внутреннего представления вещественного числа в ЭВМ. Однако количество возможных значений вещественных типов настолько велико, что в компьютере невозможно сопоставить с каждым из них целое число (его номер). *Все вещественные типы данных не являются порядковыми.*

В Turbo Pascal имеются два дополнительных пользовательских порядковых типа:

- интервальный (ограниченный) тип или диапазон;
- перечисляемый тип.

Они используются для того, чтобы еще больше ограничить количество значений, принимаемых переменными этого типа.

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).

Перечисляемый тип ограничен больше, он задается перечислением своих значений.

Например, в виде *строковых констант*: `color=(red,blue,green,black)`.

В приведенном примере создается новый (нестандартный) тип данных `color`. Переменные этого типа могут принимать всего 4 значения: `red`, `blue`, `green` и `black`. Такая возможность создания новых пользовательских типов данных имеется в языке Turbo Pascal и некоторых других языках *(см. разд. 2.2.6)*.

2.1.4. Алфавит языка

Текст на естественном языке состоит из предложений, предложения — из слов, слова — из букв. Буквы образуют алфавиты русского, английского и других языков. Язык программирования организован подобным образом.

Программа на языке Turbo 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. 2.4) и комбинации специальных символов — они образуют *составные символы* (табл. 2.5).

Таблица 2.4. *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	^	Тильда (каре)
@	Коммерческое а		Пробел (не имеет обозначения)

Таблица 2.5. *Составные символы*

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

Неделимые последовательности знаков алфавита образуют *слова*, отделенные друг от друга разделителями.

Разделителями служат — пробел, символ конца строки, комментарий. Пробел, стоящий внутри строковой константы, воспринимается не как разделитель, а как ее часть. Между комбинациями специальных символов пробелы недопустимы.

Слова подразделяются на зарезервированные слова, стандартные идентификаторы (имена) и идентификаторы пользователя.

Зарезервированные слова языка Turbo Pascal являются составной частью языка, имеют фиксированное начертание и несут в программе определенный смысл (табл. 2.6).

Таблица 2.6. *Зарезервированные слова*

Слово	Смысл слова	Слово	Смысл слова
absolute	Абсолютный	if	Если
and	Логическое И	implementation	Реализация
array	Массив	in	В (входит в)
asm	Ассемблер	inherited	Наследование
begin	Начало блока	inline	Основной
case	Вариант	interface	Интерфейс
const	Константа	interrupt	Прерывание
constructor	Конструктор	label	Метка
destructor	Деструктор	library	Библиотека
div	Деление нацело	mod	Остаток от деления
do	Выполнять	nil	Отсутствие
downto	Уменьшить до	not	Логическое НЕ
else	Иначе	object	Объект
end	Конец блока	of	Из
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

Таблица 2.6 *(окончание)*

Слово	Смысл слова	Слово	Смысл слова
shl	Сдвиг битов влево	until	До
shr	Сдвиг битов вправо	uses	Использовать
string	Строка	var	Переменная
then	То	while	Пока
to	Увеличивая	with	С
type	Тип	xor	Исключающее ИЛИ
unit	Модуль		

Внутри зарезервированных слов пробелы использовать запрещено.

Например, для оператора `goto ИмяМетки;` формы записи вида:

- `goto 10;` или `goto  10;` допустимы;
- `goto10;` или `go to 10;` являются ошибочными.

Стандартные идентификаторы служат для обозначения заранее определенных разработчиками языка типов данных, констант, процедур и функций. При использовании в программе их не требуется описывать, указывая тип.

Например, стандартный идентификатор `sin(x)` вызывает функцию, вычисляющую синус угла `x`, заданного в радианах. Любой из стандартных идентификаторов, в отличие от зарезервированных слов, допускается переопределять. Пользователь может написать свою собственную функцию с именем `sin` *(см. гл. 5)*. Обычно это ведет к ошибкам. Поэтому стандартные идентификаторы лучше использовать без изменений.

Идентификаторы пользователя

Они применяются для обозначения меток, констант, переменных, процедур и функций, определенных самим программистом. Тип идентификатора пользователя должен быть указан в описательной части программы, до его использования *(см. разд. 2.2.1)*.

Общие правила написания идентификаторов (имен):

- 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 Turbo 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;

- максимальная длина — 127 символов, но значащими, которые распознает Turbo Pascal, являются первые 63, что на практике более чем достаточно;
- 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 Turbo Pascal часто используют такой способ: первая буква каждого слова прописная, остальные — строчные (например, `TextColor`). Однако в примерах этой книги будут использоваться, в основном, строчные буквы, что позволит упростить ввод текста программ для пользователя.

Имена, используемые в программе, должны быть *уникальными*, т. е. 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: **Error 4: Duplicate identifier** (Ошибка 4: Двойной идентификатор).

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.

Комментарии и директивы компилятора

В любом месте программы, где разрешен пробел, 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:

- ограничители комментария 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 `$`. По умолчанию они находятся в состоянии, гарантирующем минимальный объем исполнимого кода и минимум времени компиляции. Несколько директив компилятора могут находиться в одной строке. Часто используются следующие директивы:
  - `{$R+}` — проверять выход за границы диапазонов;
  - `{$I-}` — отмена контроля операций ввода/вывода;
  - `{$F+}` — формировать дальний тип вызова процедур и функций.

2.2. Структура программы

2.2.1. 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. Такое имя 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 Turbo 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, заключаемый в слова `begin` и `end`;
  - в конце программы ставится признак останова — . (точка).

Описательная часть предназначена для объявления всех встречающихся в программе данных и их характеристик (имена данных, их тип, возможные значения и др.).

В *исполнительной части* (разделе операторов) записывается последовательность исполняемых операторов. Каждый оператор выражает действие, которое необходимо выполнить. Исполняемые операторы, как мы уже упоминали, отделяются друг от друга символом ; (точка с запятой).

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.*

Операторы Turbo Pascal не привязаны к определенной позиции строки. В одной строке можно размещать несколько операторов, отделяя их друг от друга точкой с запятой. Допускается перенос операторов с одной строки на другую (но без разделения ключевых слов).

Если между двумя операторами отсутствует точка с запятой, то это приведет к возникновению ошибки, поскольку компилятор зачастую не может понять, что же хотел сказать автор программы, или интерпретирует оператор неверно.

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).

2.2.2. 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` (см. листинг 3.1) и других модулях речь пойдет далее *(см. гл. 7)*.

2.2.3. Раздел описания меток

Перед любым оператором Turbo Pascal можно поставить *метку*, что позволяет выполнить прямой переход на этот оператор с помощью оператора `goto` из любого места программы *(см. разд. 3.4)*. 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. Раздел описания меток начинается с зарезервированного слова `label` и имеет следующий вид:

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

Замечание

Если метка описана, но не используется, то ошибки при этом не возникает.

Обратите внимание — областью действия метки является тот блок, где она описана. Это означает, что по метке нельзя входить или выходить из процедуры или функции *(см. гл. 5)*.

2.2.4. Раздел описания констант

Хранение констант не требует памяти, 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.

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.2.2}
  nmax=100;         { целая константа }
  nmin=-nmax;
  s='абвгд';        { строковая константа }
  kod=$123;         { шестнадцатеричная константа }
```

Обратите внимание — при определении констант применяется знак =, а не :=. Идентификатор, использованный для определения константы, можно употреблять при задании следующих констант, его значение нельзя изменять по ходу программы.

Типизированные константы

Существуют так называемые *типизированные константы*, эквивалентные *переменным с заранее заданным значением*. Название вызвано тем, что при описании указывается тип:

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

Например:

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

Для их хранения память выделяется как под обычные переменные, поэтому значения типизированных констант можно *изменять* по ходу выполнения программы.

Замечание

Типизированные константы могут быть любого типа, кроме файлов *(см. гл. 12)*. Нельзя также объявить типизированную константу-запись, если хотя бы одно из ее полей имеет файловый тип *(см. гл. 11)*.

Обратите внимание — использование типизированных констант позволяет поменять сразу много значений по всему тексту (столько значений, сколько раз встречаются в тексте данные константы), внеся изменения только в одном месте программы — разделе const.

Зарезервированные константы

Без предварительного описания в программе можно использовать значения *зарезервированных* или *предопределенных констант* (табл. 2.7).

Таблица 2.7. *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

2.2.5. Раздел описания типов данных

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

Например, пусть несколько переменных программы описываются при помощи одного структурированного типа *(см. разд. 2.1.3)*. В случае внесения изменений в описание, нет необходимости делать это несколько раз, рискуя ошибиться и пропустить очередную переменную. Вся корректировка будет выполняться в одном месте — разделе описания типов данных.

В языке Turbo Pascal предусмотрено несколько стандартных типов и существует *механизм создания новых типов данных*. Каждое новое определение типа задает множество значений и связывает с этим множеством некоторое имя.

Раздел описания типов данных — это раздел описания типов, *определяемых пользователем*, поэтому в простых программах он часто отсутствует. Раздел начинается с зарезервированного слова `type` и имеет вид:

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

Например:

```
type
  matr = array [1..maxrow, 1..maxcol] of real;
  { задан тип matr — таблица с maxrow строк и maxcol столбцов }
```

Далее идентификаторы типов можно использовать для описания переменных в разделе `var`.

Замечание

Тип данных можно описывать не только ссылаясь на него с помощью имени в разделе описания типов `type`, но и непосредственно в разделе описания переменных `var`. Многие программы можно написать, вообще не используя раздел описания типов. Однако хорошим стилем программирования считается использование раздела `type` и тогда, когда обстоятельства не вынуждают делать это явно.

2.2.6. 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 пока еще не определено, поэтому переменные часто *инициализируют*, присваивая им начальное значение *(см. разд. 2.2.4)*.

Инструкция объявления переменной выглядит так:

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

Например:

```
var
   matrix: matr;    { задан массив matrix типа matr, который }
                    { был объявлен ранее в разделе type }
   x1,x2: integer; y1: longint; { целочисленные переменные }
   sum: real; root: double;     { вещественные переменные }
   znak: char;                  { символьная переменная }
   flag: boolean;               { логическая переменная }
```

Следует знать:

- если в программе используются переменные разных типов, то зарезервированное слово `var` (от английского *variable*, переменная) лучше записать только один раз, а затем привести списки имен переменных каждого типа;
- в имени переменной можно использовать буквы латинского алфавита и цифры (первым символом должна быть буква);
- наиболее часто, особенно в простых программах, связанных с обработкой числовых данных, используются типы `real` и `integer`;

- после объявления переменной в качестве комментария рекомендуется указывать ее назначение.

Описание пользовательских типов данных

Особенностью языка Turbo Pascal является предоставляемая им возможность создания новых, *пользовательских типов* данных: перечисляемого и интервального *(см. разд. 2.1.3)*. Их применение значительно улучшает наглядность программы, экономит память и облегчает поиск ошибок, благодаря возможности контроля тех значений, которые получают соответствующие переменные.

Перечисляемый тип. Задается непосредственно перечислением всех значений, которые может принимать переменная данного типа. Сами значения указываются через запятую, а весь список заключается в круглые скобки. Первая константа имеет порядковый номер 0, вторая — 1 и т. д. (при необходимости до 65 535).

Описание перечисляемого типа данных имеет следующий вид:

```
type ИмяТипа = (Значение1, Значение2,...,ЗначениеN);
var ИмяПеременной : ИмяТипа;
```

Например:

```
type days=(monday,tuesday,wednesday,thursday,friday,saturday,sunday);
var day: days; Season: (Winter, Spring, Summer, Autumn);
```

В примере приведен явно описанный тип данных пользователя — `days`. Определены его значения — обозначения дней недели, которые принимает переменная `day`. Попытка присвоить любое другое значение вызовет программное прерывание. Другой тип не имеет имени (является *анонимным*) и задается перечислением своих значений в разделе `var`. Переменной этого типа является `Season`, она может принимать значения `Winter`, `Spring`, `Summer` и `Autumn`. Так может быть задан любой тип. Имена внутри круглых скобок являются константами соответствующего типа перечисления и могут быть использованы в операторе `case` *(см. разд. 3.8.3)*.

Для значений перечисления одного и того же типа допустимы операции отношения и логические операции *(см. разд. 3.8.1)*. Упорядочение осуществляется по номеру элемента в описании типа.

Например, будет истинно выражение `Winter < Spring`, т. к. `Spring` имеет больший номер по порядку в описании типа, чем `Winter`.

Turbo Pascal не поддерживает ввод/вывод значений перечисляемого типа. При необходимости, программист должен организовать его сам (см. листинги 3.39, 4.11, 4.12). Так, попытка использовать операторы `readln(day);` или

`writeln(day);` вызовет сообщение об ошибке: **Error 64: Cannot Read or Write variables of this type** (Ошибка 64: Не могу считывать или записывать переменные этого типа). Переменным перечисляемого типа можно присваивать значения: `day:=monday;`. К перечисляемому типу неприменимы арифметические действия: оператор `day:=monday+tuesday;` вызовет сообщение об ошибке: **Error 41: Operand types do not match operator** (Ошибка 41: Тип операндов не соответствует оператору).

Интервальный тип. Он задает две константы, определяющие границы диапазона значений для данной переменной — *отрезок типа*. Для каждой операции с переменной интервального типа автоматически выполняется проверка: остается ли значение переменной внутри установленного для нее диапазона. *Автоматическая проверка объявленных границ* позволяет программисту не отвлекаться на организацию собственного контроля, что является существенным достоинством использования интервального типа. В хорошо написанных программах можно скорее увидеть оператор `var count: 10..100;`, чем `var count: integer;`, если по условию задания переменная `count` должна принимать значения от 10 до 100. Значение первой константы должно быть обязательно меньше второй. Обе константы должны принадлежать к одному типу. Тип `real` недопустим. Оператор `type price=1.99..5.99;` вызовет сообщение об ошибке: **Error 27: Invalid subrange base type** (Ошибка 27: Недопустимый исходный тип поддиапазона). В описании интервального типа могут использоваться именованные константы *(см. разд. 2.2.4)*.

```
type ИмяТипа = Константа1.. Константа2;
var ИмяПеременной : ИмяТипа;
```

Например:

```
const min=1; max=31;
type days=min..max;
var rab_day, bol_day : days;
```

Здесь переменные `rab_day` и `bol_day` имеют тип `days`, принимают любые значения из диапазона от 1 до 31. Выход за его границы вызовет программное прерывание.

Ограничения для интервального типа те же, что и для перечисляемого.

Как уже указывалось выше, перечисляемый и интервальный типы данных совместно с целыми, логическим и символьным относятся к порядковым типам *(см. разд. 2.1.3)*. *Для работы с данными порядковых типов* в языке Turbo Pascal используются функции:

- `ord(s)` — функция возвращает *порядковый номер* значения `s` в множестве, определенном типом `s`. Результат — `longint`. Для целых типов функция

возвращает само значение s. Применение ord(s) к логическому, символьному и перечисляемому типам дает положительное целое число в диапазоне от 0 до 1, от 0 до 255 и от 0 до 65 535 соответственно. Применение ord(s) к интервальному типу зависит от его свойств;

- pred(s) — функция возвращает элемент, *предшествующий* s в списке значений типа. Тип результата совпадает с типом параметра. 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 параметра. Если следующее за s значение отсутствует, возникает программное прерывание.

2.2.7. 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 делятся на *процедуры и функции*, которые бывают *стандартными и определенными пользователем*. Стандартные процедуры и функции являются частью языка и вызываются без предварительного описания. Описание процедур и функций пользователя выполняется в специальном разделе программы.

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 далее *(см. гл. 5)*.

2.2.8. Раздел операторов

Раздел операторов является основным, т. к. именно в нем с предварительно описанными константами, переменными, значениями функций выполняются действия, позволяющие получить результат, ради которого и создавалась программа. Он начинается зарезервированным словом `begin`, далее следуют операторы языка, *отделенные друг от друга точкой с запятой*. Завершает 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, а длинный оператор может занимать несколько строк экрана.

ГЛАВА 3

Операторы

Операторы языка Turbo Pascal можно разделить на простые и сложные. *Простые* не содержат внутри себя других операторов. *Сложные (структурные)* операторы представляют собой конструкции, содержащие простые операторы. К простым относятся следующие операторы: присваивания, перехода, пустой оператор, операторы ввода и вывода. К сложным операторам относятся: составной оператор, оператор условного перехода, операторы цикла, оператор выбора, оператор присоединения в записях.

3.1. 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. 12)*. В Turbo Pascal нет стандартных средств для работы с мышью.

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

Для ввода и вывода данных в языке Turbo 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 (инструкциями) по аналогии с другими операторами языка, хотя, строго говоря, называть процедуры операторами неверно. Разница между оператором и процедурой Turbo Pascal состоит

в том, что имя процедуры является стандартным идентификатором, который, в отличие от зарезервированного слова, можно переопределить, написав, например, свою собственную процедуру с тем же именем *(см. разд. 2.1.4)*. В дальнейшем, говоря о вводе/выводе, будем использовать термин "инструкция", т. к. он имеет более широкое толкование.

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`). Причем, первые два оператора, выполненные последовательно, эквивалентны третьему.

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 *(см. разд. 8.3)*. Инструкция `read` оставляет в буфере клавиатуры код клавиши <Enter>, нажатие которой завершает процесс ввода;
- в одной инструкции `read` или `readln` можно записать несколько переменных. Для того чтобы отделить их значения друг от друга, при вводе можно использовать пробел либо символ 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, по окончании 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;
- в случае несоответствия типа введенных данных типу переменной, значение которой вводится с клавиатуры, программа завершает работу, и на экран выводится сообщение об *ошибке ввода/вывода*. Если программа за-

пущена из среды разработки, т. е. из Turbo Pascal, — **Error 106: Invalid numeric format** (Ошибка 106: Неверный числовой формат). Если программа запущена из *операционной системы* — **Run time error 106** (Ошибка времени выполнения 106). В этом случае необходимо использовать *обработку ошибок ввода/вывода (см. разд. 3.9.1)*.

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

3.2. Вывод данных

Вывод данных — это передача данных после обработки из оперативной памяти на внешнее устройство (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*. Причем, первые два оператора, выполненные последовательно, эквивалентны третьему.

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 оно будет вычислено, а затем выполнен вывод полученного результата. Подробнее о вычислении арифметических выражений рассказано ниже *(см. разд. 3.3.1)*;
- процедура вывода `writeln` аналогична `write`. Отличие заключается в том, что после вывода последней переменной из списка, 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.

3.2.1. 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 экспоненциальном виде (с использованием степени десяти). Такой вид результатов на экране часто не устраивает пользователей, поэтому программисту следует позаботиться о приемлемом *формате вывода*.

В операторах 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, в этом случае результат выводится в общепринятой форме.

Обратите внимание — если `w`, заданное программистом, мало, то при выводе ширина поля будет *увеличена*. Если мало `d`, то производится *округление*. Выводимый текст прижимается к правому краю поля вывода.

Например:

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

выведет на экран:

```
xxxxxxxxC=xx1.234
```

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

3.3. 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 соответствующих переменных в ходе 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 Turbo Pascal, т. к. именно в нем выполняются практически все действия по обработке данных.

Следует знать:

- инструкция присваивания используется для *изменения значений переменных*, в том числе и для вычислений по формулам — оператор предписывает *выполнить* выражение, заданное в его правой части, и *присвоить* результат переменной, стоящей в левой части;
- имеется три вида выражений:
  - *арифметические выражения* — служат для обработки числовых данных и задают порядок вычисления значения;
  - *логические выражения* — служат для сравнения различных данных и других логических действий;
  - *символьные выражения* — нужны для обработки текстов;
- тип результата, полученного при вычислении выражения, находящегося в правой части инструкции присваивания, должен быть *совместим по типу* с переменной, которой он присваивается, для того чтобы исключить возможность какого-либо искажения при присваивании;
- при нарушении соответствия выводится сообщение об ошибке: **Type mismatch** (Несоответствие типов).

Замечание

Переменной вещественного типа можно присваивать значение целочисленного выражения, но не наоборот.

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 этом случае вещественную переменную `y` необходимо преобразовать к целому типу. Для этого, например, можно явно использовать стандартные функции `trunc` и `round`. Операторы `x:=trunc(y);` `x:=round(y);` — верны *(см. разд. 3.3.1)*.

В языке Turbo 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;`.

3.3.1. Арифметические выражения

Рассмотрим подробно арифметические выражения, т. к. именно с их помощью выполняются все вычисления в программе. О логических и символьных выражениях будет рассказано ниже.

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

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

Арифметические операции

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

Например, в выражении `(x+y)*5-10` операндами являются переменные `x` и `y`, а также константы `5` и `10`; а `+`, `*` — знаки арифметических операций сложения и умножения соответственно. Символ операции умножения — `*` (звездочка) должен присутствовать в явном виде. Если все объекты, входящие в выражение, определены в момент их использования (в нашем примере — это переменные `x` и `y`), то значение выражения считается определенным.

В простейшем случае выражение может состоять из одной переменной или константы. Круглые скобки ставятся так же, как и в обычных математических выражениях для *управления порядком выполнения операций*. *Унарные* операции, такие как смена знака, 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. 3.3).

В табл. 3.1 приведены сведения об основных арифметических операциях. Для упрощения использованы только два основных числовых типа: `integer` и `real`.

Таблица 3.1. *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
Отрицание знака	-	real	real
		integer	integer
Арифметическое отрицание	Not	integer	integer

Замечание

Операция + используется также для работы со строками *(см. гл. 9)*. Операции +, –, * используются для работы с множествами *(см. гл. 10)*.

Операции *DIV* и *MOD*

Целочисленное деление `div` (от *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). Результат `div` всегда равен нулю, если делимое меньше делителя.

Например:

```
11 div 5 = 2
10 div 3 = 3
2 div 3 = 0
123 div 4 = 30
17 div -5 = -3
-17 div 5 = -3
-17 div -5 = 3
```

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

Например:

```
10 mod 5 = 0
11 mod 5 = 1
10 mod 3 = 1
14 mod 5 = 4
22 mod 5 = 2
31 mod 16 = 15
17 mod -5 = 2
-17 mod 5 = -2
-17 mod -5 = -2
```

Аргументы 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
```

Обратите внимание — операцию `mod` можно использовать, чтобы узнать, кратно ли целое `a` целому `b`. А именно, `a` кратно `b` тогда и только тогда, когда `a mod b = 0` (см. листинг 3.6).

Арифметические процедуры и функции

В арифметических выражениях часто используются следующие стандартные функции (табл. 3.2).

Таблица 3.2. *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 (см. листинг 3.1).

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 3.1);
- результат функции arctan получается в радианах.

Кроме приведенных в табл. 3.2 также используются следующие стандартные процедуры и 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 (см. листинг 3.4). В противном случае при каждом запуске программы датчик будет выдавать одни и те же числа. Эту особенность можно использовать при отладке программы;
- 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;
- 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)`.

Типы в арифметических выражениях

При включении в текст программы процедур и функций с целочисленными параметрами следует руководствоваться *вложенностью* типов, учитывая знак — везде, где может использоваться `word`, допускается применение `byte` (но не наоборот), аналогично, в `longint` входит `integer`, который включает в себя `shortint`. В выражениях можно использовать переменные различного типа, занимающие разный объем памяти.

Например, если объявляется переменная `day` (число месяца), то для нее можно задать типы `byte`, `integer` или `longint`. В первом случае будет занят один байт памяти, во втором — два, в третьем — четыре. Однако реально будет использоваться только один байт, а остальные — просто занимать ме-

сто в памяти. Поэтому следует подбирать наиболее подходящий тип для каждой переменной, исходя из тех значений, которые она может принимать (см. табл. 2.1—2.3).

Функция `sizeof(it)` вычисляет объем основной памяти в байтах, которую занимает указанная переменная или тип `it`. Результат — целое.

Например, рассмотрим следующую программу:

```
const a:integer=123; b:real=12.3e01; c: char='c';
begin
writeln('integer: ',sizeof(a),'; real: ',sizeof(b),'; char: ',sizeof(c));
end.
```

При выполнении программа напечатает строку:

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

Если в абстрактной математике знак арифметической операции всегда обозначает одно и то же, то в программировании ее результат зачастую зависит от типа исходных данных (см. табл. 3.1 и 3.2). Компилятор будет использовать различные команды для целых и вещественных чисел. *Если вычисленное значение выходит за пределы, определенные для соответствующего типа, возникает переполнение с потерей результата*. Для его предупреждения необходимо учитывать интервалы значений операндов и результата, различные для разных типов данных (см. табл. 2.1—2.3).

Обратите внимание — если *хотя бы один* из операндов арифметического выражения является *вещественным*, то результат всегда *вещественен*. В этом случае перед выполнением операции второй операнд также преобразуется к вещественному типу, и операция выполняется по правилам, принятым для вещественных чисел. *Операция деления всегда дает вещественный результат*, даже в случае, когда оба операнда являются целыми.

Функции *TRUNC* и *ROUND*

При работе с целочисленной арифметикой нельзя смешивать операнды целого и вещественного типов. Пусть `i` — переменная целого типа, а `x` — вещественного. Тогда для получения целочисленного результата их суммирования необходимо воспользоваться одной из функций `trunc` или `round`: `i+trunc(x)` или `i+round(x)`.

- Функция преобразования типа `trunc(x)` возвращает ближайшее целое число, меньше или равное вещественному `x` для `x>=0`, и больше или равное `x` для `x<=0` (от *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 преобразования типа `round(x)` возвращает значение `x`, *округленное* до ближайшего целого числа (от *round*, круглый). Аргумент — `real`, результат — `longint`.

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

Преобразование типов. Переполнение

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

Например, 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:10:0, y:10:0);
end.
```

При выполнении программа напечатает строку: `-32767    32769`.

Почему же при вычислении значения вещественной переменной `x` возникло переполнение? Дело в том, что при действиях с целыми числами тип результата соответствует типу операндов. В том случае, если операнды относятся к различным целым типам, тип результата будет соответствовать типу операнда с максимальной мощностью (максимальным диапазоном значений). В нашем случае — это `integer`. Результат `32769` вышел за пределы этого типа, но, что очень важно, не типа `longint`, использованного при автоопределенном преобразовании типа.

Обратите внимание — для целочисленных типов `shortint`, `byte`, `word`, `integer`, `longint` возможность *переполнения* при выполнении арифметических операций *не обнаруживается*. При переполнении для переменных вещественного типа может произойти программное прерывание. Поэтому очень важно правильно описать типы величин, используемых в программе (например, вместо типа `real` воспользоваться `extended`).

Например, рассмотрим программу вычисления частного двух чисел:

```
var a,b: integer; c: real;
begin
   write('введите a = '); readln(a);
   write('введите b = '); readln(b);
```

```
   c:=a/b;
   writeln('c= ',c:5:2)
end.
```

При вводе исходных данных a=33000 и b=33 результат составит не 1000, как это можно было бы ожидать, а -985.94. Предупреждения об ошибке не возникнет. Причина состоит в том, что мы, указав в разделе описания переменных для величин a и b тип integer, зарезервировали в памяти место для хранения только целых чисел, принимающих значения в интервале от −32 768 до 32 767. С клавиатуры же было введено число 33000, которое превышает верхнюю границу этого диапазона.

Директивы проверки *{$Q+}* и *{$R+}*

Если типы переменных, участвующих в выражении, совпадают, то никаких преобразований не делается, и все вычисления происходят в том же типе. Вычисленный результат преобразуется (если это требуется) к типу переменной, стоящей в левой части оператора присваивания. Преобразование возможно только из типа integer (а также byte, shortint, word, longint) к типу real (или любому другому вещественному типу). Это может привести к ошибкам, например, при выполнении операций умножения и сложения.

Например, рассмотрим следующую программу:

```
{$Q-}
var a,b: word; c,d: real;
begin
   a:=40000; b:=40000; c:=a*b;
   writeln('a * b = c = ',c:5:2);
   a:=1; b:=2; c:=a-b;
   writeln('a - b = c = ',c:5:2);
   d:=-a+b;
   writeln('-a + b = d = ',d:5:2);
end.
```

При выполнении программа напечатает строки:

```
a * b = c = 4096.00
a - b = c = 65535.00
-a + b = d = 1.00
```

Причина неверного результата состоит в том, что при умножении переменных типа word переполнение не обнаруживается, а при их вычитании отрицательные числа не образуются (тип word имеет лишь положительные значения). В то же время смена знака переменной a ведет к перемене типа при

выполнении вычисления. Полученный результат в любом случае преобразуется к типу переменной `c`.

Для *включения проверки переполнения* необходимо использовать директиву компилятора `{$Q+}` или воспользоваться пунктом меню интегрированной среды Turbo Pascal **Options | Compiler | Overflow checking**. В этом случае при выполнении последнего примера на операторе `c:=a*b;` будет выдано сообщение об ошибке: **Error 215: Runtime error** (Ошибка 215: Ошибка арифметического переполнения). Аналогичный результат мы получим и для оператора `c:=a-b;`. Однако использование оператора `d:=-a+b;` по-прежнему не вызовет ошибки и обеспечит правильный результат. Наиболее безопасный способ:

```
c:=a; c:=c-b;
```

Для *включения проверки выхода порядковой переменной за границы типа* (см. табл. 2.1—2.3) необходимо использовать директиву компилятора `{$R+}` или воспользоваться пунктом меню интегрированной среды Turbo Pascal **Options | Compiler | Range checking**.

Например, рассмотрим следующую программу:

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

При выполнении программа напечатает строку: **`i = 64000 l = 32000 j = 0`**.

Здесь `j=0` потому, что максимальное значение `j` равно 65 и меньше 255 (в отличие от `l`). Для его представления отводится только один байт, а у 32 000 в младшем байте находится 0. Если же в примере использовать директиву `{$R+}`, то при выполнении на операторе `i:=k+k;` будет выдано сообщение об ошибке: **Error 201: Range check error** (Ошибка 201: Ошибка диапазона).

Обратите внимание — к сожалению, использование директивы `{$R+}` далеко 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).

Замечание

Директива проверки границ по умолчанию выключена — `{$R-}`. В состоянии `{$R+}` она включает генерацию кода с проверкой соответствия переменных и массивов заданным границам. Проверка невозможна для вещественных переменных. Включение проверки увеличивает размер кода и время выполнения программы. Поэтому в процессе отладки проверку необходимо использовать, что позволит устранить ошибки в программе, а перед созданием исполняемого exe-файла программы ее можно отключить.

Возведение в степень

Вычисление степени числа выполняется в Turbo Pascal с использованием свойств логарифмов (см. листинг 5.8):

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

Таким образом, нельзя возвести в степень отрицательное число. Для этого можно использовать операторы *циклов (см. разд. 3.9)*.

Полезные формулы

Для вычисления логарифма с основанием a используем:

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

В Turbo Pascal определены только три тригонометрические функции: `sin`, `cos`, `arctg` (табл. 3.2). Для вычисления остальных тригонометрических функций необходимо использовать известные соотношения:

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

Побитовые операции

В языке Turbo Pascal можно выполнить ряд побитовых арифметических операций над десятичными операндами.

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

Операнды записываются в десятичной форме, но во время выполнения переводятся в двоичную. Результат представлен в десятичной.

Например, вычислим результат выражения a and b, если a=12 и b=22. Операнды a и b являются целыми, следовательно, занимают в памяти по два байта (см. табл. 2.1). Их двоичное представление:

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

Или в десятичной форме: 12 and 22 = 4.

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 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. При сдвиге вправо отрицательных значений освободившиеся разряды заполняются единицами. Младший бит всегда расположен справа, а старший — слева. 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 +.

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`

Операнды записываются в десятичной форме, но во время выполнения переводятся в двоичную. Результат представлен в десятичной.

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) }
```

Или в десятичной форме: `12 or 22 = 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`

Операнды записываются в десятичной форме, но во время выполнения переводятся в двоичную. Результат представлен в десятичной.

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) }
```

Или в десятичной форме: `12 xor 22=26`.

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

Функция `hi(i)` выделяет старший байт значения `i` и помещает его в младший байт результата. Старший байт результата равен нулю. Результат имеет целочисленный тип. 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. Результат имеет целочисленный тип. Например, `x:=hi($7893); { после выполнения x=$0093 }`.

Функция `swap(i)` обменивает содержимое младшего и старшего байтов целочисленного выражения `i`. Результат имеет целочисленный тип. Например, `x:=swap($7893); { после выполнения x=$9378 }`.

Кроме того, в Turbo Pascal определена операция получения указателя @. Она позволяет создать указатель на переменную. Подробнее об указателях будет рассказано в *гл. 13*.

Приоритет операций

Последовательность выполнения операций в составе выражения происходит с учетом их *приоритета (старшинства)*. В табл. 3.3 приведен порядок выполнения всех основных операций (арифметических и логических). Подробнее о логических операциях будет рассказано ниже *(см. разд. 3.8.1)*.

Таблица 3.3. *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. 10)*.

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)`.

Листинг 3.1 содержит программу вычисления площади треугольника по двум сторонам и углу между ними. Угол вводится в градусах и переводится в радианы.

Листинг 3.1. Вычисление площади треугольника

```
uses crt;
var
   a,b: real;    { длины сторон }
   angle: real; { величина угла в градусах }
   area: real;  { площадь треугольника }
begin
   textbackground(blue);    { цвет фона }
   textcolor(green);        { цвет символов }
   clrscr;                  { очистка экрана }
   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.
```

Комментарии

Процедура `clrscr` очищает экран и помещает курсор в его левый верхний угол. Для того чтобы процедура `clrscr` была доступна программе, в ее начало помещается строка `uses crt;`. Стандартный модуль `crt` также содержит процедуры, используя которые можно задать цвет фона и цвет символов, выводимых на экран: `textbackground` и `textcolor` соответственно *(см. гл. 8)*.

Листинг 3.2 содержит программу, которая вычисляет месячные выплаты m по займу в s рублей на n лет под процент p. Вычисления выполняются по формулам:

$$m = \frac{sr(1+r)^n}{12((1+r)^n - 1)}; \qquad r = p/100.$$

Листинг 3.2. Вычисление ежемесячных выплат по займу

```
var
   m,s,p,r,n,a,d: real;
   rub,kop: integer;    { целая и дробная часть числа (рубли и копейки) }
begin
   writeln('Введите заем, процент и количество лет в одной строке: ');
   readln(s,p,n);
   r:=p/100;
   a:=exp(ln(1+r)*n);   { вычисление степени числа через логарифм }
   m:=(s*r*a)/(12*(a-1));
   m:=trunc(100*m+0.5)/100; { округление до копейки }
   d:=m*n*12-s;             { общая прибыль }
   writeln;
   rub:= round(s*100) div 100; kop:= round(s*100) mod 100;
   write('Взято ',rub,' руб. ',kop,' коп. ');
   write('под ',p:5:2,'% на ',n:5:2,' лет');
   writeln;
   rub:= round(m*100) div 100; kop:= round(m*100) mod 100;
   writeln('Месячная выплата = ',rub,' руб. ',kop,' коп. ');
   rub:= round(d*100) div 100; kop:= round(d*100) mod 100;
   writeln('Общая прибыль = ',rub,' руб. ',kop,' коп. ');
end.
```

Комментарии

Прочитав листинг, обратите внимание: если ввести нулевое значение процента p или количества лет n, то при выполнении программы возникнет сообщение: **Run time error 200** (Ошибка времени выполнения 200), свидетельствующее о попытке деления на ноль. Для того чтобы исключить подобную ситуацию, при вводе необходимо выполнить проверку значения p и n. Если p и n строго больше нуля, то программа должна вычислить выплаты m, в противном случае необходимо вывести сообщение об ошибке и повторить ввод исходных данных. Таким образом, *даже при осуществлении сравнительно простых вычислений, программист сталкивается с необходимостью проверки условий и выполнения ветвлений*. Для этого используются условный оператор и оператор выбора *(см. разд. 3.8)*.

Для преобразования числа в денежный формат (выделения рублей и копеек) в программе используются операции `div` и `mod`. Обратите внимание, что в тексте программы трижды повторяется пара операторов для выделения значений рублей и копеек (`rub` и `kop`), при этом меняется только имя переменной: `s`, `m`, `d` (заем, выплата и прибыль). Возникает закономерный вопрос: можно ли записать эти операторы один раз, а затем трижды "вызвать" их для вычисления результата — переменных `rub` и `kop`, подставляя каждый раз имена различных исходных данных — переменных `s`, `m` и `d`. Для решения подобных задач в Turbo Pascal применяют *процедуры пользователя* (см. листинг 5.4).

Листинг 3.3 содержит программу, вычисляющую сумму цифр трехзначного числа.

Листинг 3.3. 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 ',100*first+10*second+third,' = ',sum);
end.
```

3.4. Безусловный переход. Оператор *GOTO*

Если в программе после выполнения очередного оператора надо выполнить не следующий по порядку, а другой, помеченный для этого меткой *(см. разд. 2.2.3)*, используется оператор безусловного перехода `goto`. Он осуществляет переход к инструкции, перед которой стоит метка, объявленная в разделе `label`, и имеет вид:

```
goto ИмяМетки;
```

Например:

```
label   metka01;
...
begin
...
metka01 : Оператор;
...
       goto metka01;
end.
```

Следует знать:

- метка, на которую передается управление, должна быть описана в разделе описания меток того блока (основной программы, процедуры или функции), в которой 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 некоторых языках программирования этот оператор отсутствует *(см. разд. 6.1)*.

3.5. Оператор вызова процедуры

Оператор вызова процедуры в виде:

```
ИмяПроцедуры (Параметр1,Параметр2,..,ПараметрN);
```

служит для выполнения предварительно определенной пользователем или стандартной процедуры. Более подробно они будут рассматриваться в *гл. 5*.

3.6. Пустой оператор

Не содержит символов и не выполняет действий. Может быть помечен меткой. Чаще всего используется для выхода из программы или составного оператора.

Например:

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

Лишняя точка с запятой в тексте программы, 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.

Если помеченный пустой оператор стоит непосредственно перед зарезервированным словом `end` составного оператора или программы, то между ним и предшествующим оператором обязательно ставится точка с запятой.

Например:

```
      x:=-y; { предшествующий оператор и точка с запятой }
   label01:  { помеченный меткой пустой оператор }
end.
```

3.7. Составной оператор

Составной оператор представляет собой группу из произвольного числа операторов, отделенных друг от друга точками с запятой, и ограниченную *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` представляют собой 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 Turbo Pascal каждому `begin` соответствует свой `end`, но не наоборот, т. к. на `end` заканчиваются разделы, начинающиеся с `case` *(см. разд. 3.8.3)* и `record` *(см. гл. 11)*, но без `begin` вначале.

3.8. Условный оператор и оператор выбора

Условные операторы и операторы выбора обеспечивают выполнение или невыполнение некоторого оператора (в том числе и составного) *в зависимости* от справедливости *проверяемого условия*. Эти операторы *прерывают естественную линейную последовательность операций*, выполняемых по очереди в порядке записи, после чего алгоритм может пойти по одной из двух (или более) возможных ветвей в соответствии с заданным условием. Такая алгоритмическая конструкция называется *ветвлением*. При *обходе* несколько шагов алгоритма пропускаются в зависимости от справедливости проверяемого условия.

3.8.1. Логические выражения и отношения

Проверяемое условие обычно записывается в виде *логического выражения*, состоящего из одного или нескольких *отношений* (табл. 3.4).

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

Таблица 3.4. *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`

Используя отношения, при помощи знаков *логических операций* (табл. 3.5) 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 3.5. *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 выражении не использованы круглые скобки, то операции выполняются в порядке их старшинства или убывания приоритетов (см. табл. 3.3). Операции одинакового старшинства выполняются слева направо. Для задания явного порядка используются круглые скобки.

Обратите внимание — в Turbo Pascal логические операции имеют более высокий приоритет, чем операции отношения (см. табл. 3.3). 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). В меню интегрированной среды Turbo Pascal представлена опция, соответствующая директиве компилятора `{$B}`: **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).

Следует знать:

- в языке Turbo Pascal нельзя записать двустороннее неравенство вида `1<x<2`. Нужно 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
```

В Turbo Pascal имеются две реализации *ветвления*. Это условный оператор — инструкция `if` и оператор выбора — инструкция `case`.

3.8.2. Условный оператор *IF*

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;
```

В последнем случае говорят о сокращенной форме условного оператора.

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`. Часть оператора, стоящая после служебного слова `else`, может *отсутствовать*. В этом случае при ложности проверяемого условия просто выполняется следующая по порядку за оператором условия инструкция.

Действие условного оператора можно пояснить с помощью блок-схем (рис. 3.1, *а*, *б*).

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');
```

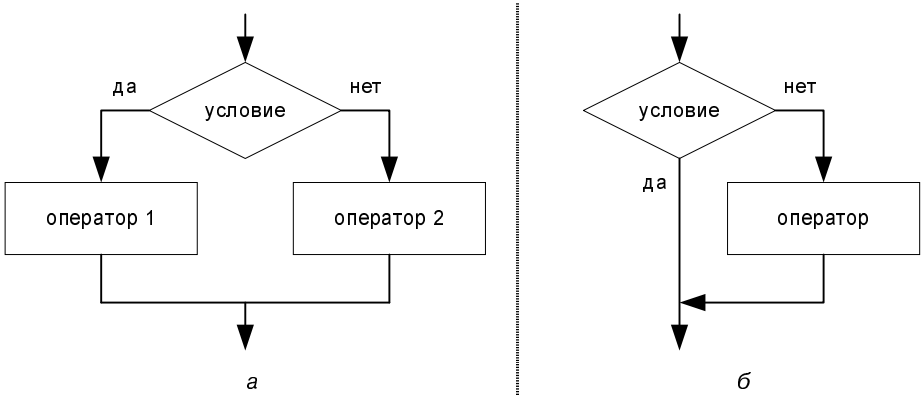

Рис. 3.1. Условные обозначения на схемах алгоритмов: *а* — ветвления и *б* — обхода

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;
```

- Вариант 3.

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

Если 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.

Следует знать:

- при вложенности операторов каждое `else` соответствует тому `then`, которое непосредственно ему *предшествует* (обратите внимание на пример в варианте 3);

- конструкций со степенью вложенности более 2—3 необходимо *избегать* из-за сложности их анализа при отладке программы;
- в условных операторах часто используют составной оператор `begin...end`. Если между `begin` и `end` находится *только одна инструкция*, слова `begin` и `end` лучше не писать;
- в условных операторах *точка с запятой не ставится* после `then` и перед `else`;
- самое простое логическое выражение — одна переменная логического типа, которая играет роль целого выражения;
- сложные логические выражения могут содержать несколько отношений и/или логических переменных, связанных операциями `and`, `or`, `not` (не забывайте заключать каждое отношение в скобки);
- с целью предупреждения ошибок выполнения *(см. разд. 1.4)* инструкцию `if` удобно использовать для соответствующей проверки (см. листинги 3.5, 3.10).

Листинг 3.4 содержит программу, которая проводит тест по таблице умножения. Формируются два случайных сомножителя, затем подсчитывается их произведение и сравнивается с тем значением, который ввел ученик.

Листинг 3.4. Тест по таблице умножения (вариант № 1)

```
var
   s1,s2: integer;   { сомножители }
   otvet: integer;   { ответ ученика }
begin
   randomize;        { инициализация датчика случайных чисел }
   { s1, s2 — случайные числа в диапазоне от 2 до 19 включительно }
   s1:=random(18)+2; s2:=random(18)+2;
   write('Сколько будет ',s1,'*',s2,'?: '); readln(otvet);
   if otvet=s1*s2 then writeln('Правильно!') else writeln('Неверно...');
end.
```

Листинг 3.5 содержит программу, которая вычисляет частное двух целых чисел. В связи с тем, что делить на ноль нельзя, организуем контроль ввода данных. Для контроля вводимых значений делителя используем оператор условного перехода `if`.

Листинг 3.5. 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.
```

Листинг 3.6 содержит программу, которая проверяет на четность введенное число.

Листинг 3.6. 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.
```

Комментарии

Для проверки на нечетность можно использовать функцию `odd`:

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

Листинг 3.7 содержит программу, которая сравнивает возраст брата и сестры и выводит соответствующее сообщение.

Листинг 3.7. 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 3.8).

Листинг 3.8. 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.
```

Тем не менее, во многих случаях применение вложенных условных операторов позволяет избежать сложных условий с использованием логических функций.

Листинг 3.9 содержит программу, которая проверяет, может ли существовать треугольник с заданными сторонами. Из геометрии известно, что сумма двух любых сторон должна быть больше третьей. Эта программа иллюстрирует использование сложного условия.

Листинг 3.9. 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.
```

Листинг 3.10 содержит программу решения квадратного уравнения с использованием сложных условий. Это единственный пример в книге, который иллюстрирует применение инструкции `goto`. Программу можно было бы написать и без нее, при помощи, например, процедуры `halt` *(см. разд. 6.2.1)*. Программа решения квадратного уравнения с использованием подпрограммы-функции приводится в листинге 5.10.

Листинг 3.10. 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 в одной строке 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.
```

3.8.3. Оператор *CASE*

Обычно при написании программы не рекомендуется использовать многократно вложенные друг в друга условные операторы `if` — 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 и в зависимости от них выполнять те или иные действия? Для этих целей в языке Turbo 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 между begin и end выполняются в том случае, если значение выражения после слова case *совпадает* с константой из соответствующего списка. Если это не так, то выполняются инструкции, идущие 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.

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 *равенства* значения *выражения-селектора* (переменной) и значения *константы выбора*, указанной перед группой инструкций;
- выражение-селектор должно иметь *порядковый тип*: чаще всего — `integer`, реже — `char`, `boolean` или один из пользовательских типов *(см. разд. 2.1.3)*. Диапазон значений выражения-селектора от −32 768 до 32 767. 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;
- точка с запятой не ставится после последнего элемента списка выбора.

Листинг 3.11 содержит программу, которая определяет день недели известной даты (по современному европейскому календарю).

Листинг 3.11. 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.
```

В листинге 3.26 приведен пример использования вложенных операторов `case`.

3.9. Операторы повтора (циклы)

С помощью условных операторов и операторов присваивания можно реализовать самый сложный алгоритм. Однако в программах, связанных с обработкой данных или 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 *(см. разд. 6.1)*.

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

В языке программирования Turbo Pascal имеется три разновидности цикла — *цикл с постусловием* (инструкция `repeat`), *цикл с предусловием* (инструкция `while`) и *цикл со счетчиком* (инструкция `for`).

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;
- 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.

3.9.1. Оператор *REPEAT*

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;
```

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. 3.2 приведена блок-схема оператора повтора `repeat`.

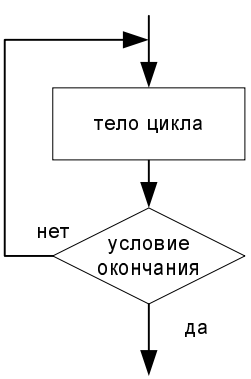

Рис. 3.2. Условное обозначение на схемах алгоритмов оператора `repeat`

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`, которая вводится с клавиатуры, не совпадет со значением переменной `comp` (листинг 3.12).

Листинг 3.12. Игра "Угадай число" (вариант № 1)

```
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.
```

Оператор `repeat` может использоваться для *проверки правильности ввода исходных данных*. 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.
```

Другим примером использования оператора `repeat` является *повтор фрагмента или всей программы*, для того чтобы произвести необходимые действия, но уже с другими данными. Чтобы каждый раз не перезапускать программу, необходимо оформить `{ Повторяемый фрагмент }` при помощи `repeat`.

Например, рассмотрим соответствующий фрагмент программы:

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

Комментарии

Функция `upcase` преобразует строчную букву в прописную, во всех остальных случаях ее результат совпадает с аргументом (кириллица не обрабатывается). Таким образом, вне зависимости от регистра клавиши, нажатой пользователем — `Y` или `y`, будет выполняться повтор группы операторов `{ Повторяемый фрагмент }`. В противном случае программа завершит работу.

Следует знать:

- *число повторений* операторов (инструкций) цикла `repeat` определяется в ходе работы программы и во многих случаях заранее неизвестно;
- инструкции цикла `repeat` будут выполняться, пока условие, стоящее после `until`, будет оставаться *ложным*;
- после слова `until` записывается условие *завершения* цикла;
- *условие* — это выражение логического типа: *простое выражение отношения* или *сложное логическое выражение*;
- для успешного завершения цикла `repeat` в его теле обязательно должны быть 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. 2.2.6)* и процедуры `inc` или `dec` *(см. разд. 3.3.1)*.

  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 напечатает 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* (условие проверяется *после* выполнения тела цикла), т. е. инструкции тела цикла будут выполнены *хотя бы один раз*. Далее мы расскажем, что остальные циклы 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.

Обратите внимание — цикл `repeat` удобно использовать для *обработки ошибок ввода/вывода* при несоответствии типа введенных данных типу переменной, значение которой вводится с клавиатуры. В этом случае программа завершает работу и на экран выводится сообщение об ошибке с кодом 106. Контроль соответствия типа базируется на использовании директивы компилятора `{$I-}` *(см. разд. 2.1.4)* и стандартной функции `IOResult`.

Замечание

Для того чтобы "поймать" ошибки ввода/вывода, используя функцию `IOResult`, нужно отключить системный контроль за ними (директива компилятора `{$I-}`). Если происходит ошибка ввода/вывода и соответствующая проверка отключена, то все последующие операции ввода/вывода игнорируются до тех пор, пока не будет сделан вызов `IOResult`. После вызова функция сбрасывает свой внутренний флаг ошибки. Анализ результата, возвращаемого `IOResult`, основывается на том, что если директива `{$I-}` задана и операция ввода/вывода была успешной, `IOResult` будет возвращать 0. Иначе — ненулевой код ошибки. Ошибочная ситуация должна быть обработана в программе пользователя.

Например:

```
var x: integer; ok: boolean;
   begin
...
      repeat
         writeln('Введите целое x');
         {$i-}   { отмена контроля операций ввода/вывода }
         readln(x);
         ok:=(ioresult=0);
         { обработка ошибочной ситуации на основе анализа значения, }
         { возвращенного ioresult }
         if not ok then writeln('Ошибка ввода. Повторите.')
         {$i+}   { включение контроля операций ввода/вывода }
```

```
      until ok;
      writeln('x= ',x:3)
...
   end.
```

Листинг 3.13 содержит программу "Тест по таблице умножения", которая уже приводилась в качестве примера при рассмотрении условного оператора (см. листинг 3.4). В результате использования 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 3.13. Тест по таблице умножения (вариант № 2)

```
uses crt;
var s1,s2,otvet,kol,prav: integer; yn: char;
begin
   randomize;   { инициализация датчика случайных чисел }
   clrscr;      { очистка экрана }
   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');
   clrscr;      { очистка экрана }
   writeln('Результаты теста: ');
   write('Задано вопросов: ',kol,'. Правильных ответов: ',prav,'.');
   readln;
end.
```

Листинг 3.14 содержит программу, которая имитирует арифметический калькулятор (используются операторы присваивания, условия и повтора).

Листинг 3.14. Арифметический калькулятор

```
var
   x,y,result: real;
   operation,ans: char;
```

```
begin
   repeat                          { начало цикла с постусловием }
      write('x = '); read(x);      { считывание первого операнда }
      write('y = '); readln(y);    { считывание второго операнда }
      write('Операция (+,-,*,/) > ');
      readln(operation);           { считывание знака операции }
      case operation of            { выбор арифметического действия }
         '+' : result:=x+y;
         '-' : result:=x-y;
         '*' : result:=x*y;
         '/' : result:=x/y;
      else
         writeln('Ошибка ввода');
      end;
      { вывод арифметического выражения }
      writeln(x:7:3,operation:3,y:7:3,' = ',result:7:3);
      write('Продолжить (y/n)');
      readln(ans);                 { считывание ответа на вопрос }
   until (ans='n')or(ans='n');  { проверка условия окончания цикла }
end.
```

Листинг 3.15 содержит программу, которая вводит и суммирует любое количество целочисленных значений. Если введено значение 999, то на экран выводится результат суммирования.

Листинг 3.15. Суммирование любого количества целочисленных значений

```
var count,x: integer; sum: real;
begin
   count:=1; sum:=0;
   repeat                       { повторять }
      { начало тела цикла }
      write('Введите значение ',count,' -го слагаемого: ');
      readln(x);                { ввести очередное значение x }
      count:=count+1;
      if x<>999 then sum:= sum+x;
   until x=999;                 { условие окончания цикла }
   writeln('Сумма введенных чисел = ',sum:7:3);
end.
```

Листинг 3.16 содержит другой вариант игры "Угадай число". Программа случайным образом выбирает число в диапазоне от 1 до 10 и предлагает пользователю угадать его за 5 попыток.

Листинг 3.16. Игра "Угадай число" (вариант № 2)

```
const npop=5;          { количество попыток }
var
   comp: integer;     { число, "задуманное" компьютером }
   igrok: integer;    { вариант игрока }
   n: integer;        { кол-во попыток, сделанное игроком }
begin
   n:=0;
   randomize;         { инициализация датчика случайных чисел }
   comp:=random(9)+1; { компьютер задумал число }
   writeln('Компьютер "задумал" число от 1 до 10');
   writeln('Угадайте его за ',npop,' попыток');
   writeln('Введите число');
   repeat
      n:=n+1;
      write('->'); readln(igrok);
   until(n=npop)or(comp=igrok);
   if comp=igrok then writeln('Вы выиграли!')
      else writeln('Вы проиграли... Компьютер задумал число ', comp);
end.
```

Листинг 3.17 содержит программу, которая определяет число и сумму цифр для любого натурального числа, не превосходящего `maxint` (см. табл. 2.7), а также выводит запись числа в обратном порядке, опуская незначимые нули в его начале.

Листинг 3.17. Работа с числом

```
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.
```

Многие из математических величин или значений функций могут быть выражены как *суммы бесконечных последовательностей.*

Например:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + \frac{(-1)^{n-1}}{2n-1} + \ldots,$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots,$$

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

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + \frac{(-1)^{n-1}}{2n-1} + \ldots,$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots,$$

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

где *n!* — *факториал* натурального числа *n*, т. е. произведение чисел $1 \times 2 \times \ldots \times n$.

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 3.18 содержит программу, которая вычисляет косинус угла.

Листинг 3.18. 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.
```

Вещественные числа в компьютере представлены приближенно. Для того чтобы оценить *точность* любых вычислений (в том числе и рассмотренных выше), необходимо знать, сколько цифр (знаков) выделено для представления числа. Листинг 3.19 содержит программу, которая выводит *число десятичных знаков*, выделяемых для представления действительного числа.

Листинг 3.19. Число 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.
```

Комментарии

Цикл заканчивается, когда 1.0=1.0...01. Это значит, что последняя цифра числа 1.0...01 исчезла. Результатом выполнения программы является число 12, что согласуется с представлением о точности вещественного типа (см. табл. 2.2).

3.9.2. Оператор *WHILE*

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;
```

Оператор `while` аналогичен оператору `repeat`, но проверка `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. 3.3 приведена блок-схема оператора повтора while.

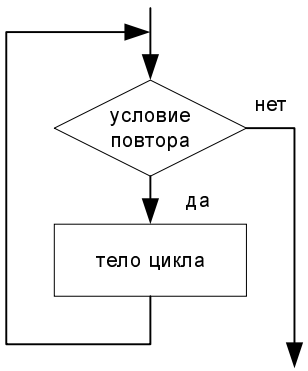

Рис. 3.3. Условное обозначение на схемах алгоритмов оператора while

Взаимосвязь операторов while и repeat:

- оператор while Условие do Инструкция; *эквивалентен* оператору
  if Условие then repeat Инструкция until Not Условие;

Например, 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;
- после слова while записывается условие *продолжения выполнения* инструкций цикла, в этом отличие цикла while от цикла repeat;
- *условие* — это выражение логического типа: *простое выражение отношения* или *сложное выражение отношения (логическое выражение)*, которое может принимать одно из двух значений: true или false;

- для успешного завершения цикла `while` в его теле обязательно должны присутствовать инструкции, оказывающие *влияние* на условие выполнения инструкций цикла.

  Для управления циклом `while` удобно использовать функции `succ` или `pred` *(см. разд. 2.2.6)* и процедуры `inc` или `dec` *(см. разд. 3.3.1)*.

  Например, рассмотрим следующую программу:

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

  При выполнении программа напечатает строки:

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

- цикл `while` — это цикл с *предусловием*, т. е. инструкции тела цикла *вообще могут быть не выполнены*, если проверяемое условие ложно с самого начала;
- исходя из последнего утверждения цикл `while` считают самым универсальным видом цикла;
- цикл `while` обычно применяется 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`.

Листинг 3.20 содержит программу "Тест по таблице умножения", измененную таким образом, чтобы, при необходимости, можно было отказаться от выполнения теста и корректно выйти из программы. Такая ситуация может

возникнуть, например, при случайном запуске программы, не преследующем цель проверки знаний.

Листинг 3.20. Тест по таблице умножения (вариант № 3)

```
uses crt;
var s1,s2,otvet,kol,prav: integer; yn: char;
begin
   randomize;   { инициализация датчика случайных чисел }
   clrscr;      { очистка экрана }
   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
         clrscr;   { очистка экрана }
         writeln('Результаты теста: ');
         write('Задано  вопросов: ',kol);
         writeln('. Правильных ответов: ',prav,'.');
         readln;
      end;         { выводим результаты, только если тест состоялся }
end.
```

Листинг 3.21 содержит программу, которая производит суммирование произвольного количества целых чисел, вводимых с клавиатуры. Концом последовательности служит ввод отрицательного числа.

Листинг 3.21. Суммирование произвольного количества целых чисел

```
var count,item,sum: integer;
begin
   count:=0;       { счетчик чисел }
   sum:=0;         { сумма чисел }
```

```
   while true do   { бесконечный цикл }
      begin
         count:=count+1;
         write('Введите ', count, ' -е целое число: ');
         readln(item); { ввод очередного числа }
         if item<0 then break; { выход из цикла }
         sum:= sum+item;
      end;
   writeln('Сумма введенных чисел равна ', sum);
end.
```

Комментарии

Для выхода из бесконечного цикла (проверяемое условие никогда не станет ложным) используется специально для этих целей предназначенный оператор `break` (прервать). Также возможно, но менее желательно, использование оператора `goto`. В этом случае в текст программы необходимо будет ввести следующие изменения:

```
label lab01;
...
if item<0 then goto lab01;
...
lab01: writeln('Сумма введенных чисел равна ', sum);
```

Листинг 3.22 содержит программу, которая выводит на экран таблицу значений функции. Вывод выполняется в два столбца: первый — значения аргумента, второй — значения функции при изменении аргумента от 0,3 до 3,7 с шагом 0,4.

$$f(x) = \lg(3) + x\sqrt{5\sin\left(\frac{\pi x}{3}\right)}.$$

Листинг 3.22. 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.
```

Комментарии

При разработке программы следует *учитывать область определения функции* и в случае необходимости, для предупреждения попытки вычисления квадратного корня из отрицательного числа, организовать вывод сообщения — "функция не определена".

Ранее была рассмотрена программа (см. листинг 3.18), вычисляющая косинус введенного угла при помощи цикла `repeat`. Листинг 3.23 содержит программу, которая, используя цикл `while`, вычисляет синус угла.

Листинг 3.23. Вычисление синуса 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:=x; u:=x; n:=1;
   while abs(u)>eps do
      begin
         n:=n+2;
         u:=-u*x*x/((n-1)*n);
         s:=s+u;
      end;
   writeln('sin(',x*180/pi:6:2,') = ',s:7:3);
end.
```

Программа, представленная листингом 3.19, вычисляла с помощью цикла `repeat` количество десятичных знаков, выделяемых для представления дей-

ствительного числа. Листинг 3.24 содержит программу, которая, используя цикл while, выводит так называемое "машинное эпсилон". Это такое минимальное, не равное нулю число, после прибавления которого к единице, компьютер уже выдает результат, отличный от единицы.

Листинг 3.24. Определение "машинного эпсилон"

```
var eps: real;
begin
   eps:=1.0;
   while eps/2+1 > 1 do eps:=eps/2;
   writeln('Машинное epsilon = ', eps);
end.
```

Листинг 3.25 содержит программу, которая представляет заданное натуральное число при помощи римских цифр. При этом 500 обозначается D, 100 — C, 50 — L, 10 — X, 5 — V, 1 — I.

Листинг 3.25. Запись натурального числа римскими цифрами (вариант № 1)

```
var i: integer;
begin
   write('Введите целое десятичное число: '); readln(i);
   write(i,'   ');
   while i>=500 do
      begin
         write('D'); i:=i-500
      end;
   while i>=100 do
      begin
         write('C'); i:=i-100
      end;
   if i>=50 then
      begin
         write('L'); i:=i-50;
      end;
   while i>=10 do
      begin
         write('X'); i:=i-10
      end;
   if i>=5 then
      begin
         write('V'); i:=i-5;
      end;
```

```
   case i of
      0:;
      1: write('I');
      2: write('II');
      3: write('III');
      4: write('IV');
   end;
   writeln; readln;
end.
```

Комментарии

Использование оператора `write`, а не `writeln` позволяет при выполнении программы формировать символы для представления числа римскими цифрами, помещая их последовательно в выходную строку, а не в столбец.

Если текущее значение числа `i` оказывается равным нулю, то выводить ничего не нужно. Поэтому для константы выбора 0 оператора `case` не указывается никаких действий. Оператор `case` будет выполняться, но число уже сформировано.

Решим обратную задачу, которая встречается при реализации *автоматных распознавателей*. Листинг 3.26 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 соответствий* (табл. 3.6).

Таблица 3.6. *Таблица соответствий 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 3.26. 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.
```

Комментарии

Перед нажатием клавиши <Enter>, завершающей ввод данных, необходимо ввести точку или пробел.

Функция `eoln` проверяет, не является ли очередной символ буфера клавиатуры символом "новая строка". 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>.

В программе используются вложенные операторы `case`.

Для работы с цифрами M=1000, D=500, C=100 и L=50 таблицу нужно расширить.

3.9.3. Оператор *FOR*

Этот вид оператора цикла называют *циклом со счетчиком* или *циклом с параметром*. 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. 3.4).

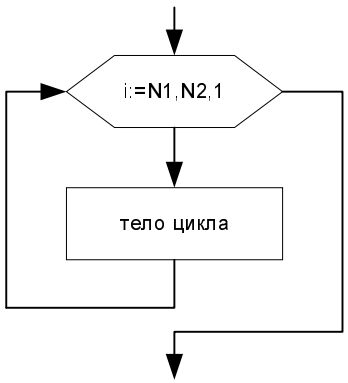

Рис. 3.4. Условное обозначение на схемах алгоритмов цикла с параметром

Инструкции между begin и end выполняются столько раз, сколько 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.

Например, выполнение цикла — фрагмента программы:

```
for i:=10 to 14 do write(i: 3);
```

выведет на экран последовательность цифр в виде:

```
10 11 12 13 14
```

Вариант 2 (с уменьшением счетчика).

```
for Счетчик := НачальноеЗначение downto КонечноеЗначение do
begin
   { Инструкции }
end;
```

Инструкции между begin и end выполняются столько раз, сколько определяет выражение [(НачальноеЗначение – КонечноеЗначение) + 1].

Если НачальноеЗначение меньше, чем КонечноеЗначение, то инструкции между begin и end не выполняются ни разу.

Например, выполнение цикла — фрагмента программы:

```
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 `char` тип, то оператор:

```
for ch:= 'a' to 'e' do write (ch: 2);
```

выведет на экран последовательность букв в виде:

```
a b c d e
```

Оператор:

```
for ch:= 'e' downto 'a' do write (ch: 2);
```

выведет на экран последовательность букв в виде:

```
e d c b a
```

Обратите внимание — цикл `for` удобно использовать для организации вывода данных программы. Вместе с оператором `if` и функцией `readln` он позволяет выполнить постраничный вывод.

Например, при выполнении следующего цикла программа будет приостанавливать вывод после заполнения экрана столбцом цифр до нажатия клавиши <Enter>.

```
for i:=1 to 50 do
   begin
      writeln(i);
      if i mod 24=23 then readln;
   end;
```

Следует знать:

- оператор (инструкция) `for` используется для организации циклов с *фиксированным*, заранее известным или определяемым во время выполнения программы числом повторений;
- *количество повторений* цикла определяется *начальным и конечным значениями переменной-счетчика*. Оператор `for` обеспечивает выполнение тела цикла до тех пор, пока не будут перебраны *все значения* параметра цикла: от начального до конечного;
- переменная-счетчик должна быть *порядкового типа:* чаще — `integer`, реже — `char`, `boolean` или одного из пользовательских типов *(см. разд. 2.1.3). 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. 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 компьютером условия выхода из цикла `i` никогда не будет равно 9 (сначала 8, а потом сразу 10).

Замечание

Для досрочного выхода из цикла можно использовать оператор `goto` или оператор `break`. Но все-таки проявлением лучшего стиля программирования в этом случае будет использование циклов `repeat` и `while`. Процедура `continue` позволяет прервать выполнение тела любого цикла и передает управление на его заголовок, заставляя цикл немедленно перейти к следующему выполнению (итерации).

Например, рассмотрим фрагмент 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;
...
```

Листинг 3.27 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 3.27. Тест по таблице умножения (вариант № 4)

```
uses crt;
var s1,s2,otvet,k,prav: integer;
begin
   randomize;   { инициализация датчика случайных чисел }
   clrscr;      { очистка экрана }
   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;
   clrscr;   { очистка экрана }
   writeln('Ваша оценка: ',prav); readln;
end.
```

Листинг 3.28 содержит программу, которая вычисляет среднее арифметическое и определяет минимальное и максимальное число в последовательности действительных чисел, вводимых с клавиатуры.

Листинг 3.28. 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.
```

Комментарии

Для решения поставленной задачи нет необходимости хранить введенные числа. Однако, зачастую, особенно для последовательности случайных чисел, важно знать не только ее 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 ввести еще раз, что трудоемко, или *запомнить при первом вводе и использовать повторно*. Для этого используются *массивы (см. гл. 4)*.

Листинг 3.29 содержит программу, которая вычисляет частичные суммы `n` членов ряда 1, −1/3, +1/5, −1/7, +..., сходящегося к значению π/4.

Листинг 3.29. 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.
```

Комментарии

Поскольку все четные члены ряда имеют отрицательный знак, поставленную задачу также можно решить при помощи функции `mod`, используемой для проверки номера очередного слагаемого на четность (см. листинг 3.6).

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;
...
```

В теле цикла не происходит изменения значения управляющей переменной `i`, хотя она принимает явное участие в расчете значения очередного члена ряда.

Цикл с уменьшением счетчика выполняет *суммирование в обратном порядке* — от слагаемых с наибольшими номерами и наименьшими значениями к слагаемым с наименьшими номерами, принимающим наибольшие значения. Такой порядок учитывает рекомендации по *уменьшению влияния ошибки округления* при выполнении арифметических операций с вещественными числами *(см. разд. 2.1.3)*. При "прямом" суммировании, начиная с некоторого номера, может возникнуть ситуация, когда при каждом выполнении цикла к относительно *большому* значению суммы будет прибавляться сравнительно *малое* значение очередного слагаемого. *Погрешность такой операции увеличивается*, в этом проявляется *особенность машинной арифметики*. При "обратном" порядке суммирования слагаемые меньше отличаются друг от друга по величине, поэтому точность вычисления полной суммы будет выше. Разумеется, для данного примера, в отличие от серьезных расчетов, различие чрезвычайно мало. При суммировании 10 000 членов ряда его сумма равна:

```
7.85373163397708E-0001 { для цикла for..downto };
7.85373163409531E-0001 { для цикла for..to };
7.85398163397448E-0001 { pi/4 }.
```

В курсе основ высшей математики большое внимание уделяется вычислению *определенных интегралов*. Есть интегралы, которые невозможно или очень трудно вычислить *аналитически*. Листинг 3.30 содержит программу, вычисляющую *приближенное значение* определенного интеграла по квадра-

турным формулам прямоугольников, трапеций и Симпсона для функции $y(x) = 3x^2$.

Листинг 3.30. Вычисление определенного интеграла (вариант № 1)

```
var
   a,b: real;          { границы отрезка }
   h: real;            { величина интервала }
   s1,s2,s3: real;     { приближенное значение интеграла }
   n: integer;         { количество интервалов }
   x,x1,x2: real;      { аргумент }
   i: integer;         { счетчик цикла }
begin
   writeln('Приближенное вычисление интеграла');
   write('Нижняя граница отрезка: '); readln(a);
   write('Верхняя граница отрезка: '); readln(b);
   write('Число разбиений отрезка: '); readln(n);
   h:=(b-a)/n;
   x1:=a+h/2; x2:=a;
   s1:=3*x1*x1;
   s2:=((3*x2*x2)+(3*b*b))/2;
   for i:=1 to n-1 do
      begin
         x1:=x1+h; x2:=x2+h;
         s1:=s1+(3*x1*x1); s2:=s2+(3*x2*x2);
      end;
   s1:=h*s1; s2:=h*s2; s3:=(2*s1+s2)/3;
   writeln('Значение интеграла по формулам:');
   writeln('прямоугольников    трапеций       Симпсона');
   writeln(' s1 = ',s1:8:5,'; s2 = ',s2:8:5, '; s3 = ',s3:8:5);
end.
```

Комментарии

Недостатком рассмотренного примера является необходимость многократно повторять формулу функции вида $3 \times x \times x$ в тексте листинга. Использование *функций пользователя* позволит значительно упростить внесение изменений в программу. Это может быть полезно при смене аналитического выражения для функции, стоящей под знаком интеграла (см. листинг 5.9).

Листинг 3.31 содержит программу, которая при помощи символа * (звездочка) симметрично относительно вертикали экрана выводит на экран график функции $\sin(x)/\exp(\alpha \cdot x)$. Для задания ширины поля при выводе используется выражение. Ширина поля изменяется от строки к строке; в каждой строке звездочка прижимается к правому краю поля.

Листинг 3.31. Вывод на экран графика функции sin(*x*)/exp(*αx*) с использованием символа *

```
uses crt;
const offset=40;     { график симметричен относительно вертикали экрана }
      scale=15;      { амплитуда }
      degreestep=40; { частота }
      alpha=0.15;
var i: integer; k: real;
begin
   clrscr;           { очистка экрана }
   { переводим угол в радианы }
   k:= degreestep*pi/180;
   for i:= 0 to 23 do
      writeln('*':round(offset+scale*sin(k*i)/exp(alpha*k*i)));
   readln;
end.
```

3.9.4. 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 применение вложенных циклов. С использованием цикла `for` соответствующий фрагмент программы имеет вид:

```
var i, j : byte;
begin
   for i:=1 to 10 do
      for j:=1 to 10 do
         writeln(i,' * ',j,' = ',i*j);
end.
```

На блок-схеме (рис. 3.5) 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 нача-

ла 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.

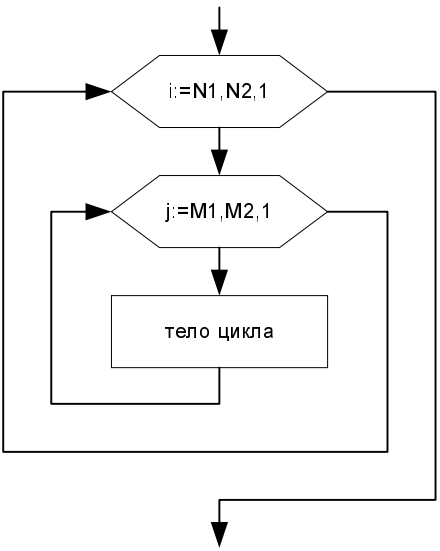

Рис. 3.5. Условное обозначение на схемах алгоритмов вложенных циклов с параметром

Еще один вариант программы, выводящей на экран таблицу умножения, но уже в виде "квадрата Пифагора", приведен ниже. Использование оператора `writeln` в теле внешнего цикла по `i` позволяет выполнить перевод курсора в начало следующей строки экрана при завершении вывода предыдущей.

Листинг 3.32 содержит программу, которая выводит на экран квадрат Пифагора — таблицу умножения.

Листинг 3.32. Вывод на экран таблицы умножения

```
var
   i,j: integer;   { номер строки и столбца таблицы }
begin
   for i:=1 to 10 do
      begin
         for j:=1 to 10 do write(i*j:5);
         writeln;   { перевод строки }
      end;
end.
```

Листинг 3.33 содержит программу, которая решает следующий числовой ребус:

```
  охохо { = O0O0O + X0X0 = O×10101 + X×1010 }
+ ахаха { = A×10101 + X×1010 }
=ахахах { = A0A0A0 + X0X0X = A×101010 + X×10101 }
```

Необходимо определить, какую цифру обозначает каждая из букв O, X и A.

Листинг 3.33. 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.
```

Комментарии

В вычислении участвуют три цифры. Программа перебирает все 9×9×10=810 вариантов ребуса и выводит единственный ответ: `90909+10101=101010`.

Листинг 3.34 содержит программу, определяющую все совершенные числа меньше *n*.

Замечание

Совершенное число — это число, равное сумме всех своих делителей, исключая себя самого, например 6=1+2+3. К настоящему времени найдено 24 совершенных числа (6, 28, 496, 8128...). Все они четные. Существуют ли нечетные совершенные числа и конечно ли их множество — неизвестно.

Листинг 3.34. Определение всех совершенных чисел, меньших заданного *n*

```
var i,j,n,m: integer;
begin
   write('Введите верхнюю границу: ');readln(n);
   for i:=2 to n do   { начало внешнего цикла }
      begin
         m:=0;        { обнуление начального значения суммы }
```

```
         { внутренний цикл: поиск и суммирование делителей }
         for j:=1 to i-1 do
   { если число i- без остатка делится на j, то число j — его делитель, }
   { поэтому увеличиваем значение m на величину значения j }
            if i mod j=0 then m:=m+j;
         if m=i then writeln(i);
      end;
end.
```

Листинг 3.35 содержит программу, которая для натурального n вычисляет сумму степеней: $(1/1)^n + (1/2)^n + ...+(1/n)^n$.

Листинг 3.35. Вычисление суммы степеней

```
var n,i,j: integer; a,s,p: real;
begin
   write('Введите показатель степени: '); readln(n);
   s:=0;              { обнуление начального значения суммы }
   for i:=1 to n do   { начало внешнего цикла }
      begin
         a:=1/i; p:=a;
         for j:=2 to n do p:=p*a;   { вычисление слагаемого }
         s:=s+p;                    { накопление суммы степеней }
      end;
   writeln('Сумма степеней: ',s);
end.
```

Листинг 3.36 содержит программу, которая определяет минимальное значение функции $f(x) = x(x-1)^2(x-1)^3$ на интервале $[-0,3; 0,5]$ с точностью 10^{-7}.

Листинг 3.36. Поиск минимального значения функции

```
const eps=1e-7;   { точность поиска минимума }
var a,b,x0,xmin,y,ymin,x,h: real;
begin
   write('Введите границы интервала: '); readln(a,b);
   write('Введите значение шага: '); readln(h);
   x0:=a;   { начальное значение аргумента при первом поиске минимума }
   while h>=eps do
      begin
         ymin:=1e7; x:=x0;
         repeat
            y:=x*sqr(x-1)*sqr(x-2)*(x-2);
            if y<ymin then
```

```
            begin
               ymin:=y; xmin:=x;
            end;
         x:=x+h;
      until (y>ymin)or(x>b);
      x0:=xmin-h; h:=h/2;
   end;
writeln('xmin = ',xmin:8:5,'; ymin = ',ymin:8:5);
end.
```

Комментарии

Максимальное или минимальное значение функции называется *экстремумом*. Внешний цикл `while` проверяет условие достижения требуемой точности. Если она достигнута и условие выполнено, осуществляется выход из внешнего цикла с выводом результата. Иначе задается новое начальное значение аргумента в окрестности минимума `x0:=xmin-h`, дробится шаг изменения аргумента `h:=h/2` и внешний цикл повторяется. Во внутреннем цикле вычисляется наименьшее значение функции с текущим значением шага изменения аргумента. Выход из внутреннего цикла выполняется в случае истинности условия `y>ymin`, т. е. прохождении экстремума, или когда цикл выполнен для всех значений аргумента.

В 1912 году американский флаг "Былая слава" имел 48 звезд (по одной на штат) и 13 полос (по одной на колонию). Листинг 3.37 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 3.37. 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.
```

3.9.5. Примеры программ, использующих циклы

Листинг 3.38 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 3.38. 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.
```

Комментарии

Условие задачи можно записать в виде системы 2 уравнений с 3 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.

Листинг 3.25 содержал программу, которая представляет заданное натуральное число при помощи римских цифр, используя последовательность циклов `while`, операторов `if` и оператора `case`. Покажем, как можно решить эту задачу, используя вложенные циклы (листинг 3.39).

Листинг 3.39. Запись натурального числа римскими цифрами (вариант № 2)

```
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;
      end;
   end;
end.
```

Комментарии

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.

ГЛАВА 4

Массивы

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 *одного типа с общим* для всех элементов *именем*.

Элементы массива *пронумерованы*, и обратиться к каждому из них можно по номеру (или 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. 12)*.

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 пример массива, в котором элементы нумеруются одним индексом.

Если в массиве хранится *таблица значений (матрица)*, то такой массив называется *двумерным*, 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, а к *последовательности* элементов. Обработка массивов производится при изменении индексов элементов.

Например, в случае использования выражения следующие переменные удобно применять для просмотра в цикле элементов массива:

- `a[i]` — всех элементов;
- `a[2*i]` — элементов, стоящих на четных местах;
- `a[2*i-1]` — элементов, стоящих на нечетных местах.

4.1. Описание и использование массивов

4.1.1. Описание массива в разделе *VAR*

Самый простой 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 }
    b: array[0..50] of char;     { 51 элемент — символы }
    c: array[-3..4] of boolean;  { 8 элементов — логические значения }
```

- 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
```

Каждое число в таблице имеет целочисленный тип (`integer`), это — `ТипЭлементов`. Для *адресации* элементов таблицы требуется два индекса — *номер строки и номер столбца* (размерность массива равна двум). Индекс 1 в данном примере принимает значения от 1 до 3. Индекс 2 может меняться от 1 до 4. Нижняя граница индекса в описании массива отделяется от верхней *двумя точками*. Нижняя граница показывает наименьшее возможное значение индекса, верхняя — наибольшее. Очевидно, что *нижняя граница не может превосходить верхнюю*. Таким образом, описание двумерного массива `y` выглядит так:

```
var y: array[1..3,1..4] of integer;
```

Например, при необходимости подсчитать частоту появления в некотором тексте различных букв латинского алфавита, можно воспользоваться вектором счетчиков, индекс которого (латинские буквы) меняется от `a` до `z`, а элементы (целые числа) после подсчета указывают, сколько раз встречается в тексте данная буква.

```
var count: array ['a'..'z'] of integer;
```

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 шесть байт — всего 600 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*.

4.1.2. Ограничение по размеру

В Turbo Pascal существует ограничение по памяти, отводимой под переменные программы — 64 Кбайта, что затрудняет программирование сложных задач. Для того чтобы его обойти, используются специальные приемы. В обычной ситуации при попытке описать массив, превышающий по размерам сегмент данных, например:

```
var huge: array[1..105,1..105] of real;
```

выводится сообщение об ошибке **Error 22: Structure too large** (Ошибка 22: Структура слишком большая). Для экономии оперативной памяти особенно полезны однобайтовые типы `byte` и `shortint` (см. табл. 2.1), но, к сожалению, не во всех массивах элементы принимают значения из ограниченных диапазонов.

4.1.3. Описание границ

При объявлении массива нельзя задавать границы индексов при помощи переменных. Память под массив выделяется компилятором до выполнения программы, а переменные получают значения только в ходе ее выполнения.

При описании массива удобно использовать предварительно определенные *именованные константы (см. разд. 2.2.4)*, которые задают количество элементов. *Употребление констант при описании массива предпочтительно*, т. к. в случае изменения размеров массива не нужно будет вносить исправления по всему тексту программы, достаточно только один раз изменить значение именованной константы.

Например, опишем двумерный вещественный массив `matrix` с `maxrow` строк и `maxcol` столбцов:

```
const
   maxrow=10; maxcol=15;
var
   m,n: integer;   { реальное количество строк и столбцов массива }
   matrix: array [1..maxrow, 1..maxcol] of real;
```

4.1.4. Задание массива типизированной константой

Массив также 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));
```

В памяти компьютера будет расположена таблица чисел:

```
1   3   5
2   4   6
```

В этом примере не просто выделяется память под массив, а происходит заполнение ячеек заданными значениями по строкам.

Элементы такого массива можно *изменять* в ходе программы, как и любые другие типизированные константы.

Описание массива как типизированной константы используется на практике:

- для задания массивов с неизменными значениями элементов, например, массив из названий дней недели или количества дней в каждом месяце года;
- при отладке программы, чтобы каждый раз не заполнять массив "вручную" при запуске программы.

4.1.5. Предварительное описание типа массива

Выше были рассмотрены способы описания массивов в разделе `var` без явного указания идентификатора типа. Такие "безымянные" типы называются *анонимными*. При описании массивов также широко используется *предварительное описание типа* в разделе описания типов данных *(см. разд. 2.2.5)*. Такая возможность может потребоваться, например, при использовании имени массива в качестве параметра процедуры или функции (см. листинг 5.17).

- Описание одномерного массива:

```
type
   ИмяТипа = array[ НижняяГраница.. ВерхняяГраница ] of ТипЭлементов;
var
   ИмяМассива : ИмяТипа;
```

- Описание двумерного массива:

```
type
   ИмяТипа = array[НижняяГраницаИндекс1.. ВерхняяГраницаИндекс1,
                   НижняяГраницаИндекс2.. ВерхняяГраницаИндекс2]
             of ТипЭлементов;
var
   ИмяМассива: ИмяТипа;
```

Например, объявим массив `matrix` типа `matr` — двумерный вещественный массив с `maxrow` строк и `maxcol` столбцов:

```
const
   maxrow=10; maxcol=15;
type
   matr = array [1..maxrow, 1..maxcol] of real;
var
   matrix: matr;
```

С необходимостью применения массивов мы сталкиваемся всякий раз, когда требуется связать, запомнить и использовать в программе ряд *родственных величин*.

Например, объявим массив `supply` типа `energy` — двумерный вещественный массив с `days` строк и `hour` столбцов для указания энергопотребления по дням недели в течение каждого часа дня:

```
type
    days=(monday,tuesday,wednesday,thursday,friday,saturday,sunday);
    hour=1..24;
    energy=array[days,hour] of real;
var supply: energy;
```

Тогда в программе именем `supply[Friday,18]` будет обозначаться количество электроэнергии, потребленной в пятницу с 17.00 до 18.00 (в течение 18-го часа суток).

4.1.6. Ошибки

Примеры *неправильного* описания массивов:

- не определены размерность и границы диапазонов:

```
a: array [] of real;
```

- значение нижней границы массива превышает значение верхней:

```
b: array [10..1] of integer;
```

- границы массива необходимо задавать константой, а не выражением:

```
c : array [1..a + b] of real;
```

- недопустимо использовать вещественные числа для границ индексов:

```
d : array [1.0..100.0] of integer;
```

Необходимо четко различать индекс элемента массива и значение элемента. Например, в операторе `a[3]:=1;` число `3` — это индекс (номер) элемента массива, а `1` — значение, которое ему присваивается.

Примеры *ошибочного* обращения к элементам массивов:

- использование вещественного индекса запрещено:

```
var f: array [1..10] of real;
...
...f[5.0] ...
```

- несоответствие размерности — к вектору обращаются как к матрице:

```
var g : array [1..100] of real;
...
...g[1,4] ...
```

Следует знать:

- границы изменения индекса должны быть константами, причем очень удобно использовать *именованные константы*;
- 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;
- при выходе индекса за границы массива возникает программное прерывание, если включена директива компилятора `{$R+}`. Если эта директива отключена, *выход за границы массива может испортить соседние с ним ячейки памяти*, что приведет к непредсказуемым ошибкам в программе.

  Например, если массив `a` описан как `var a: array [1..100] of integer;`, то обращение к несуществующему элементу массива с номером 200 — `a[200]`, означает выход индекса за границу массива;
- для ввода, вывода и обработки массивов удобно применять циклы, особенно удобен в этих случаях цикл `for`, т. к. номера элементов следуют по порядку друг за другом с единичным шагом;
- массивы, идентичные по структуре, т. е. с одинаковыми типами индексов и элементов, могут участвовать в операторе присваивания без указания индекса.

  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 **Error 26: Type mismatch** (Ошибка 26: Несоответствие типов). Здесь слова "одного типа" означают тип *с тем же именем*. Тип, имеющий такую же *спецификацию* (т. е. описание), но *другое имя*, не подходит.

4.2. Действия над массивами

4.2.1. Заполнение массива данными

Массив, описанный как типизированная константа, уже содержит данные. Массивы, объявленные в разделе описания переменных, необходимо заполнить данными, прежде чем выполнять с ними какие-либо действия.

Значения элементов массива также можно задать следующими способами:

- при вводе данных с клавиатуры (см. листинг 4.1);
- с помощью датчика случайных чисел (см. листинг 4.2);
- присваиванием заданных значений;
- считывая значения элементов из файла *(см. гл. 12)*.

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:

- вектора из 5 элементов:

```
for i:=1 to 5 do readln(a[i]);
```

- матрицы размером 3×2 (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 (см. листинг 4.2).

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 играх для создания "случайной" ситуации (см. листинг 5.16).

Нередко приходится заполнять массив нулевыми значениями — *обнулять* его.

Например, обнуление элементов двумерного массива `a` выглядит следующим образом:

```
for i:=1 to n do
   for j:=1 to m do
      a[i, j]:=0;
```

4.2.2. Вывод массива

Вывод значений элементов массива также выполняется в цикле `for` с использованием операторов `write` и `writeln`.

Например, вывод вектора из 5 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],'  ');
```

- или с заданием формата, где под каждый элемент отводится 4 позиции:

```
for i:=1 to 5 do write(a[i]:4);
```

Вывод матриц в стандартной форме записи — по строкам и столбцам — выполняется при помощи оператора `writeln;` (без параметра). Он используется после вывода текущей строки матрицы для перевода курсора в начало следующей строки экрана:

```
for i:=1 to n do
   begin
      for j:=1 to m do
         write(a[i,j]:4);
      writeln;
   end;
```

Замечание

Задание формата вывода помогает расположить матрицу на экране ровными столбцами.

4.2.3. Обработка массива

Часто требуется вычислить сумму элементов массива, их среднее арифметическое значение или найти значения и номера максимального и минимального элементов, а также изменить значения элементов массива и т. д. При этом для одномерного и двумерного массивов используются *аналогичные* алгоритмы, но в случае двумерного массива требуются *вложенные* циклы.

Действия с одномерными массивами

Условимся, что в векторе `a` содержится `n` элементов.

- Вычисление *суммы* элементов:

```
s:=0;
for i:=1 to n do s:=s+a[i]; { обычное накопление суммы в s }
```

- Вычисление *произведения* элементов:

```
s:=1;
for i:=1 to n do s:=s*a[i]; { накопление произведения в s }
```

- 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 }
```

- Поиск элемента с заданным значением. Найти элемент — это значит выяснить его номер в массиве (см. листинг 4.1).

  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');
```

Комментарии

Поскольку в данном примере требуется найти только одно, самое первое, значение, можно использовать цикл `repeat`. Заметим, что таким же образом можно было бы искать элемент, начиная поиск с конца массива и уменьшая на 1 зна-

чение 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 }
```

- Поиск максимального элемента и его номера (см. листинг 4.2).

  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.

Комментарии

Если в массиве содержится 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.

- 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.

```
minzp:=3000;
for i:=1 to n do
   if a[i]<minzp then a[i]:=minzp;
```

Листинг 4.1 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 4.1. 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 массива, имеющих значение
↪',n,' =', b:3);
               writeln('Первый элемент имеет номер', a:3);
            end;
end.
```

Листинг 4.2 содержит программу, которая формирует одномерный массив случайных чисел, выполняет поиск максимального элемента массива, а затем выводит на экран его значение и порядковый номер в массиве.

Листинг 4.2. Программа поиска максимального элемента массива и его номера

```
const count=10;
var m: array [1..count] of byte;
    max,i,numer_max: byte;
begin
   { формирование массива случайной функцией random(count*2) }
   { и вывод массива на экран }
```

```
   randomize;   { инициализация датчика случайных чисел }
   for i:=1 to count do
      begin
         m[i]:= random(count*2)+1;
         write(m[i],'  ');
      end;
   writeln;
   max:= m[1];   { начинаем с первого элемента }
   numer_max:=1;
   { проверить все элементы, начиная со второго }
   for i:=2 to count do
      begin
      { если очередной элемент массива больше max }
         if m[i]>max then
            begin
               { то присвоить его значение max }
               max:=m[i];
               { и запомнить его порядковый номер }
               numer_max:=i;
            end;
      end;
   write('Максимальный элемент ', max);
   writeln(' расположен на ', numer_max,' месте');
end.
```

Действия с двумерными массивами

Условимся, что массив `a` состоит из `n` строк и `m` столбцов.

- 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;
```

Комментарии

Аналогично вычисляется сумма в столбцах, для этого внешний цикл необходимо сделать по переменной `j` (номер столбца), а внутренний — по `i` (номер в строке).

- Поиск минимального элемента всей матрицы.

  Переменная `min` используется для хранения значения минимального элемента, `k` — номер строки, `l` — номер столбца, где он находится:

```
min:=a[1,1]; k:=1; l:=1;
for i:=1 to n do
   for j:=1 to m do
      if a[i,j]<min then
         begin
            min:=a[i,j];
            k:=i; l:=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;
```

Листинг 4.3 содержит программу, которая вводит с клавиатуры квадратный массив 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. Предусмотрен вывод на экран.

Замечание

Главную диагональ массива образуют элементы, у которых номер строки равен номеру столбца: a_{11}, a_{22}, ... a_{nn}.

Листинг 4.3. Выделение элементов массива относительно главной диагонали

```
uses crt;
const row=3;     { количество строк }
      col=row;   { количество столбцов }
var  a: array[1..row,1..col]of integer;
     b,c: array[1..row*col]of integer;
     i,j,n,m: integer;
begin
   clrscr; 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:');
   for i:= 1 to n do write(b[i],'  '); writeln;
   writeln('Элементы ниже главной диагонали:');
   for i:= 1 to m do write(c[i],'  '); writeln; readln;
end.
```

Комментарии

При вводе массива по строкам необходимо разделять числа в строке при помощи пробела, ввод строки завершать нажатием клавиши <Enter>.

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.

Для выполнения перестановок обычно используется третья переменная, которая служит для временного хранения одного из элементов и играет роль *буфера обмена*.

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 матрицы необходимо переставить местами все элементы двух строк.

Например, при перестановке двух строк, скажем, первой и второй, не обязательно использовать в качестве буфера обмена целый массив. Одной переменной вполне достаточно, т. к. перестановки всех элементов строк выполняются по очереди.

```
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 4.4 содержит программу, которая формирует матрицу случайных чисел и транспонирует ее.

Листинг 4.4. Программа транспонирования матрицы случайных чисел

```
const row=3; col=row;
var a: array[1..row,1..col] of integer;
    i,j,buf: integer;
begin
   randomize;   { инициализация датчика случайных чисел }
   writeln('Исходная матрица случайных чисел:');
   for i:= 1 to row do
      begin
         for j:= 1 to col do
            begin
               a[i,j]:=random(100);{ случайное значение элемента }
               write(a[i,j]:4);    { вывод элемента массива на экран }
            end;
         writeln;
      end;
   { транспонирование матрицы }
   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
```

```
   { обмен элементов, симметричных относительно главной диагонали }
         begin
            buf:=a[i,j]; a[i,j]:=a[j,i]; a[j,i]:=buf;
         end;
   writeln('Результат транспонирования матрицы:');
   for i:= 1 to row do
      begin
         for j:= 1 to col do write(a[i,j]:4);
         writeln;
      end;
end.
```

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. С точки зрения программиста наибольший интерес представляют: сортировка массива; сортировка строк; сортировка элементов файла. Именно эти сортировки используются при разработке компиляторов, интерпретаторов, баз данных, оформлении статистических сводок, справочных материалов и большинства прикладных пакетов.

При сортировке элементы массива *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 массивах найти нужную информацию можно гораздо быстрее по сравнению с несортированными;
- 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.

Обычная постановка задачи по сортировке массива выглядит следующим образом: программа должна вывести несортированный массив целых чисел на

экран, выполнить его сортировку по невозрастанию или неубыванию, а затем вывести отсортированный массив.

Рассмотрим три метода сортировки по невозрастанию для одного и того же массива `m` из 20 целых чисел. Использование одного массива позволит сравнить эффективность разных методов. Для сравнения эффективности различных способов сортировки введем целую переменную `a`, значение которой будет равно числу *итераций* (повторов просмотра массива). Для наблюдения текущего состояния массива после каждой перестановки элементов будем выводить его на экран. Для удобства отладки и тестирования массив задается в виде типизированной константы.

Линейная сортировка (сортировка отбором)

Идея линейной сортировки по невозрастанию заключается в том, чтобы, последовательно просматривая весь массив, отыскать наибольшее число и поменять его местами с первым элементом. Затем просматриваются элементы массива, начиная со второго, снова находится наибольший, который меняется местами со вторым и т. д. (листинг 4.5).

Листинг 4.5. Программа линейной сортировки по невозрастанию

```
const count=20;
      m: array [1..count] of byte =
         (9,11,12,3,19,1,5,17,10,18,3,19,17,9,12,20,20,19,2,5);
var i,j,buf,l: byte; a: integer;
begin
   writeln('Исходный массив:');
   for i:=1 to count do write(' ', m[i]);
   writeln; readln;
   a:=0;   { обнуляем счетчик итераций }
   { изменение размера неотсортированной части массива }
   for i:=1 to count-1 do
   { сравниваем поочередно i-й элемент неотсортированной части массива }
   { со всеми от i+1-го до конца }
      for j:=i+1 to count do
         begin
            a:=a+1;
            if m[i]<m[j] then  { если в неотсортированной части массива }
                { нашли элемент, больший, чем i-й, то меняем их местами }
               begin
                  buf := m[i];   { buf — буфер обмена }
                  m[i] := m[j];
                  m[j] := buf;
               end;
```

```
            for l:=1 to count do write(' ', m[l]);   { после сортировки }
            writeln(';  итерация # ',a);
         end;
end.
```

По последнему значению a определяем, что для данного массива линейная сортировка по невозрастанию выполняется за 190 итераций.

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 (листинг 4.6).

Листинг 4.6. Программа сортировки методом "пузырька"

```
const count=20;
      m: array [1..count] of byte =
         (9,11,12,3,19,1,5,17,10,18,3,19,17,9,12,20,20,19,2,5);
var i,j,buf,l: byte; a: integer;
begin
   writeln('Исходный массив:');
   for i:=1 to count do write(' ',m[i]);
   writeln; readln;
   a:=0;
   for i:=2 to count do
      begin
         for j:=count downto i do
            begin
               a:=a+1;
               if m[j-1]<m[j] then{ если элемент справа больше элемента }
                { слева, то "вытесняем" его влево — пузырек "всплывает" }
                  begin
                     buf:=m[j-1];   { перестановка элементов }
                     m[j-1]:=m[j];
                     m[j]:=buf;
                     for l:=1 to count do write(' ',m[l]);
                     writeln(';  итерация # ',a);
                  end;
            end;
      end;
end.
```

Комментарии

Оператором повтора `for` с параметром `j` выполняется циклический просмотр элементов массива от последнего до второго, и при этом элементы, большие, чем "соседи" слева, смещаются при обмене к началу массива, а меньшие — вправо ("всплывают" в конец массива). После каждого просмотра-сравнения элементов массива просматриваемый участок массива уменьшается на один элемент, т. к. из рассмотрения исключается самый левый — самый большой элемент. Такое уменьшение просматриваемого участка выполняется внешним циклом `for` с параметром `i`. По последнему значению `a` определяем, что для данного массива сортировка по невозрастанию пузырьковым методом выполняется за 170 итераций.

Метод быстрой сортировки с разделением

Оба выше рассмотренных метода достаточно просты и наглядны, но не эффективны. 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.

Алгоритм метода является очень сложным и требует знания процедур и рекурсии *(см. гл. 5, 6)*, поэтому приведем его здесь с минимальными комментариями в тексте самой программы (листинг 4.7).

Листинг 4.7. Программа быстрой сортировки

```
uses crt;
const count=20;
      m: array [1..count] of byte =
         (9,11,12,3,19,1,5,17,10,18,3,19,17,9,12,20,20,19,2,5);
var i,a: 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 середину массива }
   repeat
      while m[i]>x do i:=i+1;
      while x>m[j] do j:=j-1;
     { увеличить число итераций обмена элементов }
      a:=a+1;
      if i<=j then
         begin                  { обменять местами элементы }
            buf:=m[i]; m[i]:=m[j]; m[j]:=buf;
```

```
            i:=i+1; j:=j-1;
            for l:=1 to count do write(' ',m[l]);
            writeln('; итерация # ',a);
         end;
   until i>j;
   { рекурсивный вызов процедуры quicks }
   if first<j then quicks(first,j);
   { рекурсивный вызов процедуры quicks }
   if i<last then quicks(i,last);
end;                  { конец сортировки }
begin                 { начало основной программы }
   clrscr;            { очистка экрана }
   writeln('Исходный массив:');
   for i:=1 to count do write (' ',m[i]);
   writeln; readln;
   a:=0;
   quicks(1,count);  { вызов процедуры сортировки quicks }
   writeln; writeln('Отсортированный массив:');
   for i:=1 to count do write (' ',m[i]);
   writeln; readln;
end.
```

По последнему значению `a` определяем, что для данного массива сортировка по невозрастанию выполняется за 28 итераций.

Быстрый поиск в упорядоченных массивах

Поиск в неупорядоченном массиве заключается в последовательном переборе элементов массива и сравнении их значений с искомым ("ключом") до того момента, пока результат сравнения не даст значение "истина" (см. листинг 4.1). Разумеется, искомого значения в массиве может и не быть. 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 — словарный порядок облегчает поиск слова 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, т. е. сузить область дальнейшего поиска. Затем делится пополам та часть массива, в которой элемент обнаружен, и так до

тех пор, пока рассматриваемая часть массива не получится состоящей из одного элемента.

Листинг 4.8 содержит программу, которая выполняет бинарный поиск заданного элемента в отсортированном массиве целых чисел.

Листинг 4.8. 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.
```

Удаление и вставка элементов в массив

Вспомним, что элементы массива располагаются в соседних ячейках памяти. Значит, без особых усилий, при необходимости, можно удалить только последний 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 4.9 содержит программу, которая удаляет из массива все элементы, имеющие заданное значение. Оставшиеся после удаления ненужные ячейки в конце массива ("хвост массива") обнуляются.

Листинг 4.9. 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,   { параметр цикла для сдвига элементов }
    i,   { счетчик }
    n: byte; { 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.
```

Комментарии

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.

Умножение матриц

Рассмотрим задачу, связанную с нахождением результата *произведения двумерных массивов*. Три продавца продают четыре вида товаров. Количество продаваемого товара сведено в таблицу *A* (рис. 4.1).

Продавец	Товар			
	№ 1	№ 2	№ 3	№ 4
№ 1	5	2	0	10
№ 2	3	5	2	5
№ 3	20	0	0	0

Рис. 4.1. Таблица *A* — продавцы и товары

В таблице *B* (рис. 4.2) представлены цена каждого товара и комиссионные, получаемые от продажи.

Товар	Цена	Комиссионные
№ 1	1,5	0,2
№ 2	2,8	0,4
№ 3	5	1
№ 4	2	0.5

Рис. 4.2. Таблица *B* — цены и комиссионные

Вырученные от продажи деньги подсчитываются так (рис. 4.3):

Продавец			
	№ 1	5×1,5 + 2×2,8 + 0×5 + 10×2 =	33,1
	№ 2	3×1,5 + 5×2,8 + 2×5 + 5×2 =	38,5
	№ 3	20×1,5 + 0×2,8 + 0×5 + 0×2 =	30

Рис. 4.3. Выручка от продажи

А полученные комиссионные рассчитываются так, как показано на рис. 4.4.

Эти вычисления называются *умножением матриц* и традиционно записываются в форме, представленной на рис. 4.5.

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*.

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

Рис. 4.4. Комиссионные

		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					

Рис. 4.5. Умножение матриц

Листинг 4.10 содержит программу, которая вводит исходные массивы "продавец-товар" и "цена-комиссионные", находит их произведение и выводит каждую из трех матриц на экран.

Листинг 4.10. Программа умножения матриц (вариант № 1)

```
uses crt;
const mmax=5; nmax=5;      { макс.размеры массивов-сомножителей }
type massiv=array[1..mmax,1..nmax] of real;
var a,b,c: massiv;         { массивы сомножителей и результата }
    m1,n1,m2,n2: integer; { их реальные размеры }
    i,j,k: integer;
begin
   clrscr;
   writeln('Введите матрицу продавцы-товары');
   repeat
      write('Введите кол-во продавцов <=',mmax,': '); readln(m1);
      write('Введите кол-во товаров <=',nmax,': '); readln(n1);
   until (m1<=mmax)and(n1<=nmax);
   for i:=1 to m1 do
      for j:=1 to n1 do
         begin
            write('Кол-во товара ',j,' у продавца ',i,' ');
            readln(a[i,j]);
         end;
   writeln('Введенная матрица продавцы-товары');
   { вывод первого сомножителя (фрагмент вывода 1) }
```

```
   for i:=1 to m1 do
      begin
         for j:=1 to n1 do write(a[i,j]:7:3); writeln
      end;
   writeln;
   writeln('Введите матрицу цена-комиссионные');
   m2:=n1;   { число строк матрицы B равно числу столбцов матрицы A }
   n2:=2;    { число столбцов матрицы B в примере всегда равно 2 }
   for i:=1 to m2 do
      begin
         write('цена и комиссионные товара ',i,' через пробел ');
         for j:=1 to n2 do read(b[i,j]); readln;
      end;
   writeln('Введенная матрица цена-комиссионные');
   { вывод второго сомножителя (фрагмент вывода 2) }
   for i:=1 to m2 do
      begin
         for j:=1 to n2 do write(b[i,j]:7:3); writeln
      end;
   writeln;
   { вычисление произведения матриц }
   for i:=1 to m1 do
      for j:=1 to n2 do
         begin
            c[i,j]:=0.0;
            for k:=1 to n1 do c[i,j]:= c[i,j]+a[i,k]*b[k,j];
         end;
   writeln('Результат: выручка и комиссионные продавцов');
   { вывод результата (фрагмент вывода 3) }
   for i:=1 to m1 do
      begin
         for j:=1 to n2 do write(c[i,j]:7:3); writeln
      end;
end.
```

Комментарии

Именованные константы `mmax` и `nmax` позволяют, при необходимости, изменить максимальную размерность используемых матриц сразу по всему тексту программы, внеся изменения только в одном ее месте — разделе `const`.

Недостатком программы является то, что в ее тексте трижды встречается идентичный фрагмент, выполняющий вывод матрицы. Очевидно, что фрагменты вывода 1—3, выводя матрицы с разными именами и разным количе-

ством строк и столбцов во всем остальном идентичны друг другу. Избавиться от недостатка позволяет механизм подпрограмм пользователя *(см. разд. 5.2)*. Листинг 5.17 содержит текст программы перемножения матриц с использованием подпрограмм ввода/вывода и умножения двумерных массивов.

4.2.4. Примеры разных программ, использующих массивы

Листинг 4.11 содержит программу, которая вводит дату в виде трех целых чисел: число, месяц и год. Введенная дата проверяется на корректность, и если все хорошо, то выводится в виде "число название месяца год". Если введена недопустимая дата, об этом выводится соответствующее сообщение.

Листинг 4.11. Программа ввода даты

```
const a: array [1..12] of integer =
         (31,28,31,30,31,30,31,31,30,31,30,31);
{ количество дней в различных месяцах года }
      b: array [1..12] of string[10] =
         ('января', 'февраля', 'марта',
          'апреля', 'мая',     'июня',
          'июля',   'августа', 'сентября',
          'октября','ноября',  'декабря');
{ для вывода даты }
var day,month,year: integer;
    ok: boolean;   { показатель корректности даты }
begin
   writeln('Введите дату (число, номер месяца, год)');
   readln(day,month,year);
   if (day<=0)or(month<=0)or(year<=0)or(month>12) then ok:=false
   { проверка числа, месяца, года на положительность и }
   { месяца — на корректность }
      else
         begin { пока все хорошо }
            if (month=2)and(year mod 4=0) then a[2]:=29;
            { если год високосный и месяц февраль }
            if day>a[month] then ok:=false { число неправильное }
                            else ok:=true;
         end;
   if not ok then writeln('Дата некорректна')
             else write(day,' ',b[month],' ',year,' года');
end.
```

Комментарии

Применение массивов-констант позволяет избежать утомительной проверки сложных условий.

Листинг 4.12 содержит программу, которая, используя перечисляемый тип данных *(см. разд. 2.2.6)* и оператор `case` *(см. разд. 3.8.3)*, вводит информацию о загрузке станка в цехе во все дни недели и выводит суммарную загрузку в течение оставшихся дней, начиная с указанного.

Листинг 4.12. Программа расчета загрузки станка в цехе

```
type days=(monday,tuesday,wednesday,thursday,friday,saturday,sunday);
var taux: array[monday..sunday] of real;
    day,j: days;
    k: integer; sum: real;
begin
   taux[sunday]:=0; day:=monday;
   repeat
      write('Введите загрузку станка ');
      case day of
         monday   : writeln('в понедельник');
         tuesday  : writeln('во вторник');
         wednesday: writeln('в среду');
         thursday : writeln('в четверг');
         friday   : writeln('в пятницу');
         saturday : writeln('в субботу')
      end;
      readln(taux[day]);
     { переход к следующему по порядку элементу перечисления }
      day:=succ(day);
   until day=sunday;
   writeln('Введите порядковый номер дня недели'); readln(k);
   day:=days(k-1); { приведение типа переменной }
   sum:=0;
   for j:=day to saturday do sum:=sum+taux[j];
   writeln('Суммарная загрузка равна ', sum:7:3); readln;
end.
```

Листинг 4.13 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, а затем вывод остатков в обратном порядке. Массив используется для хранения остатков — двоичных цифр.

Листинг 4.13. Программа перевода числа из десятичной в двоичную систему счисления

```
var a: array[1..20] of 0..1; { массив для хранения двоичных цифр }
    n, { число, вводимое пользователем }
    k, { переменная-счетчик }
    l  { количество двоичных цифр }: integer;
begin
   writeln('Введите число для перевода'); readln(n);
   l:=0;          { пока цифр в массив не записано }
   while n>0 do   { делим n на 2 и находим остаток (0 или 1) }
      begin
         l:=l+1;
         a[l]:=n mod 2;
         n:=n div 2;
      end;
   writeln('Двоичная запись данного числа:');
   { выводим элементы массива в обратном порядке }
   for k:=l downto 1 do write(a[k]);
end.
```

ГЛАВА 5

Процедуры и функции

5.1. Общие сведения

Если в программе возникает необходимость *частого* обращения к некоторой группе операторов, выполняющих действия или вычисляющих выражение, то рационально сгруппировать эти операторы в блок, к которому можно обратиться по имени.

Такие разработанные программистом *самостоятельные* программные блоки называются *подпрограммами пользователя*. Они являются основой *модульного программирования (см. разд. 6.1)*.

Использование подпрограмм позволяет, сосредоточив в них подробное описание некоторых операций, в основной программе указывать только *имена подпрограмм*, чтобы выполнить эти операции. Такие вызовы подпрограммы возможны *неоднократно* из разных участков основной программы, причем при вызове подпрограмме можно *передать* информацию (*различную* в разных вызовах), чтобы *одна и та же* подпрограмма выполняла решение подзадачи для *разных* случаев.

Многие программы, приведенные в данной 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. Писать такие программы как нечто единое целое, без разделения на самостоятельные фрагменты, просто невозможно.

Достоинства подпрограмм:

- программы, написанные с участием подпрограмм, *легче тестировать и отлаживать*, у них более четкая *логическая структура*;
- независимость подпрограмм позволяет локализовать в них все детали программной реализации того или иного алгоритма, и поэтому их изме-

нение, например при отладке, обычно не приводит к изменению основной программы;

- самостоятельный характер подпрограмм позволяет поручать их составление различным программистам. Так осуществляется разделение работы по программированию и, тем самым, ускоряется ее завершение;
- использование подпрограмм позволяет экономить память. Память для хранения переменных, использующихся в подпрограмме, выделяется только на время ее работы и высвобождается, как только ее выполнение заканчивается.

Использование подпрограммы как обособленной именованной части программы со своими собственными объектами (константами, переменными и т. п.) является в Turbo Pascal основным средством *структурирования программ*, а сам язык называют *процедурно-ориентированным*.

5.2. Стандартные и определенные пользователем подпрограммы. Процедуры пользователя

В языке Turbo Pascal подпрограммы реализованы посредством процедур и функций. Имея один и тот же смысл и аналогичную структуру, они различаются назначением и способом их использования. Все процедуры и функции подразделяются на две группы: встроенные и определенные пользователем.

Встроенные (стандартные) процедуры и функции являются *частью* языка и могут вызываться по имени *без предварительного описания*. Некоторые, наиболее часто используемые стандартные функции, типы их аргумента и результата были представлены в табл. 3.2. Их наличие существенно облегчает разработку прикладных программ. Однако в большинстве случаев некоторые *специфичные* для данной программы действия не находят прямых аналогов в библиотеках Turbo Pascal, и тогда программисту приходится разрабатывать свои *нестандартные* процедуры и функции (см. листинг 5.3).

Процедуры и функции пользователя пишутся *самим* программистом в соответствии с синтаксисом языка в *разделе описаний процедур и функций (см. разд. 2.2.7)*. Их вызов для последующего выполнения записывается обычно *в разделе операторов*. Возможен и такой вариант, когда в процедуре или функции выполняется вызов другой процедуры или функции. *Передача* данных из главной программы в подпрограмму и *возврат* результата выполнения осуществляются с помощью *параметров (см. разд. 5.4)*.

Процедура — это независимая именованная часть программы, которую после *однократного* описания можно *многократно* вызывать по *имени* из по-

следующих частей программы для выполнения определенных действий. Процедура не может выступать как операнд в выражении.

Структура процедуры повторяет структуру программы, это "программа в миниатюре" — она также представлена заголовком и телом. *В отличие от программы для процедур и функций наличие заголовка обязательно.*

Заголовок состоит из зарезервированного слова `procedure`, идентификатора *(имени)* процедуры и *необязательного*, заключенного в круглые скобки, списка *формальных параметров* с указанием *типа* каждого параметра:

```
procedure ИмяПроцедуры(ФормальныеПараметры);
```

Например, заголовок процедуры без формальных параметров (см. листинг 5.1):

```
procedure horline;
```

Процедура, использующая формальные параметры (см. листинг 5.2):

```
procedure horline(len:integer; s:char);  { len,s — формальные параметры }
```

Имя процедуры должно быть уникально, т. е. его нельзя использовать повторно в программе для именования других процедур.

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 программы, стоит `end`, однако после него ставится *точка с запятой.*

Для обращения к процедуре используется *оператор вызова процедуры*. Он состоит из *имени* процедуры и списка *фактических параметров*, отделенных друг от друга запятыми и заключенных в круглые скобки. Список параметров *отсутствует*, если процедуре *не передается* никаких значений.

```
ИмяПроцедуры(ФактическиеПараметры);
```

Например:

```
horline;          { фактические параметры не указаны, т. к. в }
                  { вызываемой процедуре нет формальных параметров }
```

или

```
horline(10,'-');  { число и символ — фактические параметры }
```

Параметры обеспечивают механизм *замены*, который позволяет выполнять процедуру с *различными начальными данными*. Между фактическими параметрами в операторе вызова процедуры и формальными параметрами в заголовке описания процедуры устанавливается *взаимнооднозначное соответствие* в результате их перебора слева направо. Количество и тип формальных параметров равны количеству и типу фактических параметров. Соответствующие друг другу параметры не обязательно должны одинаково обозначаться.

Если процедура возвращает в программу какие-то значения, соответствующие переменные должны быть описаны как *параметры* — переменные с использованием слова `var` *(см. разд. 5.4)*.

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.

Описания меток, констант, типов и т. п. *действительны только в пределах данной пользовательской процедуры*. В теле процедуры можно использовать любые *глобальные* константы и переменные *(см. разд. 5.5)*.

Для *принудительного выхода* из процедуры используется оператор завершения `exit`, который обеспечивает выход во внешний блок (обычно — основную программу).

Листинг 5.1 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 5.1. 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;
end.
```

Комментарии

Результатом работы подпрограммы является изображение линии на экране монитора. Никакие числовые данные в подпрограмму не передаются и в основную программу не возвращаются.

Отметим еще одно преимущество использования процедур, которое сразу не бросается в глаза — если в нашем случае потребуется изменить размер линии или изобразить ее другим символом, то эти исправления *делаются один раз* в процедуре, а не три раза в программе.

Усовершенствуем процедуру вывода линии, введя в нее два параметра: длину выводимой строки и символ, которым рисуется линия (листинг 5.2).

Листинг 5.2. 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.
```

Запишем процедуру, выполняющую возведение в целую неотрицательную степень любого числа (вспомним, что в языке Turbo Pascal нет подобной *стандартной* операции). С помощью данной процедуры вычислим, сколько байтов содержится в килобайте, мегабайте и гигабайте, используя известные соотношения: 1 Кбайт=2^{10} байт, 1 Мбайт =2^{20} байт, 1 Гбайт =2^{30} байт (листинг 5.3).

Листинг 5.3. Программа возведения в целую неотрицательную степень

```
procedure degree(x: real; n: byte; var res: real);
{ процедура возведения числа x в целую неотрицательную степень n, }
{ результатом является параметр-переменная res }
var i: integer;
begin
   res:=1;
   for i:=1 to n do res:=res*x;
end;
{ основная программа }
var kb,mb,gb: real;
begin
   degree(2,10,kb);
   degree(2,20,mb);
   degree(2,30,gb);
   writeln('1 Kb = ',kb:4:0,' byte');
   writeln('1 Mb = ',mb:7:0,' byte');
   writeln('1 Gb = ',gb:10:0,' byte');
end.
```

Комментарии

Данную процедуру можно было бы оформить в виде функции, т. к. она возвращает только один результат *(см. разд. 5.3)*.

Рассмотренный выше листинг 3.2 содержал программу, которая вычисляла ежемесячные выплаты `m` по займу в `s` рублей на `n` лет под процент `p`.

$$m = \frac{sr(1+r)^n}{12((1+r)^n - 1)}; \qquad r = p/100.$$

Сложнее, однако, ответить на обратный вопрос, под какой процент `p` выдана ссуда `s`, которая гасится месячными выплатами величиной `m` в течение `n` лет (листинг 5.4).

Листинг 5.4. Программа, вычисляющая, под какой процент была выдана ссуда

```
const eps: real=1e-06;{ точность вычисления процента }
var
   s,m,m1,r,percent: real;
   n,rub,kop: integer;
procedure formula(var m: real; n: integer; s,r: real);
var a: real;
```

```
begin
   a:=exp(ln(1+r)*n); { вычисление степени числа через логарифм }
   m:=s*r*a/(12*(a-1));
end;
procedure rub_kop(money: real; var r,k: integer);
begin
   r:=round(money*100) div 100; k:=round(money*100) mod 100;
end;
{ основная программа }
begin
   writeln('Введите ссуду, выплату, количество лет в одной строке: ');
   readln(s,m,n);
   r:=0.1;            { начальное значение процента }
   repeat
      formula(m1,n,s,r);
      r:=r*m/m1;
   until abs(m/m1-1) < eps;
   percent:=r*100;
   writeln;
   rub_kop(s,rub,kop); writeln('Общая сумма: ',rub,' руб. ',kop,' коп.');
   rub_kop(m,rub,kop); writeln('Мес.выплата: ',rub,' руб. ',kop,' коп.');
   writeln('Число лет: ',n:3);
   writeln;
   writeln('Процент: ',percent:5:2,'%');   { округление до сотых }
end.
```

Комментарии

Чтобы решить приведенное выше уравнение относительно `r`, можно воспользоваться *методом проб и ошибок*. Возьмем первоначальное значение `r`, выбранное из практических соображений, подставим его в формулу и вычислим `m1`. Если `m1` совпало с `m`, то выбор `r` был верным. Если `m1` оказалось меньше, значит `r` было выбрано слишком малым, поэтому необходимо увеличить `r`, умножив его на `m/m1`, и попробовать еще раз. Если же `m1` больше чем нужно, значит `r` было выбрано слишком большим, поэтому снова умножим его на `m/m1`, на сей раз чтобы уменьшить, и вновь вычислим `m`. Короче говоря, если разница между `m` и `m1` велика, то умножаем `r` на `m/m1` и повторяем вычисления. Рано или поздно значение `r` окажется достаточно близким к точному решению уравнения.

Предложенный метод хорошо работает лишь до тех пор, пока увеличение искомой величины обуславливает увеличение (или уменьшение) результата. Если же результат колеблется или имеет разрыв, как, например, банкротство, то этот метод не годится.

Для выделения рублей и копеек используется процедура `rub_kop`.

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 год, 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 5.5 содержит программу *с процедурой*, которая вычисляет и выводит члены ряда Фибоначчи.

Листинг 5.5. Программа вычисления членов ряда Фибоначчи (вариант № 1)

```
procedure fibon(n: integer);
{ процедура вычисления и печати чисел Фибоначчи }
var fn,fn1,fn2,k: integer;
begin
   fn1:=1;
   fn:=0;
   for k:=1 to n do
      begin
         fn2:=fn1;
         fn1:=fn;
         fn:= fn1+fn2;
         writeln(fn);
      end;
end;
{ основная программа }
var n: integer;
begin
   write('Введите число членов ряда Фибоначчи: '); readln(n);
   fibon(n);   { вызов процедуры }
end.
```

Комментарии

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: $F_{-1} = 1$, $F_0 = 0$. Процедура, прежде всего, вычисляет фиктивные члены ряда. Все подлинные числа Фибоначчи вычисляются по стандартным правилам.

В связи с тем, что ряд Фибоначчи является быстрорастущим, его 24 член уже превышает верхнюю границу диапазона типа `integer` (см. табл. 2.1).

5.3. Функции пользователя

Если результатом подпрограммы является только одно значение, то имеет смысл оформить такую подпрограмму не в виде процедуры, а в виде функции. *Функция пользователя* аналогична процедуре, но имеются два отличия.

- Функция передает в программу результат своей работы — единственное значение, носителем которого является имя самой функции.
- Имя функции может входить в выражение как операнд. Функция возвращает результат в точку своего вызова.

Функция, определенная пользователем, состоит из заголовка и тела функции. *Заголовок* содержит зарезервированное слово `function`, *имя* функции, заключенный в круглые скобки *необязательный* список *формальных* параметров и, обратите внимание — в отличие от процедуры, *тип* возвращаемого функцией значения:

```
function ИмяФункции(ФормальныеПараметры) : ТипРезультата;
```

Например:

```
function fibo(n: integer): integer;    { n — формальный параметр }
function instep(a,b: real): real;      { a,b — формальные параметры }
function normrandom : double;          { формальных параметров нет }
```

Имя функции уникально в пределах программы.

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, возвращаемое как *результат работы* функции. Если таких присваиваний *несколько*, то результатом выполнения функции будет значение *последнего* оператора присваивания. Если же такой оператор отсутствует или не был выполнен, то значение, возвращаемое функцией, *не определено*.

В отличие от процедуры, вызов функции не оформляется в виде отдельного оператора. Обращение к функции осуществляется путем использования указателя функции в качестве операнда в некотором выражении. *Указатель функции* представляет собой имя функции с *необязательным* списком *аргументов* — фактических параметров. Требования к ним такие же, как и в случае процедуры.

Например:

```
writeln(fibo(i));          { параметр — значение переменной }
c:=instep(2,8);            { параметры — непосредственно значения }
write(normrandom:10:3); { фактические параметры не указаны, т. к. в }
                           { вызываемой функции нет формальных параметров }
```

Программа, представленная в листинге 5.5, использовала процедуру для вычисления членов ряда Фибоначчи. Листинг 5.6 решает эту задачу при помощи *функции*.

Листинг 5.6. Программа вычисления членов ряда Фибоначчи (вариант № 2)

```
function fibo(n: integer): integer;
{ функция вычисления чисел Фибоначчи }
var fn,fn1,fn2,k: integer;
begin
   fn1:=1;
   fn:=0;
   for k:=1 to n do
      begin
         fn2:=fn1;
         fn1:=fn;
         fn:= fn1+fn2;
      end;
   fibo:= fn; { имени функции присваивается возвращаемое значение }
end;
{ основная программа }
var i, n: integer;
begin
   write('Введите число членов ряда Фибоначчи: '); readln(n);
   for i:= 1 to n do writeln(fibo(i));   { вызов функции }
end.
```

Комментарии

Представленная программа менее эффективна, чем предыдущая (см. листинг 5.5). В соответствии с программой (листинг 5.6) числа Фибоначчи вычисляются, печатаются и "забываются". При поиске нового, большего числа прихо-

дится повторять те же самые действия. Наибольший интерес для программиста может представлять рекурсивная процедура вычисления ряда Фибоначчи, рассмотренная в листинге 6.3.

Turbo Pascal содержит стандартную функцию `random` возвращающую случайное число, *равномерно распределенное* в заданном диапазоне значений *(см. разд. 3.3.1)*. Однако давайте рассмотрим задачу, в которой использование `random` приводит к неудовлетворительным результатам. Пусть требуется сформировать ряд случайных чисел, соответствующих росту взрослых мужчин. Предположим, что их значения находятся 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 5.7 содержит программу, которая использует простейший алгоритм генерации случайных чисел со стандартным нормальным распределением.

Листинг 5.7. Генератор случайных чисел с нормальным распределением

```
uses crt;
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;
   normrandom:=s-6.0;
end;
{ основная программа }
begin
   clrscr;
   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.
```

Как уже указывалось, Turbo Pascal не имеет стандартной функции вычисления степени числа *(см. разд. 3.3.1)*. Для решения этой задачи используется умножение в цикле или свойства логарифмов. Листинг 5.3 содержал процедуру, выполняющую возведение в *целую* неотрицательную степень. Листинг 5.8 представляет функцию вычисления *действительной* степени числа с использованием логарифмов.

Листинг 5.8. Программа возведения числа в действительную степень

```
function instep(a,b: real): real;
{ функция возведения положительного основания в степень }
begin
   if a>0 then instep:= exp(b*ln(a)) else instep:=0;
end;
{ основная программа }
var a,b,c: real;
begin
   writeln('Введите положительное основание и показатель степени:');
   readln(a,b);
   c:=instep(a,b);
   if c <> 0 then
      writeln(a:6:3,' в степени ',b:6:3,' = ',c:6:3)
   else
      writeln('Основание должно быть положительным...');
end.
```

Рассмотренная ранее программа (см. листинг 3.30) позволяла выполнить вычисление приближенного значения определенного интеграла по квадратурным формулам прямоугольников, трапеций и Симпсона для $y(x)=3x^2$. Применение функции пользователя позволяет усовершенствовать программу (листинг 5.9).

Листинг 5.9. Вычисление определенного интеграла (вариант № 2)

```
var
   a,b: real;          { границы отрезка }
   h: real;            { величина интервала }
   s1,s2,s3: real;     { приближенное значение интеграла }
   n: integer;         { количество интервалов }
   x,x1,x2: real;      { аргумент }
   i: integer;
{ подинтегральная функция }
function f(x: real): real;
begin
   f:= 3*x*x;          { при изменении формулы функции под интегралом }
end;                   { изменения вносятся только здесь }
{ основная программа }
begin
   write('Нижняя граница отрезка: '); readln(a);
   write('Верхняя граница отрезка: '); readln(b);
   write('Число разбиений отрезка: '); readln(n);
```

```
   h:=(b-a)/n;
   x1:=a+h/2; x2:=a;
   s1:=f(x1);   { здесь и далее f(x) — это вызов функции пользователя }
   s2:=(f(x2)+f(b))/2;
   for i:=1 to n-1 do
      begin
         x1:=x1+h; x2:=x2+h;
         s1:=s1+f(x1); s2:=s2+f(x2);
      end;
   s1:=h*s1; s2:=h*s2; s3:=(2*s1+s2)/3;
   writeln('Значение интеграла:');
   writeln(' s1 = ',s1:8:5,'; s2 = ',s2:8:5, '; s3 = ',s3:8:5);
end.
```

Комментарии

Достоинством рассмотренной программы, содержащей функцию пользователя `f`, является то, что по сравнению с листингом 3.30 в тексте программы используются компактные вызовы функции в виде `f(x)`. При необходимости смены аналитического выражения (формулы) функции, стоящей под знаком интеграла, изменения необходимо внести лишь в одном месте — в теле функции пользователя.

Дальнейшее совершенствование программы связано с использованием процедурного типа данных (см. листинг 6.7).

Рассмотренный выше листинг 3.10 содержал программу решения квадратного уравнения. Выполним ту же задачу с использованием функции (листинг 5.10).

Листинг 5.10. Решение квадратного уравнения (вариант № 2)

```
function kvadrur(a,b,c: real; var x1,x2: real): integer;
{ функция решение квадратного уравнения }
{ a,b,c — коэффициенты уравнения }
{ x1,x2 — корни уравнения }
{ значение функции — количество корней или }
{ -1, если исходные данные неверны }
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;
{ основная программа }
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('Ошибка исходных данных');
      0: writeln('Уравнение имеет комплексные корни');
      1: writeln('x=',x1:7:3,', корни одинаковы');
      2: writeln('x1=',x1:7:3,'; x2=',x2:7:3);
   end;
end.
```

Комментарии

Параметрами функции являются коэффициенты и корни уравнения. Значение функции используется для передачи в вызывающую ее программу информации о наличии корней уравнения: 2 — два разных корня, 1 — корни одинаковые, 0 — уравнение не имеет решения. Кроме того, функция проверяет корректность исходных данных. Если исходные данные неверные, то функция возвращает –1.

Особенностью использования функции `kvadrur` является то, что результаты расчета — значения корней `x1,x2` возвращаются в программу через параметры-переменные: `var x1,x2: real;` *(см. разд. 5.4)*.

Обратите внимание — приведенный пример позволяет сделать важный вывод, расширяющий наше представление о функциях. Функция может возвращать *более одного результата*, если в ней используются *параметры-переменные*.

Листинг 5.11 содержит программу, вычисляющую значение корня x уравнения $f(x) = 4 - e^x - 2x^2 = 0$ методом половинного деления (дихотомии).

Листинг 5.11. Вычисление корня уравнения методом половинного деления

```
function f(x: real): real;
begin
   f:=4-exp(x)-2*x*x;   { анализируемое уравнение }
end;
```

```
{ основная программа }
var a,b,fa,fc,c,eps: real;
begin
   write('Границы интервала изоляции корня — a и b: ');readln(a,b);
   write('Погрешность вычисления корня: '); read(eps);
   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;
   writeln('f = ',f(a):10:7,' при x =', a:10:7);
end.
```

Замечание

Хорошо известны формулы решения линейных и квадратных уравнений вида $f(x)=0$, где $f(x)$ представляет собой функцию $kx+b$ или ax^2+bx+c (см. листинг 5.10). Однако для решения многих уравнений в области математики, физики и техники, представляющих практический интерес, не существует готовых формул нахождения их корней. В этом случае используются *численные методы* поиска корней. Одним из них является метод половинного деления. Идея метода состоит в том, что если $f(x)$ непрерывна на отрезке $[a, b]$ и $f(a)\,f(b)<0$, то $f(x)=0$ на $[a, b]$. Для определения значения корня можно сузить диапазон поиска, поделив исходный отрезок пополам. Из двух образовавшихся отрезков $[a, c]$ и $[c, b]$ выбирают тот, для которого функция на концах имеет разные знаки и т. д. Процесс продолжается до тех пор, пока длина отрезка больше погрешности: $b-a>eps$. Любая точка последнего отрезка, для которого $b-a<eps$, может считаться решением (например, один из его концов). Метод является медленным, но обладает гарантированной сходимостью и прост в реализации. В случае нескольких корней будет найден один из них. Предполагается, что с учетом погрешности результаты вычисления позволяют правильно определить, верно ли, что $f(a)\,f(c)<=0$.

5.4. Механизм передачи параметров

Как было показано выше, при создании программ с использованием процедур и функций, в заголовке процедуры или функции может быть задан список параметров, которые называются формальными. Название "формальные" эти параметры получили в связи с тем, что в этом списке заданы *только имена* для обозначения исходных данных и результатов работы процедуры, а при вызове подпрограммы на их место будут *подставлены конкретные значения* выражений и имен.

Список формальных параметров может включать в себя:

- параметры-значения, за которыми указывается их тип;
- параметры-переменные, перед которыми должно стоять служебное слово `var` и за которыми указывается их тип;
- параметры-процедуры, перед которыми должно стоять служебное слово `procedure` *(см. разд. 6.2.4)*;
- параметры-функции, перед которыми должно стоять служебное слово `function` и после которых указывается тип значения, возвращаемого функцией в основную программу *(см. разд. 6.2.4)*;
- нетипизированные параметры, перед которыми должно стоять служебное слово `var` и после которых отсутствует указание типа. При этом в процедуру или функцию передается только адрес объекта, на этом месте может стоять переменная любого типа *(см. разд. 6.2.5)*.

Структура списка определяется следующими требованиями — в нем должны быть перечислены имена формальных параметров и их типы. Имя параметра отделяется от типа двоеточием, а параметры друг от друга — точкой с запятой. Имена параметров одного типа можно объединять в подсписки, в которых имена отделяются друг от друга запятой.

Например:

```
function kvadrur(a,b,c: real; var x1,x2: real): integer;
```

Соответствие между формальными и фактическими параметрами:

- формальных и фактических параметров должно быть *одинаковое количество*;
- *порядок следования* фактических и формальных параметров должен быть *один и тот же*;
- *тип* фактического параметра *должен совпадать* с типом соответствующего ему формального параметра.

5.4.1. Параметры-значения

Параметры-значения используются только для передачи исходных данных из основной программы в процедуру или функцию.

Например:

```
procedure horline(len:integer; s:char); { len,s — параметры-значения }
```

Если формальный параметр объявлен как параметр-значение, то фактическим параметром может быть произвольное выражение. *При вызове* подпрограммы фактические параметры вычисляются и используются как началь-

ные значения формальных параметров — выполняется *подстановка значений*. При этом подпрограмме будет передана лишь *копия параметра-значения*. В процессе выполнения подпрограммы формальные параметры могут изменяться, но это никак не отразится на соответствующих фактических параметрах-переменных, которые сохранят те значения, которые имели до вызова подпрограммы, т. к. *меняются не они, а их копия*. Поэтому параметры-значения *нельзя использовать для передачи результатов* из подпрограммы в основную программу.

Существует отличие между объявлением переменных в списке формальных параметров подпрограммы и их объявлением в разделе описания переменных. Типом любого параметра в списке формальных параметров может быть только стандартный или ранее объявленный тип.

Например, нельзя объявить процедуру с заголовком:

```
procedure input_array(m,n: integer; var mas: array[1..5,1..5] of real);
```

Для передачи в подпрограмму массива необходимо предварительно описать его тип (см. листинги 5.14, 5.18).

```
const mmax=5;nmax=5   { максимальное количество строк и столбцов};
type massiv = array[1..mmax,1..nmax] of real;
procedure input_array(m,n: integer; var mas: massiv);
...
```

Листинг 5.12 содержит программу, иллюстрирующую использование параметров-значений.

Листинг 5.12. Иллюстрация использования параметров-значений

```
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 }
{ начало основной программы }
begin
   a:=1.0; b:=3.0;
   { вызов процедуры square с передачей ей }
   { значений фактических параметров a и b }
   square(a,b);
   writeln(a:7:3,'   ',b:7:3);
end.
```

При выполнении программа выведет на экран:

```
1.000   9.000
1.000   3.000
```

Комментарии

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`.

5.4.2. Параметры-переменные

Параметры-переменные необходимо использовать, прежде всего, для *возврата результатов работы* подпрограммы в основную программу.

В списке формальных параметров они перечисляются после зарезервированного слова `var` с указанием типа.

Например:

```
procedure degree(x: real; n: byte; var res: real);
{ x,n — параметры-значения, res — параметр-переменная }
```

Каждому формальному параметру, объявленному как параметр-переменная, должен отвечать фактический параметр в виде *переменной соответствующего типа*.

Если формальный параметр определен как параметр-переменная, то при вызове процедуры ей *передается сама переменная* (точнее, адрес переменной в памяти), а не ее копия, и изменение параметра-переменной приводит к *изменению* фактического параметра в вызывающей программе.

Следовательно, *исходные данные* в вызываемую подпрограмму из вызывающей ее программы могут передаваться как через параметры-значения, так и через параметры-переменные, а *результаты работы* подпрограммы возвращаются в программу *только* через параметры-переменные.

Листинг 5.13 содержит программу, иллюстрирующую использование параметров-переменных.

Листинг 5.13. Иллюстрация использования параметров-переменных

```
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
```

В программах, написанных на Turbo Pascal, для генерации случайных чисел широко используется функция `random` *(см. разд. 3.3.1)*. Однако любой программист может самостоятельно создать свою собственную функцию, формирующую случайную числовую последовательность (листинг 5.14).

Листинг 5.14. Формирование случайной числовой последовательности (вариант № 1)

```
uses crt;
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
   clrscr;
   for i:=1 to 64 do write(next(s):4);
end.
```

Комментарии

Функция `next`, будучи вызванной с параметром `s`, например равным 16, возвращает число 16. Но, одновременно, она изменяет значение, хранящееся в переменной `seed`, на 19. Если функцию вызвать снова, с полученным значением `s`, то она возвратит число 19 и изменит значение `s` на 2. Продолжая вызывать функцию `next`, мы получим определенную последовательность целых чисел, начинающихся 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 начнется новый цикл. Другими словами, начиная с любого желаемого целого, функция генерирует фиксированную перестановку чисел от 0 до 63.

Такой прием обычно и используется для генерации "случайных" чисел. Часто их называют *псевдослучайными* — для того чтобы подчеркнуть их предсказуемость.

В задачах моделирования и играх зачастую удобнее использовать случайные дроби в интервале от 0 до 1, нежели случайные целые. Для получения дробей функция должна быть незначительно изменена (листинг 5.15).

Листинг 5.15. Формирование случайной числовой последовательности (вариант № 2)

```
uses crt;
var i,s: integer;
function next(var seed: integer): real;
{ изменение: тип возвращаемого значения функции — real}
   const multiplier=37; increment=3; cycle=64;
begin
   { изменение: введена операция деления }
   next:=seed/cycle;
   seed:=(multiplier*seed+increment) mod cycle;
end;
begin
   clrscr;
   for i:=1 to 64 do write(next(s):10:3);
end.
```

Листинг 5.16 содержит программу, которая моделирует процесс бросания пары игральных костей.

Листинг 5.16. Моделирование бросания пары игральных костей

```
uses crt;
var score,throws,seed,a,b,c,d,e,f,g,h,i,j,k: integer;
function rnd(var seed: integer): real;
var a,b,c,d: integer;
begin
   rnd:=seed/32767;
   a:=seed div 256;
   b:=seed mod 256;
   c:=((b*93) mod 256)+13;
   d:=(b*26)+((b*93) div 256)+(a*93)+(c div 256)+27;
   seed:=((d mod 128)*256)+(c mod 256);
end;
begin
   clrscr;
   seed:=0;a:=0;b:=0;c:=0;d:=0;e:=0;f:=0;g:=0;h:=0;i:=0;j:=0;k:=0;
   for throws:=1 to 3600 do   { число бросков }
```

```
   begin
      { сумма двух костей при очередном броске }
      score:=trunc(1+6*rnd(seed))+trunc(1+6*rnd(seed));
      { подсчет итогов бросков }
      case score of
         2:  a:=a+1;
         3:  b:=b+1;
         4:  c:=c+1;
         5:  d:=d+1;
         6:  e:=e+1;
         7:  f:=f+1;
         8:  g:=g+1;
         9:  h:=h+1;
         10: i:=i+1;
         11: j:=j+1;
         12: k:=k+1;
      end;
   end;
   { суммы двух костей }
   writeln(2:4,3:4,4:4,5:4,6:4,7:4,8:4,9:4,10:4,11:4,12:4);
   { накопленные итоги бросков }
   writeln(a:4,b:4,c:4,d:4,e:4,f:4,g:4,h:4,i:4,j:4,k:4);
end.
```

Комментарии

Напомним, что каждая игральная кость представляет собой куб с шестью помеченными гранями. Таким образом, минимальная сумма одного броска обеих костей — 2, максимальная — 12.

В качестве датчика случайных чисел в программе используется функция `rnd`, которая генерирует циклы из 32 768 дробей, а не из 64, как в листинге 5.15.

В результате 3600 бросков двух костей будет получен результат, показывающий, насколько более выгодно ставить в этом случае на 7, чем на любую другую сумму очков:

```
   2   3   4   5   6   7   8   9  10  11  12
  99 215 327 404 479 583 489 386 308 207 103
{ сравните с "идеальным ответом" }
 100 200 300 400 500 600 500 400 300 200 100
```

Рассмотренный ранее листинг 4.10 содержал программу, которая вводит исходные массивы: "продавец-товар" и "цена-комиссионные", находит их произведение и выводит каждую из трех матриц на экран. Основным недостатком программы было то, что в ней трижды встречался идентичный фрагмент, выполняющий вывод матрицы. Листинг 5.17 содержит программу

перемножения матриц с использованием подпрограмм ввода/вывода и умножения двумерных массивов. Применение подпрограмм позволяет выполнить ввод двух матриц, их умножение и вывод трех двумерных массивов, используя по одной процедуре на каждую операцию.

Листинг 5.17. Программа умножения матриц (вариант № 2)

```
uses crt;
const mmax=5;nmax=5   { максимальное количество строк и столбцов};
type massiv = array[1..mmax,1..nmax] of real;
var
   m1,n1,m2,n2: integer;
   a,b,c: massiv;
(* раздел описания процедур *)
procedure input_array(m,n: integer; var mas: massiv);
{ input_array — ввод двумерного массива mas }
{ m,n — кол-во строк и 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: massiv; var rez: massiv; m1,n1,n2:
integer);
{ mas_mult — умножение двумерных массивов }
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:massiv);
{ 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}
(* главная программа *)
begin
   clrscr;
   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);
   { вызов процедуры ввода матрицы }
   input_array(m1,n1,a);
   writeln('Введенная матрица 1 сомножителя');
   { вызов процедуры вывода матрицы }
   out_array(m1,n1,a);
   writeln;
   writeln('Введите матрицу 2 сомножителя');
   repeat
      writeln('Введите кол-во столбцов <=',nmax:2);
      readln(n2);
   until (n2<=nmax);
   { условие существования произведения 2-х матриц A и B: }
   { число столбцов n1 массива A должно быть равно }
   { числу строк m2 массива B }
   m2:=n1;
   { вызов процедуры ввода матрицы }
   input_array(m2,n2,b);
   writeln('введенная матрица 2 сомножителя');
   { вызов процедуры вывода матрицы }
   out_array(m2,n2,b);
   writeln;
   { вызов процедуры умножения матриц }
   mas_mult(a,b, c, m1, n1, n2);
   { вызов процедуры вывода матрицы }
   writeln('Матрица – результат произведения');
   out_array(m1,n2,c);
   readln;
end.
```

Комментарии

В процедуре `mas_mult` умножения двумерных массивов введены обозначения:

- `som1` — первый сомножитель, произведение матриц не обладает свойством перестановочности;
- `m1` — количество строк, а `n1` — количество столбцов первой матрицы;
- `som2` — второй сомножитель, его число строк `m2` должно быть равно числу столбцов первого сомножителя `n1` — это необходимое условие существования произведения двух матриц;
- `n2` — количество столбцов второй матрицы;
- `rez` — результат произведения матриц (размерность: `m1` строк на `n2` столбцов).

В разделе описаний каждой из трех подпрограмм содержится один и тот же оператор: `var i,j: integer;`. Он определяет локальные вспомогательные переменные вложенного цикла `i,j` как целочисленные, а для процедуры умножения двумерных массивов `mas_mult` еще и переменную `k`. Указанные переменные можно описать в одном месте программы — разделе `var`. Это позволит вместо трех идентичных операторов использовать всего один. В этом случае переменные будут называться глобальными. Более подробно о локальных и глобальных объектах рассказано в *разд. 5.5*.

5.5. Область действия параметров

При создании программ, использующих процедуры, следует учитывать, что все объекты (метки, константы, типы, переменные, процедуры и функции), которые описываются после заголовка процедуры, называются *локальными объектами*, доступны *только* в пределах этой процедуры и недоступны вызывающей программе. Память под них выделяется при обращении к процедуре в специальной части памяти, которая именуется *программным стеком*. По окончании работы процедуры эта память освобождается и может быть занята под локальные переменные другой процедуры, поэтому все локальные объекты *создаются при входе в процедуру и уничтожаются при выходе из нее*. Если *одно и то же имя* определено в *нескольких* процедурах, вызываемых одной программой, то в каждой процедуре этому имени соответствует *свой* локальный объект.

Все объекты, описанные в вызывающей программе, называются *глобальными*. Память под глобальные переменные выделяется при компиляции программы. Эта часть памяти именуется *сегментом данных*. Глобальные переменные находятся в сегменте данных от начала до конца выполнения программы, поэтому ими можно пользоваться и в программе, и во всех процедурах, к которым обращается программа.

Следует знать:

- *обмен данными* между программой и вызываемой ею процедурой может производиться и *через глобальные переменные*;
- если же *одно и то же имя* определено и в программе, и в вызываемой ею процедуре, то в программе ему соответствует глобальный объект, но внутри процедуры глобальный объект недоступен, он как бы *экранируется (маскируется)* локальным объектом с таким же именем, над которым и выполняются необходимые действия.

Например, 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
```

Комментарии

`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. Это значение сохранится и после выхода из процедуры. `c` — локальная для данной процедуры переменная, которая маскирует глобальную переменную `c` основной программы. `d` — глобальная переменная, имеющая одинаковый смысл как в основной программе, так и в процедуре.

Может сложиться впечатление, что значительно проще иметь дело только с одними глобальными переменными. Однако использование локальных переменных позволяет оптимизировать программу, делает ее более наглядной и уменьшает вероятность появления ошибок. Процедура, не исполь-

зующая глобальные переменные, является *автономной*. С ней можно производить все необходимые действия независимо от главной программы, в том числе ее гораздо легче использовать в другой программе. Поскольку в ней нет связи с остальной программой через глобальные переменные, ее легче тестировать и отлаживать.

В Turbo Pascal допускается *любой уровень вложенности процедур и функций*. Процедура, описанная в основной программе, в свою очередь, может иметь описания внутренних процедур и функций и т. д. При этом объекты, описанные в вызывающей процедуре, являются *глобальными по отношению к вызываемой процедуре (см. разд. 6.2.3)*.

Схематическое изображение структуры блоков некоторой программы на Turbo Pascal приведено на рис. 5.1.

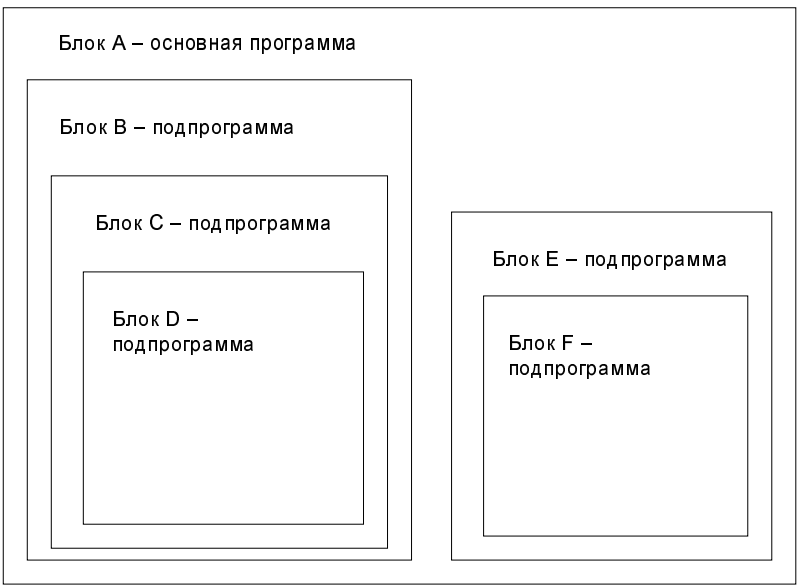

Рис. 5.1. Иллюстрация к области действия параметров

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 объемлющем блоке, то это последнее имя стано-

вится недоступным в данном блоке (оно как бы экранируется одноименным объектом данного блока).

Если применить эти правила к схеме на рис. 5.1, 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*, известны только в пределах этого блока.

Иногда при вызове подпрограмм возникают "побочные эффекты". Они выражаются в том, что при выполнении подпрограммы, наряду с вычислением ее значения, одновременно могут вноситься нежелательные изменения в значения глобальных переменных или значений фактических параметров, соответствующим формальным параметрам-переменным. Поэтому необходимо быть особенно внимательным при описании параметров-переменных, а при выборе имен контролировать наличие глобальных объектов с такими же именами.

Например, 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 }
   a:=1; b:=d(a)+d(1);
   writeln(b);   { результат 6 }
end.
```

Комментарии

Глобальная переменная `a` изменяется в теле функции `d`. Поэтому оказывается, что `d(1)+d(a)` ≠ `d(a)+d(1)`.

Использование "побочного эффекта" нежелательно, поскольку он сильно затрудняет (особенно, если не предусмотрен заранее) отладку и верификацию программы.

Локализация переменных дает программисту большую свободу в выборе идентификаторов. Так, если две процедуры полностью отделены друг от друга (не вложены одна в другую), то идентификаторы в них могут быть выбраны совершенно произвольно, в частности могут повторяться. В этом случае совпадающим идентификаторам соответствуют разные области памяти, совершенно не связанные друг с другом.

Листинг 5.18 содержит программу, определяющую количество сверхпростых чисел в натуральном ряду чисел, не превышающих 1000.

Замечание

Сверхпростым называется число, если оно простое, и число, полученное из исходного числа при записи цифр исходного числа в обратном порядке ("перевертыш"), тоже будет простым. Например, 13 и 31 — сверхпростые числа.

Листинг 5.18. Определение количества сверхпростых чисел

```
uses crt;
var x,k: integer;   { описание глобальных переменных x,k }
{ функция проверки числа x на простое число }
function prost(y: integer): boolean;
var d,i: integer;   { описание локальных переменных d,i,flag }
   flag: boolean;
begin
   d:=0; flag:=false;
   for i:=2 to y-1 do if y mod i=0 then d:=d+1;
   if d=0 then flag:=true;
   prost:=flag;
   { нашли простое число, имеющее только два делителя: }
   { единицу и само это число }
end;
{ "перевертыш" простого числа }
function povorot(y: integer): integer;
var s:integer;   { описание локальной переменной s }
begin
   if y<10 then s:=y;
   if (y>10) and (y<100) then s:=y mod 10*10+y div 10;
   if (y>=100) and (y<1000) then
      s:=y mod 10*100+y mod 100 div 10*10+y div 100;
   povorot:=s;
end;
procedure pause;   { не требует формальных параметров }
var ch: char;
begin
   writeln; writeln('Для продолжения — любая клавиша');
   repeat ch:=readkey until ch<>'';
end;
{ основная программа }
begin
   writeln('Сверхпростые числа':50);
   k:=2;   { 2 и 3 — простые числа }
   for x:=4 to 999 do
      begin
         if prost(x) then { вызов функции определения простого числа x }
```

```
{ вызов функции определения простого числа для "числа-перевертыша", }
{ полученного вычислением функции povorot(x) от простого числа }
            if prost(povorot(x)) then
               begin
                  writeln(x);   { печать сверхпростого числа }
                  k:=k+1;
                  if k mod 24=23 then pause;
               end;
      end;
   writeln('всего найдено ',k,' сверхпростых чисел < 1000');
end.
```

Комментарии

Проанализируйте текст программы, обратив особое внимание на применение функций определения простого числа, функцию получения "перевертыша" и вложенный вызов функции `prost(povorot(x))`, при котором переменные `d`, `i`, `flag`, локальные для функции `prost`, являются глобальными для функции `povorot`.

Для создания паузы, длящейся до нажатия пользователем клавиши при просмотре результатов выполнения программы на экране, использована процедура `pause`.

5.6. Основные выводы

Сведем в общий краткий перечень наиболее важные из рассмотренных выше правил работы с подпрограммами пользователя. Используя их в собственных программах, программист должен помнить, что:

- тело подпрограммы по структуре аналогично программе, но после слова `end` ставится *точка с запятой*;
- вызов подпрограммы-процедуры выполняется *отдельным оператором* главной программы. Обращение к функции выполняется по имени, которое входит в выражение как *операнд*;
- *исходные данные* в подпрограмму из вызывающей программы могут передаваться как через параметры-значения, так и через параметры-переменные. *Результаты работы* подпрограммы возвращаются в вызывающую программу только через параметры-переменные;
- *глобальные переменные*, т. е. переменные, объявленные в основной программе, доступны в любой процедуре, но применять их для обмена данными следует очень осторожно;
- *список параметров отсутствует*, если отсутствует обмен данными между подпрограммой и вызывающей программой. Функция всегда возвращает, по крайней мере, *одно значение*, носителем которого является ее имя;

- среди выполняемых инструкций функции обязательно должна быть инструкция *присваивания результата вычислений имени функции*;
- фактические параметры при вызове подпрограммы должны соответствовать формальным параметрам, указанным при ее объявлении *по количеству, порядку следования слева направо и типу*;
- если параметры подпрограммы используются *для возврата результата работы в главную программу*, то в объявлении подпрограммы перед именем соответствующего формального параметра должно присутствовать слово `var`;
- если при объявлении подпрограммы перед именем формального параметра не стоит слово `var`, то при ее вызове можно использовать в качестве фактического параметра константу или переменную (выражение) соответствующего типа. Если слово `var` указано, то параметром может быть только *переменная*.

ГЛАВА 6

Дополнительные сведения о процедурах и функциях

6.1. Структуризация в программировании

Проектирование программы начинается с установления цели *проекта*. Выбор цели оказывает большое влияние на процесс проектирования и его результат.

При разработке программного обеспечения преследуются *основные цели*:

- высокий уровень надежности — уменьшение числа *невыявленных ошибок* в программе и количества *неучтенных ситуаций*, которые могут привести к неопределенному результату или прекращению нормального функционирования программы;
- минимальное время разработки — увеличение сложности задач, решаемых с помощью компьютеров, приводит к увеличению размеров и сложности программ, что порождает дополнительные трудности при их разработке и отладке;
- удобство и простота эксплуатации — программа должна быть легка в освоении для рядового пользователя, давать возможность предупреждать и исправлять его ошибки;
- необходимый объем памяти и скорость выполнения — требуемые результаты работы программы необходимо получить в короткие сроки, учитывая реально существующие аппаратные ограничения;
- легкость модификации — с течением времени из-за изменения условий использования программ возникает необходимость внесения изменений, повышения эффективности и удобства их использования;
- универсальность.

Эти цели очень часто бывают противоречивы. В тех случаях, когда требуется одновременно достичь нескольких, их упорядочивают по степени важности.

В практике программирования широко используются *методы структурного программирования*. Идея состоит в том, что структура программы должна отражать структуру решаемой задачи, а алгоритм решения должен быть ясно виден из текста программы. Методы структурного программирования предполагают:

- проектирование программ на основе *метода пошаговой детализации* (нисходящая или восходящая разработка программ);
- модульное программирование;
- структурное кодирование.

Ныне процесс программирования превращается в *промышленное изготовление программ* на основе *технологий программирования*. Большинство специалистов придерживаются точки зрения, что *метод нисходящего проектирования программ* наиболее удобен для решения сложных проблем. Основой такого метода является идея *программирования* "сверху вниз" с постепенной разбивкой исходной задачи на ряд подзадач. Выполняется *последовательное уточнение*: сначала задача определяется в общих чертах. 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 модулей — *подпрограмм*. Используя аналогию со строительством, можно отметить, что нисходящий подход соответствует разработке проекта всего здания с последующей детализацией его отдельных частей. Восходящий — проектированию здания с учетом деталей, имеющихся в каталоге (например, если в распоряжении программиста имеется широкий набор процедур и функций).

При проектировании возможны логические ошибки. Проанализировав проект, их проще устранить на стадии его создания, чем заново переделывать всю программу. Иногда части программы замещают временными "заглушками" — синтаксически правильными подпрограммами, в которых не выполняются нужные действия.

Структурное кодирование должно обеспечить максимальное удобство для восприятия и понимания программы человеком. При прочтении программы должна четко прослеживаться логика ее работы. Программирование "без `goto`" значит, что операторы перехода не используются без крайней необходимости.

Основная структурная теорема утверждает: "Алгоритм любой сложности можно реализовать, используя только три конструкции: следование (оператор за оператором), повторение (цикл) и выбор (ветвление, альтернатива)" — Э. Дейкстра.

Любая программа получается построенной из *стандартных логических структур*: следования, ветвления и повтора, которые частично были рассмотрены ранее:

- *следование* — это последовательность операторов, групп операторов, выполняемых друг за другом в порядке их следования в тексте программы;
- *ветвление* — управляющая структура, которая в зависимости от выполнения заданного условия определяет выбор для исполнения одного из двух или более заданных в этой структуре групп операторов;
- *повторение* — цикл, в котором группа операторов может выполняться повторно, если соблюдается заданное условие.

Существенная особенность всех этих структур — 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. каждая из входящих в структуру групп операторов может быть одним оператором, группой операторов и может быть любой из допустимых структур — *допускается вложение структур*. К трем главным структурам следует добавить вспомогательные: неполную альтернативу, вызов процедуры (особенно рекурсивной), множественное ветвление и работу с функциями пользователя.

6.2. Нетрадиционное использование пользовательских подпрограмм

6.2.1. Рекурсия

В ряде приложений *оптимизация* алгоритма решения задачи требует вызова для выполнения подпрограммы из раздела операторов той же самой подпрограммы.

Предположим, что мы, например, пытаемся прочитать текст, пользуясь толковым словарем. Когда в тексте встречается незнакомое слово, то его объяснение ищется в словаре. В объяснении слова могут, в свою очередь, встретиться незнакомые слова, которые также нужно будет найти в словаре и т. д. Эту процедуру можно определить с помощью следующих правил:

1. Найдите слово в словаре.
2. Прочитайте статью, объясняющую значение этого слова.
3. Если объяснение понятно, т. е. статья не содержит слов, вызывающих затруднение, продолжите чтение с последнего прерванного места.
4. Если в объяснении встречается незнакомое слово, то прекратите чтение, запомните место прекращения и выясните значение слова, придерживаясь совокупности правил 1—4.

Указанный выше свод правил называют рекурсивным или использующим себя, поскольку в нем содержится ссылка на собственное определение. Речь идет не о порочном круге определений, а об образе действий, который пригоден для использования и вполне выполним.

В рекурсивном описании действий имеет смысл *обратить внимание* на следующие обстоятельства. Во-первых, процедура содержит всегда, по крайней мере, одну терминальную ветвь и условие окончания (пункт 3). Во-вторых, когда процедура доходит до рекурсивной ветви (пункт 4), то процесс приостанавливается, и новый аналогичный процесс запускается с начала, но уже на новом уровне. Прерванный же процесс запоминается. Он будет ждать и начнет исполнение лишь по окончании нового процесса. В свою очередь новый процесс может приостановиться, вступить в ожидание и т. д. Так образуется как бы последовательность прерванных процессов, из которых выполняется лишь последний в настоящий момент времени процесс, а по окончании его работы продолжает выполняться предшествовавший ему процесс. Как уже указывалось, при каждом очередном входе в подпрограмму ее локальные параметры размещаются в особым образом организованной области памяти — стеке. Целиком весь процесс выполнен, когда стек опустеет или, другими словами, все прерванные процессы выполнятся.

Рекурсия — это такой способ организации вычислительного процесса, при котором процедура или функция в ходе выполнения составляющих ее операторов обращается *сама к себе*. При использовании рекурсии необходимо обращать особое внимание на *выход из подпрограммы в нужный момент*.

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 $S_n = S_{n-1} + n$; $S_1 = 1$.

Рекурсия *полезна*, прежде всего, в случаях, когда основную задачу можно разделить на подзадачи, имеющие ту же структуру, что и первоначальная задача. Подпрограммы, реализующие рекурсию, называются *рекурсивными*. Для понимания сути рекурсии лучше трактовать рекурсивный вызов как вызов другой подпрограммы.

Не следует путать рекурсию с итерацией — повторяемым выполнением некоторых действий до тех пор, пока не будет удовлетворяться некоторое условие. Большинство алгоритмов можно реализовать двумя способами: итерацией и рекурсией.

Особенности рекурсии:

- использование рекурсивной формы организации алгоритма обычно выглядит *изящнее* итерационной и дает более *компактный текст программы*;

- *недостатки* рекурсии состоят в следующем:
  - если глубина рекурсии очень велика, то оттранслированная программа будет требовать во время исполнения много памяти — это может привести к переполнению стека;
  - рекурсивные алгоритмы, как правило, выполняются более медленно;
- при рекурсивном программировании велика вероятность ошибок, способных вынудить программиста к перезагрузке компьютера.

Таким образом, если у программиста есть возможность выбора между итерацией и рекурсией, то предпочтительным выбором является *итерация*.

В целях повышения безопасности работы рекомендуется:

- для *включения проверки переполнения стека* необходимо использовать директиву компилятора `{S+}` или воспользоваться пунктом меню интегрированной среды Turbo Pascal **Options | Compiler | Stack checking**;
- для *включения проверки диапазона* необходимо использовать директиву компилятора `{R+}` или воспользоваться пунктом меню интегрированной среды Turbo Pascal **Options | Compiler | Range checking**;
- в начале каждой рекурсивной процедуры и функции поместить строку `if keypressed then halt;`, позволяющую при "зависании" программы выйти из нее без перезагрузки компьютера, просто нажав клавишу.

Замечание

Функция `keypressed` возвращает результат `true`, если на клавиатуре была нажата клавиша, порождающая *символ*, и `false` — в противоположном случае. Например, оператор `repeat until keypressed;` содержит пустой оператор и обеспечивает паузу при выполнении программы до нажатия клавиши.

Стандартная процедура управления программой `halt(n)` прекращает выполнение программы и передает управление системе программирования (если выполнялся pas-файл) или операционной системе (если выполнялся exe-файл). Может указываться код завершения программы `n`, передаваемый в операционную систему. Значение `n=0` соответствует нормальному завершению. Вызов `halt` эквивалентен вызову `halt(0)`. Значение кода возврата, которое фактически не превышает 255, можно проверить в bat-файлах с помощью переменной `errorlevel` и команды `if`.

Листинг 6.1 содержит программу, которая вычисляет суммы первых `n` членов гармонического ряда 1+1/2+1/3+... с использованием процедуры `sumset`, предусматривающей традиционное накопление суммы в цикле `for` и процедуры `recursion`.

Листинг 6.1. Вычисление суммы членов гармонического ряда

```
var n: integer;    { кол-во суммируемых членов ряда }
    s: real;       { сумма элементов ряда }
```

```
procedure sumset(n: integer; var sum: real);
var i: integer;   { номер элемента ряда }
begin
   sum:=0;
   for i:=1 to n do sum:=sum+1/i;
end;
procedure recursion(n: integer; var sum: real);
begin
   if n=1 then sum:=1+sum
          else
             begin
                sum:=sum+1/n;
                recursion(n-1,sum)   { рекурсивный вызов }
             end
end;
begin
   writeln('Вычисление частичной суммы ряда 1+1/2+1/3+...');
   write('Введите кол-во суммируемых членов ряда: '); read(n);
   sumset(n,s);
   writeln('Сумма первых ',n:3,' членов ряда (цикл) равна: ', s:7:3);
   s:=0; recursion(n,s);
   writeln('Сумма первых ',n:3,' членов ряда (рекурсия) равна: ', s:7:3);
end.
```

Комментарии

Если не предусмотреть в процедуре прекращение вычислений при `n=1`, то они не завершатся (программа "зациклится").

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. Традиционная процедура `sumset` в этом случае гораздо проще, эффективнее и естественнее. Рекурсивная процедура `recursion` позволяет проиллюстрировать механизм рекурсии на простом и понятном примере.

Листинг 6.2 содержит программу, которая выводит на экран цифры целого положительного числа `n` в обратном порядке.

Листинг 6.2. 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.
```

Рассмотренный ранее листинг 5.5 использовал процедуру для вычисления и печати членов ряда Фибоначчи. В ней, прежде всего, вычислялись фиктивные члены ряда. Если превратить эти члены в параметры, то можно получить процедуру, печатающую `n` членов ряда, идущих за двумя данными членами. Модифицируем процедуру `fibon`, превратив переменные `fn1` и `fn` в параметры, не приписывая им начальных значений (листинг 6.3).

Листинг 6.3. Программа вычисления членов ряда Фибоначчи (вариант № 3)

```
procedure fibonachi(n,fn1,fn: integer);
{ рекурсивная процедура вычисления и печати чисел Фибоначчи }
begin
   if n > 0 then
      begin
         writeln(fn1+fn);
         fibonachi(n-1, fn, fn1+fn);
      end;
end;
{ основная программа }
var n, a, b: integer;
begin
   write('Введите число членов ряда Фибоначчи: '); readln(n);
   write('...следующих за двумя данными членами: '); readln(a,b);
   fibonachi(n,a,b); readln;
end.
```

Комментарии

Если пользователь захочет вывести на экран `n=10` первых членов ряда Фибоначчи, то он должен указать в качестве `a` и `b` два фиктивных члена — 1 и 0 соответственно. Если нужно вывести десять членов ряда, начиная с шестого, необходимо ввести для `n, a, b` числа 10, 3 и 5 соответственно.

Если два последних параметра не были бы соседними членами ряда Фибоначчи, то процедура напечатала бы не последовательность чисел Фибоначчи, а некоторую другую последовательность, члены которой вычисляются в соответствии с тем же правилом, что и члены ряда Фибоначчи.

Листинг 6.4 содержит программу вычисления факториала числа *n* с использованием рекурсии.

Листинг 6.4. Вычисление факториала числа с использованием рекурсии

```
var n: integer; f: longint;
{ функция }
function fakt(n: integer): longint;
{ описание функции, n — формальный параметр-значение типа integer, }
{ результат выполнения функции имеет тип longint }
begin
   if n=1 then fakt:=1              { проверка завершения рекурсии }
          else fakt:=n*fakt(n-1)    { рекурсивное вычисление n! }
end;
{ основная программа }
begin
   write('Введите число n для вычисления n! '); readln(n);
   f:=fakt(n);   { вызов функции для фактического параметра n }
   writeln('Для числа ',n,' значение факториала = ',f); readln;
end.
```

Комментарии

При вычислении факториала проверяется условие `n=1`. Если оно выполняется, то функции `fakt` присваивается значение `1`, на этом выполнение функции завершается. Если условие `n=1` не соблюдается, то выполняется вычисление произведения `n*fakt(n–1)`. Вычисление произведения носит *рекурсивный характер*, т. к. при этом осуществляется вызов функции `fakt(n–1)`, значение которой вычисляется, в свою очередь, через вызов функции `fakt`, параметром которой также будет функция `fakt`, и т. д., до тех пор, пока значение формального параметра `n=1`. Так как базовая часть описания рекурсивной функции `fakt` определяет значение `fakt` для `n=1` равным единице, то рекурсивные вызовы функции `fakt` больше не выполняются. Вместо этого выполняется вычисление функции `fakt` для чисел, возрастающих от `1` до `n`, причем функция `fakt` всякий раз возвращает значение, равное произведению очередного числа *k* на факториал от числа *k*–1. Последнее возвращение результата вычисления функции `fakt` присвоит переменной `f` значение произведения всех чисел от `1` до `n`, т. е. факториал числа `n`.

Обратите внимание, что факториал любого числа по определению является целым числом. В силу того, что *n!* — быстрорастущая функция, уже для `n=13` возникает выход за границу диапазона типа `longint` (см. табл. 2.1).

6.2.2. Опережающее объявление

Помимо *прямых рекурсий*, рассмотренных ранее, рекурсивный вызов может быть и *косвенным*, т. е. одна процедура вызывает другую процедуру, которая, в свою очередь, вызывает первую процедуру.

А т. к. до вызова процедуры она обязательно должна быть описана, то используется *опережающее объявление* — процедура содержит описание *только своего заголовка*, вслед за которым ставится зарезервированное слово `forward` (опережающий), а описание *текста процедуры* помещается *далее* в тексте программы уже *без повторения* описания списка формальных параметров и называется *определяющим объявлением*.

Опережающее и определяющее объявления должны находиться в одной и той же части объявления процедур и функций. Между ними могут быть объявлены другие процедуры и функции, и они могут вызывать процедуру с опережающим объявлением. Таким образом, возможна *взаимная рекурсия*.

Например:

```
var a,b: real;
{ опережающее объявление процедуры p1 }
procedure p1(x: real); forward;
{ объявление 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 объявление процедуры `p1` позволило использовать обращение к ней из процедуры `p2`, т. к. при трансляции компилятору до вызова процедуры `p1` из опережающего объявления становятся известными ее формальные параметры, и он может правильно организовать

ее вызов. В самом теле процедуры `p1` уже нет необходимости описывать параметры, т. к. это было сделано при опережающем описании.

6.2.3. Вложенные подпрограммы-процедуры

В качестве примера использования *вложенных подпрограмм-процедур* рассмотрим программу, в которой реализован известный алгоритм древней игры "Ханойские башни" (листинг 6.5).

Имеются три стержня (№ 1, № 2, № 3), на одном из них (например, на правом — № 3) насажены диски разных размеров, причем диски располагаются так, чтобы стержень с дисками напоминал башню, т. е. внизу находятся самые большие диски, а вверху маленькие. Цель игры — перенести башню с правого стержня (№ 3) на левый (№ 1), причем за один раз можно переносить лишь один диск и при этом можно насаживать только диск с меньшим диаметром на диск с большим диаметром, а снятый диск нельзя отложить — его необходимо сразу же надеть на другой столб. Средний стержень (№ 2) является вспомогательным для временного хранения дисков.

Листинг 6.5. 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);   { рекурсивный вызов }
         movedisk(from,tol);
         movetower(hight-1,work,tol,from);   { рекурсивный вызов }
      end;
end;
```

```
begin
   writeln('Введите число колец: '); readln(n);
   { вызов процедуры с передачей ей фактических параметров }
   movetower(n,right,left,centre);
end.
```

6.2.4. Параметры-процедуры и параметры-функции

В качестве *расширения* стандартного Паскаля, Turbo Pascal позволяет рассматривать процедуры и функции как объекты, которые можно присваивать переменным и которые могут выступать в качестве параметров — *процедурные типы* делают это возможным.

Как только процедурный тип определен, можно объявлять переменные этого типа. Такие переменные называются *процедурными переменными*. Они могут быть использованы в качестве формальных параметров при вызове процедур и функций. Подобно тому, как целочисленной переменной можно присвоить целочисленное значение, процедурной переменной можно присвоить процедурное значение. Такое значение может, конечно, быть и другой процедурной переменной, но может также быть идентификатором процедуры или функции. В этой ситуации объявление процедуры или функции допустимо рассматривать как особый вид объявления константы, значением которой является процедура или функция.

Как и во всех других операциях присваивания, переменная в левой части и переменная в правой части оператора присваивания должны быть *совместимыми по присваиванию*. Для того чтобы считаться совместимыми по присваиванию, процедурные типы должны иметь одинаковое число параметров, параметры в соответствующих позициях должны быть тождественных типов, наконец, типы результатов функций должны быть идентичны. Дополнительно к совместимости типов процедуры или функции должны удовлетворять следующим требованиям, если они присваиваются процедурной переменной:

- они должны быть объявлены с *директивой* `far` (использование дальнего типа вызова подпрограмм) и откомпилированы в состоянии `{$F+}`;
- они не должны быть:
  - стандартной процедурой или функцией;
  - вложенной процедурой или функцией;
  - `inline` процедурой или функцией;
  - `interrupt` процедурой или функцией.

Замечание

`Inline` процедуры и функции используются для включения машинных кодов в программу на Turbo Pascal. `Interrupt` процедуры и функции используются для обработки прерываний *(см. разд. 8.5)*.

При использовании параметров-процедур или параметров-функций в списке перед соответствующими формальными параметрами указывается зарезервированное слово `procedure` или `function`.

Например:

```
procedure ex(k,l:integer;var m:real;procedure pr;function st:real);
```

В списке формальных параметров процедуры `ex`:

- `k`, `l` — параметры-значения;
- `m` — параметр-переменная;
- `pr` — параметр-процедура;
- `st` — параметр-функция, результатом выполнения которой будет значение вещественного типа.

При вызове подпрограммы выполняется обычная подстановка. При этом если процедуры и функции, фигурирующие в качестве фактических параметров, имеют, в свою очередь, параметры, они могут быть только параметрами-значениями.

Параметры процедурного типа особенно полезны в ситуациях, когда над множеством процедур или функций выполняются *общие действия*. Например, процедура или функция *F* используется в качестве параметра процедуры или функции *G*, если *F* должна вычисляться во время выполнения *G* и если *F* представляет собой разные процедуры или функции в различных обращениях к *G*.

Примером использования параметров-функций может служить программа, которая с помощью одной и той же процедуры печати таблицы выводит на экран три таблицы арифметических функций (сложения, умножения и произведения суммы на разность чисел), каждая из которых выполняется отдельной функцией (листинг 6.6).

Листинг 6.6. Программа вывода таблиц арифметических функций

```
uses crt;
{ описание процедурного типа func — целой функции }
{ двух аргументов целого типа }
type func=function(x,y: integer): integer;
{ директива компилятору — использование дальнего типа вызова подпрограмм }
{$F+}
```

```
{ описание функции сложения двух целых чисел }
function add(x,y: integer): integer;
begin
   add:=x+y;
end;
{ описание функции умножения двух целых чисел }
function mult(x,y: integer): integer;
begin
   mult:=x*y;
end;
{ описание функции умножения, суммы на разность двух целых чисел }
function funny(x,y: integer): integer;
begin
   funny:=(x+y)*(x-y);
end;
{$F-}
{ описание процедуры вывода на экран таблицы для }
{ двух чисел от 1 до 10, вид арифметической операции }
{ задается значением параметра-функции operation }
procedure type_tabl(w,h: integer; operation: func);
var x,y: integer;
begin
   for y:=1 to h do
      begin
         for x:=1 to w do
         write(operation(x,y):5); writeln;
      end;
end;
{ основная программа }
begin
   clrscr;
   { вызов процедуры для печати таблицы сложения }
   type_tabl(10,10,add); readln;
   {вызов процедуры для печати таблицы умножения}
   type_tabl(10,10,mult); readln;
   { вызов процедуры для печати таблицы произведений }
   { суммы чисел от 1 до 10 на их разность }
   type_tabl(10,10,funny); readln;
end.
```

Комментарии

В данной программе процедура `type_tabl` представляет собой общее действие, выполняемое над параметрами — функциями `add`, `mult`, `funny`. После запуска программы сначала вызывается процедура `type_tabl` для фактических

параметров `10,10` и `add`, в результате чего формальным параметрам `x` и `y` присваиваются значения чисел `10` и `10` соответственно, а формальному параметру `operation` процедурного типа `func` присваивается имя фактической функции `add`. В результате этого на экран будет выведена таблица сложения от 1 до 10. Затем процедура `type_tabl` вызывается к исполнению для фактических параметров `10`, `10` и параметра — функции `mult`, в результате чего на экран будет выведена таблица умножения от 1 до 10. Аналогично вызов процедуры `type_tabl` с параметрами `10,10` и `funny` даст в результате на экране таблицу произведения суммы на разность чисел от 1 до 10.

`{$F+}` — указание *(директива)* компилятору Turbo Pascal на использование дальнего `far` типа вызова для корректной обработки вызова процедуры `type_tabl` с параметрами-функциями.

Рассмотренная ранее программа (см. листинг 5.9) позволяла выполнить вычисление приближенного значения определенного интеграла, в том числе по методу Симпсона. Программа (листинг 6.7) 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). Применение процедурного типа позволяет записать алгоритм в компактной и удобной для восприятия форме.

Листинг 6.7. Вычисление определенного интеграла (вариант № 3)

```
{ описание процедурного типа func — вещественной функции }
{ вещественного аргумента }
   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 }
{ директива компилятору — использование дальнего типа вызова подпрограмм }
{$F+}
{ подинтегральная функция #1 }
function f1(t: real): real;
   begin
      f1:=2*t;
   end;
{ подинтегральная функция #2 }
function f2(t: real): real;
   begin
      f2:=t;
   end;
```

```
{$F-}
procedure simps(f: func; a,b: real; n: integer; var int: real);
var
   sum,h: real;
   j: integer;
begin
   if odd(n) then n:=n+1;
   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);
   int:=2*h*sum/3
end;
{ основная программа }
begin
   write('Введите нижнюю и верхнюю границы отрезка: '); readln(tn,tk);
   write('Число разбиений отрезка: '); readln(n);
   { формальному параметру f процедурного типа func }
   { присваивается имя фактической подинтегральной функции f1 }
   simps(f1,tn,tk,n,s);
   writeln('Значение первого интеграла S1: ',s:10:7); square:=s;
   simps(f2,tn,tk,n,s);
   writeln('Значение второго интеграла S2: ',s:10:7); square:=square-s;
   writeln('Площадь криволинейной трапеции равна S1-S2: ',square:10:7);
end.
```

6.2.5. Нетипизированные параметры-переменные

Недостатком Turbo Pascal является невозможность использования в подпрограммах массивов произвольной длины с "плавающими" границами изменения индексов. Например, если разработана программа, обрабатывающая матрицу из 10 строк и 10 столбцов, то для обработки матрицы из 9 строк и 11 столбцов необходимо "вручную" переопределить тип в тексте программы и заново ее откомпилировать. Решение проблемы возможно при помощи *указателей и индексной арифметики (см. гл. 13)*, а также *нетипизированных* параметров.

Параметр считается нетипизированным, если тип формального параметра-переменной в заголовке подпрограммы не указан, при этом соответствующий ему фактический параметр может быть переменной любого типа. Нетипизированными могут быть только параметры-переменные.

Нетипизированные параметры в сочетании с механизмом совмещения данных в памяти очень удобно использовать для передачи подпрограмме одно-

мерных массивов переменной длины. Таким образом, *одна* подпрограмма может обрабатывать векторы *разных* размеров. В программе (листинг 6.8) функция `stdeviat` вычисляет среднеквадратическое отклонение вектора случайных чисел, длина которого также меняется случайным образом.

Листинг 6.8. Вычисление среднеквадратического отклонения

```
uses crt;
const nmax=100;   { максимальная длина вектора }
var i,j,n: integer; a: array[1..nmax] of real;
function stdeviat(var x; n: integer): real;
var a: array[1..2*maxint div sizeof(real)] of real absolute x;
    i: integer; s1,s2,average,variance: real;
begin
   s1:=0; s2:=0;
   for i:=1 to n do
      begin
         s1:=s1+a[i]; s2:=s2+sqr(a[i]);
      end;
   average:=s1/n;                          { математическое ожидание }
   variance:=(s2-n*sqr(average))/(n-1); { дисперсия }
   stdeviat:=sqrt(variance);               { средний квадрат }
end;
begin
   clrscr;
   for i:=1 to 10 do
      begin
         n:=random(nmax)+2;   { текущая длина }
         { a[j] — случайное действительное число в диапазоне от 0 до 1 }
         for j:=1 to n do a[j]:=random;
         writeln('Средний квадрат = ',stdeviat(a,n):10:7,', число
↳элементов = ',n:2)
      end;
end.
```

Комментарии

Для передачи в функцию массивов переменной длины `n` одна и та же область памяти попеременно трактуется как содержащая данные то одного, то другого типа. Используется явное совмещение в памяти данных разного типа по одному и тому же абсолютному адресу. Для этого локальная переменная — массив `a` описывается с использованием зарезервированного слова `absolute`, за которым помещается идентификатор ранее определенной переменной. Эта переменная — нетипизированный формальный параметр `x`, которому соответствует

фактический параметр — вектор переменной длины. В этом случае первые байты их внутреннего представления будут располагаться по одному и тому же абсолютному адресу памяти компьютера.

При обращении к функции `stdeviat` нетипизированный параметр — массив `x` передается по ссылке. В памяти он совмещается с локальной фиктивной переменной — одномерным массивом `a`. Его длина выбирается с учетом максимального допустимого размера структурных типов и памяти, выделяемой для хранения переменной типа `real`. Объем памяти под вещественную переменную был указан ранее (см. табл. 2.2). В программе для его определения используется функция `sizeof` *(см. разд. 3.3.1)*.

ЧАСТЬ II

На пути к вершинам

ГЛАВА 7

Библиотечные модули

При изучении темы "Процедуры и функции" внимательный читатель, конечно, заметил, что практически все подпрограммы, представленные в качестве примеров в *главах 5* и *6*, имеют универсальный характер. Их можно использовать не только в той программе, частью которой они являются, но и во многих других программах. Это было одной из целей, которые ставили перед собой авторы при подборе примеров. 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 Turbo Pascal есть замечательное средство преодоления этих неприятностей — библиотечные модули. Но прежде чем приступить к изучению модульного программирования, разберемся более обстоятельно, как происходит обработка исходного текста программы в системе программирования Turbo Pascal (Borland Pascal).

Замечание

В стандартном комплекте поставки фирмы Borland имеется два варианта среды программирования для операционной системы DOS — Turbo Pascal (файл Turbo.exe) и Borland Pascal (файл Bp.exe). Они используют один и тот же язык программирования Turbo Pascal и отличаются только отдельными элементами интерфейса. Начинающий программист разницу между этими системами может и не заметить, однако при изучении последующих разделов особенности среды Borland Pascal при необходимости будут оговариваться. Следует отметить, что Borland Pascal — более развитая среда программирования, которая имеет ряд преимуществ перед Turbo Pascal и позволяет более рационально использовать ресурсы компьютера (в первую очередь, оперативную память).

7.1. Компиляция и компоновка программы

Программа, написанная на любом языке программирования, перед выполнением должна быть приведена к виду, пригодному для исполнения, т. е. переведена с языка программирования на машинный язык. *Машинный язык* — это система команд, которую "понимает" и может выполнить процессор. Другими словами, *исходный код программы* перед выполнением должен быть преобразован в *исполнимый код*. К термину "код" необходимо привыкнуть — текст программы обычно называют ее кодом. Прилагательное "исполнимый" представляет собой перевод английского слова *executable*. Другие варианты перевода: "исполняемый", "выполнимый", от него же произошло и известное расширение исполнимых файлов — exe.

Из этого правила есть исключения, например, в очень популярном сейчас языке Java не предусмотрено преобразование исходного кода в полностью готовый к исполнению код. Однако для Turbo Pascal исходный текст, находящийся в файлах с расширением pas, всегда может быть преобразован в исполнимые файлы с расширением exe, которые затем можно запускать на выполнение непосредственно из операционной системы.

Обратное преобразование, к сожалению, невозможно — из исполнимых файлов уже не восстановить исходного текста программы на Turbo Pascal, поэтому все исходные тексты должны сохраняться (исходные тексты ценятся гораздо выше исполнимых файлов).

При работе в среде Borland Pascal исполнимый файл формируется автоматически, поэтому файл с расширением exe появляется в текущем каталоге после первого удачного запуска программы. Однако среда Turbo Pascal формирует исполнимый код в оперативной памяти и, не записывая его на диск, сразу выполняет.

Замечание

Для формирования exe-файла придется прибегнуть к небольшой хитрости — в меню **Compile** изменить установку **Destination** с **Memory** (значение по умолчанию) на **Disk**. После этого при запуске программы будет сформирован exe-файл в текущем каталоге.

Рассмотрим процесс преобразования исходного кода в исполнимый более подробно. Выделяют два этапа этого процесса:

- компиляцию;
- компоновку.

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, которые находятся в отдельном файле Turbo.tpl (это библиотека Turbo Pascal), без которой файл Turbo.exe нерабо-

тоспособен. Код программы, полученный после компиляции, называют *объектным кодом*. Библиотека Turbo Pascal также является объектным кодом.

На этапе компоновки к объектному коду программы добавляется объектный код стандартных процедур и функций из библиотеки Turbo Pascal, и в результате он превращается в полноценный исполнимый код.

Компиляция и компоновка — это два различных процесса. Первый из них выполняет программа-*компилятор* (Compiler), ее основное назначение — проверка программы на наличие синтаксических ошибок и, в случае отсутствия таковых, формирование объектного кода. Процесс компоновки выполняет программа-*компоновщик* (иначе ее называют редактор связей, Linker), ее назначение — добавить к программе весь недостающий код из других файлов, скомпоновав полноценный исполнимый код.

Из сказанного выше вытекает очень важный вывод — если компилятор имеет возможность обрабатывать не только законченные программы, но и особым образом оформленные совокупности подпрограмм, 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.

Компилятор Turbo Pascal обладает возможностью раздельной компиляции. В языке Turbo Pascal для этих целей введено понятие библиотечных модулей.

7.2. Понятие модуля

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, функций. На практике часто пользуются просто 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).

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. Удобно оформлять такие процедуры и функции (а также необходимые для их работы типы, константы, переменные) в виде отдельных модулей.

По ходу работы любой программист обычно накапливает для себя целую коллекцию таких полезных модулей — свою личную библиотеку. А это оз-

начает, что ему придется писать 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 разработать только короткие программы, для которых преимущества структурного и модульного подхода неочевидны. Тем не менее, каждый разработанный вами собственный 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.
- При выполнении программы на Turbo Pascal исполнимый код модулей размещается в отдельных сегментах памяти, следовательно, количество одновременно используемых модулей ограничивается только доступной памятью. Это предоставляет возможность создавать программы большого размера.
- Еще одно немаловажное обстоятельство — при разработке больших программ отдельные модули могут разрабатываться различными программистами, т. к. это относительно автономные программные единицы.
- Для языка Turbo Pascal уже имеется большое количество модулей. Это и стандартные модули, которые будут подробно рассматриваться в следующей главе, а также многочисленные модули, тексты которых приво-

дятся в различных книгах по программированию и/или распространяются на дискетах и компакт-дисках.

Тем не менее, начнем с правил оформления собственных модулей. Они достаточно просты и доступны начинающему.

7.3. 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. 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.

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, содержащие некоторый фрагмент программы. 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, что очень важно!

Для подключения модуля к программе или другому модулю используется предложение: `uses список модулей;`.

В программе оно располагается до начала других объявлений.

Например, `uses crt, graph, triangle;`

7.4. Пример разработки модуля

Разработаем модуль `triangle`, содержащий набор процедур и функций для выполнения расчета треугольников, используя известные формулы из геометрии. При этом предусмотрим два наиболее часто встречающихся способа задания треугольника: по трем сторонам и по координатам его вершин. Модуль позволяет выполнить следующие действия:

- вычисление сторон по координатам вершин треугольника;
- проверку существования треугольника;
- расчет площади и периметра;
- расчет радиусов вписанной и описанной окружности.

Такой модуль (листинг 7.1) может оказаться полезным при решении геометрических задач, особенно если вы забыли какую-либо из формул.

Листинг 7.1. 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.

7.4.1. Пример использования модуля

Решим 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 котором из них можно разместить идеально круглую клумбу наибольшего размера (оценим по радиусам вписанных окружностей).

В программе (листинг 7.2) приведен вариант решения данной задачи.

Листинг 7.2. 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.
```

7.5. Компиляция модулей

В среде Turbo Pascal имеются средства, управляющие способом компиляции модулей и облегчающие разработку крупных программных проектов. Результатом компиляции модуля является файл с тем же самым именем и

расширением tpu (Turbo Pascal Unit), который можно сохранить на диске так же, как и 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, там нет этой опции в меню **Compile**.

Наконец, последний пункт **Primary file...** позволяет задать файл, который будет автоматически добавляться в начало исходного текста перед компиляцией. Таким способом удобно отлаживать модули, подключая к ним исходную программу в качестве **Primary file**. При этом в процессе отладки не придется постоянно перемещаться между окнами основной программы и модуля.

Если модуль полностью отлажен и протестирован, то можно распространять его в виде tpu-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 tpu-файла его восстановить невозможно. Из исходного кода получить tpu-файл можно в считанные секунды: компилятор Turbo Pascal — один из самых быстрых.

ГЛАВА 8

Стандартные модули

8.1. Краткое описание модулей

В Turbo Pascal имеется набор стандартных модулей, их состав меняется в различных версиях, поэтому рассмотрим основные: `system`, `crt`, `graph`, `dos`, `printer`. Модуль `graph` выделен в отдельный tpu-файл, а остальные входят в состав библиотечного файла Turbo.tpl (tpl — Turbo Pascal Library, библиотека Turbo Pascal).

Обратите внимание — лишь один модуль `system` ввиду исключительной важности подключается к любой программе автоматически, все остальные необходимо подключать, указывая их за словом `uses`.

- Модуль `system`. Представляет основную библиотеку стандартных подпрограмм Turbo Pascal, без которой не может быть выполнена ни одна программа. В обычных условиях работы можно даже не подозревать о существовании этого модуля, однако в случае отсутствия или повреждения файла Turbo.tpl при запуске любой программы тут же появляется сообщение **File not found (SYSTEM.TPU)** — это компоновщик не может найти файл модуля `system`, чтобы сформировать исполнимый код.

  В модуль `system` входят все основные стандартные процедуры и функции (процедуры ввода/вывода, работа со строками, математические операции и функции и т. п.). Так как этот модуль подключается автоматически, то входящие в него процедуры и функции считаются встроенными в Turbo Pascal.

- Модуль `crt`. В него входят 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, чтением 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 нуль). 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 `graph`. Содержит обширный набор типов, констант, процедур и функций для управления графическим режимом работы экрана. С помощью подпрограмм, входящих в модуль, можно создавать разнообразные графические изображения.
- Модуль `dos`. В модуль входят процедуры и функции, организующие доступ ко всем средствам дисковой операционной системы MS-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 с помощью соответствующего *драйвера* — специально написанной программы для управления периферийным устройством (в том числе и принтером). Однако эта тема выходит за рамки данной книги.

Модули `crt` и `graph` предоставляют начинающим программистам большой набор несложных в использовании, 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 красивым оформлением.

Прежде чем познакомиться с составом модулей `crt` и `graph`, необходимо дать краткие пояснения по основным принципам формирования изображений на экране.

8.2. 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 разную 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 электронной 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.

Замечание

В Turbo Pascal есть средства прямого доступа к ячейкам оперативной памяти, пользоваться которыми требуется с величайшей осторожностью и применять лишь в случае крайней необходимости. Для качественного вывода на экран обычно хватает средств модулей `crt` и `graph`, но при желании можно воспользоваться средствами прямого доступа к памяти и портам (*см. разд. 13.1*). В данной главе будут частично затронуты *предопределенные массивы* `Mem`, `MemW` и `MemL`, которые позволяют записать информацию или прочитать ее непосредственно из ячеек памяти (один, два или четыре байта), а также массивы `Port` и `PortW` для доступа к внешним устройствам.

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 изо-

бражений — 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.

8.3. Модуль *CRT*

8.3.1. Процедуры и функции управления экраном

При работе на персональном компьютере пользователь обычно работает 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. 8.1.

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, опреде-

ленными в модуле `crt` (табл. 8.1). 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.

Рис. 8.1. Состав байта цветовых атрибутов (текстовый режим)

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. 8.2).

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 8.1. *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

Таблица 8.1 *(окончание)*

Цвет	Наименование константы	Значение константы
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 8.2. *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. 8.3).

Таблица 8.3. *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 *(см. разд. 2.1.3)* 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 8.1 содержит простую программу, которая демонстрирует использование процедур установки цветов, очистки экрана, а также позиционирования курсора (см. табл. 8.4).

Листинг 8.1. 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.
```

8.3.2. 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.

При работе с окнами полезными являются процедуры и функции, приведенные в табл. 8.4. 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. 8.4 содержит основные процедуры и функции, позволяющие управлять выводом в активное окно (или полный экран, если окно не объявлено).

Таблица 8.4. *Процедуры и функции для управления выводом в активное окно*

Процедура или функция	Назначение
`ClrScr`	Очищает активное окно и устанавливает курсор в левый верхний угол

Таблица 8.4 *(окончание)*

Процедура или функция	Назначение
`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 8.2, 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 8.2. 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.
```

8.3.3. 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 экране. Кстати, именно она является виновницей печально знаменитой ошибки в модуле `crt`.

В примере (листинг 8.3) процедура `delay` замедляет процесс заполнения экрана по спирали разноцветными звездочками. Другие примеры использования процедуры `delay` будут представлены в программах оживления изображений ниже.

Листинг 8.3. 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.
```

8.3.4. 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 соответствует никаких ASCII-кодов, 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 c 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.

С помощью программы, приведенной в листинге 8.4, можно узнать коды всех интересующих вас клавиш.

Листинг 8.4. 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.
```

Замечание

Для реализации паузы до нажатия любой клавиши в программе корректнее использовать функцию readkey, чем цикл repeat ... until keypressed, по-

скольку при этом не остается "мусора" в буфере клавиатуры. Очистить буфер клавиатуры можно командой `while keypressed do readkey;`.

Функция `readkey` широко применяется в программах не только текстового, но и графического режима. Она приведена во многих листингах программ, например, использование этой функции демонстрирует программа (листинг 8.6). В *приложении 2* приводится текст модуля `crtplus`, содержащего несколько полезных процедур и функций, которых нет в стандартном `crt`.

8.3.5. 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;
```

Программы, приведенные в листингах 8.5 и 8.6, иллюстрируют возможности Turbo Pascal по управлению звуком.

Листинг 8.5. 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 {три октавы}
      begin
         for n:=1 to 7 do {по 7 нот в каждой октаве}
            begin
               sound(octava_1[n]*koef);
               delay(300); nosound;
            end;
```

```
         koef:=koef*2;{переходим к следующей октаве}
      end;
   sound(octava_1[1]*koef);delay(300); nosound;{"до" четвертой октавы}
end.
```

Листинг 8.6. 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.
```

8.4. Модуль *GRAPH*

8.4.1. 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 может быть и файл с другим именем. При использовании графических шрифтов для вывода текста необходимо наличие файлов с расширением chr.

Переключение между текстовым и графическим 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`, устанавливающая 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 быть в том каталоге, к которому ведет указанный путь. В противном случае может появиться сообщение об ошибке **BGI Error: Graphics not initialized (Use InitGraph)**. 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 нужен 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 используется 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, который обеспечивает наилучшую разрешающую способность и палитру. Обычный графический режим, который устанавливается при подключении драйвера Egavga.bgi, обеспечивает 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 старании можно получить очень неплохие графические программы. Если все-таки очень хочется иметь лучший графический режим, то модуль `graph` позволяет подключать не только стандартные, но и пользовательские драйверы, написанные в соответствии со стандартом. В настоящее время распространяется несколько драйверов, обеспечивающих лучшее разрешение и/или палитру цветов, чем Egavga.bgi. Для их установки в модуле `graph` имеются функции `RegisterBGIDriver` и `InstallUserDriver` *(см. приложение 2)*.

Основные процедуры и функции, используемые при переключении видеорежимов, представлены в табл. 8.5.

Таблица 8.5. *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, также 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) увеличивается 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. 8.6 приведены основные функции для работы с координатами.

Таблица 8.6. *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 при изложении материала.

8.4.2. 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. 8.7.

Таблица 8.7. *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*)
`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 D*x* (по *X*) и на D*y* (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

Таблица 8.7 *(окончание)*

Процедура	Назначение
`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 с 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
`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. 8.8)

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 осво-

ить технику построения изображений с использованием графических примитивов (листинги 8.7—8.10). Во всех примерах предполагаем, что графический драйвер находится в текущем каталоге. Вероятнее всего, что на вашем компьютере он находится в каталоге BGI родительского каталога Turbo Pascal. Не забудьте указать правильный путь к графическому драйверу.

Листинг 8.7. 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 не будем проверять функцию `GraphResult`.

Листинг 8.8. 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 (листинг 8.9) приводится самый простой, но далеко не самый лучший способ "оживить" изображение — использование процедуры cleardevice для стирания картинки (в данном случае — "вагончика"). Процедура delay, которая используется для задержки картинки на экране, входит в состав модуля crt.

Листинг 8.9. 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 (листинг 8.10) несколько серьезнее. Программа изображает 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 такой график получается лучше, 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 8.10. 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)/639; { определили шаг изменения x (640 точек на графике,
                                                 а интервалов 639) }
   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.
```

8.4.3. 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. 8.8 приведены процедуры для установки цветов и стилей.

Таблица 8.8. *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-битный образец (для нестандартного стиля), толщина
`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. 8.2).

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 8.11, 8.12), демонстрирующие все цвета и стили.

Листинг 8.11. 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 8.12. 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.
```

8.4.4. 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. 8.9.

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);
```

Таблица 8.9. *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

8.4.5. 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 ошибок при выполнении. Это объясняется различными принципами 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. 8.10. Пример программы, выводящей текст, приведен в листинге 8.13.

Таблица 8.10. *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`)

Таблица 8.10 *(окончание)*

Процедура	Назначение
`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. 8.11). 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 программе при помощи функции `InstallUserFont`, в качестве параметра которой передается имя файла с новым шрифтом (без расширения chr).

Таблица 8.11. *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. 9.3)*, которая позволит преобразовать числовое значение в строку текста, а затем вывести полученную строку при помощи процедур `OutText` или `OutTextXY`.

Листинг 8.13. 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.
```

8.4.6. Сохранение и восстановление битовых образов изображений

Известно, что изображение строится в 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 памяти изображение можно переместить обратно в видеопамять, чтобы снова увидеть его на экране.

В текстовом режиме для подобных манипуляций приходится обращаться напрямую к известным адресам видеопамяти *(см. приложение 2)*. К счастью, модуль `graph` содержит средства сохранения и восстановления образов изображений (табл. 8.12). В табл. 8.13 приведены константы для параметра `BitBit`, используемого в одной из процедур, перечисленных в табл. 8.12.

Таблица 8.12. *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 в *гл. 13)*
`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 8.13. *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, 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 8.14) используется динамическое выделение памяти и работа с указателем. Эта тема будет подробно рассмотрена в *гл. 13*. Пока приведем текст с минимальными комментариями.

Листинг 8.14. 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.
```

8.5. 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 MS-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 разделе мы рассмотрим только первые три группы. Средства модуля `dos` для работы с файлами будут рассмотрены в *гл. 12*. Примеры обращений к прерываниям операционной системы можно найти в *приложении 2* (работа с мышью, изменение размера курсора в текстовом режиме).

8.5.1. Работа с системной датой и временем

Имеется четыре простых в использовании, но чрезвычайно мощных процедуры, которые обеспечивают доступ к системной дате и времени (табл. 8.14). 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 повлияет на дальнейшую работу всего программного обеспечения компьютера.

Таблица 8.14. *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.)

Таблица 8.14 *(окончание)*

Процедура	Назначение
`SetDate(year,month,day)`	Устанавливает новую системную дату (пользоваться осторожно!)
`GetTime(hour,minute,sec,sec100)`	Возвращает системное время: часы, минуты, секунды и сотые доли секунды
`SetTime(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.

8.5.2. Функции для обработки параметров командной строки

Сначала поясним назначение данной группы функций (табл. 8.15). 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 какой-нибудь *файловый менеджер* типа Far или Norton Commander), то после имени программы можно указывать через пробел один или более параметров.

Например: `format a:`. При запуске программы Format.com ей передается в качестве параметра имя диска, который нужно отформатировать.

Таблица 8.15. *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.

8.5.3. 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. 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 ноль. Другие возможные значения 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 будет подробно обсуждаться в последней главе *(см. разд. 13.1)*, но некоторые пояснения необходимо дать уже сейчас. 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 ее размера может понадобиться немалое количество памяти. В случае ее нехватки программа выполняться не будет, а переменная `DOSError` примет значение 8.

В связи с этим существуют строгие требования на распределение памяти для программ, которые будут использовать 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 (листинг 8.15) может служить образцом для запуска любой внешней программы, т. к. демонстрирует технику выполнения этого действия.

Листинг 8.15. Запуск внешней программы

```
{$M $2000,0,0} { 8 Кбайт стека и 0 байт динамически распределяемой области }
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" при запуске Turbo Pascal
векторы прерываний операционной системы}
   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.
```

ГЛАВА 9

Обработка строк текста

9.1. Типы данных *CHAR* и *STRING*

В Turbo Pascal имеется два типа данных для работы с текстами:

- `char` — литерный или символьный тип;
- `string` — строковый тип или просто строка.

9.1.1. 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 *(см. приложение 2)*. 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'`

9.1.2. 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.

Например:

```
const stradres = 'ул. Мира, 35';     { строковая константа }
      sadr: string[12] = ' ул. Мира' { типизированная константа, ее можно
      использовать как обычную переменную }
```

```
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`.

Строка в языке Turbo Pascal трактуется как массив символов. Для строки из *n* символов в памяти отводится *n*+1 байт; *n* байтов — для хранения символов строки, а 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. 9.1).

Таблица 9.1. *Распределение памяти для хранения 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 байтов, из них последние 4 байта оказались незанятыми, т. к. реальная длина строки составляет 8 символов, это значение и 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 понятно 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).

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 целиком в операторах 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 имеется функция `length`, которая возвращает значение нулевого байта строки *(см. разд. 9.3)*.

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 реализовать только в том случае, если необходимо совместить ввод и какую-то обработку символов (см. листинг 9.9). 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 9.1). В данном случае программа сначала выводит строку в "перевернутом" виде (например, вводим *РОЗА*, а выводится *АЗОР*). Затем та же строка выводится в прямом виде, но каждое слово с новой строки. Считаем, что слова разделены пробелами, причем не обязательно одиночными. Обратите внимание на условие, которое проверяет начало нового слова. Разбивка строки на слова — это типовое действие со строками.

Листинг 9.1. 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]);
   for i:=1 to length(s)-1 do
     if (s[i])=' ') and (s[i+1]<>' ') then writeln
                                      else write(s[i]);
   readln;
end.
```

Еще один пример, в котором выполняется обработка строки как массива символов. Данная программа (листинг 9.2) проверяет правильность расстановки скобок в выражении (можно считать ее маленькой частичкой компилятора, т. к. перед преобразованием любого выражения в исполнимый код всегда проверяется правильность расстановки скобок). В последней главе (см. листинг 13.3) приводится пример программы для решения аналогичной задачи в более сложной постановке.

Листинг 9.2. 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.
```

9.2. 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*.

9.2.1. Операция сцепления (+)

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", 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 компилятора Turbo 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 на следующих этапах постижения искусства программирования.

Для присваивания строковой переменной результата строкового выражения используется оператор присваивания.

Например: `str1 := 'Группа учащихся'; str2 := str1 + ' школы-лицея';`

Приведем пример использования операции конкатенации (листинг 9.3). Допустим, необходимо дополнить исходную строку справа символом `'*'` до заданной длины. Такое действие иногда выполняют при формировании важных денежных документов, чтобы нельзя было ничего вписать в пустые места.

Листинг 9.3. Дополнение строки звездочками

```
var s:string;
procedure dopstr(var s:string; n:integer);
begin
   while length(s)<n do s:=s+'*';
end;
begin
   writeln('Введите строку:');readln(s);
   dopstr(s,80); writeln(s); readln
end.
```

Комментарии

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 друг другу 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 важно!*

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 свою 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. есть риск потери символов справа, если не хватит места в той строке, которая передается в качестве фактического параметра.

9.2.2. 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*. 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. 4.2.3)*, 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 приведем в качестве примера алгоритм слияния двух отсортированных массивов строк (листинг 9.4). Допустим, происходит слияние двух предприятий и требуется объединить списки сотрудников таким образом, чтобы не нарушить алфавитного (лексикографического) порядка.

Листинг 9.4. 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.
```

9.3. Строковые процедуры и функции

Любые действия по обработке строк, вплоть до самых сложных, можно запрограммировать, используя алгоритмы обработки одномерных массивов. Однако для упрощения программирования задач обработки текстов Turbo 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 возможно, использовать стандартные процедуры и функции. Правильность их работы проверена многолетним опытом использования. Строковые процедуры и

функции располагаются в модуле system, поэтому *для их применения дополнительных модулей подключать не требуется.*

Все подпрограммы разделим на несколько групп.

9.3.1. 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 (листинг 9.3) можно было бы заменить `s:=s+'*'` на `insert('*',s,length(s)+1)`.

Обратите внимание — при вставке символов обязательно нужно контролировать длину полученной строки, чтобы не потерять последние символы. При этом никакого сообщения об ошибке не выдается даже при включенной директиве {$R+}.

9.3.2. 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, где находится первый символ подстроки str1. Если в str2 подстроки str1 не найдено, результат равен 0.

  Например:

```
s1:= 'Turbo Pascal';
n1:=pos('Pascal',s1); n2:=pos('pascal',s1);
{в результате n1=7; n2=0 (pascal и Pascal — 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));
```

9.3.3. Процедуры преобразования типов

Turbo 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 имеется две процедуры: `str` и `val`.

Процедура `str` уже упоминалась, когда речь шла о выводе текста в графическом режиме *(см. разд. 8.4)*. Удобно использовать процедуру `str` 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, значение `code` равно нулю, 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. 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. 3.9.1)*. Все же более универсальным вариантом представляется использование вспомогательной строки, т. к. в этом случае можно не просто проверить правильность ввода, но даже исправить некоторые ошибки при вводе.

9.4. Примеры программ обработки строк

9.4.1. 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 9.5) 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 9.5. Программа, сокращающая размер текста 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 т. д.', 'и тому подобное','т. к.', 'т. е.');
      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 9.6) 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 9.6. 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.
```

9.4.2. Преобразование строчных букв в заглавные

Это тоже полезное действие, которое часто применяется в системах хранения и поиска данных. Дело в том, что при сравнении строк обязательно учитывается регистр, поэтому такие способы 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 слова к буквам какого-то одного регистра, чаще к заглавным буквам.

Начнем с простого примера. Преобразуем все строчные английские буквы введенной строки в заглавные. Для этого достаточно знания одной функции `upcase`, которая преобразует к верхнему регистру один символ (листинг 9.7).

Листинг 9.7. Преобразование строчных латинских букв в заглавные

```
var  s : string;
procedure upstring(var s:string);
{ процедура  преобразования строки }
var i: integer;
begin
   for i:=1 to length(s) do
      s[i]:=upcase(s[i]);  { преобразование одного символа }
end;
begin      { начало основной программы }
   write('Введите исходную строку:');readln(s);
   upstring(s); writeln(s); readln;
end.
```

Усложним задачу. Пусть требуется преобразовать в строке строчные буквы *русского* алфавита в заглавные. Функция `upcase` с символами русского алфавита не работает. Придется писать свою функцию, работающую 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 (листинг 9.8).

Листинг 9.8. Преобразование русских букв в заглавные

```
var s:string;
function upstringrus(s:string):string;
const small:string='абвгдежзиклмнопрстуфхцчшщъыьэюя';
        big:string='АБВГДЕЖЗИКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ';
var i,n:integer;
begin
   for i:=1 to length(s) do begin
      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;
   upstringrus:=s;
end;
begin
   write('Введите строку');readln(s);
   writeln(upstringrus(s)); readln;
end.
```

Чтобы окончательно закрыть эту тему, приведем заключительный пример. Процедура `getupstr` преобразует русские и английские буквы в заглавные непосредственно при вводе. Буква вводится в любом регистре, а на экране — всегда заглавная. Такой способ ввода используется в различных программах, теперь есть возможность узнать, как это делается (листинг 9.9).

Листинг 9.9. Ввод строки с приведением всех букв к верхнему регистру

```
uses crt;
var s:string;
procedure getupstr(var st: string);
var c:char;
begin
   st:='';{ сформировали пустую строку — это действие нельзя забывать }
   repeat
      c:=readkey;{ слепой ввод без отображения на экране — модуль crt }
      case c of
{коды заглавных и прописных русских букв от А до П различаются на 32, }
         'a' .. 'п': c:=chr(ord(c)-32);
         'р' .. 'я': c:=chr(ord(c)-80); { а коды от Р до Я — на 80 }
         'a' .. 'z': c:=upcase(c);
      end;
      if c<>#13 then { не клавиша <Enter> }
      begin
         st:=st+c; write(c);{ добавляем символ к строке и выводим }
      end;
   until c=#13;{ ввод завершен нажатием клавиши <Enter> }
   writeln;
end;
begin
   writeln('Введите строку текста и нажмите <Enter>'); getupstr(s);
   writeln('Значение s:'); writeln(s); readln
end.
```

ГЛАВА 10

Множества

Множества имеют большое значение в математике, поэтому не удивительно, что в языке Turbo Pascal имеется такой тип данных.

10.1. 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 Turbo 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).

Это существенные ограничения, которые 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.

При задании значений элементов множества применяются 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 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 знаниями;
- все операции над множествами выполняются значительно эффективней, чем над другими структурами данных.

10.2. Операции над множествами

При работе с множествами допускается использование следующих операций:

- отношения (=, <>, >=, <=);
- объединения множеств (+);
- пересечения множеств (*);
- разности множеств (−);
- проверка 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. 10.1).

Таблица 10.1. *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 не равными, если они отличаются по количеству элементов или по значению хотя бы одного элемента (табл. 10.2).

Таблица 10.2. *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. 10.3).

Таблица 10.3. *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. 10.4).

Таблица 10.4. *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. 10.5).

Таблица 10.5. *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. 10.6).

Таблица 10.6. *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. 10.7).

Таблица 10.7. *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. 10.8).

Таблица 10.8. *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 10.1 содержит небольшую программу, демонстрирующую операции 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 10.1. 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 Turbo 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 1)*.

Тем не менее, нетрудно 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 в предложении, введенном с клавиатуры, определить количество гласных букв. Программа для решения этой задачи приводится в листинге 10.2.

Листинг 10.2. 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.
```

Комментарии

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 получился нагляднее, кроме того, проверка на принадлежность множеству выполняется намного быстрей, чем поиск символа в строке.

10.3. 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 для формирования случайной игровой ситуации. Существуют различные способы решения данной задачи. Интересный вариант решения получается с использованием множества. Это логично, поскольку множество не содержит одинаковых элементов по определению.

Листинг 10.3. 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 массиве. Если воспользоваться примерно тем же алгоритмом, 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 есть очень быстрый вариант решения этой задачи. Идея 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 10.4).

Листинг 10.4. 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; { меняем значениями a[k] и a[l] }
      end;
   writeln('Сформированный массив:');
   for k:=1 to n do write(a[k],' ');
   readln;
end.
```

ГЛАВА 11

Записи

При решении задач обработки большого количества значений используются массивы. Но при работе с массивами основное ограничение заключается в том, что каждый элемент массива должен иметь один и тот же тип данных. Иногда для решения задач, в которых возникает необходимость хранить и обрабатывать данные различных типов, используются отдельные массивы для каждого типа данных, а установление соответствия между ними выполняется при помощи индексов. Но есть способ лучше.

Для *записи комбинации данных разных типов* в Turbo Pascal применяется комбинированный тип данных — *запись*. Запись представляет собой наиболее общий и гибкий структурированный тип данных, т. к. она может быть образована из неоднотипных компонентов, и в ней явным образом выражена связь между элементами данных, которые характеризуют реальный объект.

11.1. Определение и правила записи

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 различного типа. Чтобы можно было ссылаться на тот или иной компонент записи, *каждое 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;
```

Можно задать в программе типизированную константу типа записи, 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 имени переменной и имени поля, разделенных точкой. Такая комбинация называется *составным именем*. Например, 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 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 операторах 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 операторах ввода/вывода запись целиком (как и массив).

Например:

```
writeln(m) { ошибочная инструкция }
```

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. Для решения этой проблемы в языке Turbo Pascal имеется оператор `with`, в виде:

```
with ПеременнаяТипаЗапись do Оператор; { обычно составной оператор }
```

Один раз указав переменную 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;
```

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 комплексных чисел или координат точки на плоскости или в пространстве.

```
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.

Записи часто применяются для работы с датами (день, месяц, год) или отрезками времени (часы, минуты, секунды). В большинстве современных языков программирования есть специальные типы данных для работы с датой и временем, однако в Turbo Pascal все действия с датой и временем приходится программировать вручную.

Рассмотрим пример использования массива записей. Составим программу, которая организует ввод данных об учащихся: имя, фамилия, возраст, школа, класс и записывает их в массив записей, а затем выводит сведения об учащихся по номеру записи и по 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 11.1).

Листинг 11.1. Пример использования массива записей

```
type pupil = 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 pupil;
    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.
```

11.2. Записи с вариантами

Записи, имеющие строго определенную структуру, весьма ограничены по возможностям применения. Однако в языке Turbo Pascal есть возможность задать тип записи, содержащий несколько вариантов структуры. Такие записи называются *записями с вариантами* и являются средствами объединения записей, которые похожи, но не идентичны по форме.

Записи с вариантами состоят из *фиксированной* и *вариантной* частей.

Фиксированная часть задается так же, как и в простых записях. Вариантная часть формируется с помощью оператора `case ... of` и может состоять из нескольких вариантов. Оператор 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:

```
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();`.

В качестве примера использования записей с вариантами разработаем программу, позволяющую создавать электронную картотеку публикаций, а именно — после запроса о виде публикации произвести ввод параметров указанной публикации. Пусть в картотеке хранятся сведения о публикациях трех видов: книги, статьи 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 — в вариантную часть (листинг 11.2).

Листинг 11.2. Пример использования записей с вариантами

```
type
   kind = (book,article,paper);
   publication=record
   { фиксированная часть записи }
      autor,title : string;
      year : word;

   { вариантная часть записи }
      case  kindp : 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.
```

ГЛАВА 12

Файлы

Мы уже знаем, что эффективным способом хранения и обработки большого количества однотипных переменных в программе являются массивы *(см. гл. 4)*. Однако как у обычных неиндексированных переменных, так и у индексированных — массивов — есть общий недостаток: они не пригодны для долговременного хранения информации. По окончании работы программы все данные будут потеряны. Результаты работы, конечно же, могут быть сохранены в виде бумажной копии — распечатаны на принтере, или некоторое время находиться на экране. Но при решении серьезных задач, связанных с вводом и/или получением большого количества данных, возникает необходимость их сохранения даже после того, как программа будет завершена, а компьютер выключен. В этом программисту могут помочь файлы (от английского слова *file* — подшивка, картотека).

Например, для расчета заработной платы сотрудников предприятия необходимы исходные данные на десятки или сотни человек: карточки персонального учета, лицевые счета, табель рабочего времени и т. д. Зарплата рассчитывается ежемесячно. Данные обновляются, достигая к декабрю расчетного года максимального объема. Вводить их с самого начала каждый месяц неразумно. Файлы обеспечивают хранение сведений, их обновление и обработку. Налоговая инспекция и фонды требуют, чтобы сведения о доходах передавались как в бумажном виде, так и на "магнитных носителях" — в виде файлов на дискетах. По сути, большая часть окружающей нас информации, в том числе и личные сведения о любом человеке, хранится как файлы.

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. 12.1)*.

Особенности файлов:

1. 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.
2. 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".
3. *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.

Замечание

Массивы и файлы — по сравнению с массивом файл обладает как недостатками, так и достоинствами. Первые две особенности файла роднят его с массивом, последняя отличает и является преимуществом. Ведь особенностью массива как, впрочем, множества и записи является то, что его размер должен быть известен компилятору заранее. Для файла этого не требуется — число его компонентов можно увеличивать. Но, с другой стороны, массив целиком располагается в быстродействующей оперативной памяти компьютера. Файл же, по сути, представляет собой именованное пространство на диске, а скорость обмена данными с винчестером, дискетой или компакт-диском гораздо ниже. Поэтому операции с файлом выполняются существенно медленнее. Для их ускорения необходимо приобретать дорогие быстродействующие внешние устройства, совершенствовать алгоритмы, уменьшая число обращений к жесткому диску или считывать данные из "медленного" файла в "быструю" оперативную память, обрабатывать их там и снова записывать на диск. В отличие от массива, файлы позволяют сохранять информацию для последующего использования не только в текущей, но и в другой программе.

Файлы *удобно использовать* потому, что:

- многократно вводить повторяющиеся или просто большие объемы данных для обработки, а потом терять их — неразумно, это трата времени. Удобно заранее создавать отдельные файлы данных, которые можно применять неоднократно;
- программа, работающая с файлами данных, достаточно автономна и не требует постоянного присутствия пользователя, освобождая его для другой работы;
- файл данных может быть подготовлен другой программой, становясь посредником по обмену данными между двумя разными задачами;
- файл может выступать как средство связи программы с внешней средой.

12.1. Некоторые сведения о файловой системе

Язык Turbo Pascal разрабатывался для программирования в *дисковой операционной системе* (Disk Operating System) фирмы Microsoft — MS-DOS. Поэтому все файлы Turbo Pascal используют ее файловую систему — FAT16 (File Allocation Table, таблица размещения файлов). Даже если программа языка Turbo Pascal будет выполняться из Windows, имя файла задается по правилам MS-DOS. Не вдаваясь в подробности, рассмотрим основные требования, предъявляемые FAT16 к именам файлов.

12.1.1. Имя и расширение файла

Имя файла состоит из двух частей: собственно имени и его расширения. Имя может содержать от 1 до 8 символов. *Расширение* отделяется от имени точкой, за которой следует от 1 до 3 символов расширения. Например, Example.pas или Example.com. MS-DOS переводит все строчные буквы в соответствующие им прописные, поэтому символы имени и расширения могут быть как прописными, так и строчными латинскими буквами, а также цифрами. Использование кириллицы — русских букв и некоторых разрешенных символов (кроме знака подчеркивания _) не рекомендуется. Расширение имени файла является необязательным, но его необходимо использовать. Оно, как правило, правильно описывает содержимое файла, что удобно. Это позволяет не только программисту, но и операционной системе быстро узнавать по стандартным расширениям типы файлов и правильно их обрабатывать, зная, какой программой был создан данный файл (табл. 12.1). В частности, система Turbo Pascal всем программам назначает расширение pas. Результат компиляции программы на диск *(см. приложение 1)* — исполняемый файл имеет расширение exe (от *execute*, исполнять).

Таблица 12.1. *Некоторые стандартные расширения имен файлов*

Расширение	О файле
pas	Файл программы на языке Pascal
c, cpp	Файл программы на языке C или C++
asm	Файл программы на языке Assembler
for	Файл программы на языке Fortran
bak	Резервная копия файла *(backup)*, создаваемая перед его изменением. Имя остается прежним. Позволяет восстановить содержимое файла *(см. приложение 1)*

Таблица 12.1 *(окончание)*

Расширение	О файле
bat	Командный *(batch)* файл. Представляет собой текстовый файл. Строки — команды MS-DOS и команды запуска файлов программ. Выполняется из командной строки как исполняемый файл
exe, com	Файлы программ, готовых к исполнению на языке команд процессора
txt	Текстовый файл. Часто файлы делят на две категории: текстовые и двоичные. *Текстовые файлы* предназначены для чтения человеком и состоят из строк символов. Строки отделяются друг от друга специальными символами #13#10 (CR/LF — возврат каретки/новая строка). Файлы, не являющиеся текстовыми, называются *двоичными*
dat	Файл данных
doc	Файл документа (возможно текстовый). Редактируется и печатается из программы обработки текстов — текстового редактора
bmp, jpg	Файлы, содержащие растровые изображения, созданные в графическом редакторе или отсканированные
zip, arj, rar	Файлы архивов. Архив содержит сжатые файлы и экономит место на диске

12.1.2. Каталоги

Подобно книгам в большой библиотеке, в компьютере одновременно может храниться множество файлов. Для быстрого поиска нужной книги читателем используется предметный или алфавитный каталог. В требовании на книгу он указывает, наряду с дополнительной информацией, ее инвентарный номер и шифр. Именно эти данные позволяют библиотекарю в считанные минуты в хранилище библиотеки, содержащей десятки и сотни тысяч изданий, найти нужный зал, стеллаж и полку, на которой стоит единственная нужная читателю книга.

Для регистрации имен файлов в файловой системе компьютера также используются каталоги (directory), а в Windows — папки (folders).

Каталог — это поименованная область на диске. Имя каталога содержит от 1 до 8 символов, которые выбираются по тем же правилам, что и имена файлов. Расширение 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)*. В этом случае каталог более высокого уровня, который содержит вложенные, выступает по отношению к ним как *родительский* ка-

талог *(надкаталог)*. Каталог, с которым в настоящий момент времени работает пользователь, называется *текущим*.

При первоначальной подготовке жесткого диска компьютера или обычной дискеты к работе выполняется операция *форматирования*. При этом поверхность диска разбивается на дорожки и сектора, а также создается служебная область и пустой корневой каталог. Каждый магнитный 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.

Кроме имени файла и его расширения каталог содержит *характеристики* файла:

- размер (в байтах);
- дату и время создания или последнего изменения файла;
- 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 на диске.

Информация о каждом конкретном файле 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 к файлу. *Путь* — это перечень имен подкаталогов, которые отделены друг от друга с помощью обратной косой черты, за которыми следует собственно имя файла: `Диск:\ИмяПодкаталога1\...\ИмяПодкаталогаN\ИмяФайла`.

Каждое имя каталога в пути соответствует входу в подкаталог с таким именем. Знак ".." соответствует входу в надкаталог. Максимально допустимая длина пути (т. е. полного имени файла) — 79 символов.

Например, `c:\turbo\pas\example.pas`

В этом примере файл программы `example.pas` содержится в каталоге `\pas`, предназначенном программистом для хранения программ и данных. Каталог

\pas зарегистрирован в каталоге \turbo. Следовательно, \pas — подкаталог (или вложенный каталог), а \turbo по отношению к нему выступает как *родительский* каталог. Но для корневого каталога c:\ уже сам \turbo будет являться вложенным.

Диск: представляет собой имя магнитного диска, на котором, в конечном счете, хранится файл. Двоеточие указывает на то, что символ слева — имя диска. Для его обозначения применяются латинские буквы от A до Z. Жесткий диск — это C:. Если, для удобства пользователя, большой винчестер разбивают на части *(логические диски)*, для их обозначения используются следующие по порядку буквы латинского алфавита (D:, E:, и т. д.). Дисковод для гибких дисков (дискет) именуется A:. Если существует второй, 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 доступа начинается с обратной косой черты, то поиск файла начинается с корневого каталога диска, иначе — с текущего.

Например, \turbo\pas\table.txt или table.txt

В программе на Turbo Pascal имя файла задается в виде текстовой константы, заключенной в апострофы, которая может быть значением строковой переменной:

Например, '\turbo\pas\table.txt' или 'table.txt'

Замечание

В операционных системах MS-DOS и Windows используются различные правила записи имен файлов. В Windows имя файла может быть представлено на любом языке и состоять из нескольких слов, разделенных пробелами. Его максимальная длина — 255 символов.

12.1.3. Устройства

В составе компьютера принято выделять отдельные компоненты — *устройства*. Те из них, что предназначены для ввода, вывода и хранения данных, называют *внешними*, или *периферийными*.

Использование файлов в Turbo Pascal было вызвано, в том числе, и необходимостью обмена данными с окружением компьютера, его аппаратными средствами: клавиатурой, дисплеем, устройством печати, каналами ввода/вывода. Все они рассматриваются в Turbo Pascal как файлы, с которыми можно работать точно так же, как с обычными файлами. Это значительно упрощает задачу программиста. Для того чтобы избежать путаницы, файлы на внешних устройствах (магнитных дисках) часто называют *физическими (внешними)* файлами. Обращение к устройствам выполняется с помощью

специальных имен, которые запрещено использовать для обычных файлов — так называемых *имен логических устройств* компьютера.

- CON. 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 *(см. разд. 2.1.3)*.
- PRN. Это 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 COM2. Это устройства, подключенные к *последовательным портам*. Используются для связи с другими компьютерами и для подключения мыши. Возможен как прием, так и передача данных, однако в каждый момент времени может использоваться либо прием, либо передача. Обычно вместо COM1 можно применять синоним AUX.
- NUL. *Нулевое*, или *"пустое"* устройство. Чаще всего используется при отладке, например, позволяет не создавать отдельный файл, когда в программе требуется указать имя входного или выходного файла. 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.

12.2. Описание файлового типа

12.2.1. Виды файлов. Файловая переменная

В Turbo Pascal имеются три вида файлов:

- текстовый файл (определяется типом `text`);
- типизированный файл (задается предложением `file of Тип`);
- нетипизированный файл (определяется типом `file`).

Для работы с файлами в программе необходимо определить *файловую переменную (файловый тип)* в разделе описаний программы в виде:

```
type
   ИмяТипа1 = text;
   ИмяТипа2 = file of Тип;
   ИмяТипа3 = file;
var
   ИмяПеременной1 : ИмяТипа1;
   ИмяПеременной2 : ИмяТипа2;
   ИмяПеременной3 : ИмяТипа3;
```

или

```
var
   ИмяПеременной1 = text;
   ИмяПеременной2 = file of Тип;
   ИмяПеременной3 = file;
```

где, как уже указывалось выше, `Тип` — любой тип Turbo Pascal, кроме файлов. Например:

```
type filetype=text;     { файловый тип }
var ftmp,f: filetype;   { файловые переменные }
```

или

```
var f1,lst: text; f2: file;
```

Замечание

Вид файла определяет способ хранения информации в файле. К сожалению, в Turbo Pascal нет средств контроля вида ранее созданных файлов. При объявлении *уже существующих* файлов программист должен *сам следить* за соответствием вида объявления характеру файла.

Файловые переменные, которые описаны в программе, называют *логическими файлами*. Все основные процедуры и функции, обеспечивающие ввод/вывод данных, работают только с логическими файлами *(см. разд. 12.3)*. Поэтому перед использованием физический файл должен быть обязательно связан с логическим файлом (файловой переменной).

Любой программе доступны *стандартные текстовые файловые переменные* `input` и `output` *(см. разд. 2.2.1)*. `Input` (ввод) — это доступный только по *чтению* файл, для ввода с клавиатуры. `Output` (вывод) — доступный только по *записи* файл, для вывода на дисплей.

Замечание

Файлы `input` и `output` и все файлы, которым присвоено пустое имя, ссылаются на устройство консоль *(см. разд. 12.1)*.

Другие файлы становятся *доступными* программе на Turbo Pascal для ввода/вывода только после выполнения процедуры *открытия* файла, которая *связывает* имя существующего или создаваемого файла с файловой переменной.

12.2.2. Указатель. Доступ к файлам

В любой момент времени программе доступен только один элемент файла, на который ссылается *указатель текущий 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 его автоматическое перемещение. 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.

Замечание

Программа на Turbo Pascal, предназначенная для работы с файлами, не зависит от вида используемого магнитного носителя информации (лента или диск). Поэтому представьте, что файл — это магнитная лента, у которой есть начало, а конец не фиксирован. Элементы файла записываются на эту ленту последовательно, друг за другом с помощью некоторого устройства — указателя файла. При чтении или записи этот указатель перемещается к следующему элементу и делает его доступным для обработки. В каждый момент доступен для записи (чтения) только тот элемент файла, на который установлен указатель.

По способу доступа к элементам различают файлы последовательного и прямого доступа *(см. разд. 12.4)*.

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 одновременно читать и записывать данные в файл.

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 поиск. Для файла прямого доступа разрешается одновременная запись и считывание данных.

12.3. Средства обработки файлов

12.3.1. Общая схема работы с файлом

Вне зависимости от файлового типа, любая программа работы с файлом должна:

1. Определить переменные файлового типа (логические файлы). При описании типа необходимо учитывать, что нетипизированные файлы можно рассматривать как частный случай типизированных, которые, вдобавок, нечасто используются. Следовательно, выбор, как правило, делается между текстовым и типизированным (двоичным) файлом. Очевидно, что для работы с текстами (наборами строк символов) наиболее подходит тип `text` *(см. разд. 12.4)*.
2. Каждому из используемых физических файлов при помощи процедуры `assign` (в переводе — назначить) программист указывает переменную файлового типа.
3. Инициировать файл — это значит указать направление передачи данных. Работа с файлом возможна, если он существует. Такой файл открывают, делая его доступным для ввода и/или вывода. В случае отсутствия файл должен быть создан. В первом случае используется процедура `reset` (переустановка), во втором случае — `rewrite` (перезапись). При открытии и создании файла для временного хранения его данных автоматически выделяется область в оперативной памяти компьютера, которая называется *буфером файла*.

Замечание

Для того чтобы ускорить обмен информацией с внешними устройствами, в файловой системе используется механизм буферизации. Операционная система устанавливает каждому открываемому файлу так называемый *обработчик файлов* с определенным номером. Он осуществляет операции обмена данными через буфер ввода/вывода. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока он полностью не будет заполнен. 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. 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 для этих переменных. В случае необходимости ее результаты записываются в новый или дописываются к уже существующему файлу. Чтение (ввод) данных или запись (вывод) выполняется при помощи инструкций `read` и `readln`, `write` и `writeln` *(см. разд. 3.1. и 3.2)*.

Обратите внимание — их первым параметром должна быть файловая переменная *(см. разд. 12.4 и 12.5)*. Она указывает на то, что при чтении значения переменных вводят из файла, а не с клавиатуры. При выводе переменные будут записываться в файл, а не выводиться на экран.

5. Закрытие файловой переменной и сохранение данных на диске по окончании работы с файлом осуществляет процедура `close`. Если файл закрыт, над ним нельзя выполнять никаких действий, пока он вновь не будет открыт.

12.3.2. Общие процедуры и функции

В Turbo Pascal существует ряд процедур и функций, применимых для файлов любых типов: `assign`, `reset`, `ioresult`, `rewrite`, `close`, `rename`, `erase`, `eof`. Рассмотрим их подробно.

- Процедура `assign(ФайловаяПеременная, ИмяФайла)` предшествует другим процедурам, т. к. ставит в соответствие физическому файлу на внешнем устройстве логический файл — файловую переменную, к которой впоследствии будут обращаться все другие файловые процедуры (связывает их). `ИмяФайла` должно представлять собой выражение строкового типа. Дальнейшие операции с переменной `ФайловаяПеременная` будут выполняться над *физическим* файлом `ИмяФайла`. Это полное имя внешнего файла, удовлетворяющее требованиям операционной системы MS-DOS *(см. разд. 12.1)*. Процедуру `assign` недопустимо использовать для открытого файла. Прежде чем использовать файловую переменную повторно, необходимо закрыть файл с помощью процедуры `close`. После вызова `assign` связь файловой переменной с внешним 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. При отсутствии внешнего файла с указанным именем возникает сообщение об ошибке **Error 2: File not found** (Ошибка 2: Файл не найден). Если при чтении файла возникнет необходимость вернуть указатель в его начало, достаточно будет просто применить процедуру `reset` к этому файлу еще раз.

- Функция `ioresult` проверяет существование файла на диске. По умолчанию при всех обращениях к стандартным функциям и процедурам ввода/вывода, используемым при работе с файлами, *автоматически* производится проверка на наличие ошибок. Программист должен предусмотреть возможность ввода неверных исходных данных пользователем

программы, например, имени файла, предназначенного для чтения. Это приведет к завершению работы программы, что нежелательно, особенно при вводе больших объемов данных. Использование директивы компилятора `{$I-}` *(см. разд. 2.1.4)* и стандартной функции `ioresult` *(см. разд. 3.9.1)* в цикле `repeat` позволит программе корректно обработать эту исключительную ситуацию (см. листинг 12.3).

- 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 внешний 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 создавать резервные копии файлов, над которыми могут производиться опасные действия (см. листинг 12.3). Обычно им назначается расширение bak (см. табл. 12.1).

- Процедура `close(ФайловаяПеременная)`. Используя процедуру `close`, программист должен закрыть файл, после того как в программе будет завершена его обработка. В противном случае может произойти *потеря данных*. При закрытии внешний файл обновляется, 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. Их необходимо предварительно закрыть. Если файл не существует, возникает ошибка выполнения программы (см. листинг 12.3).

- Логическая функция `eof(ФайловаяПеременная)` выполняет проверку, не достигнут ли конец файла (End Of File) при чтении из него данных. Функция возвращает `true`, если конец файла обнаружен, и указатель текущей позиции находится в конце файла за его последним символом. Это значит, что последний элемент в файле уже прочитан, или файл после открытия оказался пуст. В противном случае функция возвращает `false`.

  Функция `eof` находит широкое применение в задачах обработки файлов, поскольку позволяет задать условие выполнения цикла для чтения дан-

ных из файла (см. листинг 12.1). Особенно она важна для текстовых файлов. Если параметр ФайловаяПеременная отсутствует, подразумевается консоль.

Для примера создадим новый текстовый файл с именем file01.txt, содержащий одну строку текста, которая записывается в него в виде текстовой константы. Для чтения используется строковая переменная str.

```
var f: text; str: 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, str);
   close(f);
   writeln('Проверка ввода/вывода в файл:'); writeln(str);
end.
```

12.3.3. Использование логических устройств как файлов

В случае пустого имени файла файловая переменная будет связана с логическим устройством консоль con *(см. разд. 12.2)*. Тогда при вводе данные поступают с клавиатуры (стандартного файла ввода input), а при выводе размещаются на дисплее (файл вывода output).

Например, программа вводит данные с клавиатуры и выводит их на экран.

```
var f : text; str: string;
begin
   assign(f, '');   { стандартное устройство ввода input }
   reset(f);
   readln(str);
   close(f);
   assign(f, '');   { стандартное устройство вывода output }
   rewrite(f);
   writeln(f, 'Вы ввели: ',str);
   close(f);
end.
```

Напомним, что, используя стандартный модуль printer, можно следующим образом организовать вывод информации из программы на устройство печати *(см. гл. 8)*:

```
uses printer;
begin
   writeln(lst,'Проверка печати...');
end.
```

Поставив в соответствие логическому устройству `lpt1` файловую переменную `lst` текстового типа, можно добиться того же результата, используя рассмотренные выше процедуры работы с файлом:

```
var lst: text;
begin
   assign(lst,'lpt1'); { или assign(lst,'prn'); }
   rewrite(lst);
   writeln(lst,'Printing...');
   close(lst);
end.
```

Программа выведет строку **Проверка печати...** на устройство печати — принтер, хотя внешне посылает данные в файл `lpt1` *(см. разд. 12.1)*.

12.3.4. Вспомогательные процедуры и функции

Для работы с файлами применяют следующие вспомогательные процедуры и функции (табл. 12.2).

Таблица 12.2. *Некоторые процедуры и функции для работы с файлами*

Процедура, функция	Описание
	Процедуры
`Getdir`	Определяет текущий каталог на заданном диске
`Chdir`	Меняет текущий каталог
`Mkdir`	Создает подкаталог
`Rmdir`	Удаляет пустой подкаталог
`Settextbuf`	Назначает для текстового файла буфер ввода/вывода
`Flush`	Очищает буфер текстового файла, открытого для вывода (модуль `dos`)
`Fsplit`	Разделяет имя файла на путь, имя и расширение (модуль `dos`)
`Getfattr`	Возвращает атрибуты файла (модуль `dos`)
`Getftime`	Возвращает дату и время последней записи файла (модуль `dos`)
`Setftime`	Назначает новую дату и время последней записи файла (модуль `dos`)
`Setfattr`	Устанавливает атрибуты файла (модуль `dos`)

Таблица 12.2 *(окончание)*

Процедура, функция	Описание
	Функции
`Filesize`	Возвращает текущий размер файла (не используется с текстовыми файлами)
`Diskfree`	Возвращает число свободных байтов на заданном диске (модуль `dos`)
`Disksize`	Возвращает общий объем дисковой памяти на диске (модуль `dos`)
`Fexpand`	Расширяет имя файла до полностью определенного (модуль `dos`)
`Findfirst`	Ищет в заданном каталоге первый элемент, совпадающий с заданным именем файла и его атрибутами (модуль `dos`, `findnext` — следующий)
`Fsearch`	Ищет файл в списке каталогов (модуль `dos`)

12.4. Текстовые файлы

Они используются для хранения текстов, например программ Turbo Pascal. Компонентами текстовых файлов являются символы кодовой таблицы компьютера *(см. разд. 2.1.3)*, сформированные 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 сгенерировать при помощи процедур `readln`, `writeln` и распознать функцией `eoln`.

Замечание

Стандартные файлы ввода `input` и вывода `output` являются текстовыми. Перед началом выполнения программы они 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.

Текстовый файл — это последовательность символов `char`, 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) *(см. разд. 2.1.3)*.

Объявление текстовых файлов в программе осуществляется заданием файловой переменной типа `text` (текст) в виде:

```
type ИмяТипа = text;
var ФайловаяПеременная : ИмяТипа;
```

или

```
var ФайловаяПеременная : text;
```

Замечание

Текстовый файл можно подготовить и прочитать при помощи обычного текстового редактора для MS-DOS, который не использует операции форматирования текста, например, при помощи редактора Turbo Pascal *(см. приложение 1)*. Windows использует другую кодировку кириллицы — русских букв. Если они не употребляются в тексте, файл может быть подготовлен при помощи приложения "Блокнот". Программа MS Word позволяет сохранить текстовые файлы (при этом теряется форматирование) с помощью команды **Файл | Сохранить как**. Далее следует выбрать вариант **текст DOS (*.txt)**.

Длина строк файла может быть различной — от 0 до 255 символов. Позицию конкретной строки в текстовом файле нельзя получить сразу. 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*.

12.4.1. 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. 12.3)* 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*.

- 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 открытию его вновь, 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 при этом невозможно 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. 3.2.1)*. 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 того типа 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 внешнего файла, связанного с файловой переменной `f1`, последовательности цифр, затем интерпретируемой как три вещественных числа, значения которых будут присвоены переменным `a,b,c`. В случае, 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. 12.3)*, 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 пропускает пробелы и позиции табуляции перед проверкой на конец строки. Функцию можно использовать только для открытых текстовых файлов;

- функция `seekeof(ФайловаяПеременная)` аналогична `eof`, но пропускает пробелы, позиции табуляции и маркеры конца строки перед проверкой на конец файла. Функцию можно использовать только для открытых текстовых файлов.

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 12.6).

12.4.2. Задачи на текстовые файлы

Одной из основных проблем при решении подобных задач является выбор типа буферной переменной, необходимой для временного хранения строк, считываемых из файла.

Для файлов с произвольной текстовой информацией можно рекомендовать следующие варианты выбора *типа буферной переменной*:

- массив или связанный список строк, 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*.

Первый вариант требует больших затрат оперативной памяти и применяется в редких случаях, когда максимальный размер файла известен заранее, а обработка столь сложна, что требует присутствия в памяти сразу всех строк (например, сортировка по алфавиту). Второй и третий варианты не требуют больших затрат памяти, выбор типа буферной переменной здесь определяется исходя из условия задачи.

Листинг 12.1 содержит программу, которая вводит текстовый файл `file.txt` с клавиатуры, выводит его содержимое на экран дисплея, подсчитывает количество символов и строк в данном текстовом файле.

Листинг 12.1. Ввод, вывод и подсчет элементов текстового файла

```
const sumstr: word=0;    { число строк }
      sumchr: word=0;    { число символов }
var f: text;             { файловая переменная типа text }
    stroka: string;      { буферная переменная для строк файла }
begin
   writeln('Введите текст из строк, ввод закончите символом #');
   assign(f,'file.txt');
```

```
   { создание нового файла, ввод данных с клавиатуры }
   rewrite(f);
   repeat
      readln(stroka);
      if (stroka <>'#') then writeln(f,stroka);  { запись строки в файл }
   until stroka='#';
   close(f);
   { открытие, вывод и обработка файла }
   reset(f);
   writeln('Файл содержит:');
   while not eof(f) do
      begin
         readln(f,stroka);              { чтение строки из файла }
         writeln(stroka);
         inc(sumstr, 1);                { подсчет числа строк }
         inc(sumchr, length(stroka));   { подсчет числа символов }
      end;
   writeln('В файле: строк — ',sumstr,', символов — ',sumchr);
   close(f); readln;
end.
```

В листинге 12.2 приведен пример программы, которая находит в существующем файле все строки, содержащие набор символов, введенных с клавиатуры. Результат поиска выводится на экран и одновременно записывается в файл с именем `analysis.txt`.

Листинг 12.2. 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);
   { связываем f1 и f2 с именами файлов }
   assign(f1,name);assign(f2,'analysis.txt');
   { открываем f1 и создаем f2 }
   reset(f1); rewrite(f2);
   writeln('Протокол поиска:'); writeln(f2,'Протокол поиска:');
   { пока не конец файла f1, выполняем цикл }
   while not eof(f1) do
      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;
   { закрываем файлы }
   close(f1); close(f2); readln;
end.
```

Комментарии

Недостаток программы состоит в том, что проверка существования физического файла не выполняется. В случае его отсутствия программа будет аварийно завершена *(см. разд. 12.3)*. Обработка этой исключительной ситуации возможна при помощи функции `ioresult` (листинг 12.3).

Как указывалось выше, процедура `append` выполняет дозапись *в конец* текстового файла. Листинг 12.3 содержит программу, которая дозаписывает файл *в его начало*. Выполняется *проверка наличия файла* в текущем каталоге диска.

Листинг 12.3. Добавление данных в начало текстового файла

```
type filetype=text;     { файловый тип }
var
   ftmp,f: filetype;    { 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, }
{ иначе – false; после проверки файл закрывается }
var f: file;
begin
   {$i-} assign(f, filename);
   reset(f); close(f); {$i+}
   fileexists:=(ioresult=0)and(filename<>'');
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='';
```

```
   close(ftmp); writeln('Ввод окончен');
   { проверка существования файла }
   repeat
      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');
   { добавление в начало временного файла }
   assign(ftmp,'tmpfile.txt'); append(ftmp);
   assign(f,name); reset(f);
   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.
```

Комментарии

Создается новый временный файл `tmpfile.txt`, в него записываются данные, которые необходимо разместить в начале существующего файла `name`. Булевская функция `fileexists` проверяет наличие файла `name` в текущем каталоге

диска. Запрос на ввод имени файла продолжается до тех пор, пока пользователь не введет имя реально существующего файла. Если `name` найден, он открывается, но *для чтения*. Для *дозаписи* в конец при помощи `append` открывается файл `tmpfile.txt`. В цикле `while` происходит чтение файла `name` и дозаписывание файла `tmpfile.txt` строка за строкой. Условие в теле цикла позволяет пропускать при записи пустые строки файла `name`.

Оператор `case` предлагает пользователю выбор. Удаление старого файла без создания его резервной копии приводит, на самом деле, к удалению файла `name`, а `tmpfile.txt` переименовывается в `name`. При удалении с созданием резервной копии файл `name` переименовывается, получая расширение bak, а `tmpfile.txt` получает имя `name`. Выход без удаления и изменения названия оставляет файлы `name` и `tmpfile.txt`, позволяя просмотреть в любом текстовом редакторе (например Turbo Pascal) их содержимое.

В практике обработки текстов часто сталкиваются с необходимостью удаления лишних пробелов в строке. Листинг 12.4 содержит программу, которая выполняет эту операцию над текстовым файлом `file01.txt`, помещая результат в `file02.txt`.

Листинг 12.4. Удаление лишних пробелов в текстовом файле

```
var chr: char; f1,f2: text;
begin
   assign(f1,'file01.txt'); assign(f2,'file02.txt');
   reset(f1); rewrite(f2);
   while not eof(f1) do           { цикл по строкам }
      begin
         while not eoln(f1) do    { цикл по символам строки }
            begin
               read(f1,chr); write(f2,chr);
               if chr=' ' then    { символ в буфере — пробел?}
                  begin
                     repeat read(f1,chr); until chr<>' ';
                     write(f2,chr);
                  end;
            end;
         readln(f1); writeln(f2); { перевод строки }
      end;
   close(f1);close(f2);
end.
```

Удобство работы с файлами во многом определяется возможностью использования *исходных файлов данных*. Например, при вводе в текстовом редакторе или как результат работы другой программы создан непустой текстовый

файл, содержащий исходные данные — длины сторон треугольников a,b,c. Листинг 12.5 содержит программу, которая вводит эти значения из файла и рассчитывает соответствующие высоты h по известным формулам площади треугольника и формуле Герона. Результаты работы программа записывает в *файл протокола*, дополняя его.

Листинг 12.5. Работа с исходным файлом данных и файлом протокола

```
var f1,f2: text;
    filename1,filename2: string[12];
    a,b,c,ha,hb,hc,p,buf: real; i:integer;
begin
   write('Имя файла исходных данных: '); readln(filename1);
   write('Имя существующего файла протокола работы программы: ');
readln(filename2);
   assign(f1,filename1); reset(f1);
   assign(f2,filename2); append(f2);
   i:=0;
   while not eof(f1) do
     begin
       readln(f1,a,b,c); inc(i);
       p:=(a+b+c)/2; buf:=2*sqrt(p*(p-a)*(p-b)*(p-c));
       ha:=buf/a; hb:=buf/b; hc:=buf/c;
       { выполнение дозаписи в файл }
       writeln(f2,'--',i:2,'-й расчет. Данные из:',filename1:12,'--');
       writeln(f2,'a= ',a:7:3,' b= ',b:7:3,' c= ',c:7:3);
       writeln(f2,'ha=',ha:7:3,' hb=',hb:7:3,' hc=',hc:7:3);
     end;
   close(f1);close(f2);
end.
```

Комментарии

Задайте при первом выполнении программы в качестве файла исходных данных infile1.txt, где числа отделены друг от друга пробелом, при втором — infile2.txt, где используется символ табуляции.

```
{ файл infile1.txt }    { файл infile2.txt }
3 4 5                   3       6       8
3 2.2 5                 4       6.2     7
7 8 9
```

Ознакомьтесь с содержимым файла протокола. К началу работы программы файлы исходных данных и протокола должны существовать.

Текстовые файлы удобны для *наглядного* хранения массива чисел.

Рассмотрим следующую задачу. Известно, что многие насекомые ориентируются по солнцу. Предположим, что у нас есть квадратная поверхность, поделенная на клетки. В одной из угловых клеток сидит "паучок", а солнце светит из противоположного угла. "Паучок" идет к солнцу. Из одной клетки в другую он не может попасть по диагонали, а лишь через отверстия в стенках клетки. Меняя направления, "паучок" может пройти в противоположный угол квадрата различными путями. Записав число путей, ведущих в указанную клетку, получим таблицу, называемую *арифметическим квадратом*. Каждое записанное в ней число равно сумме двух чисел: находящегося непосредственно сверху и ближайшего слева. Программа (листинг 12.6), используя процедуры, *формирует арифметический квадрат* в виде массива целых чисел, *записывает* его в текстовый файл, *читает* массив из файла и *выводит* на экран.

Листинг 12.6. 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.
```

12.5. 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, 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 ограничивается при его задании. 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 автоматически 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 к нему *прямой доступ*.

Замечание

Массивы и файлы — для того чтобы обратиться к компоненту массива, достаточно указать его индексы. Компонент текстового файла становится доступным лишь после того, как мы последовательно пройдем все предшествующие ему компоненты. К компонентам типизированного файла возможен произвольный доступ. Это свойство объединяет его с массивом. Фактически типизированный файл — это массив на диске с неограниченным размером и сравнительно невысокой скоростью обработки. Очень часто в качестве элементов типизированного файла выступают *записи (см. гл. 11)*, что позволяет программисту работать с базами данных (см. листинг 12.9).

В отличие от текстового файла типизированный файл не делится на строки. Данные в нем изначально записаны во внутреннем двоичном представлении компьютера — в виде, в котором они хранятся в оперативной памяти. Вообще, все файлы, кроме файлов типа `text`, рассматриваются как двоичные. Непосредственный просмотр такого файла, например в редакторе Turbo Pascal, не позволит пользователю разобраться в его содержимом. Для

этого необходимо предварительно знать формат хранения данных в типизированном файле и уметь обрабатывать его при помощи программы Turbo Pascal.

Объявление типизированных файлов в программе осуществляется заданием файловой переменной вида `file of Тип`, где `Тип` — любой тип Turbo Pascal, 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. Отличительной особенностью последних является автоматическое прямое и обратное преобразование цепочек литер и обнаружение конца строки процедурой чтения. Процедуры `readln` и `writeln` можно применять только к текстовым файлам.

12.5.1. Процедуры и функции для типизированных файлов

При обработке таких файлов могут использоваться некоторые дополнительные процедуры и функции, а ряд известных нам общих *(см. разд. 12.3)* имеют свои особенности.

Открытие типизированного файла можно произвести стандартными способами: `reset` и `rewrite`. Процедура `reset` используется для открытия *существующего* файла, процедура `rewrite` — 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 применяются 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.

- Функция `filepos(ФайловаяПеременная)` возвращает целое число — текущую позицию в файле. Функцию нельзя использовать для текстовых

файлов. Файл должен быть открыт. Если текущей позицией является начало файла — его *первый* компонент, например, после выполнения `reset`, то функция возвращает значение *ноль*, что важно. При переходе от одного элемента к другому его значение увеличивается на единицу. Но номер физической записи будет по-прежнему на *единицу меньше* номера логической записи, хотя их *общее число совпадает*. Для случая конца файла, когда `eof` возвращает `true`, `filepos` возвращает номер последнего элемента файла (совпадает со значением функции `filesize`). Результат — `longint`.

- Функция `filesize(ФайловаяПеременная)` возвращает число элементов файла, но не текущий размер файла в байтах. Если файл пуст, возвращает 0. Функцию нельзя использовать для текстовых файлов. Файл должен быть открыт. Результат — `longint`.
- Процедура `seek(ФайловаяПеременная, НомерПозиции)` перемещает указатель файла из текущей позиции к позиции с указанным номером, не выполняя чтение или запись. Поскольку номер первого элемента файла равен 0, то оператор `seek(ФайловаяПеременная, filesize(ФайловаяПеременная))` ставит указатель файла за его конец, что используется при добавлении данных к файлу. Функцию не используют для текстовых файлов. Файл должен быть открыт. `НомерПозиции` имеет тип `longint`.
- Процедура `truncate(ФайловаяПеременная)` усекает размер файла до его текущей позиции. Функцию не используют для текстовых файлов. Файл должен быть открыт. Все элементы файла после текущей позиции удаляются, после чего текущая позиция становится концом файла.

12.5.2. Задачи на типизированные файлы

Рассмотрим ряд задач, связанных с этим типом файлов, в том числе задачу на создание простейшей базы данных.

Процедуры записи массива целых чисел в текстовый файл и его чтения из файла уже были рассмотрены (см. листинг 12.6). Не повторяя программу, приведем процедуры записи и чтения массива вещественных чисел для *типизированного* файла.

```
const max=10;
type massiv=array[1..max,1..max] of real;
var m:integer; matrix1,matrix2:massiv;
...
procedure inmatrixfiletyp(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  outmatrixfiletyp(var m,n:integer; var mas:massiv;
name:string);
{ вывод 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;
begin
   write('Введите длину стороны арифметического квадрата: ');readln(m);
   { формирование арифметического квадрата — массив matrix1 }
   square(m,matrix1);
   { запись арифметического квадрата в типизированный файл square.dat }
   inmatrixfiletyp(m,m,matrix1,'square.dat');
   writeln('Арифметический квадрат ',m,' на ',m,':');
   { чтение из типизированного файла square.dat }
   outmatrixfiletyp(m,m,matrix2,'square.dat');
   { вывод арифметического квадрата на экран — массив matrix2 }
   outmas(m,m,matrix2);
end.
```

Листинг 12.7 содержит программу, которая формирует типизированный файл из целых чисел, вводимых с клавиатуры. Их количество заранее не известно. Признаком конца ввода является 0. Программа находит: сумму и произведение чисел из файла, разность между предпоследним и вторым по счету числами, наибольшее из чисел.

Листинг 12.7. 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.
```

Комментарии

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 сумм и произведения с первого элемента (нулевого компонента) файла.

Кроме сортировки массива *(см. разд. 4.2.3)* можно выполнить сортировку файла. Она заключается в изменении порядка размещения элементов в файле в соответствии со значением определенного признака, например числовым значением. Листинг 12.8 содержит программу, которая формирует типизированный файл из `k` случайных чисел и сортирует его методом "пузырька". Дополнительный файл не создается.

Листинг 12.8. Сортировка файла методом "пузырька"

```
var  f: file of integer; buf1,buf2: integer;
     n,k: longint; ok: boolean;
```

```
begin
   assign(f,'bubble.dat'); rewrite(f);
   write('Введите количество чисел в файле: '); readln(k);
   for n:=1 to k do
      begin
         buf1:=random(100); write(f,buf1); write(' ',buf1);
      end;
   repeat
     ok:=true;
     for n:=0 to k-2 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:');
   { чтение файла после сортировки }
   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. 11)*. Листинг 12.9 содержит программу, которая создает простейшую базу 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 12.9. База данных на основе типизированных файлов записей

```
uses crt;
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
   clrscr;
   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
   clrscr;
   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.
```

Комментарии

Для каждого студента данные о номере его группы, фамилии, имени и отметках содержатся в полях записи `studrec`: переменных `group`, `surname`, `name` и массиве `oc` 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`.

12.6. Нетипизированные файлы

При выполнении копирования или обработке баз данных приходится иметь дело с файлами, состоящими из компонентов одинакового размера, структура которых не известна 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*. Для нетипизированных файлов 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 заданием 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, 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 используются соответственно процедуры `blockread` (считывает из файла в переменную одну или более записей) и `blockwrite` (записывает одну или более запись 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.

ГЛАВА 13

Динамические переменные и структуры данных

Материал данной 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 этой темой читатель уже достиг требуемого уровня и готов подняться еще на одну ступеньку в освоении профессионального программирования.

13.1. Указатели и динамические переменные

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 усложнению программы, а также существенно усложняет и сам процесс программирования. В данной главе мы покажем примеры использования указателей, когда все дополнительные затраты на их хранение и обработку окупаются в полной мере. Работа с указателями предусмотрена не только в Turbo Pascal, но и некоторых других языках программирования. Например, в языке C указатели используются практически в любой программе.

В Turbo Pascal роль указателей несколько скромнее, тем не менее начинающим программистам важно освоить базовые принципы работы с указателями, чтобы глубже понять внутренний механизм обработки и выполнения любой компьютерной программы.

13.1.1. Указатели и адреса

Определение указателя как адреса переменной несколько поверхностно. Дело в том, что многие переменные занимают несколько байтов памяти (мас-

сивы могут занимать очень большие области памяти). Каждый байт памяти имеет свой адрес (порядковый номер байта, начиная с нуля). Определим указатель на переменную как адрес *первого байта* области памяти, которую занимает переменная.

В Turbo Pascal есть 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).

В качестве индекса в этих массивах используется адрес, записанный в виде, принятом в MS-DOS: `Сегмент : Смещение относительно начала сегмента`. Такой странный способ записи адреса связан с тем, что в операционной системе MS-DOS вся память разбита на сегменты, размеры которых не более 64 Кбайт. Для получения абсолютного адреса из пары `Сегмент : Смещение` система прибавляет к сегменту справа шестнадцатеричный нуль (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 адресу его начала. Конечно, в данном случае лучше воспользоваться средствами модуля `crt` *(см. разд. 8.3.4)*.

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 — с помощью служебного слова `absolute` при описании переменной. В этом случае переменная будет располагаться именно по тому адресу в оперативной памяти, который указан после `absolute`. Пример использования данного служебного слова для совмещения двух переменных уже приводился в *гл. 6* (см. листинг 6.8). В *приложении 2* данный прием применяется для обращения к видеопамяти (см. листинг П2.1). Разумеется, использование служебного слова `absolute` — столь же опасный способ, как и обращение к памяти через предопределенные массивы.

В Turbo Pascal есть специальная операция получения указателя на переменную (или процедуру), она обозначается как `@`, а также эквивалентная ей функция `addr`.

Например, `@x` или `addr(x)` — адрес переменной `x`.

Однако в повседневной практике средства работы с адресами используются довольно редко. Указатели применяют в основном для других целей. Их основное назначение состоит в том, чтобы обеспечить механизм использования в программе *динамических переменных*, т. е. переменных, которые создаются в памяти и удаляются из нее по ходу выполнения программы. Это определение нуждается в серьезном пояснении.

13.1.2. Распределение памяти для программы на Turbo Pascal

Всем известно, что перед выполнением программы, написанной на таких языках, как Turbo Pascal или C, выполняются этапы ее компиляции и компоновки *(см. разд. 7.1)*. При компоновке программы в памяти формируется несколько специальных областей, предназначенных для хранения всех используемых в данной программе переменных. Распределение памяти для исполнимого модуля показано на рис. 13.1.

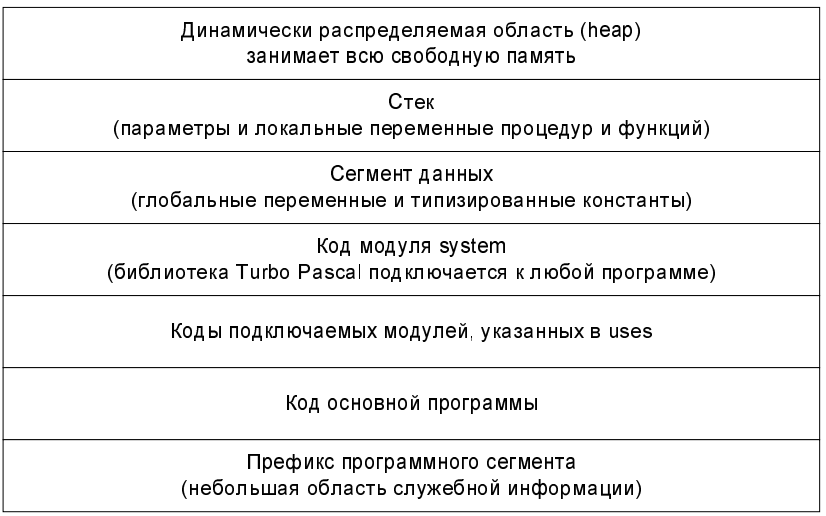

Рис. 13.1. Карта распределения памяти для исполнимого модуля программы на Turbo Pascal

Для хранения глобальных переменных, а также типизированных констант, которые в Turbo Pascal играют роль обычных переменных с заданным начальным значением, формируется область, которую называют *сегментом*

данных. Каждая переменная в зависимости от типа получает определенное количество байтов в сегменте данных. Под массив отводится столько байтов (обязательно непрерывная область), сколько нужно для хранения всех его элементов.

Суммарный размер сегмента данных определяется суммой затрат памяти на хранение всех переменных. Размер сегмента данных не изменяется от начала до конца выполнения программы. Во многих системах программирования его размер ограничен (например, в Turbo Pascal не может быть более 64 Кбайт), что приводит к существенным ограничениям на размеры используемых в программе массивов данных. Очевидно, читатель уже видел сообщение компилятора **Structure too large** при попытке объявить массив большого размера *(см. разд. 4.1)*.

Для хранения локальных переменных и параметров процедур и функций используется другая область памяти, которая получила название *стек* (stack). Смысл самого термина "стек" будет пояснен ниже. Сейчас важно уяснить, что при каждом вызове подпрограммы ее параметры и локальные переменные помещаются в стек, а по окончании работы подпрограммы — автоматически из него удаляются. Однако память под стек отводится еще до начала выполнения программы, и размер этой памяти не может быть изменен по ходу ее выполнения. Отсюда идут корни печально известной ошибки **Stack overflow**, которая порой сводит на нет отдельные гениальные идеи начинающих программистов (опытной программист с любой ошибкой справится).

Суммарный размер памяти, которая отводится под стек, в Turbo Pascal по умолчанию составляет всего 16 Кбайт, однако позже мы покажем, как можно увеличить этот размер. Тем не менее, здесь также действует ограничение в 64 Кбайт.

Для того чтобы в какой-то мере преодолеть реально существующие ограничения на размеры памяти под переменные программы, а также в целях более рационального использования памяти Turbo Pascal предоставляет программисту замечательную возможность создавать и уничтожать переменные по ходу выполнения программы. Такие переменные не объявляются в разделе объявлений и не имеют своего собственного имени, 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 имеется еще одна дополнительная область памяти, которая называется *динамически распределяемой областью*, или *кучей*. К сожалению, работать с переменными в куче можно только через указатели. Однако для тех,

кто освоит технику работы с указателями, в дальнейшем эта работа 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. разд. 8.5.3)*. В этом случае нужно использовать директиву компилятора:

```
{$M РазмерСтека, Мин.РазмерКучи, Макс.РазмерКучи}
```

Применяя эту директиву, можно заодно изменить и размеры стека, о чем уже шла речь выше, например: `{$M 65520, 0, 65536}` — увеличили стек до максимально возможного размера, а размер кучи уменьшили до 64 Кбайт.

13.1.3. Описание указателей

В Turbo Pascal имеется два различных вида указателей: типизированные и нетипизированные. *Типизированный указатель* — это указатель на переменную определенного типа, например, целого, строкового или типа массива. *Нетипизированный* — 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 стеке (если он объявлен в подпрограмме) и занимает там 4 байта (вспомним, что все адреса хранятся в виде `сегмент : смещение`). Это дополнительные накладные расходы памяти. Чем больше размер динамической переменной, тем меньше доля накладных расходов. Например, при хранении в динамической памяти массивов больших размеров лишние четыре байта на указатель несущественны. Объявить указатель на массив можно так:

```
type massiv=array[1..1000] of real;
var pmassiv=^massiv;
```

Обратите внимание — при таком объявлении память будет выделена *только под указатель*. Под массив выделится память при его создании в программе.

13.1.4. 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, которую необходимо выделить, в байтах. Размер `size` не может превышать 64 Кбайт. Это существенное ограничение, которое не позволяет, например, сохранить в одной переменной образ всего графического экрана *(см. разд. 8.4.7)*. Но для многих задач таких размеров достаточно.

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 (см. табл. 2.1—2.3). При этом указателю, который является параметром-переменной, присваивается адрес первого байта переменной.

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, 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 состоит в том, что они пропускают (забывают) процедуры выделения памяти и обнуления указате-

ля. 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 сразу очистить 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, `release`, наоборот, в конец.

В случае нетипизированных указателей можно использовать только одну процедуру: `freemem(p,size);`, которая 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`.

Все изложенное выше пока не дает ответа на вопрос: "А зачем это нужно?" Отвечаем. Динамические переменные используются в основном в двух ситуациях:

- для работы с массивами больших размеров;
- для работы с особыми структурами переменных размеров, которые получили название динамических структур данных.

Наибольший интерес для программиста представляют динамические структуры данных, поэтому на них следует остановиться более подробно.

13.2. Динамические структуры данных

В предыдущих главах читатель познакомился со всеми структурными типами данных Turbo Pascal. Очевидно, наиболее фундаментальной структурой в оперативной памяти следует считать массивы, но они имеют крайне неприятный недостаток — память под массив выделяется в полном объеме до начала выполнения программы, и размеры ее ограничены. Файлы, конечно, предоставляют уникальные возможности обработки больших объемов информации, однако работа с файлом напрямую пока выполняется непозволительно медленно.

В поисках решения проблемы быстрой обработки больших объемов данных были предложены *динамические структуры данных*. Они характеризуются следующими особенностями:

- для отдельных элементов структуры память выделяется в тот момент, когда в них появляется необходимость (а не сразу и одним блоком как для массивов);
- число элементов динамической структуры заранее не объявляется и может изменяться от нуля до некоторого значения, определяемого спецификой соответствующей задачи или доступным объемом памяти;
- память, занимаемая структурой, *не представляет собой непрерывную область*, т. е. элементы могут быть разбросаны в памяти хаотическим образом;
- логическая последовательность элементов задается в явном виде с помощью одного или нескольких указателей, хранящихся в самих элементах. Как правило, каждый элемент, кроме своего значения, хранит указатель на следующий элемент или на два соседних с ним 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.

Другое дело — живая очередь (в магазине, железнодорожной кассе). Количество человек в очереди постоянно меняется, люди приходят и уходят, но никакой неразберихи не происходит — каждый знает, за кем он стоит. Образуется связанная цепочка людей. Очевидно, кому-то из стоящих в очереди и пришла идея создания динамических структур данных.

Простая по своей сути идея оказалась не столь простой в реализации, поэтому дальнейший материал потребует повышенного внимания. Рассмотрим наиболее распространенную динамическую структуру, которая называется *связанным списком*. 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. Каждый элемент списка состоит из двух различных по значению частей: *содержательной* (информационной) и *указующей* (рис. 13.2). В минимальном варианте содержательная и указующая части занимают по одному полю записи, но могут представлять собой и более одного поля.

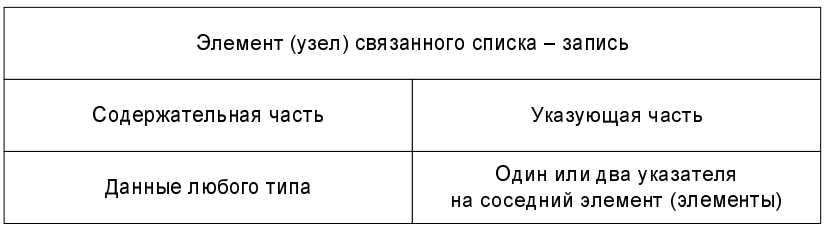

Рис. 13.2. Элемент связанного списка

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 иногда называют *линейными*. Логическая структура односвязного списка представлена на рис. 13.3.

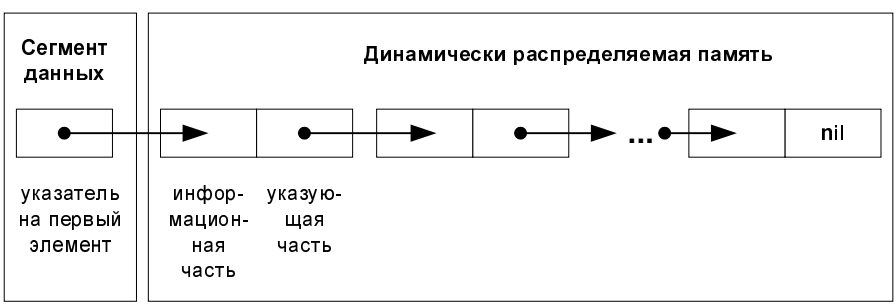

Рис. 13.3. Структура односвязного списка

В 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 двумя противоположными путями, поэтому в конце каждого из них в поле соответствующего указателя находится признак пустого указателя (`nil`).

Из 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) делаются в одном конце списка.
- *Очередь* — линейный список, в котором все добавления производятся на одном конце списка, а все удаления делаются на его другом 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 будем пользоваться при описании соответствующих алгоритмов и структур данных (табл. 13.1).

Таблица 13.1. *Обозначения переменных, используемых в листингах 13.1—13.3*

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)

Таблица 13.1 *(окончание)*

Элемент	Описание
`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.

Перейдем к примерам. Наиболее просто реализуются действия со *стеком*, поэтому первый пример (листинг 13.1) демонстрирует использование стека. Все действия со стеком выполняются только на одном 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. Именно благодаря устройству стека возможны такие приемы программирования, как вложенные подпрограммы и рекурсия.

Конечно, в примере реализуется учебный стек, содержащий целые числа.

Листинг 13.1. 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.
```

Комментарии

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`) — 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 13.2) при формировании списка указующие поля заполняются таким образом, что каждый элемент списка действитель-

но указывает на следующий за ним элемент, т. е. 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 13.2. 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.
```

Комментарии

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 13.3) 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 открывающихся скобок строится по

принципу 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 13.3. 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.
```

Подведем итоги. Итак, связанные списки и массивы — две основные 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. 4.2.3)*. 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, 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 рекомендуем не бояться указателей и динамических структур, т. к. программирование действий с ними — хороший способ развития логического мышления.

Заключение

Жизнь современного человека трудно представить без компьютера. В особенности это относится к школьникам и студентам, для многих из которых компьютер уже успел превратиться в обыденный предмет домашней обстановки. Однако далеко не все из них умеют полноценно использовать его возможности. Подлинным знатоком современных компьютерных технологий можно стать, только начав самостоятельно программировать. Лучшим языком для обучения программированию по праву считают Turbo Pascal. Вместе с тем он представляет мощное инструментальное средство написания прикладных программ, которые можно использовать в реальной работе. Стоит ли говорить, что современные технологии программирования, такие как Delphi, в буквальном смысле слова "стоят на плечах" Turbo Pascal.

Ознакомившись с содержанием, любознательный читатель без труда отметит, что книга, которую он держит в руках, насыщена большим количеством тщательно подобранных программ, имеющих отношение к решению задач из самых разнообразных областей — от простейших игр до полезных расчетов. Изучив представленные алгоритмы, начинающие программисты получат в свое распоряжение компактный и мощный инструментарий, который пригодится им в дальнейшем, вне зависимости от того, на каком языке они будут программировать завтра.

Авторы будут считать, что цель данной книги достигнута, если она поможет начинающим программистам — школьникам старших классов и студентам вузов освоить программирование на языке Turbo Pascal. Мы надеемся, что полученные при работе с книгой практические навыки позволят широкому кругу читателей быстро и качественно использовать богатые возможности языка программирования в своих целях — будь то решение школьных задач и увлекательных 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. Хочется выразить глубокую признательность генеральному директору МООИ "Инфакто" Кириллу Сергеевичу Ярошу за идею написания книги и предоставленную поддержку. Многолетняя работа связывает автора с заведующим кафедрой радиопередающих и телевизионных устройств Санкт-Петербургского государственного университета аэрокосмического приборостроения Борисом Семеновичем Тимофеевым и всем коллективом кафедры. Сердечная признательность всем тем, без чьей помощи этот труд никогда бы не состоялся: Дине Ивановне и Герману Георгиевичу Рапаковым, Людмиле Рапаковой, Наталье Невмывака (Федоровой) и многим другим. Особая благодарность — детям С. Ю. Ржеуцкой, Александру и Марине, — это те самые школьники и студенты, для которых и писалась книга, поэтому их замечания оказались очень ценными. Спасибо и вам, уважаемый читатель, за то, что вы проявили интерес и нашли время ознакомиться с этой книгой.

ПРИЛОЖЕНИЕ 1

Основы практической работы в интегрированной среде Turbo Pascal

П1.1. Работа в окне интегрированной среды, текстовый редактор Turbo Pascal

П1.1.1. 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 (версии не ниже 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 большими возможностями. В основном тексте книги, там, где это имеет значение, особенности Borland Pascal оговариваются отдельно. В дальнейшем, 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 библио-

течные модули размером до 64 Кбайт (1 Кбайт = 1024 байта). Если программа требует больше памяти, следует воспользоваться оверлейными структурами и библиотечными tpu-файлами *(см. гл. 7)*.

П1.1.2. 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, когда 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 идет 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.

П1.1.3. Создание новой программы

После запуска интегрированной среды на экране должно появиться пустое активное окно редактирования. Если окно не пусто, 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 — букв и цифр), 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 своих 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...**.

П1.1.4. 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. П1.1).

Таблица П1.1. *Комбинации клавиш для редактирования*

Клавиши	Команда меню	Функция
<Shift>+<↑>,<↓>,<→>,<←>	—	Выделение фрагмента текста
<Ctrl>+<Del>	**Edit	Clear**	Удаление выбранного текста

Таблица П1.1 *(окончание)*

Клавиши	Команда меню	Функция
<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**	Повторение последней команды **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* выполнение команды 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 команду **File** | **Open** или нажать клавишу <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 *.*.

П1.1.5. 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 можете 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. П1.2 приведены комбинации клавиш для управления окнами.

Таблица П1.2. *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
<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 клавишей <Esc> или <Ctrl>+<Break>. Для выхода из системы используется команда **File | Exit** (<Alt>+<X>).

П1.2. 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 его, но вместо нажатия клавиши <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. П1.3).

Таблица П1.3. *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**	Показ предыдущего экрана справки
<Ctrl>+<F1>	**Help | Topic Search**	Вызов контекстной информации по языку Паскаль (например, по имени стандартной функции)

П1.3. 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 *(см. разд. 1.4)*. 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 т. д., 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 ";").

П1.4. 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 клавиши <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 **Compile | Destination** и, нажав клавишу <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 переменных — отслеживание их значений в специальном окне на разных шагах выполнения алгоритма (табл. П1.4).

Таблица П1.4. *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 клавиши <Ctrl>+<F4>
<Ctrl>+<F7>	**Debug | Add watch...**	Просмотр значения переменной, на которую помещен курсор, в окне просмотра (при выполнении программы)

П1.4.1. Пример работы с отладчиком

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 клавиши <F7> приводит к переходу к очередной строке и помечает следующий оператор `readln` и т. д. Таким образом, каждое нажатие клавиши <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:
   - нажать комбинацию клавиш <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 клавишу <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`, нажать клавиши <Ctrl>+<F7> и <Enter>.

В окне будем иметь:

```
c: 0.0
a: 0
```

- нажать комбинацию клавиш <Ctrl>+<F7>, набрать `b*a` и нажать клавишу <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.

П1.5. Перечень ошибок

В ходе своей работы программист неизбежно сталкивается с ошибками *(см. разд. 1.4)*. 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. П1.5).

Таблица П1.5. *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`

Таблица П1.5 *(продолжение)*

Код	Описание
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
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

Таблица П1.5 *(окончание)*

Код	Описание
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

ПРИЛОЖЕНИЕ 2

Дополнения к модулям Turbo Pascal

П2.1. Дополнения к модулю *CRT*

П2.1.1. Кодовая таблица

Символы с кодами 0—127

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

Символы с кодами 128—255

128 А	144 Р	160 а	176 ▒	192 └	208 ╨	224 р	240 Ё
129 Б	145 С	161 б	177 ▒	193 ┴	209 ╤	225 с	241 ё
130 В	146 Т	162 в	178 ▓	194 ┬	210 ╥	226 т	242 Є
131 Г	147 У	163 г	179 │	195 ├	211 ╙	227 у	243 є
132 Д	148 Ф	164 д	180 ┤	196 —	212 ╘	228 ф	244 Ї
133 Е	149 Х	165 е	181 ╡	197 ┼	213 ╒	229 х	245 ї
134 Ж	150 Ц	166 ж	182 ╢	198 ╞	214 ╓	230 ц	246 Ў
135 З	151 Ч	167 з	183 ╖	199 ╟	215 ╫	231 ч	247 ў
136 И	152 Ш	168 и	184 ╕	200 ╚	216 ╪	232 ш	248 °
137 Й	153 Щ	169 й	185 ╣	201 ╔	217 ┘	233 щ	249 •
138 К	154 Ъ	170 к	186 ║	202 ╩	218 ┌	234 ъ	250 ·
139 Л	155 Ы	171 л	187 ╗	203 ╦	219 █	235 ы	251 √
140 М	156 Ь	172 м	188 ╝	204 ╠	220 ▄	236 ь	252 №
141 Н	157 Э	173 н	189 ╜	205 ═	221 ▌	237 э	253 ¤
142 О	158 Ю	174 о	190 ╛	206 ╬	222 ▐	238 ю	254 ■
143 П	159 Я	175 п	191 ┐	207 ╧	223 ▀	239 я	255

П2.1.2. Модуль *CRTPLUS*

Данный модуль (листинг П2.1) 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. Надеемся, что любознательному читателю они пригодятся.

Листинг П2.1. 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);
        intr($10,regs)
      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.
```

П2.2. Модуль для подключения мыши к программе на Turbo Pascal

К сожалению, в Turbo Pascal нет стандартных средств для работы с компьютерной мышью, к которой все так привыкли. Данный модуль (листинг П2.2) восполняет эту проблему, обращаясь к прерыванию MS-DOS с номером $33.

Листинг П2.2. 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}
                      var 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; {Текущие координаты курсора}
      var left, {Состояние левой кнопки мыши
                 (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 с момента
  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=0 then success:=false
    else success:=true;
    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;
begin
end.
```

П2.3. Дополнение к модулю *GRAPH* — вывод рисунков в формате BMP

Модуль BMP (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.dgi, их легко найти в Интернете, 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 П2.3. 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;            { установка регистров палитры }
  Reg.al := $12;
  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.
```

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.
3. Алкок Д. Язык Паскаль в иллюстрациях. Пер. с англ. — М.: Мир, 1991. — 192 с.
4. Бородич Ю. С. Паскаль для персональных компьютеров: Справ. пособие / Ю. С. Бородич, А. Н. Вальвачев, А. И. Кузьмич. — Мн.: Выш. шк., БФ ГИТМП "НИКА", 1991. — 365 с.
5. Вирт Н. Алгоритмы + структуры данных = программы. Пер. с англ. — М.: Мир, 1985. — 406 с.
6. Вирт Н. Алгоритмы и структуры данных. Пер. с англ. — М.: Мир, 1989. — 360 с.
7. 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.
8. Йенсен К. Руководство для пользователя и описание языка Паскаль. Пер. с англ. / К. Йенсен, Н. Вирт.— М.: Финансы и статистика, 1982. — 150 с.
9. Культин Н. Б. Программирование в Turbo Pascal 7.0 и Delphi. Второе издание, переработанное и дополненное / Н. Б. Культин. — СПб.: БХВ — Санкт-Петербург, 1999. — 416 с.
10. Культин Н. Б. Turbo Pascal в задачах и примерах. — СПб.: БХВ — Санкт-Петербург, 2000. — 256 с.

11. Мизрохи С. В. Turbo Pascal и объектно-ориентированное программирование. — М.: Финансы и статистика, 1992. — 192 с.

12. Немнюгин С. А. Turbo Pascal: практикум. — СПб: Питер, 2001. — 256 с.

13. 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.: Выш. шк., 1993. — 256 с.

14. Перминов О. Н. Программирование на языке Паскаль. — М.: Радио и связь, 1988. — 224 с.

15. Попов В. Б. Turbo Pascal для школьников. Версия 7.0. — М.: Финансы и статистика, 1996. — 446 с.

16. Фаронов В. В. Программирование на персональных ЭВМ в среде Турбо-Паскаль. 2-е изд. — М.: Изд-во МГТУ, 1992. — 448 с.

Предметный указатель

К

Л

М

Н

О

П

Продолжение рубрики см. на с. 348

Р

С

Т

У

Ф

Х

Ц

Э

Я