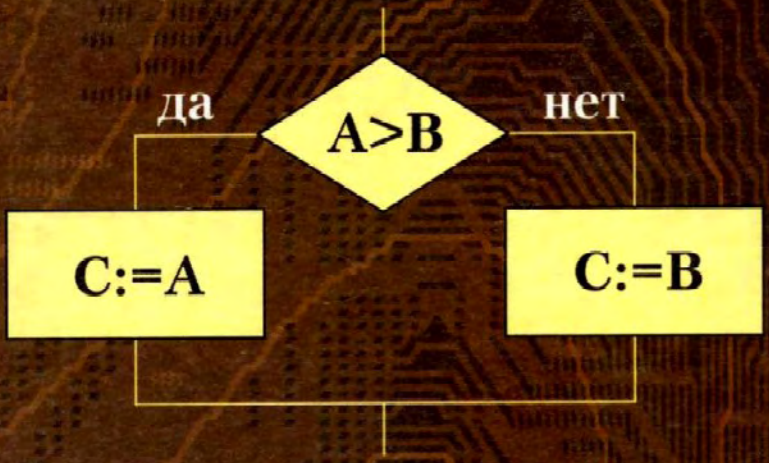
Информатика в техническом университете

Г.С. Иванова

Основы программирования

Издание второе



Издательство МГТУ имени Н.Э. Баумана

Информатика в техническом университете

Информатика в техническом университете

Серия основана в 2000 году

Г.С. Иванова

Основы программирования

Издание второе,
переработанное и дополненное

Допущено Министерством образования
Российской Федерации
в качестве учебника для студентов
высших учебных заведений, обучающихся по направлению
«Информатика и вычислительная техника»,
специальностям: «Вычислительные машины, комплексы,
системы и сети», «Автоматизированные системы обработки
информации и управления», «Программное обеспечение
вычислительной техники и информационных систем»

Москва
Издательство МГТУ имени Н.Э. Баумана
2002

УДК 681.3.06(075.8)
ББК 32.973-018
И201

Рецензенты:
профессор Л.Д. Забродин (Московский государственный инженерно-физический институт); кафедра «ЭВМ, комплексы и сети» Московского государственного авиационного института (зав. кафедрой профессор О.М. Брехов)

Иванова Г.С.

И201 Основы программирования: Учебник для вузов. – 2-е изд., перераб. и доп. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. – 416 с.: ил. (Сер. Информатика в техническом университете.)

ISBN 5-7038-1957-1

Изложены основные теоретические положения разработки программного обеспечения с использованием структурного и объектно-ориентированных подходов. Подробно рассмотрены основные приемы решения задач различных классов, в том числе приемы создания и обработки динамических структур данных, без которых невозможно современное программирование. Особое внимание уделено оценке точности получаемых результатов и анализу вычислительной сложности алгоритмов и методов. Большое количество примеров и поясняющих рисунков помогает лучшему усвоению материала.

Во втором издании (1-е – 2001 г.) для описания объектно-ориентированных программ использован Универсальный язык моделирования (UML). Добавлен материал по разработке приложений в Delphi, проиллюстрированный примерами.

Содержание учебника соответствует курсу лекций, которые автор читает в МГТУ им. Н.Э. Баумана.

Для студентов вузов, обучающихся по специальностям, связанным с информатикой. Может быть полезен всем изучающим программирование самостоятельно.

УДК 681.3.06(075.8)
ББК 32.973-018

ISBN 5-7038-1957-1

Оглавление

ПРЕДИСЛОВИЕ

Преподавание основ программирования в вузах сопряжено с целым рядом проблем. Во-первых, современное программирование – сложная и быстро развивающаяся наука. Если сравнить то, что студент должен знать в этой области сейчас и 20 лет назад, то разница окажется ошеломляющей. В то же время реальные часы, отводимые в программах вузов для изучения основ программирования, практически не изменились. Во-вторых, подготовка студентов, осуществляемая в данной области школой, очень различна: от полного отсутствия каких-либо знаний по предмету до относительно свободного владения каким-либо языком программирования.

Кроме того, программирование – наука, неразрывно связанная с практикой. Невозможно научиться программировать, не проведя много часов за составлением алгоритмов, написанием и отладкой программ. Причем учебно-практическую работу желательно совмещать с процессом изучения методов разработки программ и освоением особенностей конкретного языка программирования. Следовательно, элементы технологии программирования и алгоритмизации должны изучаться параллельно с языком программирования. Таким образом, один курс как бы включает в себя несколько курсов.

Решение перечисленных проблем потребовало тщательного отбора и структуризации материала, включенного в учебник. Данный учебник – результат 20-летнего преподавания программирования в МГТУ им. Н.Э. Баумана. Курс, читаемый автором в настоящее время, построен следующим образом.

Лекционно излагаются основы технологии программирования, сведения, необходимые для решения тех или иных задач, и поясняются конкретные языковые средства. Лекции иллюстрируются большим количеством рисунков и примеров (программ и схем алгоритмов), желательно минимального размера, чтобы конкретные возможности и особенности было легко понять и запомнить.

Семинары посвящаются обсуждению определенных проблем, связанных с решением некоторого класса задач. Как правило, на семинарах анализируются не программы, а алгоритмы или подходы. Например, рассматривается метод пошаговой детализации и его применение для разработки алго-

ритмов, понятие и способы оценки точности полученных результатов, вычислительной и емкостной сложности разрабатываемого программного обеспечения и т. д.

Во время лабораторного практикума студенты самостоятельно под контролем преподавателей разрабатывают программы решения индивидуального набора задач по изучаемым темам. Задание каждому студенту выдается в начале семестра, поэтому он имеет возможность выполнять задания по мере освоения материала, что обеспечивает определенную степень индивидуализации обучения.

Изложение материала курса в учебнике следует той же схеме. Главы содержат необходимые сведения из теории программирования, описание конкретных средств Borland Pascal и особенностей взаимодействия программ с техническими и программными средствами. При этом особое внимание уделено наиболее важным моментам, без рассмотрения которых дальнейшее изучение программирования практически невозможно. Это, в частности, проблемы создания рекурсивных программ, работа с динамическими структурами данных и объектно-ориентированный подход. Материал проблемных семинаров курса выделен в специальные разделы, названные практикумами. В конце большинства разделов приведены вопросы и задачи для самопроверки.

Данная книга представляет собой второе издание учебника. В связи с новой редакцией программ обучения основам программирования в него включены материалы по основам событийного программирования и отличиям Delphi Pascal от Borland Pascal 7.0. Кроме того, изменена графическая нотация, используемая для пояснения основ объектно-ориентированного программирования, что связано с практическим утверждением UML (Unified Modeling Language – Универсальный язык моделирования) в качестве международного стандарта описания объектно-ориентированных разработок.

Автор глубоко признателен канд. техн. наук, доценту Т.Н. Ничушкиной за предоставленные материалы и огромную помощь в подготовке книги, а также рецензентам: заведующему кафедрой «Компьютерные системы и технологии» МИФИ д-ру техн. наук, профессору Л.Д. Забродину и коллективу кафедры «ЭВМ, комплексы и сети» МАИ во главе с д-ром техн. наук, профессором О.М. Бреховым за полезные замечания и советы.

Хочется также выразить особую благодарность студентам, принявшим активное участие в обсуждении первого издания учебника, за их советы и замечания, учтенные автором в данном издании.

ВВЕДЕНИЕ

Язык программирования Паскаль был создан в 1971 г. профессором Цюрихского университета Никлаусом Виртом и предназначался для обучения студентов как основам алгоритмизации и программирования, так и основам конструирования компиляторов. Язык полностью отвечал принципам структурного программирования, сформулированным к тому моменту, имел ярко выраженную блочную структуру и развитое представление данных. Однако, будучи учебным, он имел ограниченные средства реализации ввода-вывода и создания библиотек подпрограмм.

В разные годы было разработано несколько вариантов компиляторов с Паскаля для различных типов ЭВМ. Наибольшее распространение получил Turbo (Borland) Pascal, предложенный фирмой Borland Internation (США). Существовало несколько версий. Последняя версия, предназначенная для создания программного обеспечения «под MS DOS» – версия 7.0, включает:

- интегрированную среду разработки программ, ставшую в некоторой степени прототипом создания аналогичных сред для других языков программирования;
- средства разработки многомодульных программ;
- средства управления экраном в текстовом и графических режимах;
- средства объектно-ориентированного программирования;
- усовершенствованную систему типов данных.

Современным программистам приходится иметь дело с огромным количеством разнообразных языков программирования различных уровней и назначений. Но по-прежнему начинать изучение программирования целесообразно на базе Паскаля, так как при использовании этого языка у будущего программиста быстрее формируется четкое алгоритмическое мышление.

Весомым аргументом в пользу изучения основ программирования именно на базе Паскаля также является существование профессиональной визуальной среды разработки программного обеспечения Delphi, которая использует в качестве базового языка именно Паскаль. Практика показывает, что переход к разработке программного обеспечения в этой среде после изучения базового курса происходит достаточно безболезненно, хотя и требует некоторых дополнительных знаний.

В настоящее время при разработке сложного программного обеспечения обычно используют одну из двух технологий: *структурное* программирование или *объектно-ориентированное* программирование.

Первая технология для разработки сложных программ рекомендует разбивать (*декомпозировать*) программу на подпрограммы (процедуры), решающие отдельные подзадачи, т.е. базируется на *процедурной декомпозиции.*

Вторая технология использует более сложный подход, при котором в предметной области задачи выделяют отдельно функционирующие элементы. Поведение этих объектов программно моделируется с использованием специальных средств, а затем, уже из готовых объектов, опять же специальным способом, собирается сложная программа. Таким образом, в основе второй технологии лежит *объектная декомпозиция.*

Именно объектная технология лежит в основе используемой Delphi библиотеки стандартных компонентов, поэтому переход в эту среду целесообразно осуществлять только после изучения основных принципов объектного подхода, изложенных в данном учебнике.

Кроме объектного подхода для работы в Delphi необходимо иметь представление об основных отличиях Delphi Pascal и визуальных средах, использующих принцип событийного программирования. Этот материал добавлен во второе издание учебника в виде приложений 4 и 5.

Изучение объектной технологии требует наличия базовых знаний, поэтому на первых этапах мы будем придерживаться принципов процедурного программирования.

Часть 1. ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

1. ЭТАПЫ СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В процессе разработки программ с использованием процедурного подхода можно выделить следующие этапы:

- *постановка задачи* – определение требований к программному продукту;
- *анализ* – осуществление формальной постановки задачи и определение методов ее решения;
- *проектирование* – разработка структуры программного продукта, выбор структур для хранения данных, построение и оценка алгоритмов подпрограмм и определение особенностей взаимодействия программы с вычислительной средой (другими программами, операционной системой и техническими средствами);
- *реализация* – составление программы на выбранном языке программирования, ее тестирование и отладка.
- *модификация* – выпуск новых версий программного продукта.

1.1. Постановка задачи

Процесс создания нового программного обеспечения начинают с постановки задачи, в процессе которой определяют требования к программному продукту.

Прежде всего устанавливают набор выполняемых функций, а также перечень и характеристики исходных данных. Так, для числовых данных может задаваться точность, для текстовых – возможно, размер текста, способ кодировки и т. п. Затем определяют перечень результатов, их характеристики и способы представления (в виде таблиц, диаграмм, графиков и т. п.). Кроме того, уточняют среду функционирования программного продукта: конкретную комплектацию и параметры технических средств, версию используемой операционной системы и, возможно, версии и параметры другого установ-

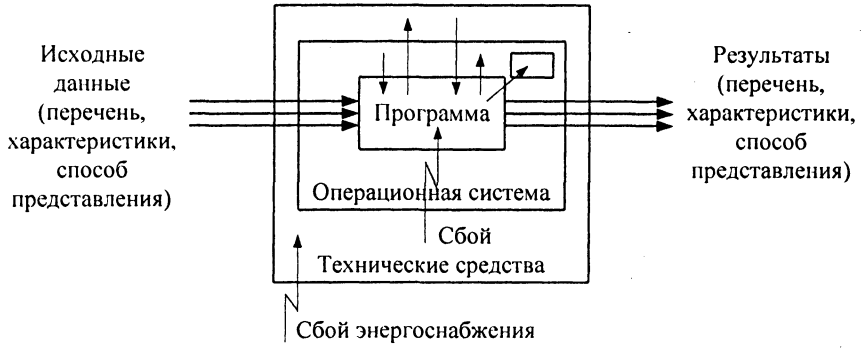


Рис. 1.1. Факторы, определяющие параметры разрабатываемого программного обеспечения

ленного программного обеспечения, с которым предстоит взаимодействовать будущему программному продукту.

В тех случаях, когда разрабатываемое программное обеспечение собирает и хранит некоторую информацию или включается в управление каким-либо техническим процессом, необходимо также четко регламентировать действия программы при сбоях оборудования и энергоснабжения (рис. 1.1).

В результате согласования между заказчиком и исполнителем всех перечисленных вопросов составляют *техническое задание* в соответствии с ГОСТ 19.201–78, которое служит основанием для дальнейшей работы.

1.2. Анализ, формальная постановка и выбор метода решения

На данном этапе по результатам анализа условия задачи выбирают математические абстракции, *адекватно*, т.е. с требуемой точностью и полнотой, представляющие исходные данные и результаты, строят *модель* задачи и определяют метод преобразования исходных данных в результат (*метод решения задачи*).

Пример 1.1. Разработать программу, которая по заданным длинам сторон прямоугольника определяет его площадь.

Исходными данными в этом случае являются длины сторон прямоугольника, т.е. некоторые числовые значения, для которых должны быть заданы диапазон изменения и точность. Математические абстракции для представления исходных данных – некие изменяемые значения – *переменные*. Результат – площадь прямоугольника – также некоторое числовое значение, диапазон возможных значений и точность которого зависят от соответствующих характеристик исходных данных. Математической абстракцией результата также является *переменная*. Модель задачи можно представить в виде:

$$S = a \times b,$$

где S – площадь; a, b – длины сторон.

Результат получают перемножением аргументов.

Однако полученная модель не является полной и, следовательно, адекватной, так как в ней не определены типы используемых переменных (целые или вещественные), что может привести к получению неверных результатов. Например, допустим, что нас интересует площадь с точностью «до сотых», тогда получение результата с точностью «до целых» следует считать ошибкой. Полная модель должна включать также указание типов переменных.

Часто формальная постановка задачи однозначно определяет метод ее решения. В тех случаях, когда задача может быть решена несколькими методами, выбирается один из них с учетом сложности и эффективности его реализации, обеспечиваемой методом точности результата, а также других параметров и характеристик.

При использовании процедурного подхода сложные задачи в процессе анализа разбивают на подзадачи, для каждой из которых может строиться своя модель и выбираться свой метод решения. При этом результаты решения одной подзадачи могут использоваться в качестве исходных данных в другой.

Определив методы решения, следует для некоторых вариантов исходных данных вручную или на калькуляторе подсчитать ожидаемые результаты. Эти данные в дальнейшем будут использованы при тестировании программы. Кроме того, выполнение операций вручную позволяет точно уяснить последовательность действий, что упростит разработку алгоритмов.

Целесообразно также продумать, для каких сочетаний исходных данных результат не существует или не может быть получен данным методом, что тоже необходимо учесть при разработке программы.

1.3. Проектирование

Принято различать логическое и физическое проектирование. *Логическое* проектирование не учитывает особенностей среды, в которой будет выполняться программа (технические и программные средства компьютера). При выполнении *физического* проектирования все эти параметры должны быть учтены.

Логическое проектирование. Логическое проектирование при процедурном подходе предполагает *детальную проработку последовательности действий* будущей программы. Его начинают с определения структуры будущего программного продукта: отдельная программа или программная система, состоящая из нескольких взаимосвязанных программ. Затем переходят к разработке алгоритмов программ.

Алгоритмом называют *формально описанную* последовательность действий, которые необходимо выполнить для получения требуемого результата.

Различают последовательности действий (вычислений) линейной, разветвленной и циклической структуры.

Линейная структура процесса вычислений предполагает, что для получения результата необходимо выполнить некоторые операции в определенной последовательности. Например, для определения площади треугольника по формуле Герона необходимо сначала определить полупериметр треугольника, а затем по формуле его площадь.

Разветвленная структура процесса вычислений предполагает, что конкретная последовательность операций зависит от значений одного или нескольких параметров. Например, если дискриминант квадратного уравнения не отрицателен, то уравнение имеет два корня, а если отрицателен, то действительных корней нет.

Циклическая структура процесса вычислений предполагает, что для получения результата некоторые действия необходимо выполнить несколько раз. Например, для того, чтобы получить таблицу значений функции на заданном интервале изменения аргумента с заданным шагом, необходимо соответствующее количество раз определить следующее значение аргумента и посчитать для него значение функции.

Процессы вычислений циклической структуры в свою очередь можно разделить на три группы:

- циклические процессы, для которых количество повторений известно – *счетные циклы* или *циклы с заданным количеством повторений*;
- циклические процессы, завершающиеся по достижении или нарушении некоторых условий – *итерационные циклы*;
- циклические процессы, из которых возможны два варианта выхода: выход по завершении процесса и досрочный выход по какому-либо дополнительному условию – *поисковые циклы*.

Формальное описание алгоритмов осуществляют с использованием схем алгоритмов и псевдокодов.

На изображение схем алгоритмов существует ГОСТ 19.701–90, согласно которому каждой группе действий ставится в соответствие блок особой формы. Некоторые часто используемые обозначения приведены в табл. 1.1.

При разработке алгоритма каждое действие обозначают соответствующим блоком, показывая их последовательность линиями со стрелками на конце. Для простоты чтения схемы желательно, чтобы линия входила в блок сверху, а выходила снизу. Если линии идут не слева направо и не сверху вниз, то стрелка в конце линии обязательна, в противном случае ее можно не ставить.

В случае, когда схема алгоритма не умещается на листе, используют соединители. При переходе на другой лист или получении управления с друго-

Таблица 1.1

Название блока	Обозначение	Назначение блока
Терминатор	Действие	Начало, завершение программы или подпрограммы
Процесс	Действие	Обработка данных (вычисления, пересылки и т. п.)
Данные	Данные	Операции ввода-вывода
Решение	Условие	Ветвления, выбор, итерационные и поисковые циклы
Подготовка	Действия	Счетные циклы
Граница цикла	Начало Конец	Любые циклы
Предопределенный процесс	Имя	Вызов процедур
Соединитель	Имя	Маркировка разрывов линий
Комментарий	--- Комментарий	Пояснения к операциям

го листа в комментарии указывается номер листа, например «с листа 3» «на лист 1».

В теории программирования доказано, что для записи любого сколь угодно сложного алгоритма достаточно трех базовых структур:

- *следование* – обозначает последовательное выполнение действий (рис. 1.2, *а*);
- *ветвление* – соответствует выбору одного из двух вариантов действий (рис. 1.2, *б*);
- *цикл-пока* – определяет повторение действий, пока не будет нарушено условие, выполнение которого проверяется в начале цикла (рис. 1.2, *в*).

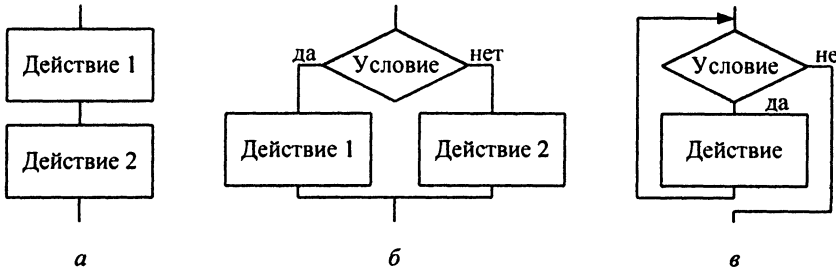


Рис. 1.2. Базовые алгоритмические структуры: следование (*а*), ветвление (*б*) и цикл-пока (*в*)

Помимо базовых структур используют три дополнительные структуры, производные от базовых:

- *выбор* – выбор одного варианта из нескольких в зависимости от значения некоторой величины (рис. 1.3, *а*);
- *цикл-до* – повторение некоторых действий до выполнения заданного условия, проверка которого осуществляется после выполнения действий в цикле (рис. 1.3, *в*);
- *цикл с заданным числом повторений* (*счетный цикл*) – повторение некоторых действий указанное число раз (рис. 1.3, *д*).

На рис. 1.3, *б*, *г* и *е* показано, как каждая из дополнительных структур может быть реализована через базовые структуры.

Перечисленные структуры были положены в основу *структурного программирования* – технологии, которая представляет собой набор рекомендаций по уменьшению количества ошибок в программах [4, 8]. В том случае, если в схеме алгоритма отсутствуют другие варианты передачи управления, алгоритм называют *структурным*, подчеркивая, что он построен с учетом рекомендаций структурного программирования.

Схема алгоритма детально отображает особенности разработанного алгоритма. Иногда такой высокий уровень детализации не позволяет выделить суть алгоритма. В этих случаях для описания алгоритма используют псевдокод.

Псевдокод – описание алгоритма, которое базируется на тех же основных структурах, что и структурные схемы алгоритма. Описать на псевдокоде неструктурный алгоритм нельзя.

Для каждой структуры используют свою форму описания. В литературе были предложены несколько вариантов форм псевдокодов. Один из вариантов приведен в табл. 1.2.

Пример 1.2. Разработать алгоритм определения наибольшего общего делителя двух натуральных чисел.

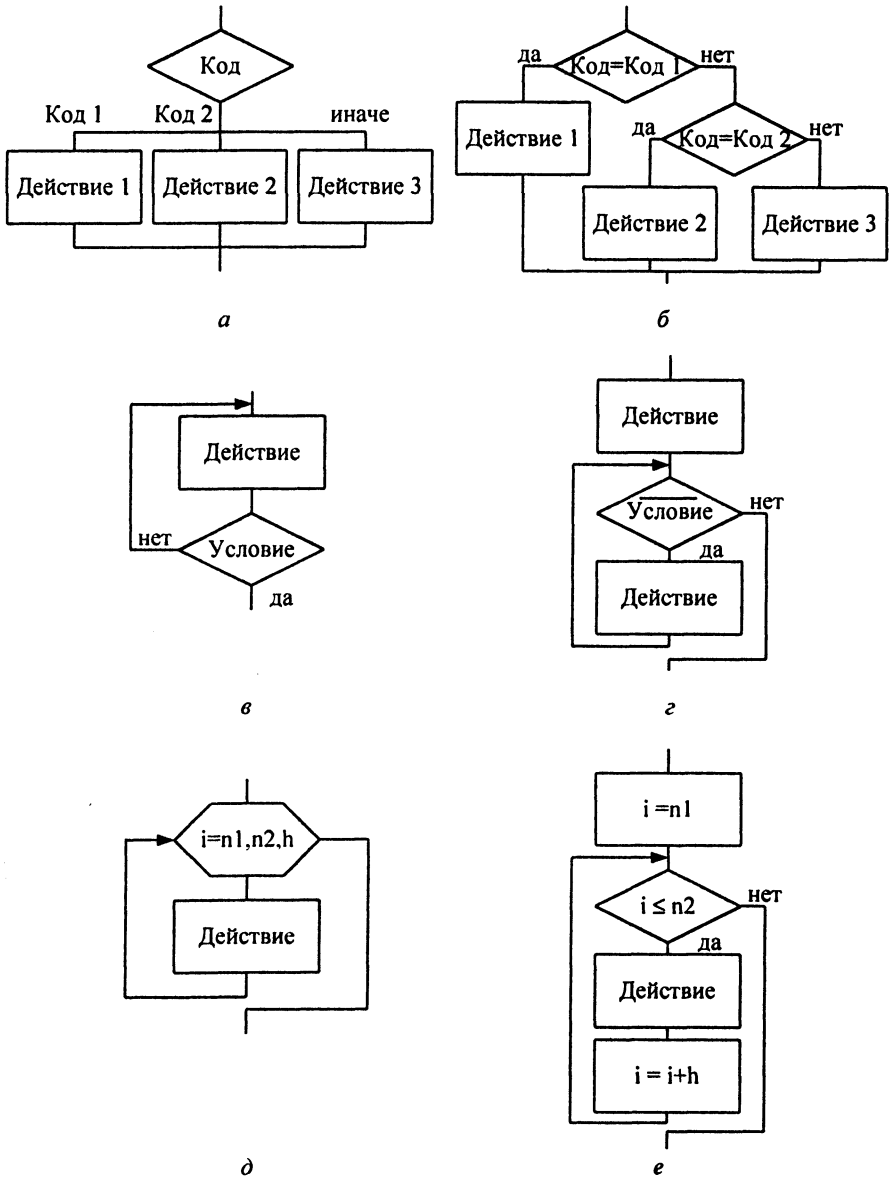


Рис.1.3. Дополнительные структуры и их реализация через базовые структуры: выбор (*а–б*), цикл-до (*в–г*) и цикл с заданным числом повторений (*д–е*)

Существует несколько способов определения наибольшего общего делителя двух натуральных чисел. Самым простым из них является так называемый алгоритм Евклида. Суть этого метода заключается в последователь-

Таблица 1.2

Структура	Псевдокод	Структура	Псевдокод
Следование	<действие 1> <действие 2>	Выбор	**Выбор** <код> <код1>: <действие 1> <код2>: <действие 2> ... **Все-выбор**
Ветвление	**Если** <условие> **то** <действие 1> **иначе** <действие 2> **Все-если**	Цикл с заданным количеством повторений	**Для** <индекс> = <n>,<k>,<h> <действие> **Все-цикл**
Цикл-пока	**Цикл-пока** <условие> <действие> **Все-цикл**	Цикл-до	**Выполнять** <действие> **До** <условие>

ной замене большего из чисел на разность большего и меньшего. Вычисления заканчиваются, когда числа становятся равны. Например:

а)	**A**		**B**	б)	**A**		**B**
	225		125		13		4
	225-125 = 100		125		13-4=9		4
	100		25-100 = 25		9-4=5		4
	100-25 = 75		25		5-4=1		4
	75-25 = 50		25		1		4-1 = 3
	50-25 = **25**	=	**25**		1		3-1 = 2
					1	=	2-1 = **1**

Программа должна начинаться с ввода чисел. Заметим, что любой ввод данных пользователем должен сопровождаться запросом на ввод, чтобы пользователь знал, чего от него ждет программа после запуска. На схеме алгоритма и при записи псевдокодов этот запрос часто не указывают.

В основе алгоритма лежит циклический процесс, количество повторений которого заранее не известно (итерационный). Условие выхода из цикла – получение одинаковых чисел. Поскольку нельзя исключить, что пользователь введет равные числа, проверку будем осуществлять на входе в цикл, т.е. имеет смысл использовать цикл-пока. Если числа не равны, то при каждом проходе цикла одно из чисел (большее) должно заменяться разностью большего и меньшего. Для реализации этой замены потребуется описать оба варианта, т.е. использовать ветвление с проверкой, какое из чисел больше. После выхода из цикла можно выводить пользователю любое из двух полученных чисел, так как они равны между собой.

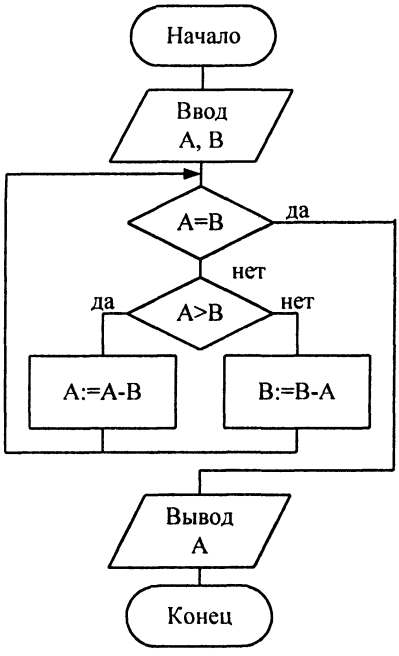


Рис. 1.4. Схема алгоритма Евклида

На рис.1.4 показана схема алгоритма, а ниже приведено его описание на псевдокоде.

Алгоритм Евклида:
Ввести A,B
цикл-пока A ≠ B
если A > B
то A := A - B
иначе B := B - A
все-если
все-цикл
Вывести A
Конец алгоритма.

Алгоритмы простых программ разрабатывают, продумывая последовательность действий для решения некоторой задачи, как это было выполнено в примере. Для разработки алгоритмов более сложных программ целесообразно использовать *метод пошаговой детализации* (см. параграф 1.6). Параллельно с разработкой алгоритма уточняют диапазон изменения, точность и структуры представления переменных, используемых для хранения исходных данных и результатов, а также для временного размещения промежуточных результатов, и описывают их в специальных таблицах.

Физическое проектирование. При выполнении физического проектирования осуществляют *привязку* разрабатываемого программного обеспечения к имеющемуся набору технических и программных средств. Так, если при выполнении логического проектирования определено, что при возникновении некоторой ситуации пользователь должен получить сообщение об ошибке, то при физическом проектировании уточняют, что это сообщение должно быть передано пользователю, например, посредством синтезатора речи.

1.4. Реализация

Разработанные алгоритмы реализуют, составляя по ним текст программы с использованием конкретного языка программирования. Язык может быть определен в техническом задании, а может выбираться исходя из особенностей конкретной разработки.

Рис. 1.5. Схема процесса подготовки программы к выполнению

На рис. 1.5 представлена схема процесса подготовки программы к выполнению.

Вначале осуществляют ввод программы в компьютер. Для ввода используют специальную программу – *текстовый редактор*, с помощью которого создают файл, содержащий текст программы.

Затем программу необходимо перевести в последовательность машинных команд (машинный код). Для этого запускают специальную программу-переводчик – *компилятор*. В процессе разбора и преобразования программы компилятор может обнаружить ошибки. Тогда он аварийно завершает работу, выдав программисту сообщения об *ошибках компиляции*. Для исправления этих ошибок обычно достаточно внимательно изучить соответствующий фрагмент с учетом текста сообщения об ошибке и внести требуемое изменение в программу. После исправления ошибок процесс компиляции повторяют. Если с точки зрения компилятора программа написана правильно, то он строит так называемый *объектный* код, содержащий текст программы на машинном языке. В среде программирования Borland Pascal этот код не переписывается в файл, а сохраняется в памяти до выполнения следующего этапа. В других средах и языках программирования на диске создается объектный файл, как правило, с расширением .obj.

граммы в пошаговом режиме и проверить содержимое интересующих нас переменных (рис. 1.6, *б*).

Современные языки программирования, как правило, имеют так называемые *среды*. Среда языка программирования объединяет специализированный текстовый редактор, компилятор, компоновщик, программу выдачи справочной информации, отладчик и другое программное обеспечение, используемое при разработке программ в единый пакет. Таким образом, среда языка программирования обеспечивает программисту все необходимые средства для реализации программы.

Помимо указанных выше типов ошибок, обнаруживаемых автоматически компилятором, компоновщиком или операционной системой, существует еще группа очень опасных *логических* ошибок. Наличие таких ошибок в программе приводит к выдаче неправильных результатов. Для их обнаружения параллельно с отладкой программы осуществляют ее тестирование.

Тестированием называют процесс выполнения программы при различных тестовых наборах данных с целью *обнаружения ошибок*. Правильный подбор тестовых данных – отдельная и достаточно сложная задача. Некоторые аспекты тестирования программ будут обсуждаться в параграфе 3.2.

Для поиска логических ошибок также можно использовать отладчик: по шагам отследить процесс получения результата. Однако полезно бывает выполнить программу вручную, фиксируя результаты выполнения команд на бумаге. При этом очень поможет пример расчета, выполненный вручную на этапе анализа и выбора методов.

Параллельно с процессом разработки программного продукта на всех этапах должно выполняться составление документации как для выполнения следующего этапа, так и для последующего сопровождения и модификации. Кроме того, важной составляющей этапа реализации является создание необходимой документации для пользователей.

1.5. Модификация

Первоначально этап модификации не включался в процесс разработки программного обеспечения, но практика показала, что в большинстве случаев разработанное программное обеспечение через некоторое время обновляется, и, следовательно, в процессе разработки программного продукта необходимо учитывать возможность его модификации.

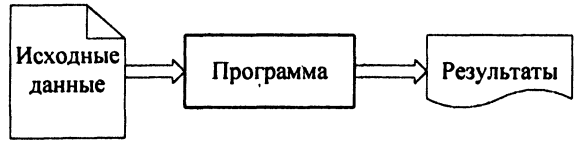
Причинами выпуска новых версий являются:

- необходимость исправления ошибок, выявленных в процессе длительной эксплуатации;
- необходимость совершенствования, например, улучшения интерфейса или расширения состава выполняемых функций;
- изменение среды (появление новых технических средств и/или программных продуктов).

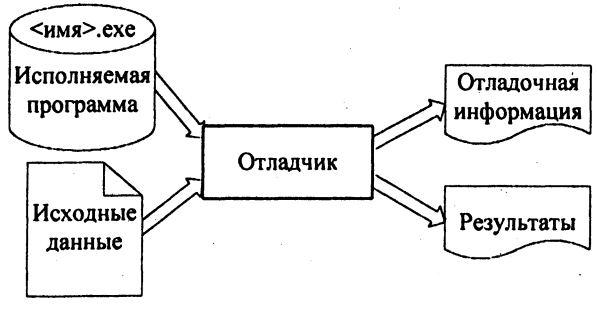
Обычно программа состоит из нескольких частей, каждая из которых компилируется отдельно. Для объединения нескольких фрагментов в единую программу используют специальную программу – *компоновщик*. В процессе связывания также могут быть зафиксированы ошибки, которые называют *ошибками компоновки*. Для исправления таких ошибок, как правило, необходимо сверить заголовки используемых подпрограмм и обращения к ним. Исправив обнаруженные ошибки, вновь запускают компилятор и компоновщик. В результате компоновки получается готовая к выполнению программа, которую при желании можно сохранить в файле с расширением .exe.

В процессе выполнения программа запрашивает, если это предусмотрено программистом, ввод исходных данных, осуществляет требуемую обработку и производит вывод результатов (рис. 1.6, *а*). При этом могут быть обнаружены ситуации, когда продолжение работы программы теряет смысл, например, обнаружено «деление на нуль» или попытка открыть не существующий файл для чтения из него и т. п. Такие ошибки называют *ошибками выполнения*. Для исправления этих ошибок может потребоваться их *локализация*, т.е. уточнение, при выполнении какого фрагмента программы зафиксировано нарушение нормального вычислительного процесса.

Процесс локализации и исправления ошибок получил название *отладки* программы. При отладке программы часто используют специальные программы – *отладчики*, которые позволяют выполнить любой фрагмент про-



а



б

Рис. 1.6. Процесс выполнения программы (*а*) и ее отладки с помощью отладчика (*б*)

На этом этапе, используя проектную документацию, в программный продукт вносят необходимые изменения, которые могут потребовать пересмотра проектных решений, принятых на предшествующих этапах.

1.6. Практикум. Разработка алгоритмов методом пошаговой детализации

Создание программы – процесс сложный, поэтому практически с любого этапа возможен возврат на предыдущие этапы для исправления ошибок или принятия других проектных решений. Чаще всего такого рода возвраты являются следствием ошибок, допущенных при логическом проектировании программы. Поэтому в процессе программирования необходимо особое внимание уделять разработке алгоритмов.

Для разработки алгоритмов программ часто используют метод пошаговой детализации [4, 9]. С использованием данного метода разработку алгоритмов выполняют поэтапно. На первом этапе описывают решение поставленной задачи, выделяя подзадачи и считая их решенными. На следующем – аналогично описывают решение подзадач, формулируя уже подзадачи следующего уровня. Процесс продолжают до тех пор, пока не дойдут до подзадач, алгоритмы решения которых очевидны. При этом, описывая решение каждой задачи, желательно использовать не более одной-двух конструкций, таких как цикл или ветвление, чтобы четче представлять структуру программы.

Пример 1.3. Разработать программу, которая с заданной точностью ε находит значение аргумента x по заданному значению функции y при известном значении n

$$y = \frac{(x+1)^n - 1}{x},$$

где $n > 1$, $x > 0$.

При $n > 1$ данная функция является монотонно возрастающей. Для нахождения значения x можно применить метод половинного деления. Суть данного метода заключается в следующем. Вначале определяют отрезок $[x_1, x_2]$ такой, что $f(x_1) \le y \le f(x_2)$. Затем делят его пополам $x_t = (x_1+x_2)/2$ и определяют, в какой половине отрезка находится x, для чего сравнивают $f(x_t)$ и y. Полученный отрезок опять делят пополам и так до тех пор, пока разность x_1 и x_2 не станет меньше заданного значения ε.

Для разработки алгоритма программы используем метод пошаговой детализации.

Шаг 1. Определяем общую структуру программы.

Программа:
Ввести y, n, eps.
Определить x.
Вывести x, y.
Конец.

Шаг 2. Детализируем операцию определения x.

Определить x:
Определить x1 такое, что $f(x1) \le y$.
Определить x2 такое, что $f(x2) \ge y$.
Определить x на интервале [x1, x2].
Все.

Шаг 3. Детализируем операцию определения x1. Значение x1 должно быть подобрано так, чтобы выполнялось условие $f(x1) \le y$. Известно, что $x > 0$, следовательно, можно взять некоторое значение x, например, x1=1, и последовательно уменьшая его, например в два раза, определить значение x1, удовлетворяющее данному условию.

```
Определить x1:
    x1:=1
    цикл-пока f(x1) > y
        x1:=x1/2
    все-цикл
Все.
```

Шаг 4. Детализируем операцию определения x2. Значение x2 определяем аналогично x1, но исходное значение будем увеличивать в два раза.

```
Определить x2:
    x2:=1
    цикл-пока f(x2) < y
        x2:=x2*2
    все-цикл
Все.
```

Шаг 5. Детализируем операцию определения x. Определение x выполняется последовательным сокращением отрезка [x1, x2].

```
Определить x:
    цикл-пока x2-x1>eps
        Сократить отрезок [x1, x2].
    все-цикл
Все.
```

Шаг 6. Детализируем операцию сокращения интервала определения x. Сокращение отрезка достигается делением пополам и отбрасыванием половины, не удовлетворяющей условию $f(x1) \le y \le f(x2)$

```
Сократить интервал определения x:
    xt := (x1 + x2) / 2
    если f(xt) > y
       то     x2 := xt
       иначе x1 := xt
    все-если
Все.
```

Таким образом, за шесть шагов мы разработали весь алгоритм, который выглядит следующим образом.

```
Программа:
    Ввести y, n, eps.
    x1:=1
    цикл-пока f(x1) > y
       x1 := x1 / 2
    все-цикл
    x2:=1
    цикл-пока f(x2) < y
       x2 := x2/2
    все-цикл
    цикл-пока x2-x1 > eps
       xt := (x1 + x2) / 2
       если f(xt) > y
          то     x2 := xt
          иначе x1 := xt
       все-если
    все-цикл
    Вывести xt, y.
Конец.
```

Таким образом, на каждом шаге решается одна достаточно простая задача, что существенно облегчает разработку алгоритма и является основным достоинством метода пошаговой детализации.

При разработке алгоритма методом пошаговой детализации мы использовали псевдокод, но можно было использовать и схемы алгоритма, в которых решение каждой подзадачи обозначено блоком «предопределенный процесс».

Задания для самопроверки

Задание 1. Разработайте алгоритм программы, определяющей первые 10 чисел последовательности Фибоначчи, которая формируется следующим образом:

$$F_1 = F_2 = 1, F_n = F_{n-1} + F_{n-2},$$

где $n > 2$. Алгоритм представьте в виде схемы и запишите псевдокодом.

Задание 2. Разработайте алгоритм программы, которая определяет квадратный корень из числа A с точностью до целой части, учитывая, что сумма первых n нечетных натуральных чисел равна n^2:

$1 = 1^2$
$1 + 3 = 4 = 2^2$
$1 + 3 + 5 = 9 = 3^2$
$1 + 3 + 5 + 7 = 16 = 4^2$ и т. д.

Алгоритм представьте в виде схемы и запишите псевдокодом.

2. ПРОСТЕЙШИЕ КОНСТРУКЦИИ ЯЗЫКА

К простейшим конструкциям языка относятся способы представления скалярных данных, конструкции выражений, оператор присваивания и операторы ввода-вывода, без которых не обходится ни одна программа. Однако прежде чем рассматривать эти конструкции, выясним, что собой представляет язык программирования и каким образом выполняется его описание.

2.1. Синтаксис и семантика языка программирования

Любой язык, в том числе и язык программирования, подчиняется ряду правил. Их принято разделять на правила, определяющие синтаксис языка, и правила, определяющие его семантику.

Синтаксис языка – совокупность правил, определяющих допустимые конструкции (слова, предложения) языка, его *форму*.

Семантика языка – совокупность правил, определяющих смысл синтаксически корректных конструкций языка, его *содержание*.

Языки программирования относятся к группе формальных языков, для которых в отличие от естественных языков однозначно определены синтаксис и семантика. Описание синтаксиса языка включает определение алфавита и правил построения различных конструкций языка из символов алфавита и более простых конструкций. Для этого обычно используют *форму Бэкуса-Наура* (БНФ) или *синтаксические диаграммы*. Описание конструкции в БНФ состоит из символов алфавита языка, названий более простых конструкций и двух специальных знаков:

«::=» – читается как «может быть заменено на»,

«|» – читается как «или».

При этом символы алфавита языка, которые часто называют *терминальными символами* или *терминалами*, записывают в неизменном виде. Названия конструкций языка (*нетерминальные символы* или *нетерминалы*), определяемых через некоторые другие символы, при записи заключают в угловые скобки («<», «>»).

Например, правила построения конструкции <Целое>, записанные в БНФ, могут выглядеть следующим образом:

<Целое> ::= <Знак> <Целое без знака> | <Целое без знака>

<Целое без знака> ::= <Целое без знака> <Цифра> | <Цифра>

<Цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Знак> ::= + | -

Для отображения того, что конструкция <Целое без знака> может включать неограниченное количество цифр, использовано правило с левосторонней рекурсией. Многократное применение этого правила позволяет построить целое число с любым количеством цифр.

Синтаксические диаграммы отображают правила построения конструкций в более наглядной форме. На такой диаграмме символы алфавита изображают блоками в овальных рамках, названия конструкций – в прямоугольных, а правила построения конструкций – в виде линий со стрелками на концах. При этом, если линия входит в блок, то в описываемую конструкцию должен входить соответствующий символ. Разветвление линии означает, что при построении конструкции возможны варианты.

На рис. 2.1 представлена синтаксическая диаграмма, иллюстрирующая первые два правила описания конструкции <Целое>. Из диаграммы видно, что целое число может быть записано со знаком или без и включать произвольное количество цифр.

Для описания синтаксических конструкций своего языка Н. Вирт использовал именно синтаксические диаграммы, поэтому в тех случаях, когда словесное описание синтаксиса конструкции длинно и нечетко, мы будем использовать синтаксические диаграммы.

Алфавит языка программирования Borland Pascal 7.0 включает:

- строчные, прописные буквы латинского алфавита (a..z, A..Z) и знак подчеркивания (_), который также во многих случаях считается буквой; кроме того, существенно то, что *строчные и прописные буквы не различаются*: *a* неотличимо от *A*, *b* – от *B* и т. д.;
- цифры (0...9);
- специальные знаки, состоящие из одного и двух символов:

. , + - * / = : < > [ ]

{ } ( ) ^ @ $ # <>

<= >= := (* *);

- служебные слова (эти сочетания считаются единым целым и их нельзя использовать в программе в другом качестве):

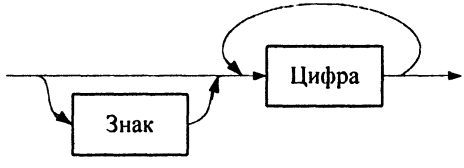


Рис. 2.1. Синтаксическая диаграмма конструкции <Целое>

absolute	*end*	*inline*	*procedure*	*to*
and	*external*	*interface*	*program*	*type*
array	*file*	*interrupt*	*public*	*unit*
begin	*for*	*label*	*record*	*until*
case	*forward*	*mod*	*repeat*	*uses*
const	*function*	*nil*	*set*	*var*
div	*goto*	*not*	*shl*	*while*
do	*if*	*of*	*shr*	*with*
downto	*implementation*	*or*	*string*	*xor*
else	*in*	*private*	*then*	

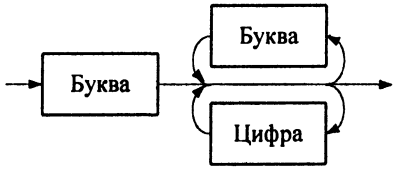


Рис. 2.2. Синтаксическая диаграмма <Идентификатор>

Примечание. Обратите внимание, что русские буквы в конструкциях языка использовать нельзя. Они допускаются только при определении строковых и символьных данных.

Из символов алфавита в соответствии с правилами синтаксиса строят различные конструкции. Простейшей из них является конструкция <Идентификатор>. Эта конструкция используется во многих более сложных конструкциях для обозначения имен программных объектов (полей данных, процедур, функций и т. п.). В Borland Pascal идентификатор представляет собой последовательность букв латинского алфавита (включая символ подчеркивания) и цифр, которая обязательно начинается с буквы, например: aaaa, b121, Parametr_1, _a и т. п. Синтаксическая диаграмма идентификатора приведена на рис. 2.2. Остальные конструкции будут рассмотрены в последующих разделах.

Семантику языка программирования закладывают в его компилятор. Таким образом, синтаксически корректная программа, написанная на языке программирования, после преобразования ее в последовательность машинных команд обеспечит выполнение компьютером требуемых операций.

2.2. Структура программы

Программа на Borland Pascal состоит из трех частей: заголовка, раздела описаний и раздела операторов.

Заголовок программы не является обязательным, он состоит из служебного слова program и идентификатора – имени программы.

Раздел описаний содержит описания всех используемых программой *ресурсов* (полей данных, подпрограмм и т.д.).

Раздел операторов заключается в так называемые *операторные скобки* begin ...end и заканчивается точкой. Между операторными скоб-

ками записывают управляющие операторы программы, которые разделяют специальным знаком – точкой с запятой «;». Если точка с запятой стоит перед end, то считается, что после точки с запятой стоит «пустой» оператор.

В тексте программы возможны комментарии, которые помещают в фигурные скобки.

Посмотрим, как выглядит на Borland Pascal программа, которая реализует алгоритм Евклида для определения наибольшего общего делителя двух натуральных чисел, разработанный в примере 1.2:

```
Program example;      {заголовок программы}
{раздел описаний}
 Var a,b:integer;     {объявление переменных}
{раздел операторов}
 Begin
  Write('Введите два натуральных числа:');  {запрашиваем ввод
                                              данных}
  Readln(a,b);        {вводим значения}
  while a<>b do              {цикл-пока a ≠ b}
    if a>b then  a:=a-b      {если a>b, тогда a:=a-b}
         else  b:=b-a;                {иначе b:=b-a}
  Writeln('Наибольший общий делитель равен ',a); {выводим результат}
 End.                 {конец программы}
```

Программа названа «example». Раздел описаний в данном случае включает только описание переменных (см. параграф 2.3). Раздел операторов содержит операторы ввода исходных данных, вычислений и вывода результатов.

Начнем рассмотрение особенностей программирования на языке Borland Pascal с проблемы описания данных.

2.3. Константы и переменные. Типы переменных

Любая программа оперирует с некоторыми данными, используемыми в расчетах или определяющими последовательность выполнения действий. Все данные, с которыми оперирует программа на Borland Pascal, должны быть описаны.

Данные в программе могут присутствовать в виде констант и переменных.

Константы. Константы определяются один раз и не изменяются во время выполнения программы.

Используют следующие типы констант:

- целые и вещественные десятичные числа, например, 25, 6.12, 0.125e10 (см. примечание);
- шестнадцатеричные числа – должны начинаться со знака «$», например, $64;
- логические константы – true (истина) и false (ложь);
- символьные константы – записываются либо в апострофах, например 'A' , либо в виде соответствующих кодов по таблице ASCII (русский вариант таблицы символов см. в приложении 2), причем в последнем случае перед кодом ставится знак «#», например #65 (этот код соответствует символу A латинское);
- строки символов – записываются в апострофах, например 'ABCD' (см. параграф 4.5);
- конструкторы множеств (см. параграф 4.7);
- «нулевой» адрес – nil (см. параграф 7.1).

Примечания. 1. В программировании принято при записи вещественных чисел вместо запятой для разделения целой и дробной частей числа использовать точку.

2. Обычно при записи в программе или выполнении операций ввода-вывода вещественные числа записывают в так называемом *формате с фиксированной точкой*, указывая в начале целую часть числа, а затем, после точки, дробную, например: 0.5, -3.85 . Но иногда бывает удобно задавать числа в *формате с плавающей точкой*, т.е. в виде мантиссы и порядка. При этом мантиссу записывают перед порядком и отделяют от него строчной или прописной латинской буквой «e», например: запись 1.5e-10 соответствует значению $1{,}5 \times 10^{-10}$, а запись 0.5E23 соответствует значению $0{,}5 \times 10^{23}$.

Константы используются в двух формах: как литералы и как поименованные константы.

Литерал представляет собой значение константы, записанное непосредственно в программе (например, в выражении 2+5.1*x использованы два литерала «2» и «5.1»).

Поименованные константы объявляются в инструкции раздела описаний const. Обращение к ним осуществляется по имени (идентификатору). Под выражением при этом (рис. 2.3) понимают запись, состоящую из ранее

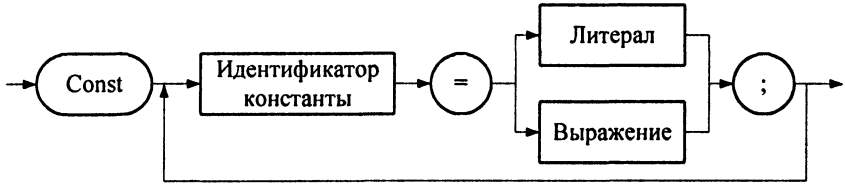


Рис. 2.3. Синтаксическая диаграмма конструкции <Объявление констант>

объявленных констант, литералов, знаков операций (см. параграф 2.4) и стандартных функций abs, chr, hi, length, lo, ord, odd, pred, round, sizeof, str, succ, trunc (см. приложение 1).

Например:

```
Const min=-23;  max=45;   {десятичные константы}
   a16=$10;               {шестнадцатеричная константа}
   ch1=#94; ch2='a';      {символьные константы}
   stroka= 'end';         {строковая константа}
   v1=[3,6,8..9];         {конструктор множества}
   center=(max-min) div 2; {выражение}
```

Переменные. Переменные – поименованные значения, которые могут изменяться в процессе выполнения программы. Их объявление также выполняют в разделе описаний программы, причем при этом указывается не только идентификатор переменной, но и ее тип (рис. 2.4). Обращение к переменным также осуществляют по идентификатору.

Тип переменной определяет возможный набор значений данной переменной, размер ее внутреннего представления и множество операций, которые могут выполняться над переменной.

На рис. 2.5 показана классификация типов переменных Borland Pascal. В соответствии с ней различают простые и структурные типы переменных.

Простые (скалярные) типы описывают упорядоченные наборы значений. Они делятся на порядковые и вещественные.

Группа *порядковых* типов объединяет типы переменных, набор значений которых конечен, группа *вещественных* типов – типы с условно бесконечным набором значений.

Порядковые типы переменных делятся на стандартные, перечисляемые и отрезки. *Стандартно* заданы следующие типы:

- целые типы – см. табл. 2.1;
- булевский тип Boolean включает только два значения – false (0) и true (1), но в памяти значения данного типа занимают целый байт;
- символьный тип Char определяет набор символов по таблице ASCII (см. приложение 2). Всего в таблице указано 255 кодов, для большинства из

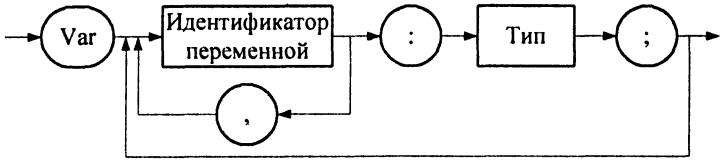


Рис. 2.4. Синтаксическая диаграмма конструкции <Объявление переменных>

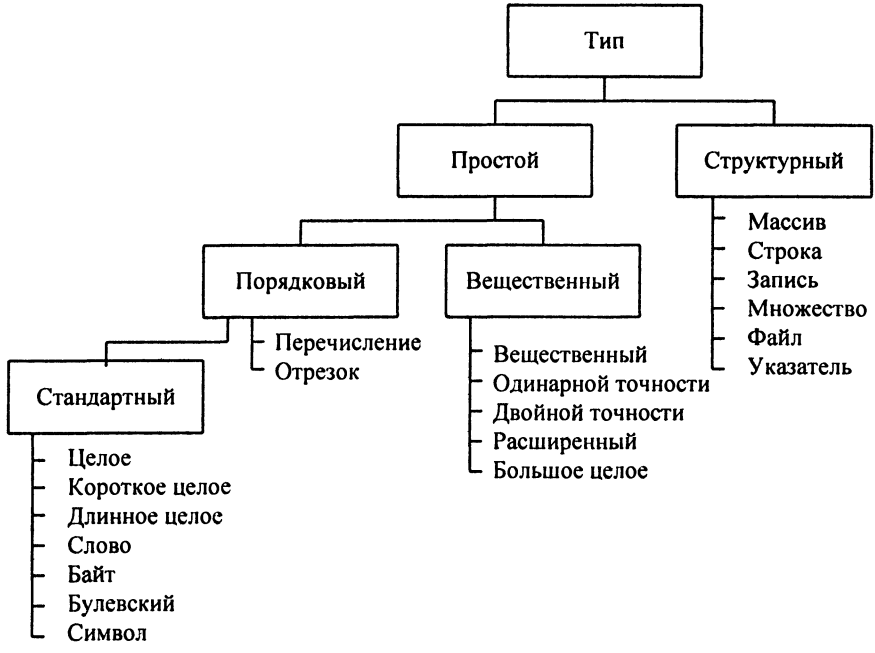


Рис. 2.5. Классификация типов

которых существует символьное представление. Это, например, буквы русского и латинского алфавитов, цифры и специальные знаки, такие как точка, запятая и т. п.

Нестандартные порядковые типы необходимо описывать при объявлении переменных или используя инструкцию объявления типа (рис. 2.6).

Таблица 2.1

Название	Обозначение	Диапазон значений	Длина внутреннего представления, байт
Целое	*Integer*	-32768..32767	2 (со знаком)
Короткое целое	*ShortInt*	-128..127	1 (со знаком)
Длинное целое	*LongInt*	$-2^{31}..2^{31}-1$	4 (со знаком)
Байт	*Byte*	0..255	1 (без знака)
Слово	*Word*	0..65535	2 (без знака)

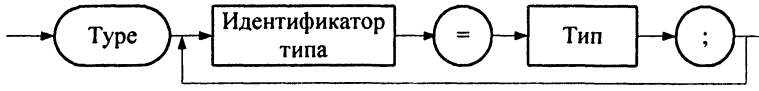


Рис. 2.6. Синтаксическая диаграмма конструкции <Объявление типа>

Перечисляемый тип формируется из значений, определенных программистом при объявлении типа. Перечень значений задают через запятую в круглых скобках, например:

Var D:(Mon,The,Wed,Thu,Fri,Set,Sun); ...{переменная D может принимать только указанные значения}

Примечание. Во внутреннем представлении значения перечисляемого типа кодируются целыми числами, начиная с нуля. Так, идентификатору Mon будет соответствовать 0, The – 1 и т. д.

Объявляя переменную перечисляемого типа, можно сначала определить новый тип, а затем уже переменную этого типа, например:

Type Day=(Mon,The,Wed,Thu,Fri,Set,Sun); {объявление нового типа}
Var D:Day;... {объявление переменной данного типа}

Тип переменной *отрезок* определяется как диапазон значений некоторого уже определенного типа. При его описании также можно использовать конструкцию объявления типа, например:

Type Data=1..31; {диапазон одного из целых типов}
Var DataN:Data;...

или, не описывая тип отдельно, ту же переменную можно объявить следующим образом:

Var DataN:1..31; ...

Вещественные типы используют для представления чисел, содержащих дробную часть. Во внутреннем представлении мантисса и порядок вещественных чисел хранятся раздельно, причем количество разрядов под мантиссу и порядок регламентируются типом числа. Соответственно *обработка вещественных чисел в компьютерах выполняется с некоторой конечной точностью*, которая зависит от количества двоичных разрядов, отведенных для размещения мантиссы. Количество разрядов для записи порядка числа определяет диапазон чисел, для представления которых можно использовать разрядную сетку данного типа. В табл. 2.2 приведены характеристики вещественных типов Borland Pascal.

Таблица 2.2

Название	Обозначение	Количество десятичных значащих цифр	Диапазон изменения порядка	Длина внутреннего представления, байт
Вещественный	*Real*	11...12	-39...+38	6
Одинарной точности	*Single*	7...8	-45...+38	4
Двойной точности	*Double*	15...16	-324...+308	8
Расширенный	*Extended*	19...20	-4951...4932	10
«Большое целое»	*Comp*	19...20	$-2^{63}+1...\ 2^{63}-1$	8

Примечание. Следует иметь в виду, что:

- работа со всеми вещественными типами, кроме Real, требует установки особого режима компиляции (указания директивы {$N+} или соответствующей опции компилятора);
- для типа Real используется самая медленная арифметика.

Структурные типы данных будут рассмотрены в соответствующих разделах.

Инициализированные переменные. В Borland Pascal имеется возможность объявления переменных с заданными начальными значениями. Такие переменные называют *инициализированными* и объявляют в специальной конструкции const (рис. 2.7).

Примечание. С точки зрения идеологии языка объявление инициализированных *переменных* в конструкции *const* является не корректным. В последующих версиях языка эта некорректность была исправлена.

Инициализированные переменные в программе можно изменять так же, как и обычные, например:

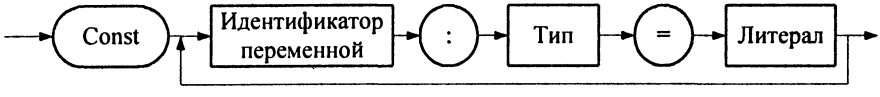


Рис. 2.7. Синтаксическая диаграмма конструкции <Объявление инициализированных переменных>

```
Const a:real=5.6;...
      a:=(n-1)/k;...
```

Наложенные переменные. Иногда возникает необходимость объявления переменных, размещенных по конкретным физическим адресам памяти или в том же месте, что и другие переменные программы. Наложение переменных выполняют также с использованием конструкции var, но после типа указывают зарезервированное слово absolute. На рис. 2.8 представлена полная синтаксическая диаграмма данной конструкции, из которой следует, что возможны два варианта наложения переменной.

1. *Наложение по абсолютному адресу.* В этом случае за словом absolute следует пара чисел типа word, разделенных двоеточием. Первое число трактуется как адрес сегмента, а второе как смещение (см. параграф 7.1). Такое объявление соответствует физическому связыванию переменной и области памяти по указанному физическому адресу.

Например:

```
Var  A: word absolute $0000:$00FF;
     L:array[1..2] of char absolute 128:0; ...
```

Данный вариант применяют, например, для обращения к таблицам операционной системы.

2. *Наложение на ранее определенную переменную.* В этом случае за словом absolute размещают идентификатор ранее определенной переменной. При этом переменной, описанной с absolute, присваивается адрес переменной, идентификатор которой стоит после него. Таким образом, происходит совмещение в памяти данных с разными именами и, возможно, типами. Например:

```
Var   c:byte;
      a:real absolute c;...
```

Вследствие наложения любое изменение одной переменной отражается на значении другой. При несовпадении размеров областей внутреннего пред-

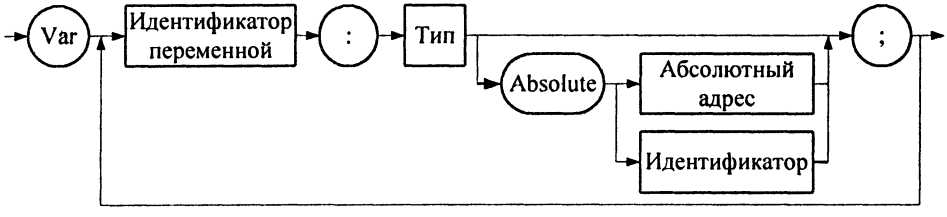


Рис. 2.8. Полная синтаксическая диаграмма конструкции <Объявление переменной>

ставления переменных, связанных по absolute, как в примере выше, корректность полученных результатов не контролируется. Пример использования такого варианта наложения рассмотрен в параграфе 5.5.

2.4. Выражения

Все вычисления и другие преобразования данных в программе записываются в виде *выражений*. Обычно выражение включает несколько операций, которые выполняются в порядке их приоритетности. Различают:

- *арифметические* операции: + (сложение), - (вычитание), * (умножение), / (деление вещественное), **div** (деление целочисленное), **mod** (остаток целочисленного деления) – эти операции применяют к вещественным и целым числам, результат – также число;
- операции *отношения*: > (больше), < (меньше), = (равно),<> (не равно), >= (не меньше), <= (не больше) – эти операции применяют к числам, символам, символьным строкам и некоторым другим типам данных, результат – значение логического типа;
- *логические* операции: **and** (и), **or** (или), **xor** (исключающее или), **not** (не) – эти операции выполняют с логическими переменными и константами, результат – значение логического типа;
- *поразрядные* операции: **and** (и), **or** (или), **xor** (исключающее или), **not** (не), **shr** (сдвиг вправо), **shl** (сдвиг влево) – эти операции выполняют с целыми числами, результат – целое число;
- *строковая* операция: + (сцепление строк) – выполняется над символами и строками, результат – строка (см. параграф 4.5);
- операции *над множествами*: + (объединение), - (дополнение), * (пересечение), результат – множество; **in** (определение принадлежности элемента множеству), результат – значение логического типа (см. параграф 4.7);
- операция *над указателями*: @ (определение адреса программного объекта), результат – адрес (см. параграф 7.1).

Таблица 2.3

Операции	Приоритет
@, not	1
*, /, div, mod, and, shr, shl	2
+, -, or, xor	3
>, <, <>, =, <=, >=, in	4

В табл. 2.3 приведены приоритеты, присвоенные этим операциям.

Для изменения порядка выполнения операций в выражении используют круглые скобки. В выражениях также допускается использование стандартных (см. приложение 1) и определенных программистом функций (см. главу 5). Им присваивается высший приоритет.

Арифметические операции. Запись выражений, содержащих арифметические операции, выполняется «в строку», порядок выполнения операций

определяется скобками. Особенно внимательно следует программировать выражения, включающие операции различных приоритетов. Например:

1) запись a+b/c предполагает, что вначале выполняется операция деления, а затем сложения;

2) запись (a+b)/c*d предполагает, что сумма a+b делится на c, а затем умножается (!) на d.

При программировании арифметических выражений также следует учитывать правила выполнения операций, перечисленные ниже.

1. Операции «целочисленное деление» и «определение остатка от деления» применимы только к операндам целых типов, например: 6 div 4 =1, а 6 mod 4 = 2. Если в операции участвуют переменные, то они должны быть объявлены как целые, например:

```
Var i, n:integer; ...
    n mod 2; ...
```

Для получения при делении целых значений результата с точностью до дробной части необходимо использовать операцию вещественного деления: 6 / 4 = 1.5.

2. При выполнении арифметических операций над числами различных типов выполняется *неявное преобразование типов*:

а) если один операнд целого типа, а другой – вещественного, то переменная целого типа преобразуется к вещественному типу; результат операции – значение вещественного типа;

б) если в качестве операндов использованы вещественные или целые переменные различных типов, то их значения преобразуются к типу с наибольшей разрядной сеткой; результат операции того же типа. Так, если в выражении есть переменные double, extended и real, то значения будут преобразованы в тип extended и того же типа будет полученный результат.

Операции отношения. Операции отношения определены для вещественных и целых чисел, логических значений, кодов символов, строк и множеств. Результат этих операций – значение логического типа, true, если отношение истинно, и false – в противном случае.

Следует помнить, что из-за ограниченной разрядной сетки вещественные числа представляются в памяти не точно, и, соответственно, *проверка равенства или неравенства вещественных чисел должна выполняться с некоторым допуском*, например:

```
(x-y)>1e-10     {вместо x<>y}
(x-y)<1e-10     {вместо x=y}
```

Если такой допуск не указан, то он определяется автоматически, исходя из количества значащих цифр в представлении числа (см. табл. 2.2), и может

оказаться слишком строгим для задачи с неточными данными или методами решения (более подробно см. параграф 2.7).

Логические операции. Логические операции выполняют над значениями типа boolean. Если в логических операциях в качестве операндов используют результаты операций отношения, которые имеют более низкий приоритет, то необходимы скобки. Например, логическое выражение, которое должно быть истинно, если значение x попадает в интервал [a, b], должно быть записано следующим образом:

(x>=a) and (x<=b).

Поразрядные логические операции и операции сдвигов. Поразрядные логические операции и операции сдвигов выполняются над целыми числами. Полученный результат – число того же типа. Второй операнд операций сдвига определяет количество двоичных разрядов, на которое необходимо сдвинуть первый операнд:

5 shl 4 – число 5 в своем внутреннем двоичном представлении сдвигается влево на 4 двоичных разряда, что соответствует умножению числа на $2^4 = 16$. Следовательно, результат данного выражения – число 80.

Остальные типы операций рассмотрены в соответствующих разделах.

2.5. Оператор присваивания

С помощью оператора присваивания в программе записываются действия, связанные с изменением значений переменных (рис. 2.9). При выполнении этого оператора вычисляется выражение, приведенное в правой части, и его результат заносится в переменную, имя которой указано слева. Если оператор присваивания записывается в последовательности операторов, то после него ставится точка с запятой.

Например:

а)
```
Var a,b,c:real;
Begin ...
      c:=(a*a-sin(b))/(a+25.1); ...
```

б)
```
Var v:boolean; a:integer; b:real;
Begin a:=8; b:=1.5;
      v:=(a>5)and(b>=8);   {v получит значение false}...
```

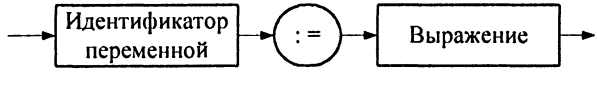


Рис. 2.9. Синтаксическая диаграмма конструкции <Оператор присваивания>

Для корректного выполнения операции присваивания результат выражения и переменная, записанная в правой части оператора присваивания, должны иметь одинаковые или совместимые типы.

Совместимыми считаются:

- все целые типы;
- все вещественные типы;
- отрезок некоторого базового типа и базовый тип;
- два отрезка одного базового типа;
- символ и строка.

При несовпадении типов правой и левой частей оператора присваивания для совместимых типов происходит неявное преобразование результата выражения к типу переменной, указанной в правой части. Например:

```
Var
   L:longint; {переменная типа longint}
   E,x: extended; {переменные типа extended}
   I:integer;   {переменная типа integer}
   R:real;   {переменная типа real}
Begin ...
  R:= I*E/(x+L); ... {результат выражения, записанного в правой
        части оператора присваивания, будет иметь тип extended,
        однако, так как переменная R типа real, то результат будет
        преобразован в этот тип}
End.
```

Если типы правой и левой частей оператора присваивания не совместимы, то необходимо использовать *явное преобразование* типов.

Явное преобразование обычно выполняют посредством использования специальных функций:

Trunc(x) – преобразует значение вещественного типа в значение целого типа, отбрасывая дробную часть;

Round(x) – преобразует значение вещественного типа в значение целого типа, округляя его до ближайшего целого;

Ord(x) – преобразует значение порядкового типа в его номер;

Chr(x) – преобразует номер символа по таблице ASCII в сам символ.

Например:

а)
```
Var n,n1:integer; xn,xk,h:real;
  Begin
     xn:=1; xk:=5.7; h:=0.3;
      n:=Round((xk-xn)/h);       {n получит значение 16}
      n1:=Trunc((xk-xn)/h); ... {n1 получит значение 15}
```

б) *Var c:char; x,y:integer;*
Begin x:=3;
y:= ***Ord('A');*** {y получит значение 65 – код символа A по таблице ASCII}
c:= ***Chr(Ord('A')+x);*** ... {c получит значение 'D'}

Кроме того, для явного преобразования типов можно использовать функции, имена которых соответствуют идентификаторам стандартных или определенных пользователем типов. Этот вид преобразования иногда называют *автоопределенным*, например:

```
Var h:char;
  ... h:=Char(65);  ...  {h получит значение 'A'}
```

Следует отметить, что при данном виде преобразования изменения значения не происходит, выполняется просто изменение типа, связанного с данным значением. В результате может произойти усечение или увеличение размера значения по сравнению с исходным. В случае усечения значения (приведение к меньшему по размеру типу) возможно изменение знака преобразуемого числа. В случае же расширения значения (приведение к большему по размерам типу) знак числа всегда сохраняется.

Например:

```
Type  Month=(Jan,Fab,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
Var  M:Month;
     A,B:integer;
     C:char;
     L:longint;
Begin
   A:=10;    C:='E';
   B:=Integer(C);      {число 69 – код символа E – длиной 2 байта}
   M:=Month(A-2);    {значение Sep}
   L:=Longint(M); .... {значение 8}
```

2.6. Процедуры ввода-вывода

Ввод значений. Для ввода значений с клавиатуры используют специальные процедуры Read и ReadLn (рис. 2.10). Эти процедуры позволяют вводить значения стандартных типов, кроме boolean, и строки (string).

Вводимые значения (кроме значений символов и строк) разделяют пробелами или записывают на разных строках. Отдельные символы и строки символов при вводе записывают подряд, так как пробел в этом случае также считается символом.

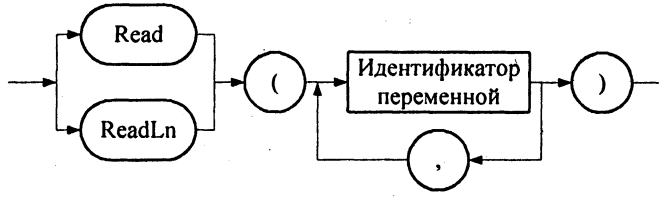


Рис. 2.10. Синтаксическая диаграмма
<Процедуры ввода с клавиатуры>

Физически операции ввода выполняются с использованием *буфера*, т. е. вводимая с клавиатуры последовательность символов сначала помещается в память, а затем, уже из памяти, читается программой. Последовательность передается в буфер ввода по нажатию клавиши Enter. При этом в буфер вместе с кодами введенных символов помещается и код Enter, состоящий из двух символов «#13, #10».

Если ввод осуществляется процедурой ReadLn, то буфер ввода после выполнения операции очищается, причем символы, оставшиеся не обработанными, игнорируются. Если ввод осуществляется процедурой Read, то очистка не выполняется и, следовательно, следующий оператор ввода начнет читать символы из той же строки. Последнее существенно только, если вводятся значения типа char (или string, см. параграф 4.5), так как при вводе чисел пробелы и переход на следующую строку игнорируются.

Например:

а)
```
Var a,b,c:real;  n:integer;
 Begin
   Read(a,b);     {числа могут быть введены в одной строке или в
                   разных}
   ReadLn(c,n);... {числа могут быть введены в той же строке, что и
                   предыдущие числа}
```

б)
```
Var a:real; c:char;
 Begin ...
   Read(a);  ...
   Write('Продолжить? (y/n)');
   Read(c);  {приводит к тому, что после запроса компьютер не
     переходит в ожидание ввода, как мы предполагали, а вводит
     следующий символ из буфера ввода, т.е. символ #13 (рис. 2.11)}
```

Чтобы избежать «игнорирования ввода», необходимо для выполнения предыдущей операции ввода использовать вместо процедуры Read процедуру ReadLn:

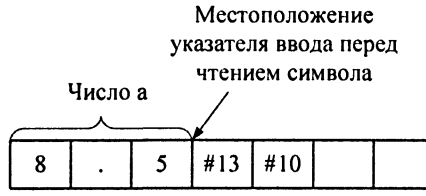


Рис. 2.11. Ситуация «игнорирования» ввода

```
Var a:real; c:char;
  Begin ...
    ReadLn(a); ...{очистим буфер ввода после выполнения операции}
    Write('Продолжить? (y/n)');
    Read(c);  {в данном случае все в порядке: после вывода запроса
              программа ожидает ввода символа}
```

Вывод значений. Для вывода значений на экран используют процедуры Write и WriteLn (рис. 2.12). Эти процедуры предназначены для вывода значений стандартных типов и строк.

Целочисленный литерал <Целое 1> интерпретируется как ширина поля, в которое выводится значение, причем выводимые значения прижимаются к правой границе. Если указанной ширины поля недостаточно, то она автоматически увеличивается до нужного значения. Если <Целое 1> не указано, то его значение определяется количеством выводимых символов.

Целочисленный литерал <Целое 2> указывается только для вещественных чисел: он определяет количество цифр дробной части числа. Если <Целое 2> указано равным 0, то ни дробная часть числа, ни десятичная точка не выводятся. Если <Целое1> и <Целое 2> не указаны, то вещественные числа выводятся в виде мантиссы и порядка, причем ширина поля вывода по умолчанию принимается равной 23, а количество дробных цифр – 14.

Логические значения выводятся как TRUE или FALSE.

Символы и строки выводятся без изменения, но дополняются пробелами, если ширина поля вывода больше, чем необходимо.

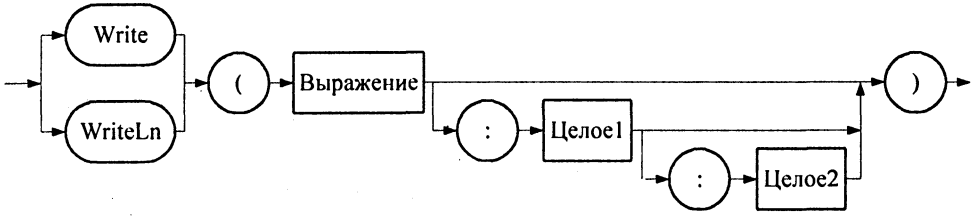


Рис. 2.12. Синтаксическая диаграмма <Процедуры вывода на экран>

После вывода значений процедурой WriteLn курсор переводится на следующую строку.

Пример 2.1. Разработать программу вычисления корней уравнения $Ax^2+Bx+C=0$ при условии, что дискриминант – неотрицательное число.

Формула корней уравнения известна:

$$x_{1,2} = \frac{-b \pm \sqrt{d}}{2a},$$

где $d = b^2 - 4ac$.

Алгоритм программы выглядит следующим образом:

Корни уравнения:
Ввести a, b, c
$d := b^2 - 4ac$
$e := b / (2a)$
$x_1 := - e + \sqrt{d} / (2a)$
$x_2 := - e - \sqrt{d} / (2a)$
Вывести x_1, x_2
Конец алгоритма.

Ниже приведен текст программы.

```
Program ex;
 Var a,b,c,x1,x2,e,d:real; {описываем переменные}
 Begin
  WriteLn('Введите коэффициенты уравнения:');
  ReadLn(a,b,c);     {вводим параметры}
  d:=b*b-4*a*c;    {определяем дискриминант}
  e:=b/(2*a);      {определяем значение вспомогательной переменной}
  x1:=-e+sqrt(d)/(2*a);    {определяем x1}
  x2:=-e-sqrt(d)/(2*a);    {определяем x2}
  WriteLn('x1=',x1:6:2,' x2=',x2:6:2); {выводим результаты}
End.
```

2.7. Практикум. Оценка точности результатов

При программировании вычислений необходимо помнить о том, что во многих случаях результат этих вычислений является числом *приближенным*.

Пусть «А» – точное значение числа, а «а» – его приближенное представление, тогда *ошибкой* или *абсолютной погрешностью* приближенного представления числа А называют значение

$$\Delta = |A - a|.$$

Обычно при оценке точности полученных результатов точное значение неизвестно. Поэтому для оценки погрешности используют ее приближение «сверху», т.е. максимально возможное значение погрешности, которое называют *предельным значением абсолютной погрешности*

$$\Delta_a \geq \Delta = |A - a|.$$

Абсолютная погрешность не является единственной характеристикой ошибки. Сравним два варианта результата с одинаковой погрешностью: 100±1 и 1±1. Очевидно, что с точки зрения практики необходимо иметь характеристику, позволяющую при оценке ошибки учитывать само значение. Такой характеристикой является относительная погрешность.

Относительной погрешностью называют отношение абсолютной погрешности числа к его модулю ($A \neq 0$):

$$\delta = \Delta / A.$$

С учетом того, что точное значение A обычно не известно, в качестве *предельной относительной погрешности* или «оценки сверху» относительной погрешности можно использовать значение

$$\delta_a = \Delta_a / (a - \Delta_a).$$

Погрешность результата вычислений складывается из погрешностей:

- допущенных при постановке задачи за счет ее упрощения (погрешности задачи);
- связанных с использованием приближенных методов решения задачи (погрешности метода);
- связанных с использованием приближенных значений параметров, например, любых физических констант (начальные погрешности);
- связанных с ограниченным количеством разрядов, используемых для представления чисел (погрешности округления);
- возникающих при выполнении операций над приближенными числами (погрешности операций).

Примечание. При программировании следует помнить, что относительная погрешность вычислений резко возрастает при вычитании двух близких чисел. Это связано с тем, что при этом резко уменьшается значение результата и соответственно также резко возрастает относительная погрешность.

Естественно, при решении конкретной задачи какие-то погрешности могут отсутствовать или быть несущественными.

Пример 2.2. Выполнить оценку погрешности представления числа 1/3 и вычислений над числами типа real.

```
Program ex;
  Var y,y1,y2,y3,y4,y5,y6,p:real;
  Begin
     y:=1;
     y1:=y/3;
     WriteLn('y1=',y1:16:14);   {выводит y1=0.33333333333348}
     y2:=sqrt(y1);
     y3:=sqr(y2);
     y4:=y3/14;
     y5:=y4*14;
     y6:=y5*3;
     WriteLn('y6=',y6:16:14);   {выводит y6=1.00000000000182}
     WriteLn('y =',y:16:14);    {выводит  y =1.00000000000000}
End.
```

Откуда погрешности представления числа 1/3 в формате real:

$$\Delta_{1/3} = |1/3 - y1| \cong |\ 0.33333333333333 - 0.33333333333348| = 0.15 \cdot 10^{-12},$$

$$\delta_{1/3} \cong 0.15\ 10^{-12}\ /(1/3) = 0.45 \cdot 10^{-10},$$

а погрешность выполнения операций над числами, представленными в формате типа real, в конкретном случае:

$$\Delta_{y6} = |y - y_6| = |1 - 1.00000000000182| = 0.182 \cdot 10^{-11},$$

$$\delta_{y6} = \Delta_{y6}\ /y = 0.182 \cdot 10^{-11}/1 = 0.182 \cdot 10^{-11}.$$

Пример 2.3. Из математики известно, что $ch^2\ x - sh^2\ x = 1$. Разработать программу, «проверяющую» это равенство.

Наша программа должна вводить значение x и для него считать

$$y_1 = (e^x + e^{-x})/2,$$
$$y_2 = (e^x - e^{-x})/2,$$
$$y = y_1^2 - y_2^2.$$

Полученные значения y_1, y_2 и y выведем на экран.

```
Program ex;
  Var x, y, y1, y2:real;
  Begin
     Write('Введите значение x:');
     ReadLn(x);
     y1:=(exp(x)+exp(-x))/2;
```

```
    y2:=(exp(x) - exp(-x))/2;
    y:= sqr(y1) - sqr(y2);
    WriteLn('y1=', y1:13:11);
    WriteLn('y2=', y2:13:11);
    WriteLn('y =', y:13:11);
End.
```

Последовательно вводя x = 5, 6, 7, ..., 14, получаем, что y ≅ 1, хотя погрешность результата растет. Однако при x=15 y=0 (!!!). Попробуем понять, почему такое происходит, для чего сведем результаты при x = 0, 5, 10, 14, 15, 20 в табл. 2.4 и рассчитаем абсолютную и относительную погрешности результата. Из таблицы видно, что значения функций ch x и sh x с увеличением x быстро растут. А чем больше число, тем длиннее запись его мантиссы. И, наконец, при x=15 разрядной сетки для записи мантиссы числа перестает хватать. При этом младшие разряды мантиссы, которые различны для ch x и sh x, отбрасываются, и при возведении чисел в квадрат мы получаем одинаковые результаты.

Для того чтобы избежать подобных ситуаций, в каждом конкретном случае используют разные приемы, например, в данном примере можно использовать тип данных с большим количеством значащих цифр (см. табл. 2.2) или вычислять результат преобразованного выражения:

$$ch^2 x - sh^2 x = (ch x - sh x)(ch x + sh x),$$

в котором значения уменьшаемого и вычитаемого растут не так быстро.

Таблица 2.4

x	y1, y2	y	Δ, δ
0	y1=1.00000000000 y2=0.00000000000	1.00000000000	Δ=0, δ=0%
5	y1=74.2099485248 y2=74.2032105778	0.99999999989	Δ=0.00000000011 δ=0. 000000011 %
10	y1=11013.2329201 y2=11013.2328747	1.00008608813	Δ=0.00008608813 δ=0. 008608813 %
14	y1=601302.142083 y2=601302.142082	1.14689281583	Δ=0.14689281583 δ=14.689281583 %
15	y1=1634508.68623 y2=1634508.68623	**0.00000000000**	Δ=1 **δ=100 %**
20	y1=242582597.704 y2=242582597.704	**0.00000000000**	Δ=1 **δ=100 %**

Тема оценки погрешностей вычислений будет продолжена в параграфе 3.5.

Задания для самопроверки

Задание 1. Измените в программе примера 2.3 тип x на double. Объясните полученные результаты.

Задание 2. Разработайте программу, которая «проверяет» формулу

$$\sin^2 x + \cos^2 x = 1.$$

Убедитесь, что при любых допустимых значениях x мы получаем правильные результаты. Почему?

3. УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ ЯЗЫКА

Программы, содержащие в разделе операторов только операторы ввода-вывода и операторы присваивания, выполняются последовательно оператор за оператором. Такие программы называют *линейными*, они реализуют линейный процесс вычислений. Для организации разветвленных и циклических процессов вычислений используют *управляющие операторы* языка, определяющие последовательность выполнения операторов программы. В данной главе мы рассмотрим управляющие операторы языка Borland Pascal, к которым относят оператор условной передачи управления, оператор выбора, операторы организации циклов, а также неструктурные операторы и процедуры передачи управления.

3.1. Оператор условной передачи управления

Оператор условной передачи управления (рис. 3.1) используют для программирования ветвлений, т. е. ситуаций, когда возникает необходимость при выполнении условия реализовывать одни действия, а при нарушении – другие. Условие записывают в виде логического выражения, в зависимости от результата которого осуществляется выбор одной из ветвей: если результат true, то выполняется оператор, следующий за служебным словом then, иначе – оператор, следующий за служебным словом else.

В каждой ветви допускается запись одного оператора (в том числе и другого if) или составного оператора.

Составным оператором в Borland Pascal называют последовательность операторов, заключенную в операторные скобки begin...end. Операторы последовательности отделяют друг от друга точкой с запятой «;». Перед end точку с запятой можно не ставить. Перед else точка с запятой *не ставится никогда*, так как в этом случае запись условного оператора продолжается.

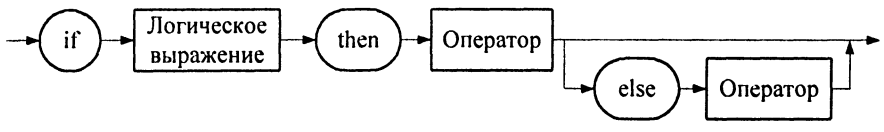


Рис. 3.1. Синтаксическая диаграмма <Оператор условной передачи управления>

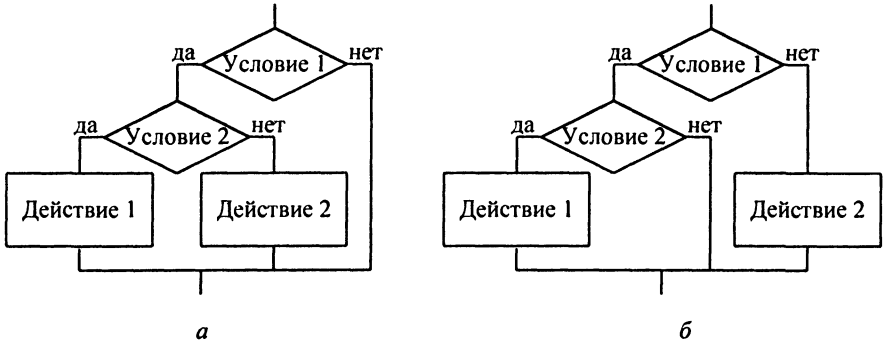


Рис. 3.2. Фрагменты алгоритмов

В соответствии с синтаксической диаграммой допускается использовать оператор условной передачи управления с неуказанной (пустой) ветвью else. В некоторых случаях использование укороченных конструкций может привести к неоднозначности, например, не понятно, какому из двух вариантов схем алгоритма (рис. 3.2) соответствует фрагмент:

```
if <условие1> then
    if <условие 2> then
            <действие 1>
    else <действие 2>;
```

В этих случаях используется так называемое правило вложенности: *альтернатива else всегда относится к ближайшему if*, что соответствует варианту алгоритма на рис. 3.2, *а*. Если необходимо реализовать вариант алгоритма, изображенный на рис. 3.2, *б*, то используют операторные скобки:

```
if <условие1> then
   begin
      if <условие 2> then
            <действие 1>
   end
else <действие 2>;
```

Пример 3.1. Разработать программу, которая вычисляет значение функции, заданной следующим образом:

$$y = \begin{cases} |x| & \text{при } |x| \le 1; \\ x^2 & \text{при } 1 < |x| \le 2; \\ 4 & \text{иначе.} \end{cases}$$

Программа должна начинаться с ввода значения аргумента. Затем в зависимости от того, в какой интервал попадает введенное значение, вычисля-

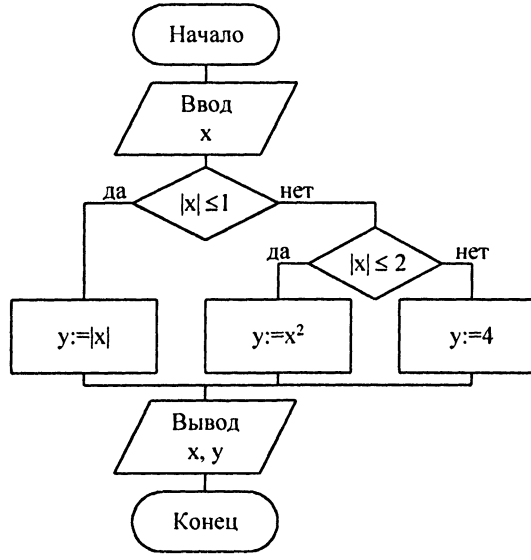


Рис. 3.3. Схема алгоритма программы вычисления функции, заданной на отрезках, в заданной точке

ем значение функции по одному из заданных выражений. Алгоритм решения данной задачи представлен на рис. 3.3. Текст программы имеет следующий вид:

```
Program ex;
 Var x,y:real;
 Begin
    WriteLn('Введите значение аргумента.');
    ReadLn(x);
    if abs(x)<=1 then y:=abs(x)        {первый отрезок}
    else
        if abs(x)<=2 then
                y:=sqr(x)                {второй отрезок}
        else    y:=4;                    {третий отрезок}
    WriteLn('При x=', x:8:5, ' y=', y:8:5);
 End.
```

3.2. Практикум. Тестирование программ

Как уже упоминалось в параграфе 1.4, тестированием называют процесс выполнения программы с различными исходными данными, для которых заранее известны результаты. Интуитивно начинающие программисты обычно

целью тестирования считают проверку правильности программы, что совершенно не верно. В большинстве случаев перебрать все возможные комбинации данных невозможно, а выборочное тестирование не доказывает правильности программы, так как то, что программа работает на десяти наборах данных, не означает, что она будет давать правильные результаты на одиннадцатом наборе. Поэтому целью тестирования является *обнаружение ошибок*.

Соответственно хорошим следует считать тест, обнаруживающий ошибку. Для формирования таких тестов определены две стратегии:

- «белого» или «прозрачного ящика» (тестирование маршрутов);
- «черного ящика».

При тестировании с использованием стратегии «белого ящика» тесты стараются подобрать так, чтобы *хотя бы один раз пройти по каждой ветви алгоритма*. Стратегия имеет существенный недостаток: по ней принципиально невозможно обнаружить пропущенный маршрут.

При тестировании с использованием стратегии «черного ящика» структура программы считается неизвестной, и тесты подбирают так, чтобы проверить выполнение всех функций программы, а затем отследить реакцию на ввод некорректных данных.

На практике лучшие результаты получают, используя при разработке тестов обе стратегии.

Пример 3.2. Даны длины сторон треугольника, определить вид треугольника и его площадь. Выполнить контроль вводимых чисел.

Сама задача решается просто. Вид треугольника определим, сравнивая стороны, а площадь вычислим по формуле Герона. Задание «выполнить контроль вводимых чисел» означает, что программа в случае ввода чисел, которые не могут интерпретироваться как стороны треугольника, должна выдавать сообщение об ошибках данных. Три числа нельзя интерпретировать как стороны треугольника, если хотя бы одно из них меньше или равно 0, или сумма двух любых чисел больше третьего.

На рис. 3.4 представлена схема алгоритма данной программы, а сама программа имеет следующий вид:

```
Program ex;
Var A,B,C,P,S:real;
Begin
  Write('Введите длины сторон треугольника:');
  ReadLn(A,B,C);
  if (A<=0)or(B<=0)or(C<=0) then
    WriteLn('Числа должны быть положительными.')
  else
      if (A+B<=C) or (A+C<=B) or (B+C<=A) then
          WriteLn('Треугольник с такими сторонами не существует.')
```

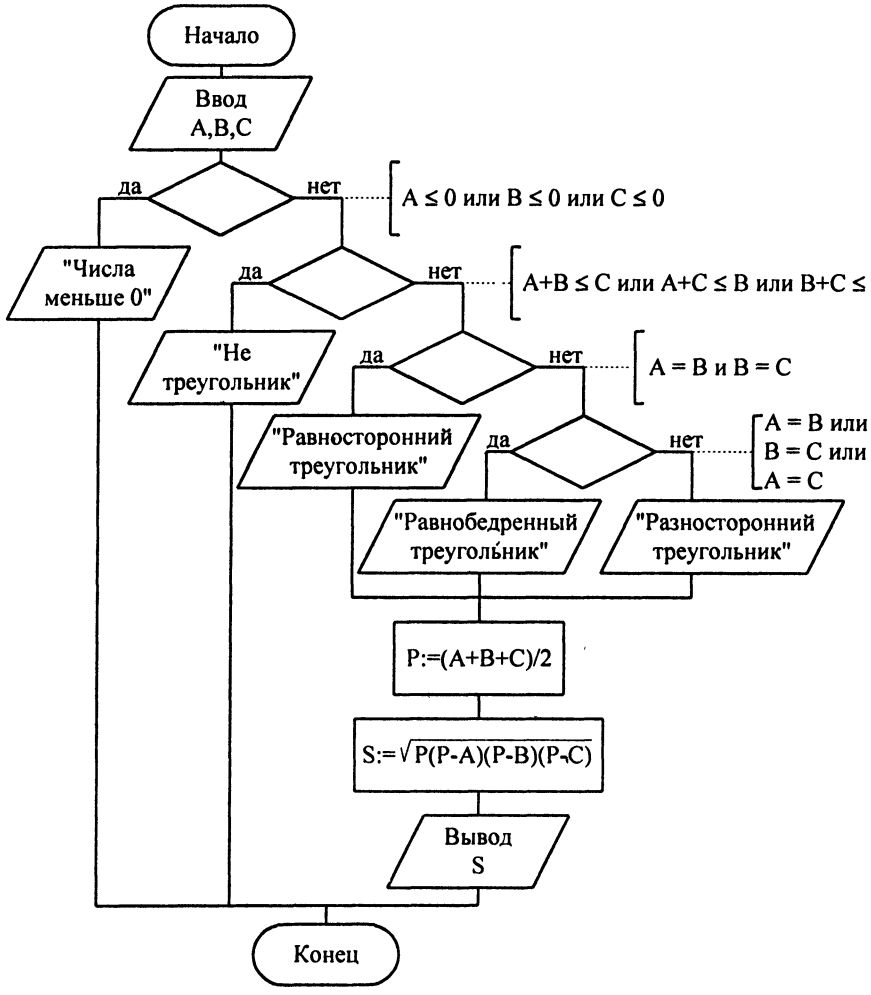


Рис. 3.4. Схема алгоритма программы определения вида треугольника

```
else
  begin
    if (A=B)and(B=C) then
            WriteLn('Треугольник равносторонний,')
    else
        if (A=B)or(A=C)or(B=C) then
            WriteLn('Треугольник равнобедренный,')
        else WriteLn('Треугольник разносторонний,');
    P:=(A+B+C)/2;
    S:=sqrt(P*(P-A)*(P-B)*(P-C));
```

Таблица 3.1

A	B	C	Ожидаемый результат	Объект проверки
1	1	1	Равносторонний, S=0.43	Маршрут
2	2	3	Равнобедренный, S=1.98	Маршрут
4	3	4	Равнобедренный, S=5.56	Маршрут
2.5	2.5	4	Равнобедренный, S=3.00	Тип данных и результата
3.1	4	2.6	Разносторонний, S=4.03	Тип данных и результата
0	0	0	Числа должны быть >0	Недопустимые данные
1	-1.5	1	Числа должны быть >0	Недопустимые данные
0	1	1	Числа должны быть >0	Недопустимые данные
1	2	0	Числа должны быть >0	Недопустимые данные
8	4	3	Треугольник не существует	Недопустимые данные
2	2	5	Треугольник не существует	Недопустимые данные
3	9	4	Треугольник не существует	Недопустимые данные

```
        WriteLn('площадь треугольника: ',S:8:2, ' единиц.');
    end
End.
```

Примерный набор тестов для обнаружения ошибок в рассматриваемой программе приведен в табл. 3.1

Задания для самопроверки

Задание 1. Разработайте программу, которая вводит вещественные числа x, y и определяет, принадлежит ли точка с координатами (x, y) закрашенной части плоскости (рис. 3.5). Выполните тестирование полученной программы.

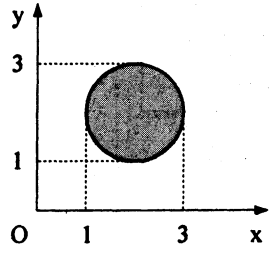


Рис. 3.5. Вид области к заданию 1

Задание 2. Заданы целые числа a, b, где a < b. Разработайте программу, которая определяет, имеет ли точки разрыва функция:

$$g(x) = \begin{cases} x^4 & \text{при } x < a; \\ x & \text{при } a \le x \le b; \\ 1 & \text{при } x \ge b. \end{cases}$$

Выполните тестирование полученной программы в точках предполагаемого разрыва.

Задание 3. Периодическая функция f(x) на отрезке [-1, 1] совпадает с функцией x^2+1. Разработайте программу, которая определяет значение функции f(x) для заданного значения x_0. Выполните тестирование полученной программы.

3.3. Оператор выбора

Оператор выбора используется для реализации нескольких альтернативных вариантов действий, каждый из которых соответствует своим значениям некоторого параметра. Синтаксическая диаграмма этого оператора приведена на рис. 3.6.

Пример 3.3. Разработать программу, которая вычисляет значение одной из заданных функций в указанной точке.

Предоставим пользователю возможность выбрать функцию через простейшую имитацию меню, в котором каждой функции соответствует некоторое число (код):

Введите код функции:
1 – y=sin(x)
2 – y=cos(x)
3 – y=exp(x)

В зависимости от значения введенного кода выбирается одна из функций. На рис. 3.7 представлена схема алгоритма программы.

Ниже представлена программа, реализующая разработанный алгоритм.

```
Program ex;
 Var x,y:real; Kod:byte; Key:boolean;
 Begin
   WriteLn('Введите код функции:');
   WriteLn('1 - sin(x)');
   WriteLn('2 - cos(x)');
   WriteLn('3 - exp(x)');
   ReadLn(Kod);
```

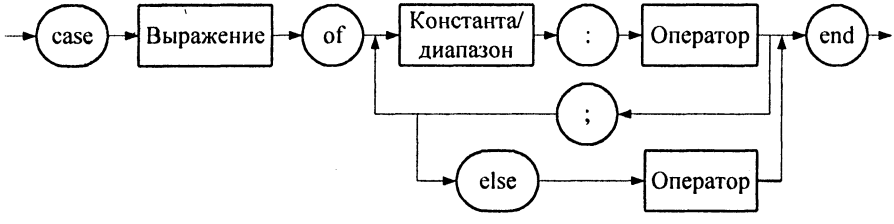


Рис. 3.6. Синтаксическая диаграмма <Оператор выбора>

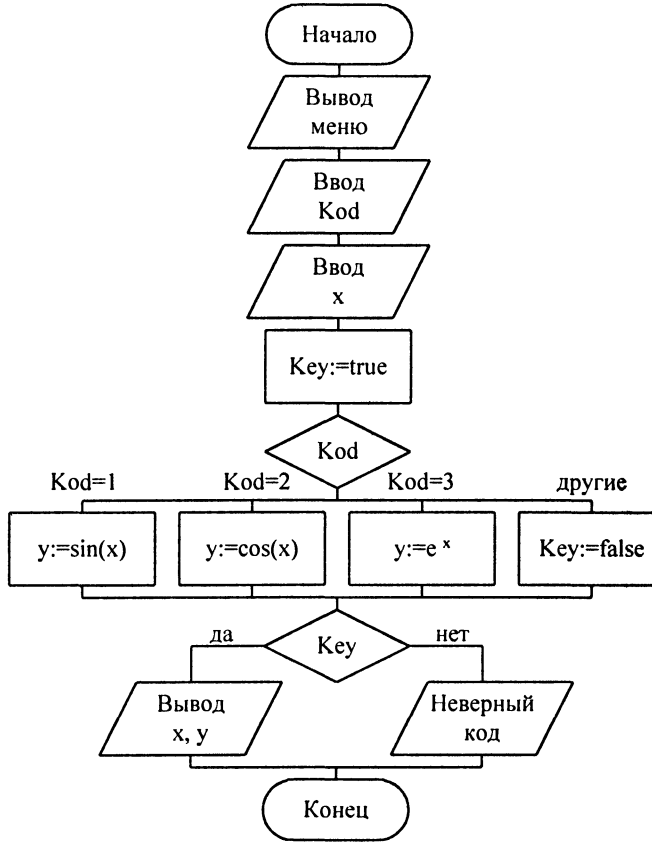


Рис. 3.7. Алгоритм программы вычисления значения одной из заданных функций

```
Write('Введите значение аргумента:');
ReadLn(x);
Key:=true;   {признак правильности кода}
case Kod of
    1:   y:=sin(x);
    2:   y:=cos(x);
    3:   y:=exp(x);
    else Key:=false;  {код не соответствует функции}
end;
if Key then
            WriteLn('При x=',x:12:6,' y=',y:12:6)
else
            WriteLn('Введен неверный код функции.');
End.
```

3.4. Операторы организации циклической обработки

Для реализации циклических процессов используют операторы циклов.

Как уже пояснялось в параграфе 1.3, в теории программирования выделяют несколько основных видов циклов:

- цикл-пока (рис. 3.8, *а*);
- цикл-до (рис. 3.8, *б*);
- счетный цикл (рис. 3.8, *в*).

В Borland Pascal реализованы все три указанных вида циклов. Цикл-пока и цикл-до используют для реализации итерационных циклических процессов. Счетный цикл – для реализации циклических процессов с заданным количеством повторений. Для реализации циклических процессов поискового типа используют циклы-пока или циклы-до со сложными условиями или неструктурные передачи управления (см. параграф 3.6).

Цикл-пока. Синтаксическая диаграмма оператора «цикл-пока» приведена на рис. 3.9. Условие записывают в виде логического выражения. Оператор тела цикла повторяется, пока условие истинно. Проверка осуществляется на входе. Если при входе в цикл условие не выполняется, то оператор тела цикла игнорируется.

Если в тело цикла необходимо поместить несколько операторов, то используют составной оператор.

Цикл-до. Операторы тела цикла повторяются до выполнения условия, условие проверяется на выходе, т.е. тело цикла всегда выполняется хотя бы

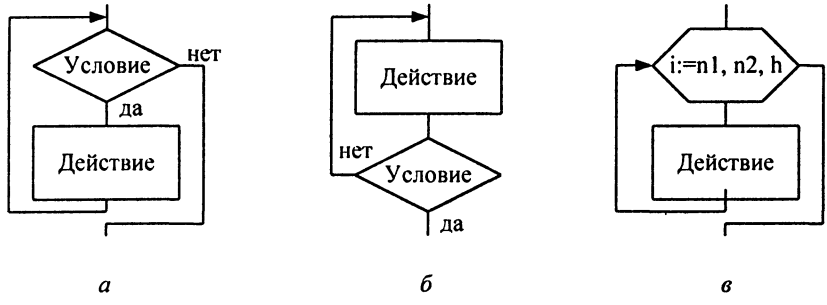


Рис. 3.8. Структура циклов, реализованных в Borland Pascal:
а – цикл-пока; *б* – цикл-до; *в* – счетный цикл

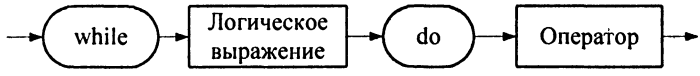


Рис. 3.9. Синтаксическая диаграмма <Цикл-пока>

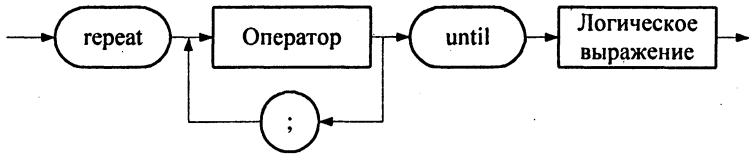


Рис. 3.10. Синтаксическая диаграмма <Цикл-до>

один раз. Синтаксическая диаграмма оператора «цикл-до» приведена на рис. 3.10. В тело цикла можно поместить несколько операторов, разделив их точкой с запятой «;».

Счетный цикл. Цикл выполняется, пока переменная (*параметр*) цикла принимает значения в заданном диапазоне с определенным шагом. Синтаксическая диаграмма оператора приведена на рис. 3.11. *Переменная цикла* должна иметь порядковый тип. Выражение 1 определяет начальное значение параметра цикла, выражение 2 – конечное значение параметра цикла. Соответственно начальное и конечное значения должны принадлежать к тому же типу, что и параметр цикла. Если используется служебное слово to, то при каждом выполнении цикла переменной цикла присваивается следующее значение порядкового типа переменной. Если используется служебное слово downto, то при каждом выполнении цикла переменной цикла присваивается предыдущее значение порядкового типа переменной. Если диапазон значений переменной цикла пуст, то цикл не выполняется.

Примечание. По сравнению с теоретическим представлением оператор счетного цикла, реализованный в Borland Pascal, является менее мощной конструкцией, так как шаг цикла ограничен (фактически только +1 и -1).

Пример 3.4. Разработать программу вычисления суммы n первых натуральных чисел.

Сумма определяется методом *накопления*. Количество суммируемых чисел известно, поэтому используем цикл с заданным количеством повторений. При каждом проходе к сумме будем добавлять переменную цикла, которая будет изменяться от 1 до n. Перед циклом переменную суммы необходимо

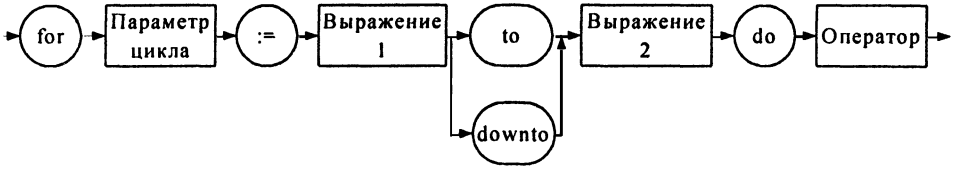


Рис. 3.11. Синтаксическая диаграмма
<Цикл с заданным количеством повторений>

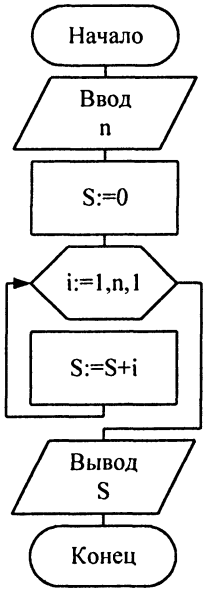


Рис. 3.12. Алгоритм вычисления суммы n натуральных чисел

обнулить. На рис. 3.12 приведена схема алгоритма программы. Текст программы приведен ниже.

```
Program ex;
  Var
      i, n, S:integer;
  Begin
     Writeln('Введите n');
     ReadLn(n);
     S:=0;
     for i:=1 to n do
                        S:=S+i;
     WriteLn('Сумма ', n ,' чисел равна ', S);
  End.
```

Пример 3.5. Разработать программу, определяющую сумму ряда $1 - 1/x + 1/x^2 - 1/x^3 + \ldots$ с заданной точностью ε (епсилон).

Из соответствующих разделов математики известно, что суммой ряда называется предел, к которому стремится последовательность частичных сумм данного ряда, если он существует. Если такой предел существует, то ряд называется *сходящимся*, в противном случае – *расходящимся*. Также известно, что знакопеременный ряд сходится, если

$$|r_n| > |r_{n+1}| ,$$

где r_n и r_{n+1} – соответственно n-й и n+1-й члены ряда.

Кроме того, доказано, что

$$|S-S_n| \le |r_{n+1}| ,$$

где S – сумма ряда, а S_n – сумма n членов ряда.

Следовательно, для получения требуемого результата будем накапливать частичную сумму элементов ряда, пока очередной член ряда не станет меньше заданной погрешности:

$$|r_n| < \varepsilon .$$

На рис. 3.13 представлен алгоритм программы. Анализ алгоритма показывает, что он неструктурный, так как в нем присутствует цикл суммирова-

ния, который не является ни циклом-пока, ни циклом-до, ни циклом с заданным количеством повторений. Для реализации данного алгоритма необходимо его преобразовать в структурный, чтобы можно было использовать один из имеющихся операторов цикла.

Поскольку количество повторений цикла определить нельзя, попробуем преобразовать неструктурный цикл к циклу-пока. Для этого необходимо операцию S=S+R продублировать: одну копию поместить до цикла, а вторую – в конце цикла. Операторы S=0 и S=S+R, записанные до цикла, заменим оператором S=1, так как в этот момент R=1. Условие выхода из цикла также необходимо заменить на противоположное. Окончательный вариант фрагмента алгоритма с циклом-пока показан на рис. 3.14, *а*. Его реализация представлена ниже:

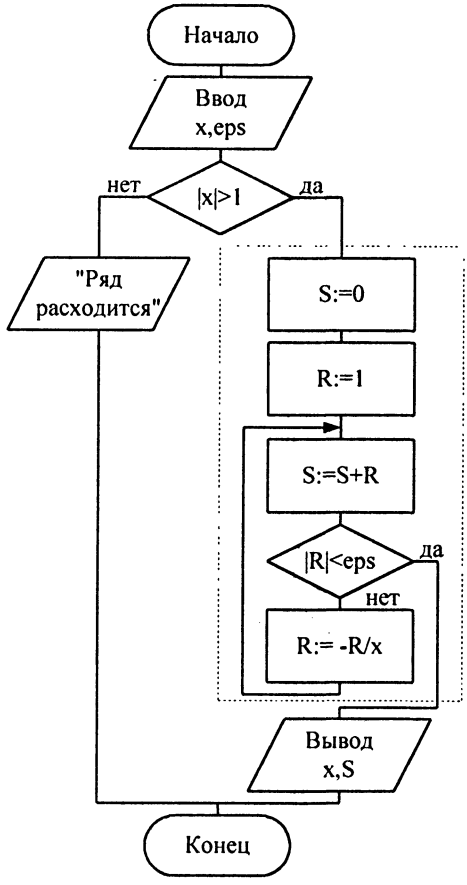


Рис. 3.13. Алгоритм определения суммы ряда с заданной точностью

```
Program ex;
 Var S,R,X,eps:real;
 Begin
     WriteLn('Введите x и эпсилон:');
     ReadLn(X,eps);
     if abs(x)>1 then   {если x > 1, то ищем сумму ряда}
       begin
         S:=1;
         R:=1;
         while abs(R)>eps do
           begin
             R:=-R/X;
             S:=S+R;
           end;
```

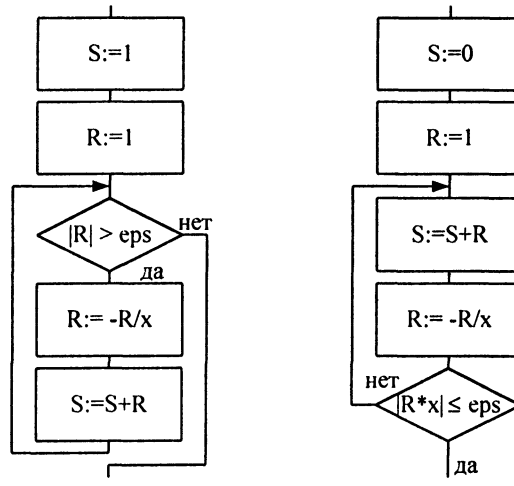


Рис. 3.14. Структурные варианты алгоритма:
а – с использованием цикла-пока; *б* – с использованием цикла-до

```
        WriteLn('При x=', x:6:2,' S=', S:8:2,', a R=', R:8:6)
      end
    else Writeln('Ряд расходится');
End.
```

Тот же алгоритм можно преобразовать так, чтобы цикл можно было реализовать с использованием ц и к л а - д о (рис. 3.14, *б*). Ниже представлена соответствующая программа.

```
Program ex;
 var S,R,X,eps:real;
 Begin
     WriteLn('Введите значение x и эпсилон:');
     ReadLn(X,eps);
     if x>1 then
        begin
          S:=0;
          R:=1;
          repeat
             S:=S+R;
             R:=-R/X
          until abs(R*x)<=eps;
         WriteLn('При x=',x:6:2,' S=',S:8:2,', a R=',R:8:6)
        end
```

```
    else Writeln('Ряд расходится');
End.
```

Другие способы реализации неструктурных алгоритмов более подробно будут рассмотрены в параграфе 3.4.

3.5. Практикум. Точность решения задач вычислительной математики

Вычислительная математика занимается рассмотрением численных методов вычисления значений функций, решения алгебраических и трансцендентных уравнений, систем уравнений, интерполяции функций и т.д. К той же группе задач относится и задача о нахождении суммы ряда, рассмотренная в предыдущем параграфе. Естественно в рамках одного параграфа невозможно рассмотреть алгоритмы решения всех задач данного раздела математики, но попробуем на примере двух из них продемонстрировать проблемы достижения заданной точности результатов, которые возникают при решении задач этой области.

Пример 3.6. Разработать программу вычисления определенного интеграла функции f(x) на заданном отрезке [a, b] методом прямоугольников.

Определенный интеграл вычисляется как площадь фигуры, ограниченной графиком функции, отрезком оси абсцисс и вертикальными линиями, проходящими через границы отрезка. Метод прямоугольников предполагает, что площадь определяется как сумма площадей n прямоугольников, основанием которых является n-я часть отрезка [a,b], а стороной – значение функции на одном из концов отрезка (например, левом, как на рис. 3.15).

Итак

$$S1 = f(x_1)\times\delta + f(x_2)\times\delta + f(x_3)\times\delta + \ldots + f(x_n)\times\delta = \delta\times\sum_{i=1}^{n} f(x_i),$$

где $\delta = (b-a) / n$.

Значение определенного интеграла S1, определенное по данной формуле, является не точным, причем с увеличением количества отрезков n точность значения S1 увеличивается. Считают, что значение определено с заданной точностью, если абсолютная величина разности двух последовательных приближений результата, полученных при разных значениях n, не превышает заданной погрешности.

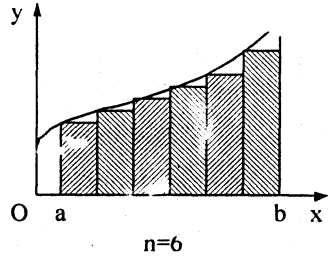


Рис. 3.15. Вычисление определенного интеграла методом прямоугольника

Для определения момента достижения заданной точности необходимо организовать

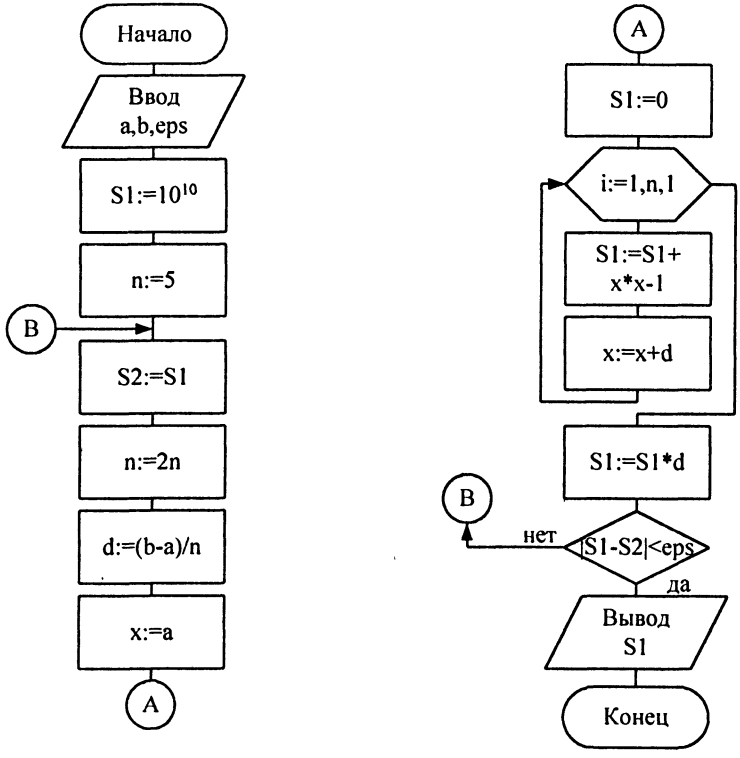


Рис. 3.16. Схема алгоритма вычисления определенного интеграла

еще один – внешний цикл, в котором значение n будем увеличивать, например, в два раза, и рассчитывать значение S1, предварительно сохранив предыдущее значение S1 в переменной S2. Цикл должен завершаться, когда абсолютная величина разности двух приближений S1 и S2 станет меньше ε. Исходное значение S2 примем достаточно большим, чтобы цикл не завершился при первой проверке условия.

Схема алгоритма вычисления определенного интеграла для функции $f(x) = x^2 - 1$ приведена на рис. 3.16.

Ниже приведена программа, реализующая данную схему алгоритма.

```
Program ex;
Var a,b,S1,S2,d,eps,x:real;n,i:longint;
 Begin
    WriteLn('Введите a, b и эпсилон:');
    ReadLn(a,b,eps);
    S1:=1E+10;
    n:=5;
```

```
    repeat
      S2:=S1;
      n:=n*2;
      d:=(b-a)/n;
      x:=a;
      S1:=0;
      for i:=1 to n do
        begin
          S1:=S1+x*x-1;
          x:=x+d;
        end;
      S1:=S1*d;
    until abs(S1-S2)<eps;
    WriteLn('I=', S1:10:6);
End.
```

При выполнении программы с разными значениями погрешности eps и отрезком [0, 2] получаем разные приближения результата I (табл. 3.2). Точное значение определенного интеграла равно 2/3. Определим абсолютную и относительную погрешности результата.

В данном случае абсолютная погрешность результата всегда будет меньше eps, так как это определяется самим методом. Последнее верно не для любого метода.

Пример 3.7. Разработать программу, которая определяет корень непрерывной функции f(x) на заданном отрезке [a, b] методом хорд.

Метод работает в том случае, если значения функции на концах отрезка разных знаков, т.е. если функция только касается оси абсцисс, то ее корень данным методом определить нельзя. Если на заданном отрезке у функции несколько корней, то, используя данный метод, найдем один из них.

Метод хорд заключается в следующем. Значения функции на концах отрезка соединяют отрезком прямой. В точке пересечения этой прямой с осью абсцисс определяют значение функции. Если это значение меньше заданной погрешности функции, то абсцисса точки пересечения считается корнем

Таблица 3.2

eps	n	I	Δ_I	δ_I, %
0.01	640	0.660420	0.006	0.9
0.001	5120	0.665885	0.0002	0.06
0.0001	40960	0.666569	0.00004	0.012

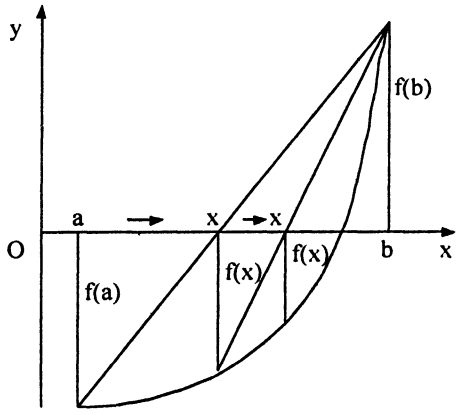


Рис. 3.17. Нахождение корня методом хорд

функции, иначе корень функции ищется на том отрезке из двух полученных, для которого значения функции на концах разных знаков. На рис. 3.17 показана графическая интерпретация метода.

На рис. 3.18 представлена схема алгоритма программы для функции $f(x) = x^2 - 1$.

Полученный алгоритм является не структурным, так как цикл, который в нем использован, не является ни циклом-до, ни циклом-пока. Для преобразования алгоритма в структурный используем при-

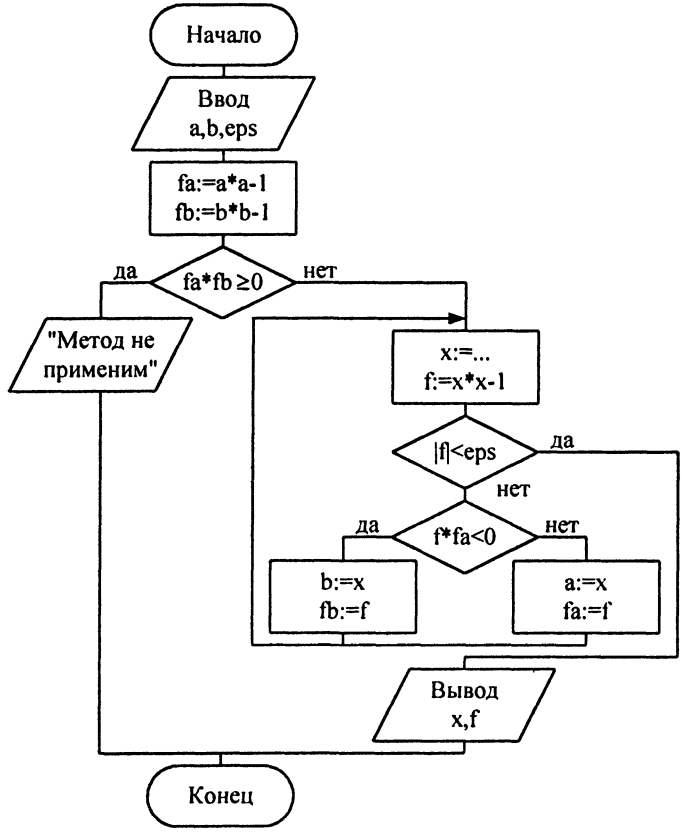
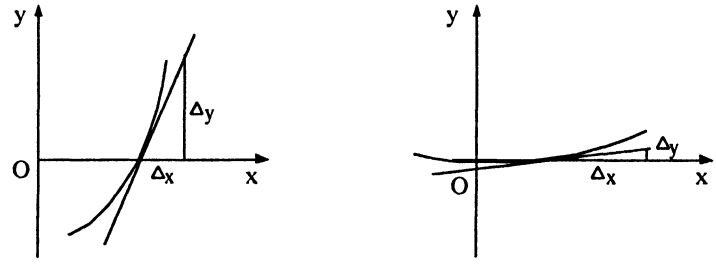


Рис. 3.18. Схема алгоритма нахождения корня методом хорд



а *б*

Рис. 3.19. Соотношение Δ_y и Δ_x в зависимости от вида функции:
а – угол наклона производной в корне больше 45°; *б* – меньше 45°

функции. Однако, если бы производная функции в точке корня была близка к нулю, это соотношение было бы обратным (рис. 3.19), и, следовательно, метод оказался бы неприменимым.

В качестве альтернативы можно предложить алгоритм, в котором выход из цикла вычислений выполняется, когда разность между двумя соседними приближениями значений корня становится меньше заданного eps (по аналогии с примером 3.6). Однако этот алгоритм был бы не применим для поиска корня, если угол наклона производной функции в корне больше 45°.

Следовательно, применяя тот или иной методы вычислительной математики, необходимо проверять обеспечение точности получаемых результатов.

Задания для самопроверки

Задание 1. Разработайте программу вычисления

$$\text{arctg } x = x - x^3/3 + x^5/5 - x^7/7 + \ldots$$

при заданном значении x и точности 10^{-3}, 10^{-4}. Оцените количество итераций в каждом случае.

Задание 2. Разработайте программу, которая определяет сумму первых k чисел последовательности Фибоначчи. Последовательность задана законом

$$F_0 = F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}$$

для $n \geq 2$.

Задание 3. Разработайте программу вычисления длины кривой на отрезке [0,4]. Кривая задана уравнением $y=x^{3/2}$. Вычисления выполните с точностью 10^{-3}, 10^{-4}. Оцените количество итераций в каждом случае.

Задание 4. Разработайте программу, которая определяет, является ли введенное число n простым.

Задание 5. Разработайте программу вычисления: $\sqrt{3 + \sqrt{6 + \ldots + \sqrt{60 + \sqrt{63}}}}$.

ем, рассмотренный в примере 3.3. А именно: повторяем в конце тела цикла операторы, выполняемые в начале цикла до условия, и организуем цикл, как цикл-пока. Поскольку выход из цикла-пока осуществляется по нарушению условия цикла, придется также изменить условие цикла. Программа, реализующая структурный вариант алгоритма, имеет следующий вид:

```
Program ex;
Var a,b,f,fa,fb,eps,x:real;
Begin
   WriteLn('Введите a, b и эпсилон:');
   ReadLn(a,b,eps);
   fa:=a*a-1;
   fb:=b*b-1;
   if fa*fb>=0 then WriteLn('Метод не применим.')
   else
     begin
       x:=(b-a)*abs(fa)/abs(fa-fb)+a;
       f:=x*x-1;
       while abs(f)>=eps do    {условие выхода из цикла}
        begin
           if fa*f<0 then
                  begin  b:=x;  fb:=f;  end
         else begin  a:=x;  fa:=f;  end;
         x:=(b-a)*abs(fa)/abs(fa-fb)+a;
         f:=x*x-1;
        end;
      WriteLn('При x=', x:9:6, ' y=', f:10:8);
    end;
End.
```

Меняя значение погрешности, получим разные значения корня и значения функции в корне (табл. 3.3). Зная точное значение корня x=1, определим предельную абсолютную и относительную погрешности в каждом случае. Из таблицы видно, что погрешность корня меньше погрешности значения

Таблица 3.3

$eps = \Delta_y$	x	y	Δ_x	δ_x, %
0.01	0.997260	-0.00547195	0.003	0.3
0.001	0.999695	-0.00060948	0.00031	0.03
0.0001	0.999966	-0.00006774	0.000034	0.003

3.6. Неструктурные алгоритмы и их реализация

С точки зрения теории программирования неструктурные оператор и процедуры передачи управления являются «лишними», так как любой алгоритм может быть преобразован в структурный и реализован без них. Однако интуитивно построенные алгоритмы, как это видно на предыдущих примерах, часто получаются неструктурными, и для их реализации может потребоваться использование неструктурных вариантов передач управления. Проанализируем достоинства и недостатки реализации неструктурных алгоритмов.

Для организации неструктурных передач управления используют оператор безусловной передачи управления goto и специальные процедуры.

Оператор безусловной передачи управления. Этот оператор передает управление в точку, определенную специальной меткой (рис. 3.20).

Все метки в программе должны быть описаны инструкцией объявления меток label (рис. 3.21). Метка ставится перед любым выполняемым оператором программы, причем на один оператор можно поставить несколько меток.

Неструктурную передачу управления также осуществляют следующие процедуры:

Break – реализует выход из цикла любого типа;

Continue – осуществляет переход на следующую итерацию цикла, игнорируя оставшиеся до конца тела цикла операторы;

Halt (<код завершения>***)*** – осуществляет выход из программы, возвращая операционной системе заданный код завершения. Считается, что программа завершилась нормально, если код завершения равен нулю. Возвращение кода завершения, отличного от нуля, обычно означает,что программа завершена по обнаружении каких-либо ошибок. Коды завершения назначаются программистом, а информация о них помещается в программную документацию;

Exit – осуществляет выход из подпрограммы (см. главу 5). Если процедура использована в основной программе, то она выполняется аналогично Halt.

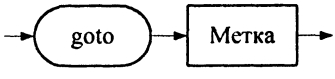


Рис. 3.20. Синтаксическая диаграмма <Оператор безусловной передачи управления>

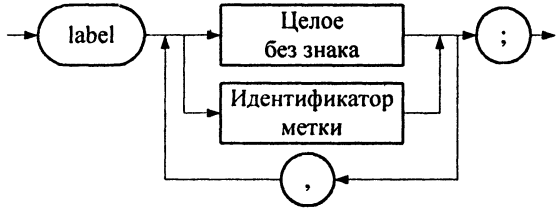


Рис. 3.21. Синтаксическая диаграмма <Объявление меток>

Чаще всего проблема разработки структурного варианта алгоритма возникает при работе с поисковыми циклами. Как уже упоминалось в параграфе 1.3, в таких циклах просматривают некоторые последовательности элементов, пока не будет обнаружен элемент с заданными характеристиками. Отличительной особенностью поискового цикла является то, что элемент с заданными характеристиками в последовательности может отсутствовать, следовательно, необходимо предусмотреть два варианта выхода из цикла: досрочный, когда нужный элемент найден, и обычный – по завершении просмотра всех вариантов.

Пример 3.8. Разработать программу, которая определяет первый отрицательный элемент последовательности значений функции sin x при заданных n, шаге h и диапазоне изменения x [a, b].

Для получения требуемого результата необходимо последовательно вычислять значение функции и анализировать его знак. Количество элементов последовательности на заданном отрезке и с заданным шагом можно определить, следовательно, для вычисления значений функции можно использовать счетный цикл. Для конкретных вариантов исходных данных может оказаться, что такого элемента в заданной последовательности нет. Следовательно, после завершения цикла необходимо вывести соответствующее сообщение. Неструктурный вариант алгоритма решения задачи показан на рис. 3.22.

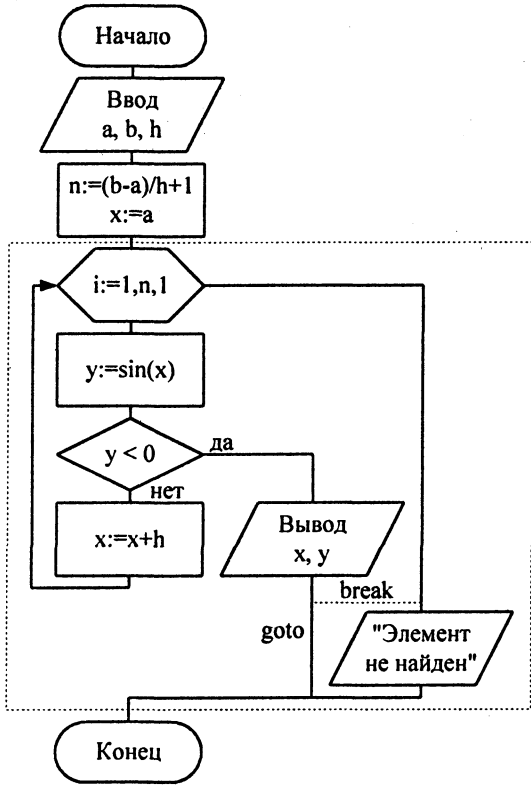
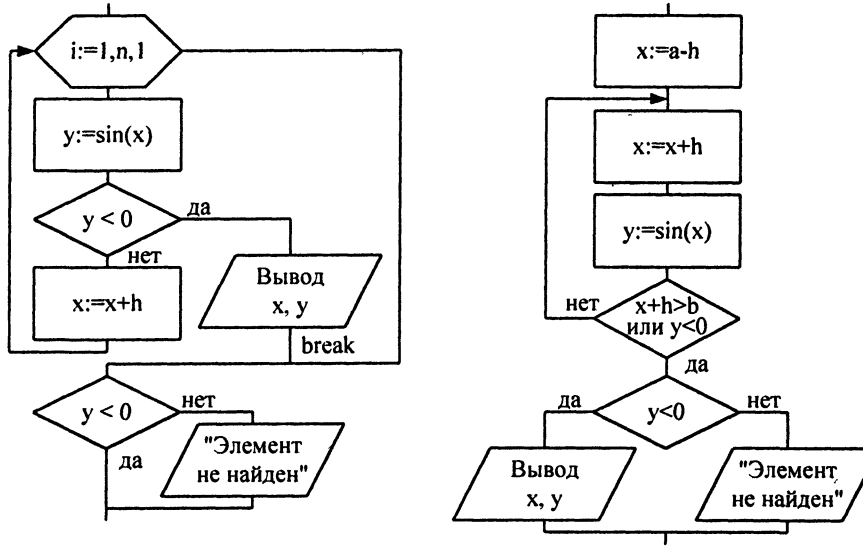


Рис. 3.22. Схема алгоритма поиска первого отрицательного элемента последовательности (пунктиром выделен поисковый цикл)

Без преобразований этот вариант алгоритма можно реализовать только с использованием оператора goto, так как он позволяет передать управление в любое место программы, а процедура break – только оператору, следующему после цикла (пунктир на рис. 3.22). Поэтому реализация с использованием break потребует дополнительной проверки (найден ли элемен) на вы-



а *б*

Рис. 3.23. Варианты организации поискового цикла для реализации:
а – с использованием процедуры break; *б* – структурный вариант

ходе из цикла, чтобы вывести сообщение о том, что элемент не найден, только в том случае, если он действительно не найден (рис. 3.23, *а*)

Для построения структурного варианта алгоритма меняем вид цикла: вместо счетного цикла будем использовать цикл-до со сложным условием, объединяющим оба условия выхода. После выхода из цикла неизвестно, по какому из условий цикл завершился, поэтому в данном варианте алгоритма также приходится проверять, найден нужный элемент или нет (рис. 3.23, *б*). Рассмотрим соответствующие варианты программ.

Вариант с использованием оператора goto:

```
Program ex;
Var i,n:integer; x,y,a,b,h:real;
Label konec;          {объявляем метку}
 Begin
    Write('Введите a,b,h:');
    Readln(a,b,h);
    n:=round((b-a)/h+1.5); {определяем количество элементов}
    x:=a;
   for i:=1 to n do
     begin
       y:=sin(x);
```

```
        if y<0 then
          begin
            WriteLn('y=',y:8:6,' при x=',x:6:3);
            goto konec;  {управление передаем на конец программы по-
                          сле вывода сообщения о том, что элемент не найден}
          end;
        x:=x+h;
      end;
  WriteLn('Элемент не найден.');
 konec:      {место, куда передается управление}
End.
```

Вариант с использованием процедуры break:

```
Program ex;
var i,n:integer; x,y,a,b,h:real;
Begin
  Write('Введите a,b,h:');
  Readln(a,b,h);
  n:=round((b-a)/h+1.5); {определяем количество элементов}
  x:=a;
  for i:=1 to n do
    begin
      y:=sin(x);
      if y<0 then
        begin
          WriteLn('y=',y:8:6,' при x=',x:6:3);
          break;    {осуществляем досрочный выход из цикла}
        end;
      x:=x+h;
    end;
{место, куда будет передано управление при выполнении break}
  if y>=0 then WriteLn('Элемент не найден.');
End.
```

Структурный вариант:

```
Program ex;
var x,y,a,b,h:real;
Begin
  Write('Введите a,b,h:');
  Readln(a,b,h);
  x:=a-h;
  repeat
```

```
    x:=x+h;
    y:=sin(x);
  until (x+h>b) or (y<0);   {комбинированное условие выхода из цикла}
  if y<0 then WriteLn('y=', y:8:6, ' при x=', x:6:3)
         else WriteLn('Элемент не найден.');
End.
```

Разрабатывая структурный вариант цикла, мы были вынуждены однозначно сформулировать условие выхода из цикла и точно определить признак, по которому делается вывод о том, найдено ли решение. В то время как реализация «естественного» варианта алгоритма с использованием goto или break требует учета всех возможных последствий неструктурной передачи управления. Кроме того, при структурном варианте получена самая простая программа, что в соответствии с рекомендациями структурного программирования существенно сокращает количество возможных ошибок.

Рассмотрим еще один пример, при разработке программы которого *теоретически* можно использовать оператор continue.

Пример 3.9. Разработать программу, которая должна вводить и суммировать 10 целых чисел, не превышающих 50. При вводе отрицательного числа она должна выдавать предупреждение, игнорировать число и ожидать ввода правильного значения.

В данном случае в программе нельзя использовать счетный цикл, так как количество его повторений заранее неизвестно, поэтому используем итерационный цикл, конкретно цикл-пока. Алгоритм решения задачи представлен на рис. 3.24. Этот алгоритм может быть реализован с использованием оператора goto, процедуры continue или оператора ветвления if. Последний вариант является структурным.

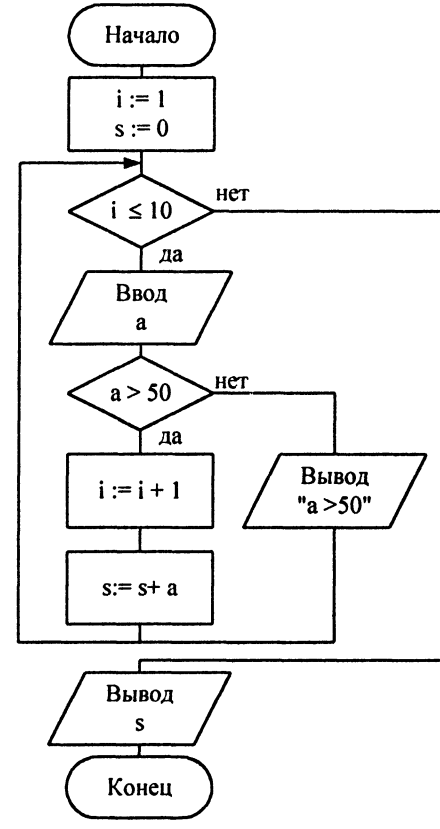


Рис. 3.24. Схема алгоритма определения суммы 10 чисел, не превышающих 50

Рассмотрим эти варианты.
Вариант с использованием оператора goto:

```
Program ex;
Var i,s,a:integer;
Label cycl;
Begin
    i:=1;
    s:=0;
    while i<=10 do
      begin
         Read(a);
         if a>50 then
            begin
               WriteLn('Число не должно превышать 50');
               goto cycl;    {передача управления на конец цикла}
            end;
         s:=s+a;   {операторы, которые необходимо обойти}
         i:=i+1;
  cycl: end;
 WriteLn(s);
End.
```

Вариант с использованием процедуры continue:

```
Program ex;
Uses crt;
Var i,s,a:integer;
Begin
 i:=1;
 S:=0;
 while i<=10 do
   begin
    Read(a);
    if a>50 then
      begin
        WriteLn('Число не должно превышать 50');
        continue;   {передаем управление на следующую итерацию}
      end;
    s:=s+a;
    i:=i+1;
  end;
 WriteLn(s);
End.
```

Структурный вариант:

```
Program ex;
Uses crt;
Var i,s,a:integer;
Begin
 i:=1;
 S:=0;
 while i<=10 do
  begin
    Read(a);
    if a>50 then
            WriteLn('Число не должно превышать 50')
    else
      begin
        s:=s+a;
        i:=i+1;
      end
  end;
 WriteLn(s);
End.
```

Структурная реализация алгоритма в данном случае является самой простой. Использование вариантов с оператором goto и процедурой continue не целесообразно. Применение continue в аналогичных ситуациях может быть оправдано только при большом количестве пропускаемых операторов.

Еще один тип алгоритмов, для реализации которых неопытные программисты обычно используют оператор goto, можно продемонстрировать на следующем примере.

Пример 3.10. Разработать программу, которая многократно вычисляет значение функции по вводимому пользователем значению аргумента. После выдачи результата программа должна спрашивать, нужно ли продолжить работу. Если необходимость продолжения работы существует, пользователь вводит «y», иначе – «n». Соответственно в первом случае работа с программой продолжается, а во втором – программа завершает работу.

В основе программы лежит циклический процесс. По типу это цикл-до (рис. 3.25), однако его достаточно часто пытаются реализовать с использованием goto. Чтобы не делать таких ошибок, необходимо запомнить следующее: если в схеме алгоритма присутствует возврат управления на уже выполненный фрагмент, то необходимо выделить цикл и соответственно определить условие выхода из цикла.

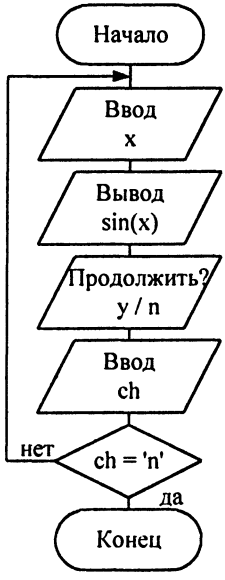


Рис. 3.25. Схема алгоритма вычисления значения функции

Ниже представлена реализация алгоритма с использованием цикла-до.

```
Program ex;
Var x:real;ch:char;
Begin
  repeat
    Write('Введите x: ');
    ReadLn(x); {ни в коем случае не Read, так как
       далее идет ввод символа (см. параграф 2.6)}
    WriteLn('Результат ', sin(x):8:4);
    WriteLn('Продолжить? (y/n)');
    ReadLn(ch);
  until ch='n';
End.
```

Окончательно можно сделать следующие выводы.

1. Оператор безусловной передачи управления goto в программах на Borland Pascal желательно вообще не использовать. Если у вас получился неструктурный алгоритм, при реализации требующий goto, попробуйте преобразовать его в структурный. Такое ограничение связано с тем, что использование goto, как правило, приводит к появлению в программе большого количества ошибок [3, 4, 9].

2. Процедуры неструктурной передачи управления break и continue могут быть полезны, однако при их использовании необходимо четко представлять себе, куда будет передано управление. Цикл, в котором использованы эти процедуры – это место повышенной вероятности наличия ошибок. Он должен специально тестироваться.

Процедуры Halt и Exit обычно используют для аварийного завершения соответственно программы или подпрограммы, поэтому их присутствие в программе практически не влияет на ее структурность.

Задания для самопроверки

Задание 1. Разработайте программу, которая определяет сумму первых трех отрицательных чисел последовательности значений, рассчитанных по формуле $y_n = x - \sin x$ при заданных диапазоне [a,b] изменения x и шаге h.

Задание 2. Разработайте программу, которая строит таблицу значений функции $y = (\ln x) / \operatorname{tg} x$ при заданных диапазоне [a, b] изменения x и шаге h. Если значение функции в очередной точке не существует, то в соответствующей строке таблицы выведите сообщение «значение не существует».

4. СТРУКТУРНЫЕ ТИПЫ ДАННЫХ

Достаточно часто возникает необходимость программирования обработки однотипных данных: таблиц, текстов, множеств и т.д. Для их представления используют структурные типы данных. В Borland Pascal определены следующие структурные типы данных:

- *массивы* – для представления однотипных или табличных данных;
- *строки* – для представления символьной (текстовой) информации;
- *множества* – для представления абстрактных математических множеств;
- *записи* – для представления таблиц с данными различных типов.

4.1. Массивы

Массив – это упорядоченная совокупность однотипных данных. Каждому элементу массива соответствует один или несколько *индексов*, определяющих положение элемента в массиве. Индексы образуют упорядоченные последовательности. Синтаксическая диаграмма объявления массива представлена на рис. 4.1.

Тип индекса определяет его допустимые значения. В качестве типа индекса может быть указан любой порядковый тип (boolean, char, integer, перечисляемый тип, а также диапазоны этих типов), кроме типа longint и его производных.

В зависимости от количества типов индексов различают: одномерные, двумерные, трехмерные и n-мерные массивы. Двумерные массивы обычно называют *матрицами*, считая первый индекс – номером строки, а второй – номером столбца.

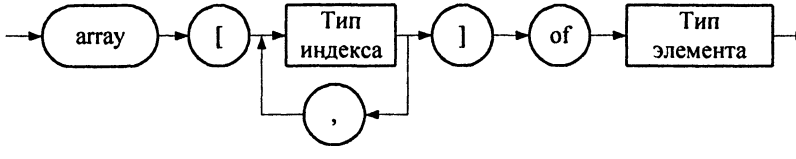


Рис. 4.1. Синтаксическая диаграмма <Объявление массива>

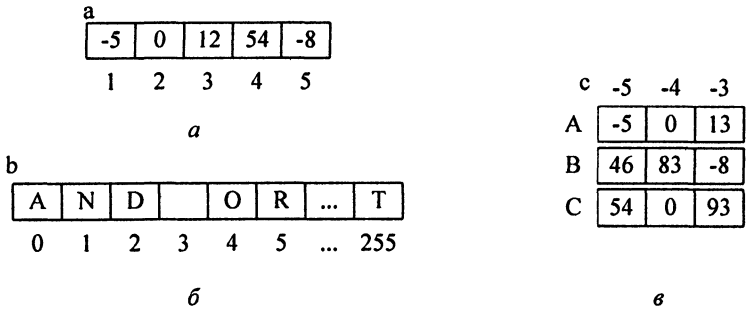


Рис. 4.2. Одномерный массив из 5 целых чисел (*а*), одномерный массив из 256 символов (*б*), матрица из 9 чисел (*в*)

Тип элементов массива – любой допустимый в Borland Pascal тип (в том числе и массив), кроме файла.

Объявление переменных типа массив выполняется двумя способами:

- в операторе объявления переменных, например:

```
Var a:array[1..10] of integer;  {массив из 5 целых чисел, см. рис. 4.2, а}
    b:array[byte] of char;      {массив из 256 символов, индекс элемента
                                 массива изменяется от 0 до 255, см. рис. 4.2, б}
    c:array['A'..'C',-5..-3] of byte; {матрица из 9 чисел, см. рис. 4.2, в}
    d:array['A'..'C'] of array[-5..-3] of byte;  {матрица из 9 чисел, по
                                 структуре эквивалентная предыдущей}
```

- с предварительным объявлением типа, например:

```
Type mas=array[1..10] of integer;  {объявляем тип}
Var  a:mas;                        {объявляем переменную}
```

Ограничения на количество индексов в Borland Pascal нет. Однако суммарная длина массива не должна превышать 65537 байт.

Значения элементов массива в программе можно определить тремя способами. Во-первых, массив может быть инициализирован с использованием типизированных констант или просто присваиванием значений элементам. Во-вторых, элементы массива могут быть введены с клавиатуры или из файла (см. главу 6). В-третьих, элементы массива могут быть вычислены, например, сгенерированы с использованием датчика случайных чисел, рассчитаны по заданным формулам и закономерностям, а также скопированы из другого массива.

Инициализация массивов. Для объявления инициализированных массивов в Borland Pascal используют типизированные константы. При этом соответствующие значения указывают в скобках через запятую. Значения элементов многомерных массивов перечисляют в порядке возрастания индексов

справа налево, заключая в скобки каждый подмассив. Для матриц такой порядок соответствует построчному указанию значений. Например:

```
Const a: array[1..5]of real = (0,-3.6,7.8,3.789,5.0);
      b: array[boolean, 1..5,]of real = ((0,-3.6,7.8,3.789,5.0),
                                         (6.1,0,-4.56,8.9,3.0));
         {массив будет инициализирован следующим образом:
          b false,1 = 0, b false,2= -3.6, b false,3 = 7.8, ... , b true,1 = 6.1, и т.д.}
      c:array[1..3,0..1,-2..1] of byte = (((3,6,9,6),(0,4,3,9)),
                                         ((5,7,3,1),(45,8,0,2)),
                                         ((5,9,2,3),(1,5,8,4))); ...
```

Операции над массивами. Над массивом в целом определена единственная операция – операция присваивания.

Присваивание массивов заключается в копировании элементов одного массива в другой. Эту операцию можно выполнять только над массивами одного типа.

Массивы считаются совпадающими по типу, если они объявлены через запятую в одной строке, например:

```
Var a, b:array[boolean] of real;
... a:=b;...
```

или, если вначале объявлен тип массива, а затем массивы этого типа:

```
Type mas=array[boolean] of real;
Const a:mas=(3.6,-5.1);
Var b:mas;
... b:=a;...
```

Доступ к элементам массива. Работа с массивом, как правило, сводится к действиям над его элементами. Для обращения к конкретному элементу массива необходимо указать имя массива и значения индексов элемента в квадратных скобках через запятую, например:

```
Var a:array[char,boolean] of real;  {объявляем матрицу}
... a['A',true]:=5.1; ...  {присваиваем значение элементу a A,true}
```

Значения индексов можно указать непосредственно литералом, например a[3], или *косвенно*, указав идентификатор переменной, которая содержит значение индекса, например a[i] (рис. 4.3).

Косвенное задание индексов позволяет реализовывать последовательную обработку элементов массивов. Причем, поскольку интервал изменения индекса определен при объявлении массива, для этого обычно применяют

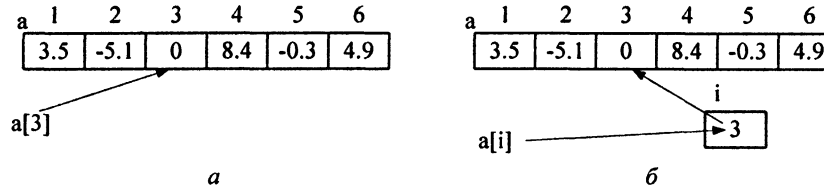


Рис. 4.3. Прямая (*а*) и косвенная (*б*) адресация элементов массива

циклы с заданным количеством повторений. Параметр же цикла используют в качестве переменной косвенной адресации массива, например:

```
Var a:array[1..6] of integer; ...
for i:=1 to 6 do a[i]:=i; ... {при i=1  a1 присваивается 1,
                               при i=2  a2 присваивается 2,
                               при i=3  a3 присваивается 3 и т.д.}
```

Количество переменных, необходимых для косвенной адресации массивов, совпадает с размерностью массива. Так, для работы с матрицами используют две переменные, хранящие индексы: одну – для хранения номеров строк, а вторую – номеров столбцов.

Примечание. Из многомерных массивов допускается *выделять подмассивы*, отбрасывая индексы, записанные справа, и оставляя индексы, определяющие данный подмассив. Так, из матрицы можно выделить строку, но нельзя – столбец, например:

```
Type mas=array[boolean] of real;  {массив из двух вещественных чисел}
Const  a:array[1..2]of mas=((3.6,-5.1),(7.0,-4.2)); {матрица из четырех
                                                     вещественных чисел}
Var    b:mas;
Begin  b:=a[1];...{в массив b скопирована первая строка матрицы a}
```

Ввод-вывод массивов. Ввод-вывод массивов выполняют поэлементно, используя циклы с заданным числом повторений, например:

```
Var a:array[1..5] of real;
Begin
   for i:=1 to 5 do Read(a[i]); {осуществляем ввод массива}
   ReadLn;    {очищаем буфер ввода, чтобы далее значения вводились
               со следующей строки (см. параграф 2.6)}
```

Значения элементов массива вводят в порядке обращения к ним из цикла, например для цикла, показанного выше: a_1, a_2, a_3, a_4, a_5. Эти значения могут задаваться в одной строке через пробел или с нажатием клавиши Enter после ввода одного или нескольких чисел.

Например:

а) **2_-6_8_56_34** ↵ – символ «↵» означает нажатие клавиши Enter;

б) **2** ↵
-6 ↵
8 ↵
56 ↵
34 ↵

При выполнении операций ввода-вывода матриц и массивов большой размерности целесообразно вводить и выводить значения построчно.

Например:

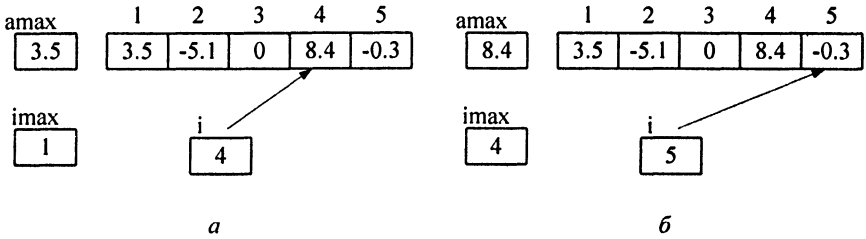
```
Var a:array[1..5, 1..7] of real;   {матрица а из 5 строк по 7 элементов}
Begin
  for i:=1 to 5 do      {цикл ввода строк массива: a1, a2, a3, a4, a5}
    begin
      for j:=1 to 7 do          {цикл ввода элементов i-й строки:}
            Read(a[i, j]);      {ai,1, ai,2, ai,3, ai,4, ai,5, ai,5, ai,7}
    ReadLn; {очищаем буфер ввода}
    end; ...
```

Пример 4.1. Разработать программу определения максимального элемента массива A(5) и его номера.

Вначале элементы массива необходимо ввести. Для выполнения этой операции используем цикл с заданным числом повторений.

Поиск максимального элемента выполним следующим образом. Запомним в качестве максимального, т. е. запишем в amax первый элемент и зафиксируем в imax его номер. Затем будем последовательно просматривать элементы массива, сравнивая их со значением, хранящимся в amax. Если очередной элемент больше значения в amax, то сохраняем его в качестве максимального в amax и запоминаем его номер в imax (рис. 4.4). Для организации последовательного просмотра используем цикл с заданным числом повторений, изменяя переменную цикла от 2 до 5, так как первый элемент мы уже учли. Просмотрев все элементы массива, найдем максимальный элемент и его номер. После этого необходимо вывести на экран исходный массив, максимальный элемент и его номер.

На рис. 4.5 представлена схема алгоритма программы. Поскольку операции ввода-вывода массивов выполняют однотипно, на схеме алгоритма соответствующих циклов, так же как и запросов на ввод данных, обычно не показывают. Вместо этого в схему вставляют блок операции ввода/вывода, в



а *б*

Рис. 4.4. Поиск максимального элемента массива:
а – состояние на момент проверки четвертого элемента массива; *б* – изменение текущего значения максимального элемента и его номера по результатам проверки четвертого элемента массива

котором указано имя массива и количество элементов, участвующих в операции. Ниже приведен текст программы.

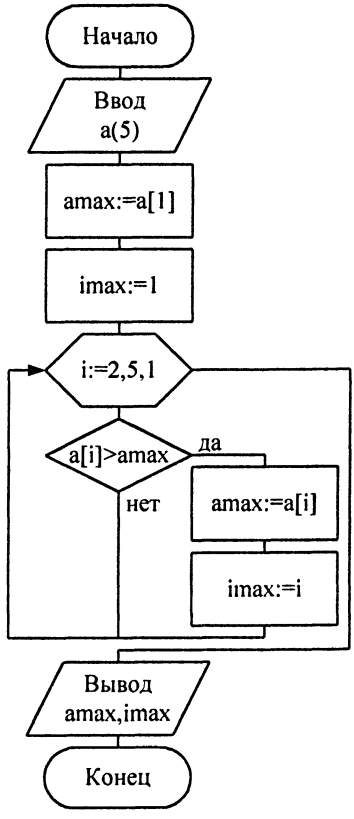


Рис. 4.5. Схема алгоритма поиска максимального элемента массива

```
Program ex;
 Var a:array[1..5] of real;
     amax:real;
     i, imax:byte;
 Begin
   {запрос на ввод массива}
   WriteLn('Введите 5 чисел:');
    {ввод элементов массива}
   for i:=1 to 5 do Read(a[i]);
   ReadLn;
    {поиск максимального элемента}
   amax:=a[1];
   imax:=1;
   for i:=2 to 5 do
       if a[i]>amax then
           begin
               amax:=a[i];
               imax:=i;
           end;
    {вывод массива}
   WriteLn('Исходные данные:');
   for i:=1 to 5 do Write(a[i]:5:2);
   WriteLn;
   {вывод результата}
  WriteLn('Максимальный элемент равен ',
       amax:5:2,', его номер равен ', imax);
End.
```

A	1	2	3	4	5
1	3.1	5.7	8.1	-0.7	3.6
2	4.3	6.8	-0.3	5.7	9.2
3	6.4	2.7	5.5	-5.3	2.7
4	5.1	-2.7	7.7	1.7	5.1

а

B	
1	19.8
2	25.7
3	12.0
4	16.9

б

Рис. 4.6. Исходные данные (*а*) и результат (*б*) примера 4.2

Пример 4.2. Разработать программу вычисления сумм элементов строк матрицы A(4,5). Полученные суммы записать в новый массив B.

Итак, нам задана матрица, имеющая 4 строки и 5 столбцов (рис. 4.6, *а*). Требуется сформировать одномерный массив B из четырех элементов, который будет содержать суммы элементов строк (рис. 4.6, *б*). Распечатать результат лучше так, чтобы суммы были выведены после соответствующей строки матрицы, как на рис. 4.6.

Программа должна начинаться со ввода матрицы. Основной цикл программы – цикл по строкам. Переменная цикла i в нем будет изменяться от 1 до 4. Для каждой i-й строки в этом цикле должно выполняться суммирование элементов. Суммирование будем осуществлять методом накопления, для чего перед суммированием обнулим соответствующий i-й элемент массива B, а затем в цикле выполним добавление элементов строки. После завершения цикла суммирования эту строку и ее сумму можно сразу выводить. На рис. 4.7 представлена схема алгоритма программы (пунктиром выделено суммирование элементов i-й строки). Ниже приведен ее текст.

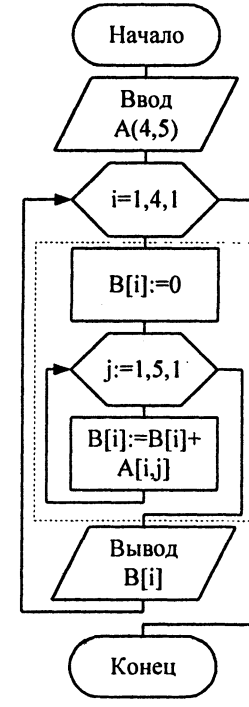


Рис. 4.7. Схема алгоритма программы нахождения сумм элементов строк

```
Program ex;
 Var A:array[1..4,1..5] of real;
     B:array[1..4] of real;
     i,j:byte;
 Begin
   WriteLn('Введите матрицу построчно:');
   for i:=1 to 4 do     {вводим матрицу}
       begin
         for j:=1 to 5 do Read(A[i,j]);
         ReadLn;
       end;
```

```
 WriteLn('Результаты:');
 for i:=1 to 4 do     {для каждой строки}
   begin
     B[i]:=0;     {обнуляем накапливаемую сумму}
      for j:=1 to 5 do B[i]:=B[i]+A[i,j];  {суммируем элементы строки}
      for j:=1 to 5 do Write(A[i,j]:7:2);      {выводим строку}
       WriteLn('  Сумма равна ',B[i]:7:2); {выводим сумму}
   end;
End.
```

Символьные массивы. Символьными называют массивы, элементами которых являются символы. Такие массивы традиционно использовались для представления символьной информации, например различных текстов. Обработка символьных массивов в Borland Pascal имеет некоторые особенности.

1. Объявляя символьный массив как типизированную константу, значения символов можно указывать поэлементно:

```
Const d:array[1..10] of char = ('0', '1', '2', '3', '4', '5', '6',' '7', '8', '9');
```

или целиком, используя строковую константу, длина которой должна строго соответствовать размеру массива:

```
Const d:array[1..10] of char = '0123456789'; ...
```

2. Присвоить значение символьному массиву также можно целиком, используя строковую константу, длина которой должна совпадать с длиной массива:

```
Var  S: array[1..11] of char; ...
     S:= 'Пример один';
```

3. При вводе элементы символьного массива нельзя разделять пробелами, так как пробел будет восприниматься как символ:

```
Var S: array[1..10] of char; ...
for i:=1 to 10 do  Read(S[i]);    {вводим строку «ABCDE FILN ↵»}
```

4. Символьный массив можно выводить поэлементно в цикле, как обычный одномерный массив, а можно – целиком, одним оператором Write или WriteLn:

```
WriteLn(S);  {вывод символьного массива одним оператором}
```

5. В операторе вывода допускается использование операции конкатенации (слияния) символьных массивов, обозначаемой символом «+». Результатом этой операции будет новый символьный массив, число элементов кото-

рого равно сумме размеров исходных массивов, а значениями элементов – элементы исходных массивов, последовательно записанные друг за другом:

WriteLn(st1 + ' ' + st2); {конкатенация символьных массивов}

Работа с одномерными символьными массивами осуществляется поэлементно, как с обычными массивами. Рассмотрим пример использования символьных массивов.

Пример 4.3. Дана строка не более 40 символов, состоящая из слов, разделенных пробелами, и завершающаяся точкой. Разработать программу удаления «лишних» пробелов. Лишними считать пробелы в начале строки, второй и более пробелы между словами и пробелы в конце строки.

Например:

Исходная строка: ___ABC__DE___FGH__.
Результат: ABC_DE_FGH.

Удалить пробелы в начале строки не трудно: просто не нужно переписывать пробелы до первого значащего символа из исходной строки в результирующую. Несколько сложнее дело обстоит с пробелами между словами, так как удалить нужно не все пробелы, а только повторяющиеся. Для решения задачи используем специальный признак «первый пробел». Этот признак будем устанавливать, встретив первый пробел, и гасить, встретив символ, отличный от пробела (на рис. 4.8). Используя этот признак, мы сможем отличить первый пробел, который необходимо перенести в массив результата от последующих, которые переносить не надо. Диаграмма показывает, что если в конце строки есть пробелы, то с использованием признака «первый пробел» мы получим в строке результата один лишний пробел. Поэтому после завершения обработки необходимо проверить признак, и если он установлен, удалить пробел, уменьшив длину строки на единицу. В программе, приведенной ниже, вместо этого на место пробела пишется точка. В том случае, если пробела в конце нет, для точки «добавляется» элемент.

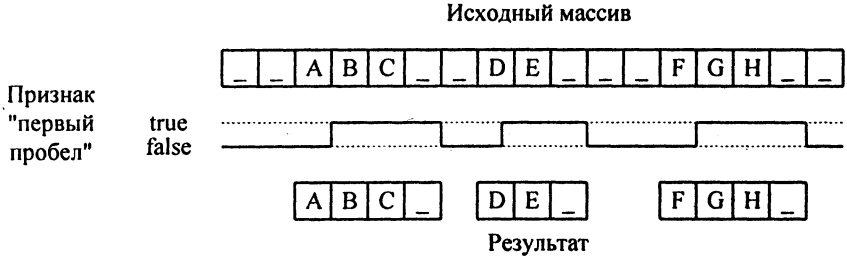


Рис. 4.8. Диаграмма установки и гашения признака «первый пробел»

```
Program Stroka;
  Var  i,j,n:byte;   key:boolean;
       s:array [1..41] of char;  {дополнительный символ – для точки}
  Begin
    WriteLn('Введите исходную строку длиной до 40 символов:');
    i:=1;   {вводим строку посимвольно до точки, но не более 40 символов}
    Read(s[i]);
     while (i<40) and (s[i]<>'.') do
      begin
        i:=i+1;
        Read(s[i]);
       end;
    ReadLn;
    if s[i]='.' then n:=i-1  else n:=i;    {определяем количество введенных
                                                    символов без точки}
    Write('Введенная строка:'); WriteLn(s);
    j:=0; {номер символа в строке результата}
    key:=false; {гасим признак "первый пробел"}
    for i:=1 to n do
      if s[i]=' ' then  {если обнаружен пробел}
        begin
          if key then  {этот пробел первый}
             begin
                key:=false; {дальше пойдут лишние пробелы}
                j:=j+1;     {пробел переписываем в результат}
                s[j]:=s[i];
             end;
        end
      else {символ}
        begin
          key:=true;   {устанавливаем признак "первый пробел"}
          j:=j+1;
          s[j]:=s[i];   {символ переписываем в результат}
        end;
     if key then j:=j+1;  {если пробела в конце нет, то увеличиваем
                                                    длину для записи точки}
     s[j]:='.';     {записываем точку}
     WriteLn('Преобразованная строка ');
     for i:=1 to j do Write(s[i]);  WriteLn;
End.
```

4.2. Практикум. Обработка одномерных массивов

Все операции, которые приходится выполнять над элементами одномерных массивов, можно разбить на следующие классы:

- последовательная обработка элементов массивов;
- переформирование массивов;
- одновременная обработка нескольких массивов или подмассивов;
- поиск элементов массива по заданным критериям.

Как правило, в реальной жизни задачи, включающие только эти операции, встречаются редко. Однако программирование более сложной обработки включает элементы указанных операций.

Для выполнения перечисленных выше операций разработаны соответствующие приемы. Рассмотрим наиболее распространенные приемы программирования обработки одномерных массивов.

Последовательная обработка элементов массивов. Особенностью операций данного класса является то, что количество обрабатываемых элементов массива и шаг изменения индексов известны. Это позволяет для выполнения операции использовать счетный цикл, через переменную которого обеспечивается косвенный доступ к элементам. Если просматриваются все элементы массива, то обращение выполняют, используя переменную цикла в качестве индекса, а если с заданным шагом, то для адресации элементов строится выражение, в которое входит переменная цикла, например 2*i+1. Однако возможно применение и других типов циклов.

Примерами задач, требующих выполнения последовательной обработки, являются: ввод и вывод массивов, нахождение сумм элементов как целиком массива, так и его определенной части, произведения элементов, среднего арифметического, среднего геометрического, подсчет количества элементов, отвечающих определенному условию или обладающих некоторыми признаками, а также их суммы, произведения и т.д. Кроме того, к этой группе могут быть отнесены задачи формирования значений и замены значений всех элементов значениями, подчиняющимися определенному закону.

Пример 4.4. Разработать программу определения среднего арифметического значений положительных элементов целочисленного массива A(n), где $n \le 40$, кратных трем.

Количество элементов массива в условии не определено, но ограничено. Для реального массива, который будет обрабатываться программой, это количество естественно должно быть известно. Следовательно, прежде чем вводить элементы массива, можно запросить у пользователя ввод количества элементов n. Массив при этом будем описывать на максимально возможное количество элементов, так как в Borland Pascal *выделение памяти под массивы реализовано статически*, т. е. память под массивы, строки и другие структурные типы данных резервируется на этапе компиляции программы. Если

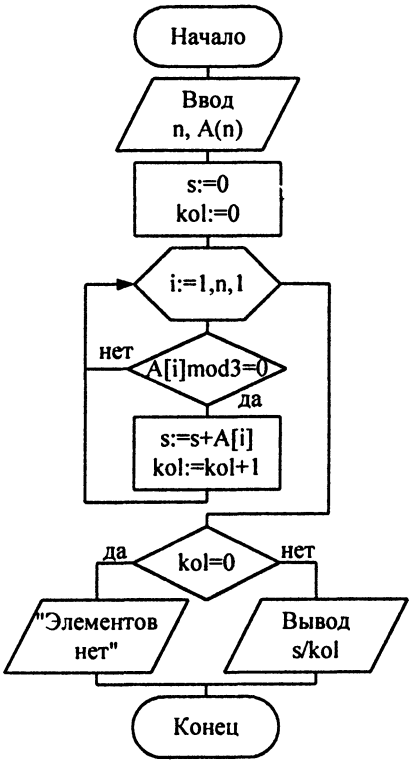


Рис. 4.9. Алгоритм определения среднего арифметического элементов массива, кратных 3

необходимо реализовать выделение памяти во время выполнения программы, то используют указатели (см. главу 7).

Для решения самой задачи необходимо двум вспомогательным переменным s и kol, которые будут использованы для накопления суммы требуемых элементов и их количества, присвоить нулевые значения. После этого осуществляют перебор всех элементов, и если очередной удовлетворяет условию, его значение добавляют к содержимому переменной s, а значение переменной kol увеличивают на единицу. После просмотра всего массива среднее арифметическое может быть определено делением накопленной суммы на количество найденных элементов. (Обратите внимание, что среднее арифметическое определяется *после просмотра всего* массива, так как до этого момента у нас не полные сумма и количество элементов.) Положительных элементов в массиве может не быть, что необходимо предусмотреть в программе. Алгоритм решения задачи представлен на рис. 4.9. Ниже представлен текст программы.

```
Program ex;
  Var a:array[1..40] of integer;  s, kol, i, n:integer;
Begin
    WriteLn('Введите количество элементов массива <=40');
    Readln(n);   {вводим количество элементов массива}
    WriteLn('Введите ', n,' элементов массива:');
    for i:=1 to n do  Read(a[i]);
    ReadLn; {вводим массив}
    WriteLn(' Исходный массив');
    for i:=1 to n do  {выводим массив по 10 элементов в строке}
      if (i mod 10)=0 then WriteLn(a[i]:5)
      else  Write(a[i]:5);
    WriteLn;
    kol:=0;      {обнуляем количество элементов, кратных 3}
    s:=0;    {обнуляем начальное значение суммы элементов}
```

```
for i:=1 to n do
  if (a[i] mod 3)=0  then  {если элемент кратен 3}
        begin
          kol:=kol+1;  {увеличиваем количество на 1}
          s:=s+a[i];  {добавляем элемент к сумме}
        end;
     if kol=0 then    {если количество элементов равно нулю, то}
        WriteLn('Элементов, удовлетворяющих условию, нет')
     else WriteLn(' Среднее арифметическое', kol:3,
                  ' элементов, кратных 3 =',(s/kol):7:2);
End.
```

Пример 4.5. Для целочисленного массива A(n), где n≤10, разработать программу, определяющую количество отрицательных элементов среди элементов, стоящих в массиве на местах с четными номерами. Элементы массива пронумерованы, начиная с единицы.

Для решения задачи необходимо проверить все элементы массива, имеющие значение индекса 2, 4, 6 и т.д. Поскольку количество повторений цикла несложно определить: оно равно n div 2, используем счетный цикл, выразив индекс элемента через переменную цикла: 2*i.

Для подсчета количества искомых элементов вводим вспомогательную переменную kol, начальное значение которой равно 0. Если проверяемый элемент отрицателен, то значение переменной kol увеличивается на единицу. После обхода всего массива переменная kol будет содержать искомое значение.

```
Program ex;
 Var  a:array[1..10] of integer;   kol, i, j, n:integer;
 Begin
   WriteLn('Введите количество элементов массива <=10');
   ReadLn(n);   {вводим количество элементов}
   WriteLn('Введите ',n,' элементов массива');
   for i:=1 to n do Read(a[i]); {вводим массив}
   WriteLn('Введенный массив');
   for i:=1 to n do Write(a[i]:3); WriteLn; {выводим исходный массив}
   kol:=0; {обнуляем количество отрицательных элементов}
   for i:=1 to n div 2 do
       if a[2*i]<0  then                {если элемент отрицателен, то}
                       kol:=kol+1;  {увеличиваем количество на 1}
   if kol=0 then
     WriteLn('Отрицательных элементов на четных местах нет')
   else
     WriteLn('Количество отрицательных элементов kol=', kol:3);
End.
```

Переформирование массива. Переформирование массива предполагает изменение порядка элементов посредством их перемещения, удаления или вставки. При переформировании массива его размер может изменяться, а может оставаться без изменений.

Следует учесть, что вставка или удаление элементов осуществляется за счет сдвига всех элементов той части массива, которая расположена после удаляемого или вставляемого элемента. В зависимости от конкретных условий иногда такой сдвиг может быть совмещен с последовательной обработкой оставшихся элементов. В остальных случаях он должен выполняться в специальном вложенном цикле.

Пример 4.6. Дан массив A(n), где n ≤ 10, и число B. Разработать программу удаления из массива всех элементов, меньших заданного значения B.

Для получения требуемого результата необходимо последовательно перебрать все элементы массива. Если очередной элемент меньше B, то его следует удалить, сдвигая на один элемент все элементы, расположенные после него. Если в массиве были найдены элементы меньше B, то полученный массив будет иметь меньший размер.

Для решения этой задачи можно предложить два алгоритма: с отдельной и встроенной реализацией сдвига элементов.

Первый алгоритм представляет собой решение задачи «в лоб»: нашли элемент – исключили его, сдвинув остальные элементы, нашли следующий – исключили его и т. д. (рис. 4.10).

Второй алгоритм базируется на том, что нам не требуется, чтобы в момент анализа элементов массива все элементы располагались подряд без пропусков. Следовательно, можно, используя две переменные для косвенной адресации, отслеживать как исследуемые элементы исходного массива, так и уже полученные элементы результирующего массива. Например, пусть переменная i изменяется в цикле и обеспечивает адресацию следующего анализируемого элемента. Тогда переменную k увеличиваем на едини-

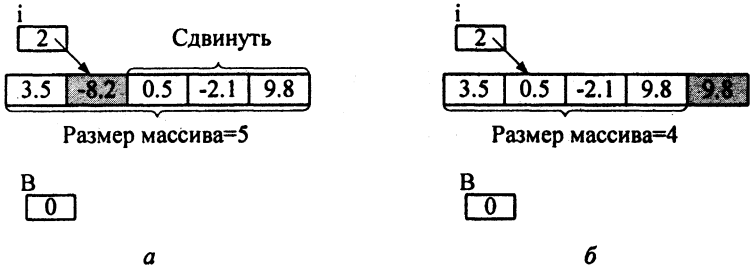


Рис. 4.10. Этапы выполнения программы:

а – обнаружен элемент меньше заданного; *б* – остальные элементы сдвигаем на его место, уменьшаем размер массива и вновь анализируем элемент с тем же номером i

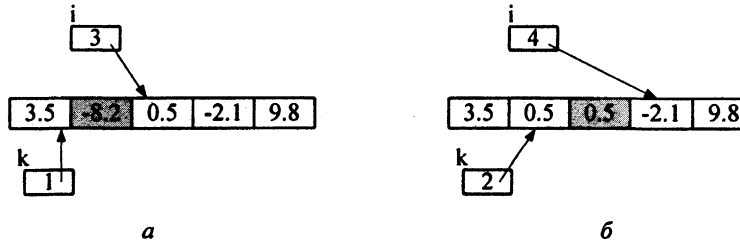


Рис. 4.11. Этапы выполнения программы:
а – пропускаем элемент, который необходимо удалить, и переходим к анализу следующего; *б* – оставляем в массиве следующий элемент: увеличиваем k на единицу и переписываем элемент его на k-е место, затем переходим к анализу следующего элемента

цу, если обнаружен еще один элемент, который необходимо оставить в массиве. При этом оставляемый элемент переписывается на место, указанное данной переменной (рис. 4.11). После завершения цикла просмотра массива переменная k содержит размер массива. Элементы, содержащиеся в остальной части массива, на экран не выводятся.

Если в массиве не содержится элементов, удовлетворяющих заданному условию, то при просмотре массива мы перепишем элементы сами в себя, а значение k будет равно n.

Алгоритмы с отдельной и встроенной реализацией сдвига элементов показаны на рис. 4.12.

Реализуем второй более простой вариант.

```
Program ex;
 Var a:array[1..10] of integer;
     B, i, k, n:integer;
 Begin
  WriteLn('Введите количество элементов n <= 10');
  ReadLn(n);
  WriteLn('Введите ',n,' элементов массива');
  for i:=1 to n do Read(a[i]);
  ReadLn;   {вводим массив}
  WriteLn('Введите B:');
  ReadLn(B);      {вводим B}
  WriteLn(' Исходный массив ');
  for i:=1 to n do  Write(a[i]:5);
  WriteLn;{выводим исходный массив}
  k:=0;  {пока не найдено ни одного элемента массива}
```

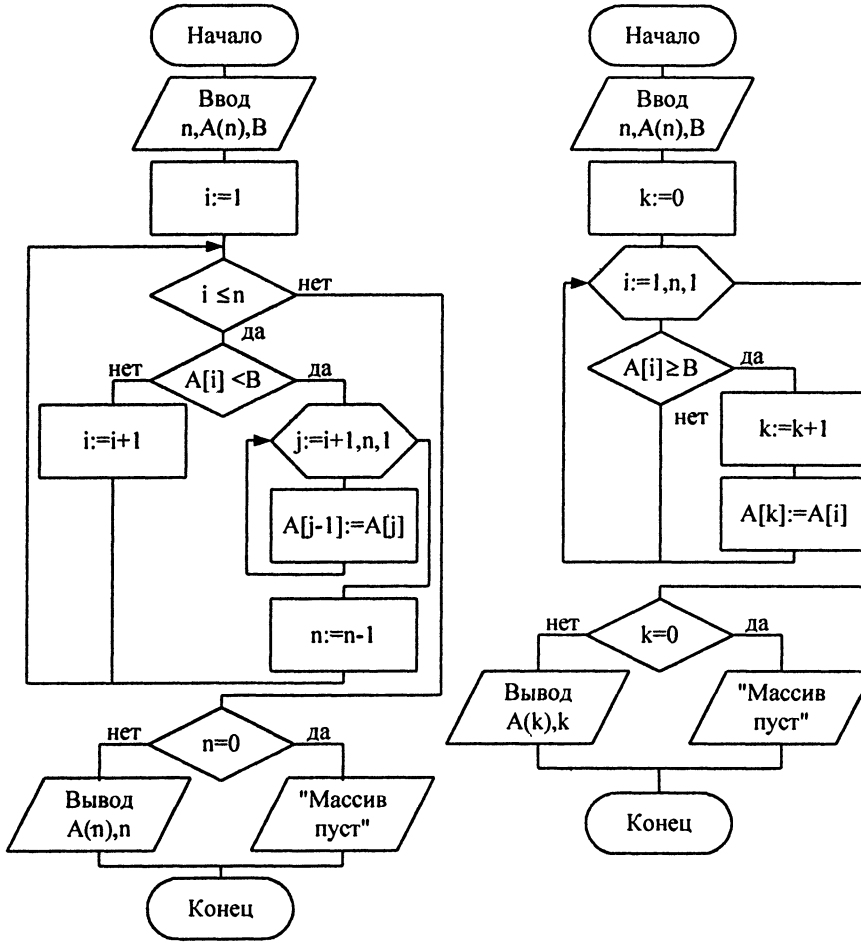


Рис. 4.12. Два алгоритма удаления из массива элементов меньше B:
а – с отдельным циклом сдвига; *б* – со встроенным циклом сдвига

```
for i:=1 to n do
  begin
    if A[i]>=B then
       begin
            k:=k+1;
            A[k]:=A[i];
       end
  end;
if k=0 then WriteLn('Все элементы вычеркнуты. Массив пуст.')
  else
```

```
    begin
      WriteLn(' Результирующий массив из ', k, ' элементов:');
      for i:=1 to k do Write(a[i]:5);
      WriteLn;
    end;
End.
```

В некоторых случаях при переформировании массива его размер не меняется. Примерами подобной обработки служат перестановки различного характера (см., например, сортировку массивов в параграфе 4.3).

Одновременная обработка нескольких массивов или подмассивов. К этому классу относятся задачи слияния массивов, переписи элементов одного массива, отвечающих определенному условию, в другой, формирования нового массива из элементов исходного в соответствии с заданным законом преобразования и т. п. Особенностью таких задач является то, что у каждого массива свой индекс, свой закон и диапазон его изменения. При программировании таких действий можно использовать как счетные, так и итерационные циклы, причем выбор зависит от закона изменения индексов. При этом, если индексы обрабатываемых массивов связаны, то их получают один из другого (см. пример 4.2), а если не связаны, то формируют независимо.

Пример 4.7. Разработать программу формирования из массива целого типа A(n), где $n \le 40$, нового массива B, содержащего только положительные элементы массива A.

Для решения этой задачи необходимо перебрать все элементы массива A, выбирая и записывая в массив B только положительные элементы. В данном случае индексы массивов *не связаны*. Для просмотра элементов массива A используем счетный цикл с индексом i, а для обращения к элементам нового массива используем индекс k, меняющийся только при записи нового элемента массива. Размер формируемого массива должен быть не меньше исходного, так как все элементы массива могут оказаться положительными.

```
Program ex;
  Var a,b:array[1..40] of integer;
      i, k, n:integer;
  Begin
     WriteLn('Введите количество элементов массива <=40');
     ReadLn(n);
     WriteLn('Введите ',n,' элементов массива A');
     for i:=1 to n do Read(a[i]);
     ReadLn;
     WriteLn('Исходный массив:');
     for i:=1 to n do  Write(a[i]:3);
     WriteLn;
```

```
k:=0;{начальное значение индекса формируемого массива}
for i:=1 to n do
   if a[i]>0  then {если элемент > 0}
     begin
        k:=k+1; {изменение индекса формируемого массива}
        b[k]:=a[i]; {перепись найденного элемента}
     end;
if k=0 then
        WriteLn('В массиве А нет положительных элементов.')
else
  begin
       WriteLn(' массив результат B');
       for i:=1 to k do Write(b[i]:3); WriteLn;
  end;
End.
```

Поиск элементов массива по заданным критериям. Примерами подобного рода задач могут служить поиск первого отрицательного, первого положительного и любого первого элемента, отвечающего некоторому условию, а также поиск единственного или определенного количества элементов, равных некоторому конкретному значению. Особенность задач этого класса в том, что нет необходимости просматривать весь массив. Просмотр можно закончить сразу, как только требуемый элемент будет найден. Однако в худшем случае для поиска элемента требуется просмотреть весь массив, причем нужного элемента в нем может не оказаться.

Существует несколько методов поиска. Самый простой заключается в последовательном просмотре элементов массива. Если массив не очень большой, затраты времени линейного поиска не столь заметны. Но при солидных объемах информации время поиска становится критичным. Поэтому существуют методы, позволяющие уменьшить время поиска, например двоичный поиск, который применяется только, если элементы массива сортированы по возрастанию или убыванию (см. пример 4.19).

Чаще всего при программировании поисковых задач используют циклы-до или циклы-пока, в которых условие выхода формируется из двух условий (см. параграф 3.6): первое условие – пока искомый элемент не найден, а второе – пока есть элементы массива. После выхода из цикла осуществляют проверку, по какому из условий произошел выход.

Пример 4.8. Разработать программу, определяющую первый отрицательный элемент массива.

Для решения задачи необходимо разработать поисковый цикл, т.е. организовать последовательный просмотр массива, пока не будет обнаружен первый отрицательный элемент. Из материала параграфа 3.6 известно, что эту операцию можно выполнить структурно и неструктурно.

Неструктурный алгоритм, в котором просмотр осуществляется с помощью счетного цикла, а выход обеспечивается операторами goto или break, рассматривать не будем.

Реализуем структурный алгоритм, в котором для просмотра элементов используется цикл-пока со сложным условием: пока элементы не отрицательны и индекс элемента не вышел за границы массива. Элемент, на котором прервался цикл, если его индекс не превышает размера массива, и есть искомый.

```
Program ex;Var a:array[1..100] of integer;
      i,j,n:integer;
 Begin
  WriteLn('Введите количество элементов n <= 100');
  ReadLn(n);
  WriteLn('Введите ',n,' элементов массива');
  for i:=1 to n do  Read(a[i]);
  ReadLn;
  WriteLn(' Исходный массив ');
  for i:=1 to n do Write(a[i]:5);  WriteLn;
  i:=1;  {начальное значение индекса массива}
  while (a[i]>=0) and (i<n) do i:=i+1; {пока элемент не отрицателен
                и индекс меньше n – переходим к следующему элементу}
  if i<=n then
       WriteLn('Первый отрицательный элемент ',a[i]:5,
                                    ' имеет индекс  ',i:4)
   else WriteLn('Таких элементов в массиве нет.');
End.
```

Задания для самопроверки

Задание 1. Дан одномерный массив вещественных чисел A(n), где n ≤ 50. Разработайте программу, формирующую новый массив B из элементов массива A, которые превышают среднее арифметическое элементов массива A, стоящих на местах с четными индексами. Выведите среднее арифметическое значение элементов массива A, исходный и сформированный массивы.

Задание 2. Дан одномерный целочисленный массив C(n), где n ≤ 40, содержащий как положительные, так и отрицательные элементы. Разработайте программу, которая определяет номер первого отрицательного элемента, по абсолютной величине превышающего максимальный элемент этого массива. Выведите массив C, а также номер найденного элемента, или соответствующее сообщение, если такого элемента нет.

Задание 3. Разработайте программу, которая формирует массив B(n), n ≤ 30, содержащий элементы целого типа в диапазоне от –20 до 130, используя датчик слу-

чайных чисел. В сформированном массиве определите количество и среднее арифметическое положительных и отрицательных элементов массива. Переменной логического типа Flag присвоить True, если среднее арифметическое отрицательных чисел по абсолютной величине больше среднего арифметического положительных чисел, и False, если нет. Выведите массив B, а также все найденные в программе величины.

Задание 4. Дан массив T(n), $n \leq 20$, вещественного типа. Разработайте программу, которая вычисляет произведение максимального по абсолютной величине элемента заданного массива на его же первый отрицательный элемент, если таковой имеется. Выведите исходный массив и произведение, или сообщение о невозможности вычисления произведения.

Задание 5. Дан массив D(n), $n \leq 10$, вещественного типа. Разработайте программу, которая вычисляет сумму трех первых положительных элементов заданного массива. Если таких элементов нет, программа должна выдавать соответствующее сообщение. Выведите на печать исходный массив и искомую сумму.

4.3. Практикум. Сортировка массивов. Оценка вычислительной сложности алгоритма

Сортировка – это процесс упорядочивания информации по определенному признаку. Цель сортировки – облегчение последующего поиска элементов. Это почти универсальный вид обработки информации, с которым мы встречаемся в жизни повсеместно.

Существует огромное количество методов сортировки и, соответственно, алгоритмов их реализации. Часть из этих алгоритмов в некотором смысле оптимальна, другие имеют свои достоинства и недостатки. Поэтому, прежде чем использовать алгоритм, реализующий какой-либо метод, следует выполнить анализ его производительности в конкретных условиях.

В качестве оценки производительности методов обычно используют *функциональную зависимость* времени работы программы от размерности исходного массива t(n). При анализе алгоритмов в первую очередь интерес представляет характер зависимости при достаточно больших значениях размерности задачи ($n \to \infty$).

В математике характер зависимости часто определяют ее порядком. *Порядком* некоторой функции t(n) при достаточно больших n называют другую функцию g(n), такую, что

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \text{const} \neq 0.$$

Это обозначается как $t(n) = O[g(n)]$.

Например, для полинома $f(n) = 2n^4 - 3n^3 + 5n - 6$, порядком является полином n^4, или $f(n) = O(n^4)$, так как

$$\lim_{n\to\infty} \frac{2n^4 - 3n^3 + 5n - 6}{n^4} = 2.$$

В программировании порядок зависимости времени работы программы, реализующей некоторый метод, от размерности исходных данных n называют *вычислительной сложностью данного метода*. Так, вычислительная сложность O(const) означает, что время решения задачи с использованием данного метода не зависит от размерности задачи, O(n) – время работы пропорционально размерности задачи, $O(n^2)$ – время работы пропорционально квадрату размерности задачи и т. д.

Примечание. В некоторых случаях целесообразно различать вычислительную сложность метода и его конкретной реализации, так как неудачная реализация может существенно ухудшить предполагаемую вычислительную сложность метода.

Временную сложность можно оценить, используя в качестве единиц измерения временные единицы (мкс, с и т. д.), а можно – используя *время выполнения* основных, характеризующих процесс *операций*, количество которых соответствует количеству итераций (повторений) цикла, например, операций сравнения, операций пересылки. Время выполнения этих операций можно считать постоянным. Следовательно, функциональная зависимость количества выполняемых операций от размерности задачи по характеру будет совпадать с временной зависимостью.

На практике интересны методы сортировки, которые позволяют экономно использовать оперативную память, поэтому целесообразно рассмотреть только методы, не требующие использования дополнительных массивов. Такие методы в практике программирования называют *прямыми*. Самыми простыми из прямых методов являются:

- метод выбора;
- метод вставки;
- метод обменов (метод пузырька).

Рассмотрим эти методы на конкретном примере.

Пример 4.9. Разработать программу сортировки элементов массива A(n), где $n \le 20$, используя метод выбора, метод вставки и метод обменов. Оценить эффективность применения указанных методов.

Метод выбора. Сортировка посредством выбора представляет собой один из самых простых методов сортировки. Он предполагает такую последовательность действий.

Сначала находим минимальный элемент массива. Найденный элемент меняем местами с первым элементом. Затем повторяем процесс с n-1 элемен-

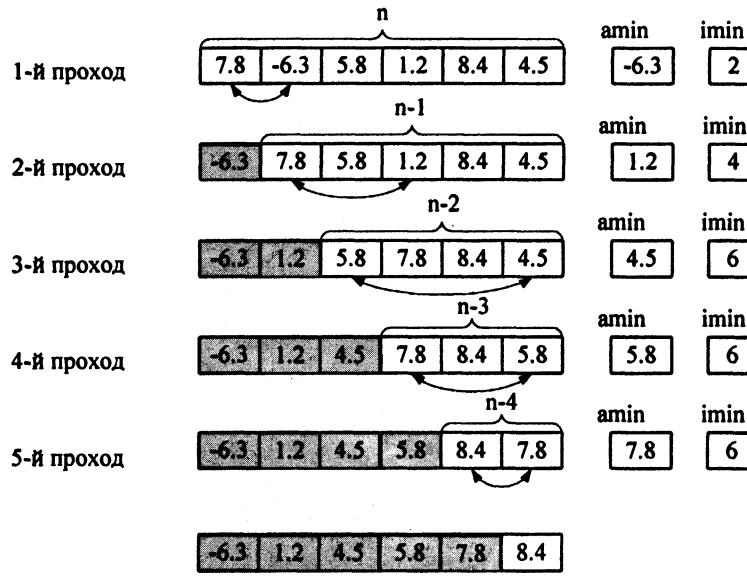


Рис. 4.13. Сортировка выбором

тами, начиная со второго, потом с n-2 элементами, начиная с третьего и т.д. до тех пор, пока не останется один, самый большой элемент массива (рис. 4.13).

Алгоритм сортировки выбором приведен на рис. 4.14. Ниже приведен текст программы, реализующий данный алгоритм.

```
Program sort1;
Var a:array[1..20] of real;
j, i, n, imin:integer;
min:real;
Begin
   Writeln('Введите количество чисел n<=20:');
   Readln(n);
   Writeln('Введите массив:');
   for i:=1 to n do Read(a[i]);
   Readln;
   for j:=1 to n-1 do {цикл поиска минимальных элементов массива}
     begin
        min:=a[j]; {начальное значение для поиска минимума}
        imin:=j; {начальное значение индекса минимального элемента}
        for i:=j+1 to n do {цикл поиска минимума и его индекса}
             if a[i]<min then {если элемент меньше уже найденного
                                                  минимального}
```

```
        begin
          min:=a[i]; {запоминаем
                      элемент}
          imin:=i    {запоминаем его
                      индекс}
        end;
      {меняем местами найденный
       минимум и первый элемент
       текущего массива}
      a[imin]:=a[j];
      a[j]:=min;
    end;
  for i:=1 to n do Write(a[i]:6:2);
  Writeln;
End.
```

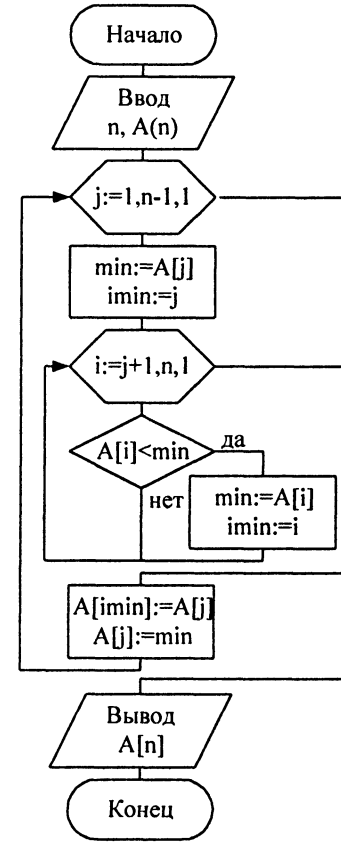


Рис. 4.14. Алгоритм сортировки выбором

Оценим временную сложность данного метода, используя в качестве основной операции операцию сравнения.

Для поиска минимального элемента в каждом проходе потребуется выполнить: n-1, n-2, ..., 1 операций сравнения, т.е. всего n(n-1)/2 операций сравнения. Следовательно, вычислительная сложность данного метода $O(n^2)$. Причем время сортировки не зависит от исходного порядка элементов.

Метод вставки. Сортировку вставками можно описать следующим образом. В исходном состоянии считают, что сортируемая последовательность состоит из двух последовательностей: уже сортированной (она на первом шаге состоит из единственного – первого элемента) и последовательности элементов, которые еще необходимо сортировать. На каждом шаге из сортируемой последовательности извлекается элемент и вставляется в первую последовательность так, чтобы она оставалась сортированной. Поиск места вставки осуществляют с конца, сравнивая вставляемый элемент a_i с очередным элементом сортированной последовательности a_j. Если элемент a_i больше a_j, его вставляют вместо a_{j+1}, иначе сдвигают a_j вправо и уменьшают j на единицу. Поиск места вставки завершают, если элемент вставлен или достигнут левый конец массива. В последнем случае элемент a_i вставляют на первое место (рис. 4.15).

Разрабатывая алгоритм, избавимся от проверки достижения начала массива. Прием, позволяющий отменить эту проверку, называется «установкой барьера». С использованием этого приема проверка организуется так, чтобы

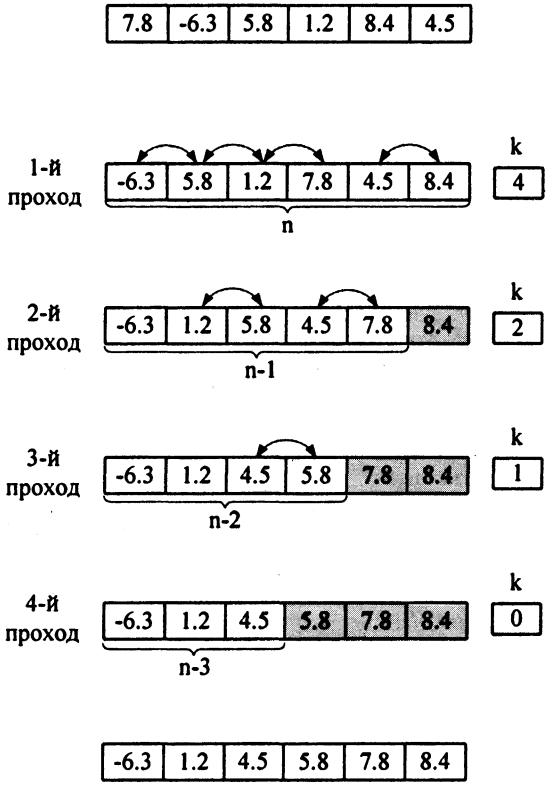


Рис. 4.15. Сортировка вставками

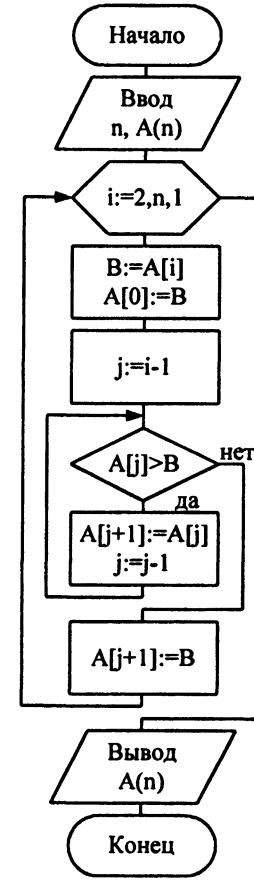


Рис. 4.16. Схема алгоритма сортировки вставками

из цикла поиска места вставки в любом случае происходил выход по первому условию. Для этого достаточно поместить вставляемый элемент перед первым элементом массива, как элемент с индексом 0. Этот элемент и станет естественным барьером для ограничения выхода за левую границу массива.

Алгоритм сортировки вставками приведен на рис. 4.16. Ниже приведен текст программы, реализующей данный алгоритм.

```
Program sort2;
Var a:array[0..20] of real; B:real;
    i,j,n:integer;
Begin
    WriteLn('Введите количество чисел n<=20.');
```

```
  ReadLn(n);
  WriteLn('Введите массив.');
  for i:=1 to n do Read(a[i]);
  ReadLn;
  for i:=2 to n do {начиная со второго элемента до конца массива}
    begin
      B:=a[i];   {запоминаем i-й элемент}
      a[0]:=B;   {этот же элемент записываем в a[0] – это барьер}
      j:=i-1;    {индекс i запоминаем в j}
      while B<a[j] do {пока очередной рассматриваемый элемент
                       больше i-го элемента}
        begin
          a[j+1]:=a[j]; {сдвигаем элемент}
          j:=j-1;       {уменьшаем j на 1}
        end;
      a[j+1]:=B;     {как только найдено место, туда записывается B}
    end;
  WriteLn('Отсортированный массив:');
  for i:=1 to n do Write(a[i]:6:2);
  WriteLn;
End.
```

Оценим временную сложность данного метода, также определив количество операций сравнения.

Для поиска места i-го элемента каждый раз потребуется выполнить от 1 до i-1 операций сравнения, т.е. в среднем i/2 операций сравнения. Значение i изменяется от 2 до n, т.е. выполняется n-1 проход, в каждом из которых происходит в среднем от 1 до n/2 сравнений. Таким образом, суммарно в среднем для решения задачи требуется выполнить $(n-1)(n/2 + 1)/2 = (n^2 + n - 2)/4$ операций сравнения. Откуда вычислительная сложность метода в среднем также равна $O_{ср}(n^2)$, хотя время выполнения примерно в два раза меньше, чем у предыдущего метода. Интересно, что в данном случае вычислительная сложность зависит от исходного расположения элементов массива.

Так, в лучшем случае, когда массив уже упорядочен, поиск места вставки требует одного сравнения для каждого элемента, и количество сравнений равно n-1. Соответственно, вычислительная сложность равна $O_{л}(n)$.

В худшем случае, если элементы массива в исходном состоянии расположены в обратном порядке, поиск места вставки для каждого элемента потребует: 1, 2, 3, ..., n-1 сравнения, следовательно, всего потребуется $n(n-1)/2$ операций сравнения, т. е. время выполнения программы примерно совпадет со временем программы, реализующей метод выбора. Значит вычислительная сложность в худшем, так же как в среднем, равна $O_{х}(n^2)$.

Таким образом, за счет ускорения сортировки в лучших случаях данный метод имеет лучшие временные характеристики, чем предыдущий.

Метод обменов. Алгоритм прямого обмена основывается на сравнении пары соседних элементов. Если расположение элементов не удовлетворяет условиям сортировки, то их меняют местами. Сравнения и перестановки продолжают до тех пор, пока не будут упорядочены все элементы. Определить, что элементы упорядочены, можно, считая количество выполненных перестановок: если количество перестановок равно нулю, то массив отсортирован (рис. 4.17).

Простейший алгоритм сортировки с помощью обмена представлен на рис. 4.18. Ниже приведена программа, реализующая данный алгоритм.

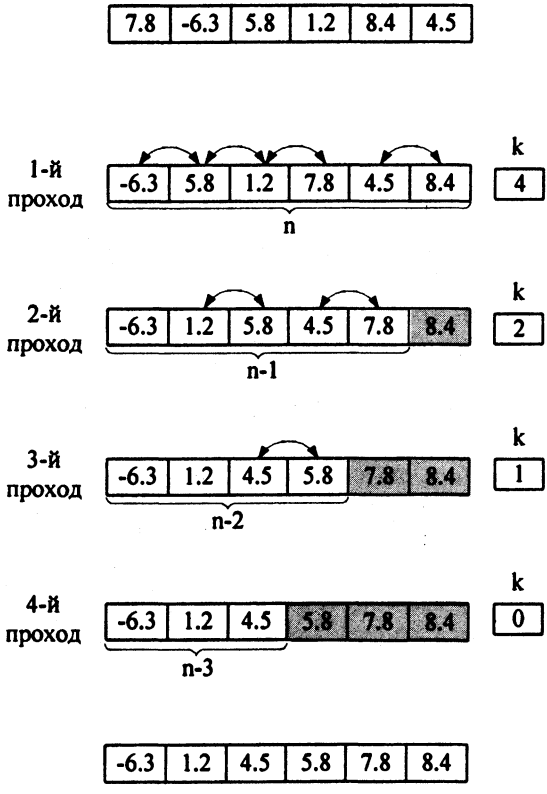


Рис. 4.17. Сортировка обменом

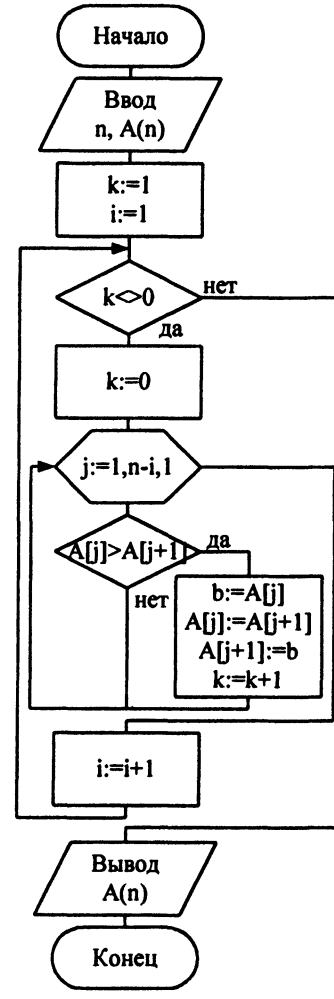


Рис. 4.18. Схема алгоритма сортировки обменом

```
Program ex;
 Var  a: array[1..20] of Real;      i,j,n,l,k: integer;      b:real;
Begin
   WriteLn('Введите размер массива N<=20');
   ReadLn(n);
   for i := 1 to n do Read(a[i]);
   ReadLn;
   WriteLn('Исходный массив:');
   for i := 1 to n do Write(a[i]:7:2);
   WriteLn;
   k:=1;  {количество перестановок, начальное значение не равно 0 }
    i:=1;   {номер очередного просмотра, в начале равен 1}
    while k<>0 do   {пока есть перестановки}
      begin
        k:=0;  {обнуляем количество перестановок}
        for j:=1 to n-i do {цикл сравнения соседних элементов}
           if a[j]>a[j+1] then {если предыдущий элемент больше, то}
               begin          {осуществляем перестановку}
                  b:=a[j];
                  a[j]:=a[j+1];
                  a[j+1]:=b;
                  k:=k+1; {счетчик перестановок увеличиваем на 1}
               end;
        i:=i+1;  {увеличиваем номер просмотра на 1}
      end;
    WriteLn('Отсортированный массив');
   for i := 1 to n do  Write(a[i]:7:2);
   WriteLn;
   WriteLn(' Количество проходов ', i:3);
End.
```

Оценим вычислительную сложность данного метода. Очевидно, что она сильно зависит от исходного расположения элементов массива. Так, в лучшем случае, если массив был уже отсортирован, потребуется выполнить n-1 сравнение для проверки правильности расположения элементов массива, т. е. вычислительная сложность в лучшем $O_л(n)$.

В худшем случае, если массив отсортирован в обратном порядке, будет выполнено n-1, n-2, ...1 операций сравнения, т. е. всего $(n^2-n)/2$ операций сравнения, следовательно, вычислительная сложность в худшем определяется как $O_х(n^2)$.

Выполнить усредненную оценку данного метода достаточно сложно, так как в отличие от предыдущих случаев зависимость времени выполнения от количества неправильно стоящих элементов не является линейной. Например, два элемента, стоящие на своем месте в начале массива, практически не

Проверка и удаление строк:
k:=0
Для i:=1, n, 1
Определение максимального элемента max i-й строки.
Если max≠B,
то k:=k+1
Перемещение i-й строки на k-е место
все-если
Все-цикл
Все.

Определение максимального элемента строки – уже известная нам операция (см. пример 4.1).

Перемещение строки выполняется поэлементно.

Перемещаем i-ю строку на k-е место:
Для j = 1, m, 1
A[k,j]:=A[i,j]
Все-цикл.
Все.

Ниже представлен полный текст программы.

```
Program ex;
 Var a: array[1..10,1..10] of integer;
     B, max, n, m, k, i, j:integer;
Begin
   WriteLn('Введите размеры матрицы n,m<=10');
   ReadLn(n,m);
   WriteLn('Введите ',n:4,' строк по',m:4,' элементов');
   for i:=1 to n do
     begin
        for j:=1 to m do Read(a[i,j]);
        ReadLn;
     end;
   WriteLn('Введите значение B:');
   ReadLn(B);
   WriteLn('Исходный массив');
   for i:=1 to n do
     begin
       for j:=1 to m do  Write(a[i,j]:4);
       WriteLn;
     end;
   k:=0;                    {количество остающихся строк}
```

повлияют на время сортировки, а два элемента, стоящих на своем месте в конце массива, вызовут ее досрочное завершение. По результатам тестирования можно считать, что в среднем этот метод сортировки требует примерно в два раза меньше времени, чем предыдущий. Вычислительная сложность в среднем данного метода $O_{ср}(n^2)$.

Примечание. По материалам данного раздела можно сделать ошибочный вывод, что все методы сортировки в среднем имеют вычислительную сложность $O(n^2)$. Методы сортировки, рассмотренные в данном разделе, не являются самыми быстрыми, они просто самые простые и потому используются чаще всего. Существуют методы быстрой сортировки, которые обеспечивают в среднем и даже в худшем вычислительную сложность $O(n \log_2 n)$. Разработаны также методы, обеспечивающие для специальных данных вычислительную сложность $O_{ср}(n)$ [3,5]. Один из методов быстрой сортировки будет рассмотрен в разделе 7.5.

4.4. Практикум. Обработка матриц

Рассмотрим наиболее распространенные приемы программирования обработки матриц. Следует отметить, что программирование операций всех классов для матриц имеет свою специфику, связанную с тем, что матрица, фактически, является массивом одномерных массивов. Это значит, что для каждой операции существует гораздо больше различных вариантов выполнения.

Использование приемов обработки одномерных массивов. При решении некоторых задач обработки многомерных массивов могут быть выделены подзадачи, при программировании которых можно использовать приемы обработки одномерных массивов.

Декомпозицию целесообразно выполнять, используя метод пошаговой детализации.

Пример 4.10. Разработать программу, удаляющую из матрицы A(n,m), где $n \le 10$, $m \le 10$, строки, максимальный элемент которых равен B.

На первом этапе определяется структура программы:

Программа:
Ввод исходной матрицы.
Проверка и удаление строк.
Вывод матрицы.
Конец программы.

Из выделенных подзадач ввод и вывод матрицы представляют достаточно простые фрагменты, реализующиеся вложенными счетными циклами. Детализируем подзадачу проверки и удаления строк. Для удаления строки будем использовать прием, рассмотренный в примере 4.6.

```
  for i:=1 to n  do      {цикл по строкам}
    begin
      max:=a[i,1];       {исходное значение максимума строки}
      for j:=1 to m do   {цикл поиска максимума строки}
            if a[i,j]>max then  max:=a[i,j];
      if max<>B then {если максимум строки не равен B}
        begin            {то оставляем строку}
          k:=k+1;        {увеличиваем количество остающихся строк}
          for j:=1 to m do a[k,j]:=a[i,j];  {копируем строку на место}
        end;
    end;
  if k<>0 then    {если в матрице осталась хоть одна строка}
     begin
          WriteLn('Сформированная матрица ');
            for i:=1 to k do
              begin
                for j:=1 to m do  Write(a[i,j]:4);
                 WriteLn;
              end;
     end
  else
    WriteLn('Все строки матрицы удалены');
End.
```

В некоторых случаях применение приемов обработки одномерных массивов к матрицам результатов не дает. В основном это задачи, связанные с различными вариантами обхода матриц, и задачи обработки разных групп элементов в матрице.

Обход элементов матрицы. На рис. 4.19 показано несколько способов обхода элементов матрицы. Для каждого из представленных способов, кроме последнего, который выполняется по правилу выхода из лабиринта, можно предложить закон, связывающий индексы между собой.

Самые простые обходы: по строкам и столбцам. Их реализацию полезно помнить. Обход по строкам реализуется вложением циклов:

```
for i:=1 to n do
    for j:=1 to m do
          <обработка элемента a[i,j]>
```

При обходе по столбцам меняются местами циклы по строкам и столбцам:

```
for j:=1 to m do
    for i:=1 to n do
          <обработка элемента a[i,j]>
```

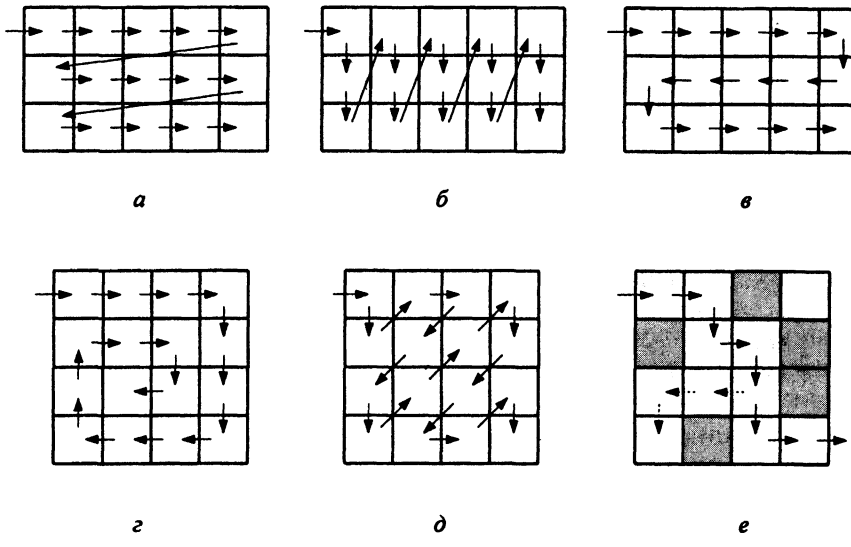


Рис. 4.19. Примеры вариантов обхода матрицы:
а – обход по строкам; *б* – обход по столбцам; *в* – обход «змейкой»;
г – обход по спирали; *д* – обход «змейкой по диагоналям»; *е* – «лабиринт»

В прочих случаях закономерности формирования индексов приходится исследовать, чтобы предложить соответствующий вариант реализации.

В качестве примера рассмотрим закономерность формирования индексов диагоналей квадратной матрицы (рис. 4.20).

Определив закономерности, сравнительно легко можно построить цикл прохода по диагонали матрицы. Например, для диагонали, проходящей через элемент [p,k] параллельно главной, получаем:

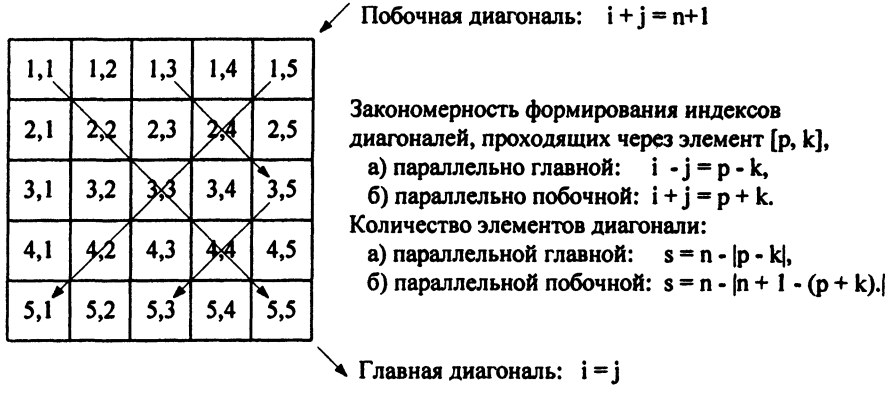


Рис. 4.20. Закономерности формирования индексов диагоналей квадратной матрицы

1	2	3	4	5
10	9	8	7	6
11	12	13	14	15

Рис. 4.21. Закон формирования матрицы

```
if p-k>0 then s:=n-p+k else s:=n-k+p;
for i:=1 to s do
        <обработка элемента a[i,i-p+k]>
```

Пример 4.11. Разработать программу, которая формирует матрицу, представленную на рис. 4.21.

Для решения задачи необходимо осуществить обход матрицы змейкой, присваивая его элементам требуемое значение. Из рисунка видно, что во всех нечетных строках значения элементов возрастают монотонно на единицу слева направо, а во всех четных – справа налево. Такое изменение можно описать формулами:

для четной строки – (i-1)*n + n - j + 1,

для нечетной – n*(i -1) + j,

где i – номер рассматриваемой строки, а j – номер столбца в ней.

Значит, в программе можно построчно обойти все элементы и присвоить им значения в соответствии с указанными формулами.

Однако можно и просто реализовать данный вид обхода, присваивая элементам значение, которое каждый раз увеличивается на единицу. Программа, приведенная ниже, реализует второй вариант.

```
Program ex;
 Var a: array[1..3,1..4] of integer;
     k, i, j:integer;
   Begin
      k:=1;
      for i:=1 to 3 do
         if (i mod 2)=0 then {если номер строки четный}
              for j:=4 downto 1 do   {обходим справа налево}
                        begin
                             a[i,j]:= k; k:=k+1;
                        end
          else        {если номер строки нечетный }
               for j:=1 to 4 do     {обходим слева направо}
                        begin
                             a[i,j]:=k; k:=k+1;
                        end;
      WriteLn('Сформированный массив:');
      for i:=1 to 3 do
            begin
               for j:=1 to 4 do  Write(a[i,j]:3);
                WriteLn;
            end;
 End.
```

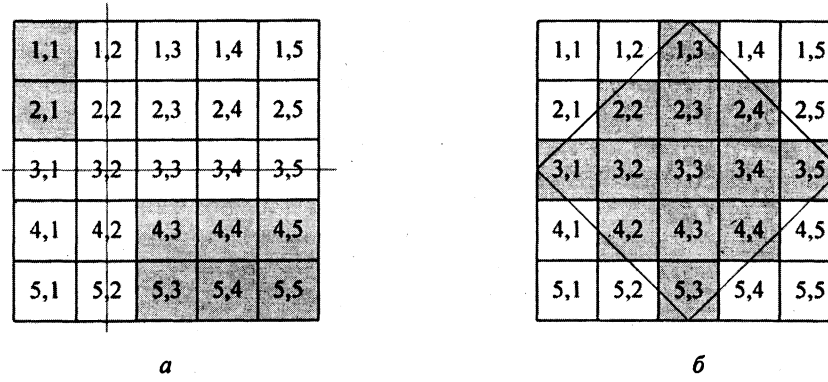


Рис. 4.22. Варианты выборки элементов матрицы:
а – фрагменты прямоугольной формы; *б* – фрагмент ромбовидной формы

Выборочная обработка элементов матриц. Выборочная обработка элементов матриц, как и в случае одномерных массивов, требует определения законов изменения индексов как строк, так и столбцов. Однако вариантов выборки, так же как и вариантов обхода, можно предложить множество (рис. 4.22).

В каждом случае также приходится искать закономерности, которые могут быть использованы в программе.

Пример 4.12. Разработать программу определения суммы элементов, расположенных в закрашенных областях (рис. 4.23), полученных при построении диагоналей, проходящих через элемент [p,k].

Из рисунка видно, что диагональные элементы матрицы исключаются из рассмотрения. Суммирование будем выполнять в два последовательно выполняемых сложных цикла. Это связано с тем, что выше элемента [p, k] суммировать необходимо элементы от 1 до диагонали, параллельной главной, и от диагонали, параллельной побочной, до n, а ниже – диагонали меняются местами. В каждом цикле отдельно записываем циклы суммирования левой и правой частей. Текст программы с комментариями приведен ниже.

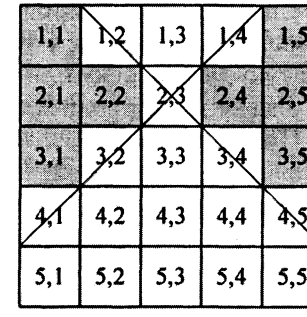


Рис. 4.23. Пример разбиения матрицы на области диагоналями, проходящими через элемент [2,3]

```
Program ex;
  Var a:array[1..15,1..15] of integer;
      s,n,k,p,i,j:integer;
  Begin
    WriteLn('Введите размер матрицы n<=15');
```

```
ReadLn(n);
WriteLn('Введите ',n,' строк(и) по ',n,' элемента(ов):');
for i:=1 to n do for j:=1 to n do Read(a[i,j]);
ReadLn;
WriteLn('Введите индексы элемента p,k:');      ReadLn(p,k);
WriteLn('Исходный массив');
for i:=1 to n do
    begin
      for j:=1 to n do  Write(a[i,j]:4);              WriteLn;
    end;
s:=0;  {начальное значение суммы верхней части}
for i:=1 to p do {цикл определения суммы верхней части}
  begin
     for j:=1 to i-p+k-1 do  {цикл обхода элементов левой части}
                             s:=s+a[i,j];
     for j:=p+k-i+1 to n do  {цикл обхода элементов правой части}
                             s:=s+a[i,j];
  end;
for i:=p+1 to n do {цикл определения суммы нижней части}
  begin
     for j:=1 to p+k-i-1 do  {цикл обхода элементов левой части}
                             s:=s+a[i,j];
     for j:=i-p+k-1 to n do  {цикл обхода элементов правой части}
                             s:=s+a[i,j];
  end;
WriteLn('Сумма элементов равна ',s);
End.
```

Связанная сортировка матриц. Сортировка матриц имеет свои особенности. На практике в виде матрицы представляют таблицы, поэтому, если какую-либо строку или столбец таблицы надо сортировать, соответствующие элементы остальных строк или столбцов перемещаются вместе с ними.

Пример 4.13. Разработать программу, определяющую суммарную «тень» отрезков, параллельных оси x, на оси x. (Тенью будем называть сумму проекций отрезков на ось x (рис. 4.24), не включающую наложений про-

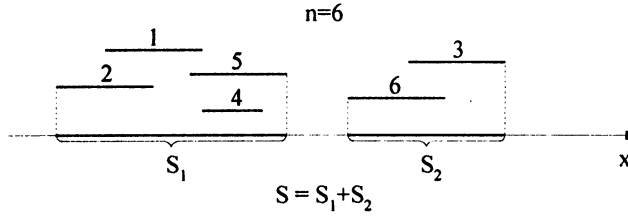


Рис. 4.24. Определение суммарной «тени»

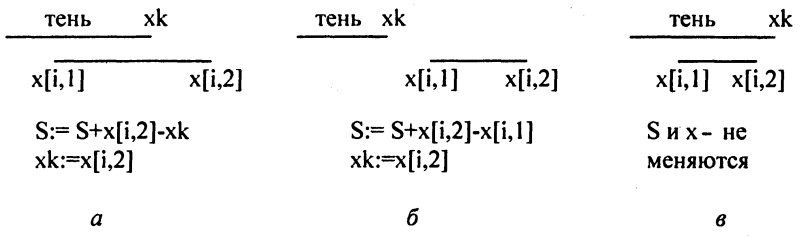


Рис. 4.25. Три случая добавления i-го отрезка к «тени»:
а – отрезок частично перекрыт «тенью»; *б* – отрезок не перекрыт «тенью»; *в* – отрезок полностью перекрыт «тенью» (*в*);
S – уже накопленная «тень», xk – правая граница этой «тени»

екций.) Количество отрезков n. Отрезки заданы координатами начала и конца проекций на ось x.

Анализ условия задачи и возможных вариантов отрезков показывает, что решение задачи «в лоб» достаточно сложно. В то же время, если бы отрезки были *сортированы* по левой границе, то вычисление тени можно было выполнять, добавляя отрезки по одному. При этом можно было бы выделить три случая (рис. 4.25).

Окончательно алгоритм будет включать сортировку отрезков по левой границе и цикл накопления тени, включающий анализ вариантов добавления. Программа в этом случае имеет вид

```
Program ten;
 Var x:array[1..100,1..2] of real;
      i,j,n,k:integer;
      xk,S,w:real;
 Begin
   Write('Введите количество отрезков:');
   Readln(n);
   WriteLn('Вводите начала и концы отрезков.');
   for i:=1 to n do ReadLn(x[i,1],x[i,2]);
   {сортировка отрезков по возрастанию левой границы}
   j:=1; k:=1;
   while k<>0 do
     begin
       k:=0;
       for i:=1 to n-j do
         if x[i,1]>x[i+1,1] then
            begin   k:=k+1;          {меняем отрезки местами }
              w:=x[i,1]; x[i,1]:=x[i+1,1]; x[i+1,1]:=w;
              w:=x[i,2]; x[i,2]:=x[i+1,2]; x[i+1,2]:=w;
     end;
```

```
      j:=j+1;
    end;
  {определение тени}
  S:=x[1,2]-x[1,1];              {длина первого отрезка}
  xk:=x[1,2];                    {правая граница первого отрезка}
  for i:=2 to n do
    if x[i,1]>=xk then     {случай б}
      begin S:=S+x[i,2]-x[i,1];
          xk:=x[i,2];
      end
    else if x[i,2]>xk then   {случай а}
          begin S:=S+x[i,2]-xk;
              xk:=x[i,2];
          end;
    Writeln('Длина тени равна', S:6:2);
End.
```

Задания для самопроверки

Задание 1. Дана матрица вещественного типа D(n,m), n, m ≤ 20. Разработайте программу, которая в заданной матрице вычеркивает все строки, содержащие более трех отрицательных элементов. Вывести на печать исходную матрицу и матрицу-результат или соответствующие сообщения, если таких строк не окажется или все строки будут удовлетворять условию.

Задание 2. Дана матрица A(n,m), n, m ≤ 15. Разработайте программу, формирующую одномерный массив B(n), элементами которого должно являться количество элементов каждой строки, превышающих среднее арифметическое значение матрицы в целом. Если в строке таких элементов нет, в соответствующий элемент одномерного массива заносится 0. Вывести исходную матрицу, значение среднего арифметического элементов матрицы и сформированный массив B.

Задание 3. Разработайте программу, формирующую квадратную матрицу D(n,n), n ≤ 15, элементы которой определяются по формуле

$$D[i,j] = \begin{cases} \sin(i+j) & \text{при } i < j; \\ 1 & \text{при } i = j; \\ (i+j)/(2i+3j) & \text{при } i > j, \end{cases}$$

где i – номер строки, а j – номер столбца элемента матрицы. В сформированной матрице поменять местами максимальный элемент среди элементов, лежащих ниже главной диагонали, с минимальным элементом среди элементов матрицы, лежащих ниже побочной его диагонали. Вывести исходную матрицу, соответствующие элементы и их координаты, а также преобразованную матрицу.

4.5. Строки

Уже на простом примере обработки символьной информации, рассмотренном в параграфе 4.1, видно, что обработка строк с использованием одномерных массивов представляет собой достаточно специфическую задачу. В то же время большинство операций, которые выполняют со строками текста, повторяются в разных программах: поиск, копирование, удаление и вставка фрагментов строки. Поэтому для упрощения работы со строками в Borland Pascal существует специальный тип данных – *строковый*, который приспособлен для обработки символьной информации.

Синтаксическая диаграмма объявления строкового типа данных представлена на рис. 4.26.

Целое без знака – это максимальная длина строки, которая не должна превышать 255 байт. Если длина не указана, то по умолчанию принимается максимальное значение – 255 символов.

Объявление переменных строкового типа, так же как и массивов, можно выполнить двумя способами:

- в операторе объявления переменных, например:

```
Var S1, S2:string[40]; {символьные строки длиной 40 байт}
    S3:string;         {символьная строка длиной 255 байт}
```

- с предварительным объявлением типов, например:

```
Type S40 = string[40]; {тип – строка длиной 40 байт}
     ST = string;      {тип – символьная строка длиной 255 байт}
Var  S1,S2: S40;       {символьные строки типа S40}
     S3:ST;            {символьная строка типа ST}
```

Внутреннее представление строки показано на рис 4.27, откуда видно, что строка представляет собой одномерный символьный массив, индексы которого изменяются от 0 до максимального значения, указанного при объявлении строкового типа. Следовательно, физическая длина строки на единицу превышает максимальную.

Инициализация строк. Для инициализации строковых переменных, так же как и переменных других типов, можно использовать типизированные

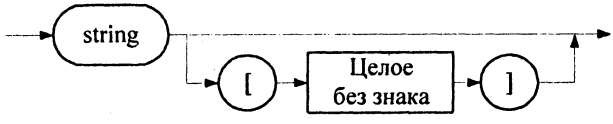


Рис. 4.26. Синтаксическая диаграмма
<Объявление строкового типа>

Рис. 4.27. Внутреннее представление строки

константы, причем строка-литерал может быть короче инициализируемой строки, например:

```
Const S:string[40]='Типизированная константа';
      S1:string=''; ...     {пустая строка нулевой длины}
```

Операции над переменными строкового типа. Над переменными строкового типа помимо операции доступа к символам определены операции присваивания, *конкатенации* (сцепления) и отношений.

Доступ к символам строки. Обращение к символам строки выполняется как к элементам массива символов, т. е. с указанием имени строки и номера элемента, например st[1] или s[i]. Нулевой байт содержит текущее значение длины строки, но так как строка – это массив символов, длина автоматически интерпретируется как символ. Для получения текущей длины строки в виде числа необходимо явное преобразование символьного типа в целый. Например, если i – переменная целого типа, а S3 – переменная строкового типа, то получить текущую длину строки S3 можно:

```
1) i:= byte(S3[0]); {явное преобразование с помощью автоопределения}
2) i:= ord(S3[0]);  {явное преобразование с помощью специальной функции}
```

Однако лучше это сделать, используя специальную функцию Length, описанную ниже.

Присваивание строк. Можно присвоить строке значение строки и значение символа. При выполнении операции символы заполненной части строки и ее длина переписываются в строку-результат, например:

```
S1:='ABCD';  {присваиваем строке значение строковой константы}
S2:= 'A';    {присваиваем строке значение символа}
S:=S1;       {переписываем одну строку в другую}
```

При присваивании строке значения символа последний интерпретируется как строка единичной длины. Если строка-источник длиннее, то при присваивании она усекается в соответствии с длиной строки-результата.

Конкатенация. Операция конкатенации позволяет сцепить строки с другими строками или символами. При сцеплении длины строк суммируются, а символы объединяются в одну последовательность. Например:

```
'fdc'+'ghj';    {получаем ' fdcghj'}
S4+ 'vvv';      {к строке S4 дописывается 'vvv'}
```

Результат этой операции можно присвоить какой-либо строке или вывести на экран.

Отношения. Над строками допускается выполнять операции отношения: = , <>, >, <, >=, <=. Сравнение строк при этом выполняется последовательно слева направо с учетом внутренней кодировки символов до первого несовпадающего символа. Большей считается та строка, код несовпадающего символа которой по таблице ASCII больше. Если длина одной строки меньше другой, то недостающие значения до длины большей строки заполняются символами #0. Результатом операций отношения для строк, как и для чисел, является значение false и true.

Допускается сравнение символов со строками, при этом символы преобразуются в строки единичной длины.

Так, если

```
S4:='ABCD';  S3:='ADFH';  C:='L';
```

то при выполнении операций отношения:

```
S4 = S3     {получим false}
S4 > S3     {получим false}
S3 > S4     {получим true}
S3 = C      {получим false}
```

Ввод-вывод строк. Ввод-вывод переменных строкового типа осуществляется одной операцией Read (ReadLn) или Write (WriteLn), например:

```
ReadLn(S1);
WriteLn(S1);
```

При вводе за строку принимается последовательность символов до кода клавиши ENTER. Если длина введенной строки больше указанной максимальной длины, то лишние символы отбрасываются, а в нулевой байт записывается значение максимальной длины. В противном случае в нулевой байт записывается количество введенных символов. Поскольку строкой считаются все символы до кода клавиши ENTER, ввести в одной строке строковое значение, а затем, например, число нельзя.

Если при вводе строки просто нажать клавишу Enter, не вводя никаких символов, то считается, что введена пустая строка.

Процедуры и функции для работы со строками. Все основные действия над строками и символами реализуют с помощью стандартных процедур и функций.

1. Функция ***Length(st):word*** – возвращает длину строки st, например:

```
n:=Length(st1);  {целочисленной переменной n присваивается значение длины строки}
```

2. Процедура ***Delete(st, index, count)*** – удаляет count символов строки st, начиная с символа с номером index, например:

```
S1: = 'dddddsssssfffff';
Delete(S1,6,5);  {получим результат 'dddddfffff'}
```

3. Процедура ***Insert(St2,St1,index)*** – вставляет подстроку символов St2 в строку St1, начиная с символа с номером index. Процедура обычно используется при формировании строк, включающих числовую информацию, например:

```
S1 = 'ddddddddddd';
S2 = 'aaaaaa' ;
Insert(S1, S2,6);      {получим 'dddddaaaaaaddddd'}
Insert('Pas', S2,6);     {получим 'dddddPasaaaaaaddddd'}
```

4. Процедура ***Str(x[:w [:d]] , St)*** – преобразует результат выражения x в строку st, содержащую запись этого числа в виде последовательности символов (как при выводе).

Примечание. По правилам описания конструкций языков программирования используемые в описании заголовков процедур и функций квадратные скобки означают, что соответствующий параметр может быть опущен.

Значение w, если оно указано, интерпретируется как длина строки, а значение d, если оно указано – как количество цифр дробной части для вещественных чисел, например:

```
x:=-5.67;
Str(x:7:3,s1);  {получим строку '  -5.670'}
```

Процедура обычно используется для формирования строк, включающих числовую информацию.

5. Процедура ***Val(St, x, Code)*** – преобразует строку St с записью числа в виде последовательности символов во внутреннее представление целого или вещественного числа и помещает его в переменную x. В целочисленной переменной Code процедура возвращает код ошибки: 0, если преобразование прошло успешно, и номер ошибочного символа, если строка st не являлась допустимой формой записи числа.

Процедура обычно используется, если необходимо предотвратить некорректный ввод чисел, например:

```
Var S:string; Code:integer; a:real;  ...
...repeat
    Write('Введите число a:');
    ReadLn(S);          {вводим строку}
    Val(S,a,Code);  {пытаемся преобразовать строку в число}
    if Code<>0 then
            WriteLn('Число введено не верно');
  until Code=0; ...  {до получения правильного значения числа}
```

6. Функция ***Copy(St,index,count):string*** – возвращает фрагмент строки St длиной count символов, начиная с символа с номером index, например:

```
S1 = 'qqqEEEEEEuuuuu';
S:= Copy(S1,4,6);     {получим строку 'EEEEEE'}
```

7. Функция ***Pos(St2,St1):integer*** – возвращает номер позиции первого вхождения подстроки St2 в строку St1. Если вхождение не найдено, то функция возвращает 0, например:

```
S1 = 'qqqEEppEEuuuuu';
 i:= Pos('EE',S1);     {получим i=4}
```

8. Функция ***UpCase(ch):char*** – возвращает символ, соответствующий символу верхнего регистра для ch, если таковой имеется, либо сам символ ch, если для него не определен символ верхнего регистра.

В качестве первого примера посмотрим, как будет выглядеть решение задачи из примера 4.3 с использованием строковых типов.

Пример 4.14. Дана строка не более 40 символов, состоящая из слов, разделенных пробелами. Разработать программу удаления «лишних» пробелов. Лишними считать пробелы в начале строки до первого символа, второй и более пробелы между словами и пробелы в конце строки.

При решении данной задачи с использованием строкового типа отпадает необходимость посимвольного анализа строки. Функция Pos, которой в качестве подстроки заданы два пробела подряд, позволит определить все места в строке, где записаны несколько пробелов подряд. Поочередно удалив лишние пробелы, получим строку, в которой останется только проверить и при необходимости удалить пробел в начале и пробел в конце (рис. 4.28). Ниже приведен текст программы.

```
Program ex;
  Var st:string[40];
      k:byte;
```

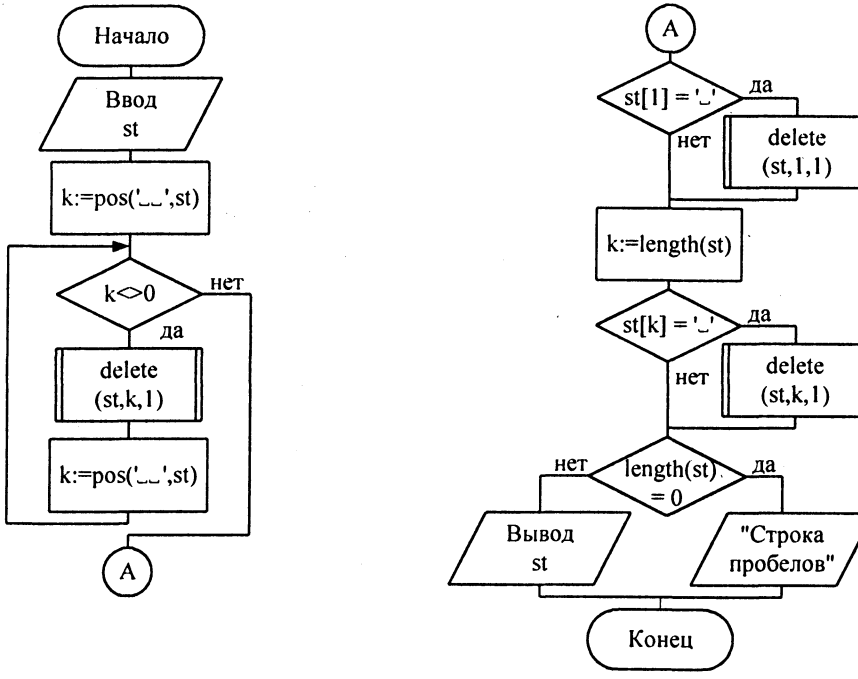


Рис. 4.28. Схема алгоритма программы удаления «лишних» пробелов

```
Begin
  WriteLn('Введите строку длиной <= 40 символов');
  ReadLn(st);
  Write('Введенная строка:');
  WriteLn(st);
  k:=pos('␣␣',st);   {проверяем, есть ли сдвоенные пробелы?}
  while k<>0 do   {пока есть сдвоенные пробелы}
    begin
      delete(st,k,1);    {удаляем первый пробел}
      k:=pos('␣␣',st);   {проверяем, есть ли сдвоенные пробелы?}
    end;
  if st[1]= '␣' then delete(st,1,1);    {удалили пробел в начале}
  k:= length(st);
  if st[k]= '␣' then delete(st,k,1); {удалили пробел в конце}
  WriteLn('Результат:');
  if length(st)<>0 then WriteLn(st)
  else WriteLn('Строка содержала только пробелы.');
End.
```

Пример 4.15. Разработать программу, которая вводит строки, содержащие фамилию, имя, отчество и год рождения, а выводит – строки, содержащие фамилию, инициалы и возраст на текущий год. Например:

Иванов Иван Иванович 1956 ⇒ Иванов И.И. 45

Завершение ввода – при чтении пустой строки.

Для выполнения операций над строками используем строковые функции. Обработку строк будем выполнять в цикле до ввода пустой строки. Начнем с определения местоположения первого пробела, который отделяет имя от фамилии. Для этого используем функцию Pos, а результат запишем в переменную c1. Затем перепишем в строку результата фамилию, пробел и первый инициал. Туда же дописываем точку.

Для поиска следующего пробела придется копировать в рабочую строку часть исходной строки, начиная с символа после первой буквы имени. В этой строке вновь определяем местоположение пробела и заносим результат в переменную c2. Теперь можно переписать в строку-результат второй инициал.

Удаляем из рабочей строки начало, включая второй инициал, и вновь определяем местоположение пробела, выделяя подстроку, содержащую год рождения. Удаляем из рабочей строки остаток отчества и преобразуем строку в число. Полученное значение вычитаем из текущего номера года, а результат вновь преобразуем в строку и дописываем к строке результата. Выводим результат на экран и вводим следующую строку.

Несколько первых шагов преобразования показаны на рис. 4.29.

```
Program stroka;
 Var st,strez,strab:string[40];
     c1,c2,c3,n,old,code:word;
```

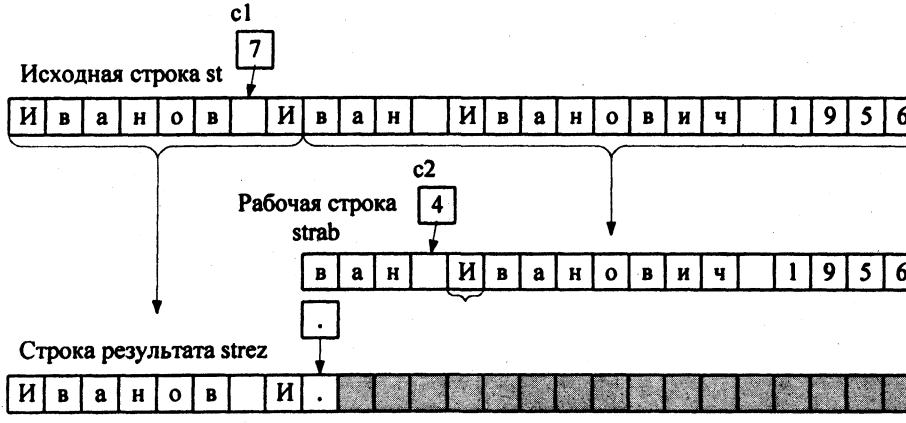


Рис. 4.29. Начало решения задачи преобразования строк

```
Begin
  WriteLn('Введите строку. Завершение – ввод пустой строки.');
  ReadLn(st);
  while st<>'' do {цикл ввода, преобразования и вывода строк}
    begin
      c1:=Pos(' ',st);{определим местоположение первого пробела}
      strez:=Copy(st,1,c1+1)+'.'; {перепишем фамилию, инициал
                                   и добавим точку}
      strab:=Copy(st,c1+2,Length(st)-c1-1); {копируем остаток строки
                                   в рабочее поле}
      c2:=Pos(' ',strab); {определяем местоположение второго пробела}
      strez:=strez+strab[c2+1]+'.'; {добавляем к результату второй
                                   инициал и точку}
      Delete(strab,1,c2+1); {удаляем распознанную часть}
      c3:=Pos(' ',strab);{определяем местоположение третьего пробела}
      Delete(strab,1,c3);          {удаляем остаток имени}
      Val(strab,n,code);           {преобразуем год рождения в число}
      old:=2001-n;                 {определяем возраст}
      Str(old,strab);              {преобразуем возраст в строку}
      strez:=strez+' '+strab;      {добавляем возраст в результат}
      WriteLn(strez);              {выводим результат}
      WriteLn('Введите строку. Завершение- ввод пустой строки.');
      ReadLn(st);
    end;
End.
```

4.6. Практикум. Обработка и поиск символьной информации

В основе обработки символьной информации, как правило, лежит разбиение текста на слова и выполнение некоторых операций со словами.

Пример 4.16. Разработать программу, которая определяет в строке количество слов длиннее четырех символов. Слова разделены одним пробелом.

Решение задачи будем осуществлять следующим образом. Вначале убедимся, что в конце строки есть пробел. Если пробела нет, тогда вставим его. Затем, пока длина строки не станет равной нулю, будем определять местоположение пробела и, соответственно, длину слова, которая на единицу меньше номера пробела. Если длина слова больше четырех символов, то добавим единицу к счетчику слов. Затем удалим обработанное слово вместе с пробелом и перейдем к обработке следующего слова.

```
Program stroka1;
 Var st: string;
     p, spos:integer;
```

```
Begin
    WriteLn('Введите строку');
    ReadLn(st);    {вводим строку}
    p:=0;          {обнуляем счетчик слов }
    if st[Length(st)] <>' ' then  st:=st+' ';   {если в конце нет пробела,
                                                  то добавим его}
    while Length(st)<>0 do
      begin
        spos:= Pos(' ', st);
        if spos>5  then p:=p+1;     {определяем длину слова}
        Delete(st,1,spos);          {удаляем слово}
      end;
   WriteLn('В строке ', p, ' слов(а), длина которых больше четырех.');
End.
```

Пример 4.17. Разработать программу, меняющую в строке одно сочетание букв на другое.

С использованием строковых функций задача решается просто. Вводим строку и оба сочетания букв. Затем определяем вхождения заменяемого сочетания, удаляем его и вставляем на это место заменяющее сочетание.

```
Program Stroka;
 Var  n:byte;  s, s1, s2:string;
Begin
  WriteLn('Введите исходную строку');
  ReadLn(s);
  WriteLn('Введите заменяемое слово: ');
  ReadLn(s1);
  WriteLn('Введите заменяющее слово: ');
  ReadLn(s2);
  n:=Pos(s1,s);     {определяем вхождение заменяемого сочетания}
  while n > 0 do
       begin
          Delete(s,n,Length(s1));  {удаляем заменяемое сочетание}
          Insert(s2,s,n);          {вставляем заменяющее сочетание}
          n:=Pos(s1,s);  {определяем следующее вхождение}
       end;
  WriteLn('Результат : ',s);
  ReadLn;
End.
```

Пример 4.18. Разработать программу, меняющую в строке местами слова с указанными номерами. Запретить ввод номеров, которые превышают количество слов в строке или равны между собой.

В а р и а н т 1. При решении данной задачи строку приходится просматривать посимвольно, так как необходимо фиксировать начало и длину каждого из слов с указанными номерами. Если очередной символ равен пробелу, то количество слов необходимо увеличить на единицу, проверить, не совпадает ли номер с одним из заданных и если совпадает, то запомнить номер первого символа и длину слова. После чего обнуляем счетчик длины слова и фиксируем начало следующего слова. Если символ не пробел, то увеличиваем длину текущего слова.

В конце строки пробела может не быть. Следовательно, завершение последнего слова необходимо проверять отдельно, к тому же учитывая, что если после последнего слова нет пробела, то его длина получается на единицу меньше, что тоже необходимо скорректировать.

После того, как местоположение слов определено, необходимо осуществить их перемещение. При этом необходимо учесть, что как только мы *удалим первое слово, начало второго слова сместится*. Следовательно, вначале необходимо удалить второе слово и вставить первое, а затем уже удалить первое слово и вставить второе. Чтобы не анализировать, какое из слов первое, а какое второе, лучше всего сортировать введенные номера слов по возрастанию.

```
Program Stroka2;
 Var  ns1, ns2, ks, n1, n2, dl1, dl2,ns ,dls, i, w:byte;
      s, s1, s2:string;
Begin
 WriteLn('Введите исходную строку'); Readln(s);
 ks:=0;   {обнуляем счетчик слов}
 for i:= 1 to Length(s)  do
     if (s[i]=' ') or (i=length(s)) then {если конец очередного  слова }
             ks:=ks+1; { увеличиваем счетчик слов}
 WriteLn('Введите номера слов для обмена');
 ReadLn(n1,n2); {вводим номера}
 while (n1>ks) or(n2>ks) or(n1=n2) do {пока номера не допустимы}
    begin
       WriteLn('Количество слов в строке ',ks:5,
                '. Одинаковые номера не допустимы.
                      Повторите ввод номеров.');
       ReadLn(n1,n2); {вводим номера}
    end;
 if n1>n2 then       {сортируем номера по возрастанию}
 begin   w:=n1;   n1:=n2;   n2:=w;   end;
 ns:=1;   {начало первого слова равно 1}
 dls:=0;  {длина первого слова равна 0}
 ks:=0;   {номер слова пока равен 0}
```

```
    for i:=1 to Length(s) do   {по всей строке}
        begin
            if (s[i]=' ') or (i=Length(s)) then   {если слово завершено}
                begin
                    if (i=Length(s)) and (s[i] <>' ') then {если в конце
                                                строки нет пробела}
                            dls:=dls+1;  {корректируем длину слова}
                    ks:=ks+1;
                    if ks=n1  then    {если это первое слово}
                        begin   {то запоминаем начало и длину}
                            ns1:=ns;    dl1:=dls;
                        end;
                    if ks=n2  then   {если это второе слово}
                        begin   {то запоминаем начало и длину}
                            ns2:=ns;    dl2:=dls;
                        end;
                    dls:=0;    {обнуляем  длину текущего слова}
                    ns:=i+1;  {запоминаем начало текущего слова}
                end
            else     dls:=dls+1; {считаем длину очередного слова}
        end;
    s1:=Copy(s, ns1, dl1);  {копируем значение первого слова}
    s2:=Copy(s, ns2, dl2);  {копируем значение второго слова}
    Delete(s, ns2, dl2);        {удаляем дальнее слово}
    Insert(s1, s, ns2);         {вместо него вставляем ближнее}
    Delete(s, ns1, dl1);        {удаляем ближнее слово}
    Insert(s2,s,ns1);           {вставляем дальнее}
    WriteLn('Результат : ', s);
End.
```

Вариант 2. Вначале разобьем текст на слова и поместим каждое слово в элемент вспомогательного массива строк. Затем выполним перестановку элементов. И, наконец, вновь объединим слова в строку.

Это решение имеет два существенных недостатка. Во-первых, оно требует дополнительной памяти для размещения вспомогательного массива. Во-вторых – выполняться такая программа будет несколько дольше, так как исходная строка будет просматриваться несколько раз. Однако этот вариант решения несколько проще, и в тех случаях, когда отсутствуют строгие ограничения на объем используемой памяти и время выполнения, он может оказаться предпочтительным.

```
Program Stroka3;
 Var  ks, n1, n2, i, k:byte; s, s1:string;
      MasStr:array[1..100] of string[20];   {рабочий массив}
```

```
Begin
 WriteLn('Введите исходную строку'); Readln(s);
 ks:=0;  {обнуляем счетчик слов}
 if s[length(s)]<>' ' then s:=s+' ';  {если в конце строки нет пробела,
                                       то дописываем его}
{разбор строки}
 while s<>'' do      {пока в строке остались слова}
   begin
     ks:=ks+1;       {увеличиваем счетчик слов}
     k:=Pos(' ',s);  {определяем конец слова}
     masStr[ks]:=Copy(s,1,k);  {копируем слово в массив}
     Delete(s,1,k);            {удаляем слово из строки}
   end;
{обмен слов}
 WriteLn('Введите номера слов для обмена');
 ReadLn(n1,n2); {вводим номера}
 while (n1>ks) or(n2>ks) or(n1=n2) do {пока номера не допустимы}
     begin
       WriteLn('Количество слов в строке ',ks:5,
       '. Одинаковые номера не допустимы.Повторите ввод номеров.');
       ReadLn(n1,n2); {вводим номера}
     end;
  s1:=MasStr[n1];      {меняем слова местами}
  MasStr[n1]:=MasStr[n2];
  MasStr[n2]:=s1;
  {объединение слов в строку}
  for i:=1 to ks do s:=s+MasStr[i];  {объединяем слова в строку}
 Delete(s,Length(s),1);     {удаляем пробел после последнего слова}
 WriteLn('Результат : ', s);  {выводим результат}
End.
```

Пример 4.19. Разработать программу, которая осуществляет поиск заданной строки в отсортированном в соответствии с латинским алфавитом массиве строк MasStr[n], n<100. Конкретное количество строк массива определять в процессе их ввода.

Ввод строк организуем в цикле-пока. В качестве условия завершения ввода будем использовать ввод пустой строки. Количество вводимых записей будем считать. Если введено меньше 99 слов, то после завершения ввода уменьшим количество введенных строк на единицу, чтобы не обрабатывать пустую строку.

Поиск строк может быть реализован несколькими способами.

Вариант 1. Самый простой способ поиска – последовательный. При последовательном способе мы запись за записью сравниваем строки в мас-

сиве с заданной строкой. Однако данный вид поиска является и самым продолжительным по времени. Оценим время поиска.

Если искомая строка совпадает с первой строкой массива, то в процессе поиска будет выполнено одно сравнение. Если искомая строка совпадает с последней строкой, то – n сравнений. В среднем в процессе поиска понадобится выполнить (n+1)/2 сравнений, т.е. вычислительная сложность последовательного поиска $O_{ср}(n)$.

Вариант 2. Для ускорения обработки можно реализовать двоичный поиск. Этот метод применим, так как массив отсортирован по возрастанию кодов символов строк. Метод двоичного поиска заключается в следующем. Определяют примерную середину массива и проверяют, совпадает ли искомый элемент с элементом в середине массива. Если совпадает, поиск завершен. Если не совпадает, то, если элемент больше среднего, то поиск продолжают в левой половине массива, иначе – в правой половине. Таким образом, диапазон элементов на каждом шаге уменьшается больше, чем вдвое. Если диапазон сократился до нуля, а элемент не найден, то такой элемент в массиве отсутствует.

На рис. 4.30 показан пример реализации двоичного поиска для массива, включающего 10 элементов. На первом шаге осуществляется проверка 5-го элемента, и массив разбивается на два подмассива: 1...4 и 6...10. На втором шаге – 2-го, если искомое значение меньше 5-го элемента, или 8-го, если искомое значение больше 5-го элемента. Затем проверяются значения третьего уровня и т. д.

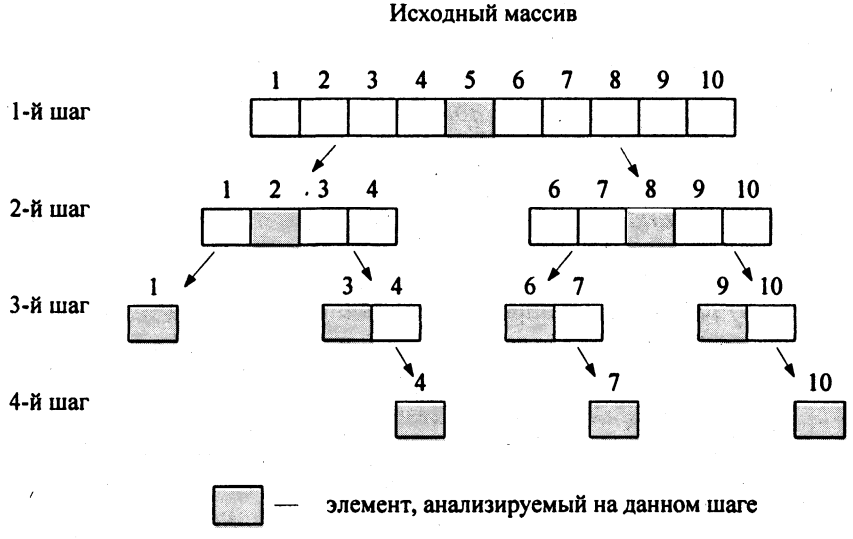


Рис. 4.30. Пример дерева двоичного поиска для исходного диапазона из 10 элементов

Оценим время поиска для данного метода. Будем считать, что дерево поиска строки получилось сбалансированным, т.е. количество записей $n = 2^j-1$, где j=1, 2, 3 и т.д., тогда количество уровней дерева равно $\log_2 n$. Считая, что искомое значение может с равной вероятностью находиться на любом уровне, получаем, что среднее количество сравнений равно $(\log_2 n +1)/2$, т.е. вычислительная сложность двоичного поиска $O_{ср}(\log_2 n)$, что при бо́льших значениях n существенно лучше, чем при последовательном поиске.

Примечание. Если бы изначально массив не был отсортирован, а поиск требовалось бы производить многократно, то массив целесообразно было бы сортировать.

Реализуем двоичный поиск.

```
Program ex;
 Var MasStr:array[1..100] of string[22];
      n, k, l: integer; st:string[22];       key:boolean;
 Begin
  n:=1;
 WriteLn('Введите до 100 строк.Завершение ввода - пустая строка');
  ReadLn(MasStr[n]);
  while (MasStr[n]<>'') and (n<100) do
      begin
        n:=n+1;
        ReadLn(MasStr[n]);
      end;
  if n<100 then n:=n-1;  {слов в массиве на одно меньше}
  WriteLn('Введите строку для поиска.');
  ReadLn(st);
  k:=1; key:=false;
  while (n-k>=0) and not key do  {пока диапазон положителен и
                                                  запись не найдена}
   begin
     l:=(n-k) div 2+k;      {определяем среднее значение индекса}
     if st=MasStr[l] then key:=true    {запись найдена}
     else                          {уменьшаем диапазон индексов}
       if st>MasStr[l] then k:=l+1   {смещаем левую границу}
                       else n:=l-1;    {смещаем правую границу}
   end;
  if key then
          WriteLn('Строка найдена. Номер равен ',l)
  else WriteLn('Строка не найдена.');
End.
```

Задания для самопроверки

Задание 1. Дана строка текста длиной не более 80 символов, состоящая из слов, разделенных пробелом, в конце точка. Разработайте программу, которая определяет номера слов, в которых содержится более трех символов «А». Вывести исходную строку и номера слов. Если слов с таким числом букв не окажется, вывести соответствующее сообщение.

Задание 2. Дана строка текста длиной не более 40 символов, состоящая из слов, разделенных пробелом, в конце точка. Разработайте программу, которая удаляет из текста слово, содержащее максимальное количество букв «В». Вывести исходную и преобразованную строки. Если в тексте нет слов с буквой «В» – вывести соответствующее сообщение.

Задание 3. Дан массив символьных строк, длиной не более 40 символов. Строки состоят из слов, разделенных пробелом, в конце точка. Разработайте программу, которая формирует одномерный массив B, содержащий в качестве элементов количество слов каждой строки, начинающихся на гласную букву. Если таких слов нет, в соответствующий элемент массива B занести 0. Вывести исходный и сформированный массивы.

Задание 4. Дана строка, состоящая из слов, разделенных одним пробелом. Разработайте программу, которая разбивает исходную строку на подстроки, размер которых не превышает заданного значения n. Перенос слов считать запрещенным.

Задание 5. Разработайте программу, которая осуществляет «выравнивание по ширине» подстрок, полученных в результате работы программы задания 4. Выравнивание должно выполняться таким образом, чтобы дополнительные пробелы между словами распределялись по подстроке равномерно.

4.7. Множества

Понятие множество является одним из основных в современной математике и трактуется как *неупорядоченная совокупность неповторяющихся объектов*. В общем случае множество может не содержать ни одного элемента. Такое множество называется пустым.

В Borland Pascal предусмотрен структурный тип, предназначенный для представления множеств. Данные множественного типа представляют собой совокупности однотипных элементов, каким-либо образом связанных друг с другом. Характер связей между элементами только подразумевается программистом и никак не контролируется.

Множественный тип объявляется как совокупность элементов некоторого базового типа (рис. 4.31).

Допускается объявлять только конечные множества, количество элементов которых может меняться от 0 до 255.

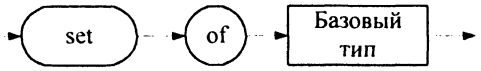


Рис. 4.31. Синтаксическая диаграмма <Объявление множественного типа>

Базовым типом может быть любой порядковый тип за исключением типов integer и longint, количество возможных значений которых превышает 255. В качестве базового типа могут использоваться только диапазоны значений этих типов.

Порядок расположения элементов во множестве никак не фиксируется. Это соответствует принятой в математике трактовке множества.

Новый множественный тип обычно сначала объявляют, а затем уже используют при описании переменных и констант, например:

```
Type
  Digits = set of 1..100;   {тип «множество целых чисел от 1 до 100»}
  Setchar = set of char;    {тип «множество символов таблицы ASCII»}
  letter=set of 'a'..'z';   {тип «множество прописных латинских букв»}
  logic = set of boolean;   {тип «множество логических значений»}
Var
  mychar:setchar;  {переменная – множество символов таблицы ASCII}
  bool: logic;     {переменная – множество логических значений}
  mydig: Digits;   {переменная – множество целых чисел от 1 до 100}
  simst: letter;   {переменная – множество прописных латинских букв}
```

Множественный тип можно определить и непосредственно при объявлении переменных программы, например:

```
Var number:set of 1..31;{переменная – множество целых чисел от 1 до 31}
  cif: set of 0..9;  {переменная – множество цифр}
  kods: set of #0..#255;{переменная – множество кодов таблицы ASCII}
  workweek, week: set of dayn;  {переменная – множество дней недели}
```

Значением переменной множественного типа является множество. Конкретные значения переменных и констант множественного типа определяются с помощью *конструктора множества*, представляющего собой заключенный в квадратные скобки список элементов множества, перечисленных через запятую. Элементы множества могут задаваться константами, переменными и выражениями базового типа, также допускается указывать интервалы значений элементов, входящих во множество, например:

[ ] – пустое множество;
[2,3,5,7,11] – множество, включающее несколько целых чисел;
['a','d','f','h'] – множество, включающее несколько латинских литер;
[1,k] – множество, состоящее из целого числа 1 и текущего значения переменной k;
*[k..2*k]* – множество целых чисел от значения переменной k до результата выражения 2*k;

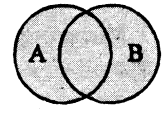
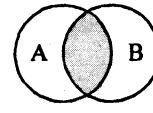
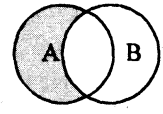
[2..100] – множество целых чисел от 2 до 100;
[1, 2, 3..7] – множество целых чисел, включающее числа 1,2,3,4,5,6,7;
[red,yellow,green] – множество, состоящее из трех элементов некоторого перечислимого типа.

Инициализация множеств. Возможна инициализация переменных множественного типа с использованием типизированных констант. Например:

```
Type setnum=set of byte;
Const S:setnum=[1..10]; {инициализированная переменная, ее исходное
                         значение в программе равно множеству, включающе-
                         му целые числа от 1 до 10}
```

Операции над множествами. Для работы со значениями множественного типа предусмотрены специальные операции, в основном соответствующие операциям, определенным в теории множеств: *объединение*, *пересечение* и *дополнение* множеств. Это двуместные операции, операндами которых являются данные множественного типа – множества и выражения, принимающие значение множественного типа. Оба операнда должны принадлежать одному и тому же множественному типу. Обозначения этих операций в Borland Pascal и их геометрическая интерпретация представлены в табл. 4.1 применительно к множествам A и B.

Таблица 4.1

Математическая запись	Операция Borland Pascal	Геометрическая интерпретация	Результат операции
A ∪ B	***A + B***		Объединение множеств A и B – множество, состоящее из элементов, принадлежащих множествам A и B
A ∩ B	***A * B***		Пересечение множеств A и B – множество, состоящее из элементов, принадлежащих одновременно и множеству A и множеству B
A \ B	***A - B***		Дополнение множеств A и B – множество, состоящее из тех элементов множества A, которые не принадлежат множеству B

Например:

[1,2] + [3,4] = [1,2,3,4];
[1..10] * [3,8,9,15,23,45] = [3,8,9];
[1..15] - [3,8,9,15,23,45] = [1,1,4..7,10..14];
[red,blue,green,black] * [blue,magenta,yellow] = [blue].

Операции отношения. Наряду с рассмотренными выше операциями над значениями множественного типа определены и операции отношения. Операндами отношений в этом случае являются переменные или выражения множественного типа. В табл. 4.2 показано соответствие между математическими операциями сравнения множеств и операциями отношения, определенными в Borland Pascal.

Операция проверки вхождения элемента во множество. Особое место занимает операция проверки вхождения элемента во множество, обозначаемая служебным словом *in* (рис. 4.32). В отличие от операций отношения, в операции *in* первый операнд должен принадлежать *базовому* типу элементов данного множественного типа. Результатом операции *in*, как во всех операциях отношения, является логическое значение.

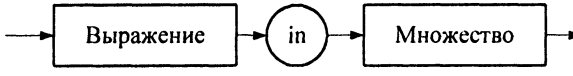


Рис. 4.32. Синтаксическая диаграмма <Проверка вхождения элемента во множество>

Ниже приведены примеры применения операций отношения и операции вхождения к различным множествам.

Таблица 4.2

Математическая запись	Операция Borland Pascal	Результат операции	
		TRUE	FALSE
A = B	***A = B***	Множества A и B совпадают	Множества A и B не совпадают
A ≠ B	***A <> B***	Множества A и B не совпадают	Множества A и B совпадают
A ⊆ B	***A <= B***	Все элементы множества A принадлежат множеству B	Не все элементы множества A принадлежат множеству B
A ⊇ B	***A >= B***	Все элементы множества B принадлежат множеству A	Не все элементы множества B принадлежат множеству A

['a', 'b'] = ['b', 'a'] – результат операции TRUE;
[4, 5, 6] = [4..6] – результат операции TRUE;
['c','b'] = ['c', 'b', 'd'] – результат операции FALSE;
[2, 3, 5,7] <= [1..9] – результат операции TRUE;
[3, 6..8] <= [2..7, 9] – результат операции TRUE;
[3, 6..8] <= [2..7, 9] – результат операции FALSE;
[5..8, 9..12] >= [6, 8, 11] – результат операции TRUE;
10 in [2, 4, 6, 8, 10, 12, 14] – результат операции TRUE;
k in [1, 3, 5, 7, 9] – результат операции TRUE при k=1,3,5
и FALSE при k=2,4,6

Следует помнить, что *значения множественного типа нельзя вводить и выводить.* Однако можно ввести значения элементов множества и добавить их к множеству, используя операцию объединения множеств, например:

```
S:=[ ];   {исходное множество пусто}
Read(n);
while not Eof do
  begin
      S:=S+[n]; {объединяем исходное множество с элементом}
      Read(n);   {вводим следующий элемент}
  end; ...
```

Для того чтобы вывести элементы множества, используют специальный прием: в цикле проверяют вхождение во множество всех элементов базового типа и выводят те, которые входят во множество, например:

```
for i:='a' to 'z' do
   if  i in S then Write(i:3);
```

Рассмотрим несколько примеров решения задач с использованием множеств.

Пример 4.20. Разработать программу, которая определяет, является ли введенное слово идентификатором, т.е. начинается ли оно с буквы или знака подчеркивания и не содержит ли специальных символов.

Строим множество символов, которые допустимы в качестве первого: это строчные и прописные буквы латинского алфавита и символ подчеркивания:

['A'..'Z','a'..'z','_'] .

Аналогично определяем множество допустимых символов, которые могут встретиться, начиная со второго символа слова:

['A'..'Z','a'..'z','_','0'..'9'].

Программа должна вводить строку, проверять допустимость первого символа, а затем в цикле проверять допустимость остальных символов.

```
Program ex;
Var st:string;
    key:boolean;
    i:integer;
Begin
  WriteLn('Введите строку');
  ReadLn(st);
  if st[1] in ['A'..'Z','a'..'z','_'] then {проверка первого символа}
    begin
      i:=2;
      key:=true;
      while (i<=length(st)) and key do  {проверка остальных символов}
      if st[i] in ['A'..'Z','a'..'z','_','0'..'9'] then inc(i)
          else key:=false;
       if key then WriteLn('Строка ',st,' – идентификатор.')
       else WriteLn('Строка ',st,' содержит недопустимые символы.');
    end
  else
    WriteLn('Строка ',st,' начинается с недопустимого символа.');
End.
```

Пример 4.21. Разработать программу для определения количества различных цифр в десятичной записи натурального числа.

Для получения требуемого результата модуль введенного числа преобразуем в строку символов, а затем сформируем множество из этих символов. Теперь проверим, входят ли во множество цифры от 0 до 9, и те, которые входят, выведем на экран.

```
Program ex;
 Var n:longint;
     st:string;
     mnoj:set of '0'..'9';
     i:integer; j:char;
  Begin
    WriteLn('Введите число:');
    ReadLn(n);
    Str(abs(n),st);
    mnoj:=[ ];       {в исходном состоянии множество пусто}
    for i:=1 to length(st) do
                mnoj:=mnoj+[st[i]]; {формируем множество}
   WriteLn('Запись числа ',n,' содержит следующие цифры:');
   for j:='0' to '9' do       {выводим цифры, вошедшие во множество}
        if j in mnoj then Write(j+' ');
  End.
```

Пример 4.22. Разработать программу, которая для строки символов, введенной с клавиатуры и состоящей из нескольких слов, разделенных пробелами, определяет множество гласных, которые:

- встречаются в каждом слове строки;
- встречаются только в одном слове строки;
- встречаются хотя бы в одном слове строки;
- встречаются более чем в одном слове строки.

Для решения задачи определим тип «множество символов ASCII». Множество гласных букв русского языка зададим с помощью типизированной константы множественного типа. Определим переменные множественного типа для хранения результатов и промежуточных значений:

res1 – множество гласных, входящих в каждое слово,
res2 – множество гласных, входящих не более чем в одно слово,
res3 – множество гласных предложения,
res4 – множество гласных, входящих более чем в одно слово,
mnsl – множество гласных, встретившихся в текущем слове.

В программе введем строку и будем последовательно выделять из нее слова. Для каждого слова построим множество встретившихся гласных букв mnsl.

Если в строке содержится одно слово, то

$$res1 = res2 = res3 = mnsl,$$

а множество res4 пусто.

Если в строке более одного слова, то каждое новое слово изменяет результирующие множества следующим образом:

1) множество гласных, входящих в каждое слово, будет равно пересечению уже найденного множества гласных, входящих в каждое слово, и множества гласных слова:

$$res1 \cap mnsl;$$

2) множество гласных, входящих более чем в одно слово, res4 увеличится (объединение) на повторяющиеся буквы нового слова:

$$res4 \cup (res3 \cap mnsl);$$

3) множество гласных в предложении res3 увеличится (объединение) на множество гласных слова:

$$res3 \cup mnsl .$$

Множество гласных, входящих только в одно слово, res2 будем определять после обработки всех слов как разность множества гласных букв предложения и множества гласных, входящих более чем в одно слово предложения:

$$res3 \setminus res4.$$

```
Program ex;
  Type setchar=set of char;
  Const G: setchar = ['а','я','у','ю','э','е','о','ё','и','ы'];  {типизированная константа «множество гласных букв»}
  Var res1,  {множество гласных, входящих в каждое слово}
      res2,  {множество гласных, входящих только в одно слово}
      res3,  {множество гласных в предложении}
      res4,  {множество гласных, входящих более чем в одно слово}
      mnsl:setchar;     {множество гласных текущего слова}
      st,slovo: string;       ch: char;
      i,k:integer;      first:boolean;
Begin
  WriteLn('Введите исходную строку:');
  ReadLn(st);  {читаем исходную строку}
  st:=st+' ';  {добавляем в конец пробел для простоты обработки}
  first:=true;  {признак «первое слово»}
  while st<>'' do  {цикл выделения и обработки слов}
    begin
      k:=pos(' ',st);
      slovo:=Copy(st,1,k-1);  {выделяем слово}
      Delete(st,1,k);              {удаляем слово из строки}
      {определяем множество гласных, входящих в данное слово}
      mnsl:=[];              {исходное состояние «пустое множество»}
      for i:=1 to k-1 do
        if slovo[i] in G then                {если гласная буква, то}
            mnsl:=mnsl+[slovo[i]];    {добавляем к множеству}
      {формируем множества результатов}
      if first then  {если первое слово, то}
        begin
          res1:=mnsl;  {входят в каждое слово}
          res2:=mnsl;  {входят не более чем в одно слово}
          res3:=mnsl;  {встретившиеся гласные}
          res4:=[];    {входят более чем в одно слово}
          first:=false;  {выключаем признак «первое слово»}
        end
      else     {если не первое слово предложения, то}
        begin
          res1:=res1*mnsl;        {входят в каждое слово}
          res4:=res4+res3*mnsl;  {входят более чем в одно слово}
          res3:=res3+mnsl;        {встретившиеся гласные}
        end
    end;
  res2:=res3-res4;  {входящие в одно слово}
```

```
{выводим результаты анализа предложения}
WriteLn('Гласные, которые входят в каждое слово:');
for ch:=#0 to #255 do if ch in res1 then Write(ch:2);
WriteLn;
WriteLn('Гласные, входящие только в одно слово:');
for ch:=#0 to #255 do if ch in res2 then Write(ch:2);
WriteLn;
WriteLn('Гласные, входящие хотя бы в одно слово:');
for ch:=#0 to #255 do if ch in res3 then Write(ch:2);
WriteLn;
WriteLn('Гласные, входящие более чем в одно слово:');
for ch:=#0 to #255 do if ch in res4 then Write(ch:2);
WriteLn;
End.
```

Задания для самопроверки

Задание 1. Дан текст, содержащий несколько слов, разделенных пробелом, в конце точка. Разработайте программу, которая, используя множественный тип, определяет количество слов текста, содержащих специальные символы «@, #, $, ^, &, _, *, %, ~». Вывести исходную последовательность, количество искомых слов, а также их номера в тексте.

Задание 2. Дан текст, содержащий несколько слов, разделенных пробелом, в конце точка. Разработайте программу, которая, используя множественный тип, удаляет из последовательности все слова, включающие хотя бы одну цифру. Вывести исходный и преобразованный текст. Если слов с цифрами не окажется, вывести соответствующее сообщение.

Задание 3. Дана строка, состоящая из последовательности целых чисел в символьном изображении. Числа разделены пробелами (например: «345 6785 1235 54 657»). Разработайте программу, которая формирует множество цифр, присутствующих в записи всех чисел последовательности. Вывести исходную последовательность и сформированное множество. (Для приведенной последовательности множество состоит из одного элемента 5.)

Задание 4. Дана строка, содержащая последовательность слов, разделенных пробелами. В словах содержатся буквы латинского и русского алфавита. Разработайте программу, которая формирует и выводит в алфавитном порядке два множества: множество латинских и множество русских строчных букв (кроме ё), встретившихся в исходной строке.

Задание 5. Дана строка, содержащая последовательность слов из латинских строчных букв, разделенных пробелами. Разработайте программу, которая, используя множество, вычеркивает из каждого слова строки буквы i, j, k, l, m, n. Вывести исходную и преобразованную строки.

4.8. Записи

Запись – это структура данных, состоящая из фиксированного числа *разнотипных* компонент, называемых *полями* записи. Записи используются для представления разнородной, но логически связанной информации. Каждое поле записи имеет имя, которое дается ему при объявлении записи.

В Borland Pascal определены записи двух типов: записи с фиксированными полями и вариантные записи (рис. 4.33).

Записи с фиксированными полями. Синтаксическая диаграмма записи с фиксированными полями представлена на рис. 4.34.

Как любой тип данных языка, записи можно определить двумя способами:

- при объявлении переменных, например:

```
Var Zap1,Zap2: record {две записи, состоящие из 5 полей}
                      F,S:real; {два поля вещественного типа}
                      A,B:integer; {два поля целого типа}
                      C:char;      {поле символьного типа}
                    end;
    Zap3: record {запись, состоящая из 3 полей}
                      S: string [80];{символьная строка длиной 80 байт}
                      A: array [1..20] of real; {одномерный массив на 20
                                                 вещественных чисел}
                      Flag:: boolean;  {поле логического типа}
                    end; ...
```

- предварительно объявив тип записи, например:

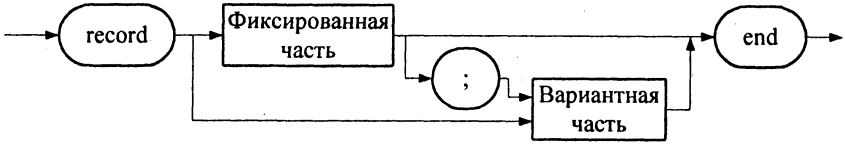


Рис. 4.33. Синтаксическая диаграмма <Объявление типа записи>

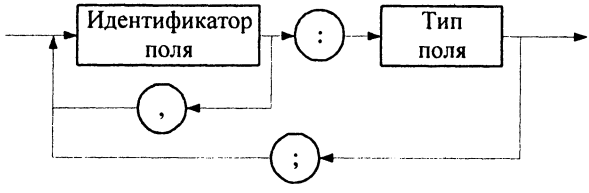


Рис. 4.34. Синтаксическая диаграмма <Фиксированная часть записи>

```
Type  Zt1 = record                {тип «запись, состоящая из 5 полей»}
              F,S:real;            {два поля вещественного типа}
              A,B:integer;         {два поля целого типа}
              C:char;              {поле символьного типа}
            end;
      Zt2 = record                 {тип «запись, состоящая из 3 полей»}
              S: string [80];      {символьная строка длиной 80 байт }
              A: array [1..20] of real;{одномерный массив на
                                        20 вещественных чисел}
              Flag: boolean;       {поле логического типа}
            end;
Var   Zap1, Zap2:Zt1;      {две переменные типа Zt1}
      Zap3:Zt2; ...        {переменная типа Zt2 }
```

В качестве полей записи можно использовать другие записи, определенные как ранее, так и внутри записи, например:

```
Type Human=record          {запись о сотруднике}
       Fio: record                 {поле типа «запись из 3 полей»}
              Fam,                 {фамилия}
              Name,                {имя}
              Otch: string;        {и отчество сотрудника»}
            end; {Fio}
       BirthDay: record            {поле типа «запись из 3 полей»}
                   Day:1..31;      {день}
                   Month: 1..12;   {месяц}
                   Year: word;     {год рождения}
                 end; {Birthday}
     end; ...{Human}
```

или

```
Type Data = record
              Day:1..31;
              Month: 1..12;
              Year: word;
            end;
     Famio = record
               Fam, Name, Otch: string;
             end;
     Human = record   {тип «запись о сотруднике»}
               Fio: Famio;     {поле типа Famio }
               BirthDay: Data {поле типа Data }
             end; ...
```

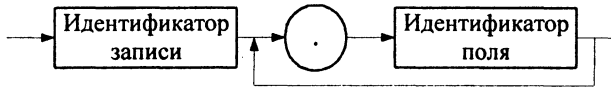


Рис. 4.35. Синтаксическая диаграмма <Доступ к полям записи >

Можно объявить как отдельные переменные программы, так и массивы записей, например:

```
Var  Sotr: Human;
     Otdel: array [1..20] of Human;
```

Инициализация записей. Присвоить начальное значение конкретной записи можно, используя типизированные константы. Начальное значение полей записи при этом указывается в скобках через точку с запятой, причем для каждого поля указывается имя и значение через двоеточие, например:

```
Const
     BirthDay: Data = (Year:1973; Month:6; Day:30);  ...
```

Операции над записями. Над записями возможно выполнение следующих операций.

Доступ к полям записи. Синтаксическая диаграмма доступа к полям записи представлена на рис. 4.35.

Например, к полям переменной Sotr типа Human, объявленного выше, можно обратиться следующим образом:

```
Sotr.BirthDay.Day:=25;
m:=Sotr.BirthDay.Year; ...
```

В том случае, если доступ к полям записи осуществляется многократно, целесообразно обращаться к полям записи с использованием оператора присоединения with (рис. 4.36). Например, для переменной Sotr типа Human возможны следующие варианты доступа к полю Day с применением оператора with:

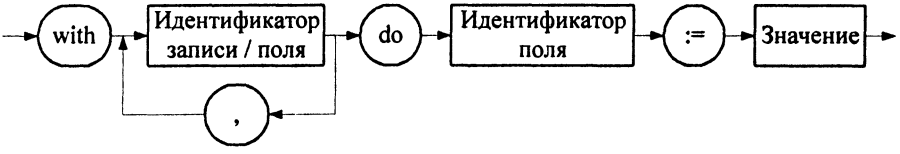


Рис. 4.36. Синтаксическая диаграмма <Оператор присоединения>

а) *with Sotr do BirthDay.Day:=30;*
б) *with Sotr.BirthDay do Day:=24;*
в) *with Sotr, BirthDay do Daay:=31;*
г) *with Sotr do*
with BirthDay do Day:= 7; ...

Присваивание записей. Операция возможна при совпадении типов записей и выполняется последовательно поле за полем. Например:

Otdel[i]:=Sotr; ...

Ввод записей с клавиатуры и *вывод* их на экран выполняются по полям по правилам ввода переменных соответствующих типов.

Пример 4.23. Разработать программу, которая вводит в массив записей информацию о студентах учебной группы: фамилию и дату рождения. Организовать поиск информации о студенте, фамилия которого вводится с клавиатуры.

```
Program ex;
Type
  data=record     {тип запись данные о дате}
           year:word;     {год}
           month:1..12;   {месяц}
           day:1..31;     {день}
         end;
  zap=record       {тип запись о студенте}
            fam:string[16];  {фамилия}
            birthday:data;   {дата рождения}
        end;
Var fb:array[1..25] of zap; { массив данных о группе студентов}
    fff:string;  {строка для ввода фамилии}      i,j,m,n:byte;
    key:boolean; {ключ поиска, если фамилия найдена - true}
Begin
 WriteLn('Введите данные о количестве студентов n<=25');
 ReadLn(n);
 m:=0;
{ввод исходных данных с клавиатуры поле за полем}
  repeat              m:=m+1;
    Write('Введите фамилию :');        Readln(fb[m].fam);
    Write('Введите год рождения : ');  Readln(fb[m].birthday.year);
    Write('      месяц      : ');      ReadLn(fb[m].birthday.month);
    Write('      день       : ');      ReadLn(fb[m].birthday.day);
 until n=m;
 WriteLn;
```

```
{вывод исходных данных на экране с помощью оператора with}
 WriteLn('Список студентов группы '); WriteLn;
 for i:=1 to m do
     with fb[i] do
       begin
            Write(i:2, fam:17);
            with birthday do
                    WriteLn(year:6, month:4, day:4);
       end;
     WriteLn;
     {поиск данных в массиве записей}
     WriteLn('Введите фамилию');
     ReadLn(fff);
     i:=0;
     key:=false;  {признак «данные не найдены»}
     repeat i:=i+1;
          if fb[i].fam = fff then key:=true
     until key or( i=m);
     {вывод результата}
     if key then   {если такой студент найден, то выводим данные}
          with fb[i] do
            begin
              WriteLn('Данные о студенте :');
              Write(fam:18,'  ');
              with birthday do
                    WriteLn(day:2,':',month:2,':',year:5,' года');
            end
       else  WriteLn('Данных о студенте :',fff:18,' нет.');
End.
```

Записи с вариантами. Иногда бывает удобно и естественно рассматривать несколько типов данных как варианты одного, т. е. в пределах одной записи иметь различную информацию в зависимости от конкретного значения некоторых полей. Для обеспечения такой возможности запись кроме фиксированного списка полей может иметь еще и вариантную часть (4.37).

Из диаграммы видно, что вариантная часть может содержать несколько альтернатив, в каждой из которых задается список полей, присущих данному варианту. Каждой альтернативе предшествует константа, идентифицирующая соответствующий вариант.

Рассмотрим несколько примеров определения записей с вариантами:

- без предварительного описания типа:

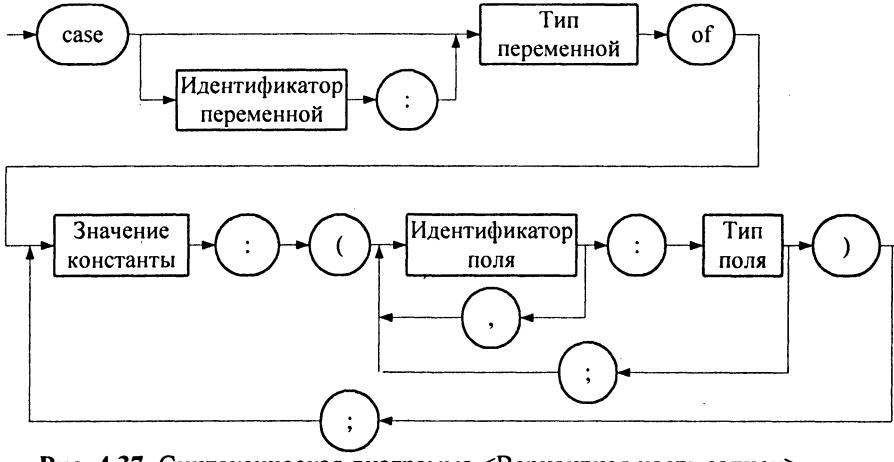


Рис. 4.37. Синтаксическая диаграмма <Вариантная часть записи>

```
Var  M: record
        case {вариантная часть}
         byte of   {тип констант для идентификации вариантов}
           0: (by: array [0..3] of byte); {массив из 4 байт}
           1: (wo: array [0..1] of word); {массив из 2 слов}
           2: (lo: longint); {переменная длиной 4 байта}
        end; ...
```

- с предварительным объявлением типа:

```
Type
    Figure = (Square, Triangle, Circle);    {перечисляемый тип}
    Paramf = record                 {тип – запись с вариантами}
                X,Y: real;          {фиксированные поля}
             case                   {вариантная часть}
                Fig:Figure of { переменная и тип
                                  идентифицирующих констант}
             Square:    (Side: real);
             Triangle:  (Side1,Side2,Side3: real);
             Circle:    (Radius: real);
             end;
Var  Param: Paramf;     {объявление переменной}
```

Конкретное значение переменной типа «запись с вариантами» присваивается точно так же, как и обычной записи: вводом с клавиатуры, с помощью литеральной константы и с помощью типизированной константы. Однако при объявлении инициализированной переменной для вариантной части задается только один вариант.

Например, пусть запись должна содержать либо данные о человеке, либо о корабле (так, Георгий Седов – это и имя человека, и название корабля). Для этого опишем следующую запись с вариантами

```
Type
  Forma = record {запись места прописки человека и корабля}
  case  {вариантная часть}
     boolean of  {тип констант, идентифицирующих варианты}
  {вариант для человека}
   True: (BirthPlace: string[40]; {место проживания}
  {вариант для корабля}
   False: (Country: string[20];    {страна}
           EntryPort: string[20]; {порт}
           EntryDate: array[1..3] of word; {дата}
           Count: word) {водоизмещение}
end; ...
```

Типизированные константы для задания исходных данных двух записей: Object2 – сведения о человеке, Object1 – сведения о корабле определяются так:

```
Const   Object1: Forma = (Country: 'Польша';
                          EntryPort: 'Ленинград';
                          EntryDate:(16,3,89);
                          Count:12);
        Object2: Forma = (BirthPlace: 'Москва'); ...
```

Следует отметить, что идентификаторы полей во всех вариантах должны быть различны, а также не совпадать с именами полей фиксированной части. Для некоторых значений констант, идентифицирующих вариант, поля в вариантной части записи могут отсутствовать, тогда после двоеточия можно поместить пустой список «( )».

Кроме того, при использовании записей с вариантами необходимо учитывать некоторые особенности. Так, для размещения переменной типа запись всегда отводится фиксированный объем памяти в соответствии с объемом, занимаемым самым большим из вариантов, т. е. различные варианты размещаются на одном участке памяти, как бы «накладываясь» друг на друга.

Следует также учитывать, что транслятор языка не содержит никаких средств контроля корректной обработки вариантных записей. Это означает, что в любое время возможен доступ ко всем полям во всех вариантах. За соответствием хранимой информации и вариантом доступа к ней должен следить сам программист. Однако именно эта особенность записей с вариантами может быть использована для неявного преобразования типов данных.

Поскольку различные варианты ссылаются на один участок памяти, как бы «накладываясь» друг на друга, можно обращаться к содержимому памяти поочередно, то как к переменной одного типа, то как к переменной другого. Например:

```
Type
 Perem= record   {запись с вариантами}
          case byte of
           0: Wo:word;      {переменная типа word}
           1: Lo:longint;   {переменная типа longint}
           2: Re:real;      {переменная типа real}
        end;
Var  C:Perem;
Begin ...
  C.Lo:=0;  {очищаем область}
  C.Wo:=10; {в вариантное поле по шаблону целого без знака записываем число 10}
  WriteLn(C.Lo:10);... {печатается содержимое вариантного поля по шаблону длинного целого}
```

В этом примере под поля будет выделено 6 байт памяти в соответствии с самым длинным типом во внутреннем представлении. Пользователь может работать с этим полем по любому из шаблонов, используя идентификаторы соответствующих полей записи.

Задания для самопроверки

Задание 1. Разработайте программу, которая, используя тип запись, формирует массив данных о сотрудниках отдела, содержащий следующую информацию: фамилию, год поступления в отдел, стаж работы в отделе, общий стаж работы. Затем сортирует полученный массив в соответствии со стажем работы в отделе и выводит первые пять фамилий из отсортированного списка. После чего определяет среди первых пяти сотрудников сотрудника, у которого общий стаж наибольший.

Задание 2. Разработайте программу, которая формирует массив записей о студентах некоторой группы, содержащий следующую информацию: фамилию, оценки за последнюю сессию по четырем предметам и размер стипендии. Фамилию и отметки программа должна вводить с клавиатуры, а размер стипендии считать исходя из оценок: все «5» – повышенная (+ 25%), есть одна «4» – повышенная (+10%), нет троек – обычная стипендия, есть одна тройка – социальная стипендия (-15%), больше одной тройки – стипендия 0. После чего программа должна сортировать массив по размеру стипендии и выводить его на экран. Предусмотреть ввод размера обычной стипендии с клавиатуры.

5. МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Большие программы обычно разрабатывают и отлаживают по частям. Целесообразно при этом, чтобы каждая такая часть, называемая *подпрограммой*, была оформлена так, чтобы ее можно было использовать при решении аналогичной подзадачи в той же программе или даже при решении других задач. В Borland Pascal реализованы два типа подпрограмм: процедуры и функции.

5.1. Процедуры и функции

Процедуры и функции представляют собой относительно самостоятельные фрагменты программы, соответствующим образом оформленные и снабженные именем (программные блоки). По правилам Borland Pascal программные блоки – такие же ресурсы, как типы и переменные. Соответственно, они также должны быть описаны перед использованием в разделе описаний программного блока, который их использует (основной программы или вызывающей подпрограммы). Каждый блок имеет такую же структуру, как основная программа, т.е. включает заголовок, раздел описаний и раздел операторов, но заканчивается не точкой, а точкой с запятой (рис. 5.1).

Заголовок блока определяет форму вызова подпрограммы. В разделе описаний блока объявляют внутренние локальные ресурсы блока (переменные, типы, внутренние подпрограммы). Раздел операторов содержит инструкции подпрограммы в операторных скобках begin...end.

Заголовки процедур и функций описываются по-разному. В отличие от процедуры функция всегда возвращает в точку вызова скалярное значение, адрес или строку. Тип возвращаемого результата описывается в заголовке функции (рис. 5.2).

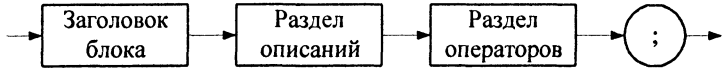


Рис. 5.1. Синтаксическая диаграмма конструкции <Программный блок>

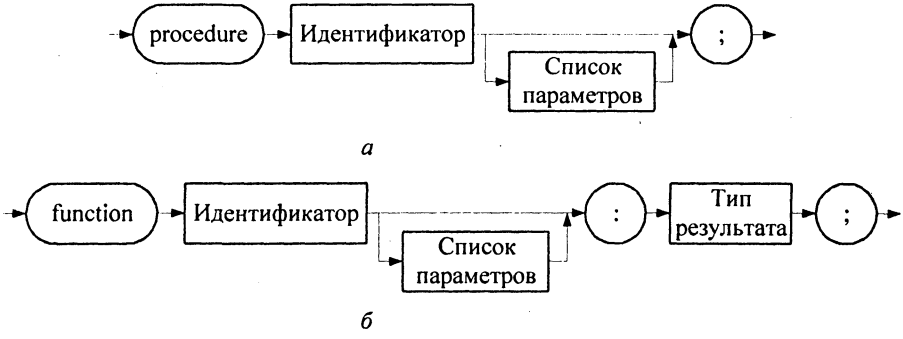


Рис. 5.2. Синтаксические диаграммы конструкций <Заголовок процедуры> (*а*) и <Заголовок функции> (*б*)

Данные для обработки процедуры и функции получают из вызвавшей их основной программы или подпрограммы. Для размещения рабочих полей подпрограммы могут объявлять новые типы и переменные в собственном разделе описаний. Результаты же они обычно должны возвращать вызвавшей программе или подпрограмме.

Из основной программы данные могут быть получены:

- неявно – с использованием глобальных констант и переменных;
- явно – через параметры.

Неявная передача данных в подпрограммы. Каждой подпрограмме доступны все ресурсы программного блока, в разделе описаний которого эта подпрограмма объявлена. Ресурсы же основной программы доступны в любой подпрограмме. Они получили название *глобальных*.

В свою очередь *локальные* ресурсы, объявленные в разделе описаний подпрограммы, из программного блока, в разделе описания которого она определена, не доступны. В том случае, если в подпрограмме объявляется ресурс, имя которого совпадает с именем глобального ресурса, соответствующий глобальный ресурс в подпрограмме становится не доступным, «перекрывается».

Например, на рис. 5.3 в разделе описаний основной программы объявлена подпрограмма A, в разделе описаний которой объявлена подпрограмма B. Переменная x доступна в обеих подпрограммах, так как она не перекрывается. Переменная z основной программы в подпрограмме A не доступна, так как эта подпрограмма перекрывает глобальную переменную z локальной. В подпрограмме B используется значение z, определенное в подпрограмме A.

Опыт показывает, что неявная передача данных в подпрограммы обычно приводит к большому количеству ошибок, которые достаточно сложно искать, так как неизвестно, какая подпрограмма с какими глобальными ресурсами работает. Кроме того, подпрограммы, использующие глобальные дан-

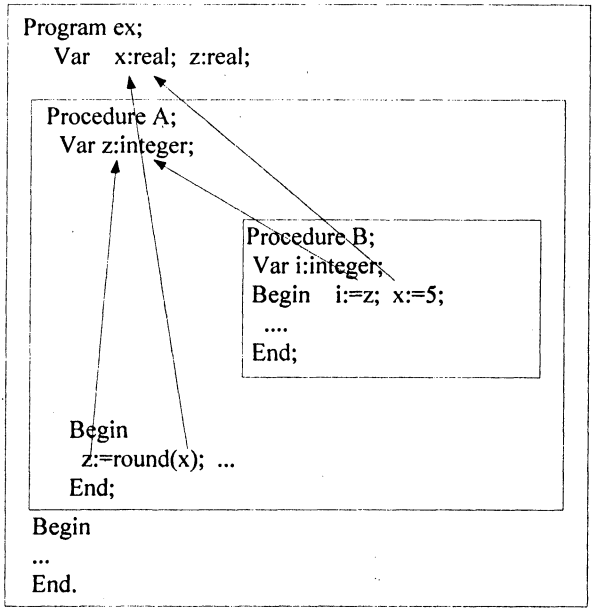


Рис. 5.3. Перекрытие ресурсов в подпрограммах

ные, невозможно перенастроить на работу с другими значениями, что сокращает возможности их применения. Поэтому желательно использовать явную передачу значений – через параметры.

Передача данных через параметры. Список параметров описывается в заголовке подпрограммы (рис. 5.4). Параметры, перечисленные в этом списке, получили название *формальных*, так как для их размещения не отводится память. При обращении к подпрограмме для каждого параметра должно быть указано *фактическое* значение – литерал, константа или переменная того же типа, что и формальный параметр. Несоответствие типов и количества формальных и фактических параметров выявляется компилятором (или компоновщиком, если вызов подпрограммы происходит из другого модуля – см. далее). Нарушение порядка следования фактических параметров, если

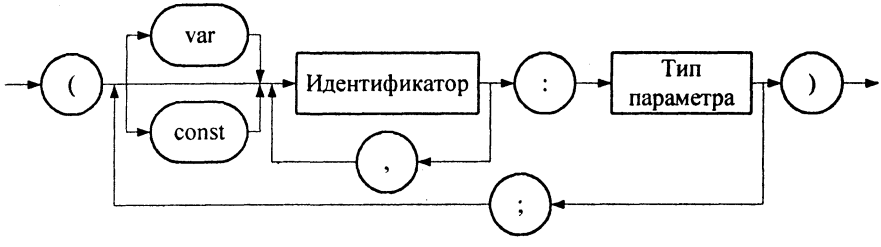


Рис. 5.4. Синтаксическая диаграмма конструкции <Список параметров>

это нарушение не связано с несовпадением количества параметров или их типов, приводит к нарушению логики работы программы и часто может быть обнаружено только при тестировании программы.

В Borland Pascal параметры в подпрограмму могут передаваться тремя способами:

- как *значения* – в подпрограмму передаются копии значений параметров, и никакие изменения этих копий не возвращаются в вызывающую программу;
- как *переменные* – в подпрограмму передаются адреса фактических параметров, соответственно все изменения этих параметров в подпрограмме на самом деле происходят с переменными, переданными в качестве фактических параметров; такие параметры при описании помечаются служебным словом var; *в качестве фактических значений параметров-переменных нельзя использовать литералы*;
- как *неизменяемые переменные* (именованные *константы*) – в подпрограмму, так же как и в предыдущем случае, передаются адреса фактических параметров, но при попытке изменить значение параметра компилятор выдает сообщение об ошибке; такие параметры при описании помечаются служебным словом const.

Вызов процедур и функций. И процедура, и функция, используя параметры-переменные, могут изменять значения переменных основной программы. Но как отмечалось выше, функция отличается от процедуры тем, что кроме изменения значений параметров-переменных всегда возвращает в точку вызова скалярное значение, строку или указатель. Поэтому в теле функции обязательно наличие специальной переменной с именем функции, которой должно присваиваться значение. Именно это значение и будет возвращено в место вызова функции в качестве ее результата. Вызов функции, таким образом, можно осуществлять в составе выражений везде, где возможно использование выражений (в операторе присваивания, в операторе вывода и т.д.), например:

<переменная>:=<имя функции>(<фактические параметры>).

Процедура же должна вызываться отдельным оператором, состоящим из имени процедуры и списка фактических параметров:

<имя процедуры>(<фактические параметры>).

Вызов процедуры и функции по-разному изображается на схеме алгоритма: вызов функции – в блоке «процесс» или блоке вывода, а для вызова процедуры используется специальный блок «предопределенный процесс». Схемы алгоритмов же самих подпрограмм в обоих случаях оформляются отдельно, причем вместо слова «начало» указывают имя подпрограммы, а вместо слова «конец» – указывают слово «возврат» или «return».

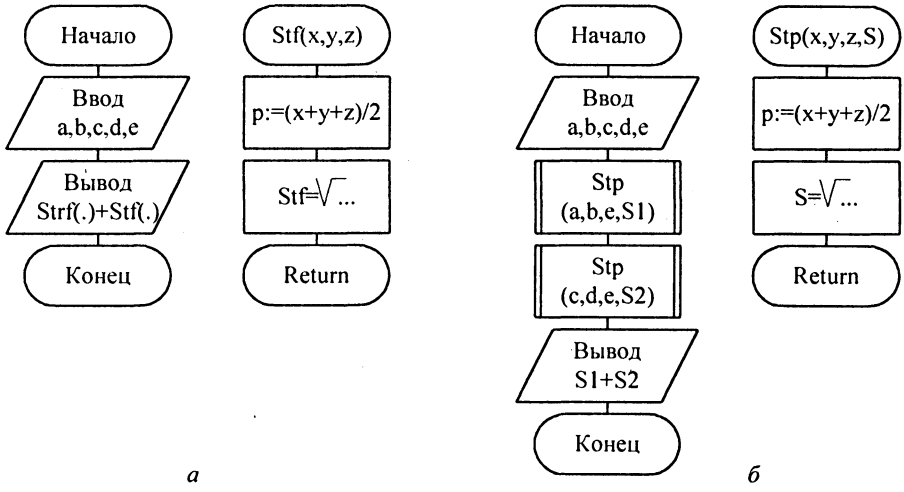


Рис. 5.5. Схемы алгоритмов программы определения площади четырехугольника с использованием функции (*а*) и процедуры (*б*)

Пример 5.1. Разработать программу, которая определяет площадь четырехугольника по заданным длинам сторон и диагонали.

Будем считать площадь четырехугольника как сумму площадей двух треугольников, определенных по формуле Герона. Вычисление площади треугольника оформим как подпрограмму. Исходные данные такой подпрограммы – длины сторон треугольника. Подпрограмма не должна менять значения параметров, поэтому их можно передать как параметры-значения или параметры-константы. Результат работы этой подпрограммы – скалярное значение, значит, она может быть реализована как функция. Однако ее также можно реализовать как процедуру, которая возвращает результат через параметр-переменную. Схемы алгоритма данной программы с использованием подпрограмм обоих типов приведены на рис. 5.5.

Ниже приведены тексты соответствующих программ.

Вариант с использованием функции:

```
Program ex;
  Var A,B,C,D,E:real;    {глобальные переменные}
{описание функции}
  Function Stf(const X,Y,Z:real):real;
    Var p:real;          {локальная переменная}
    begin                {раздел операторов функции}
        p:=(X+Y+Z)/2;
        Stf:=sqrt(p*(p-X)*(p-Y)*(p-Z));
    end;
```

```
{раздел операторов основной программы}
Begin
  WriteLn('Введите длины сторон и диагонали');
  ReadLn(A,B,C,D,E);
  WriteLn('Площадь четырехугольника', Stf(A,B,E)+Stf(C,D,E):7:3);
End.
```

Вариант с использованием процедуры:

```
Program ex;
 Var A,B,C,D,E:real; S1,S2:real; {глобальные переменные}
{описание процедуры}
 Procedure Stp(const X,Y,Z:real;var S:real);
   Var p:real;              {локальная переменная}
   begin                    {раздел операторов процедуры }
      p:=(X+Y+Z)/2;
      S:=sqrt(p*(p-X)*(p-Y)*(p-Z));
   end;
{раздел операторов основной программы}
Begin
   WriteLn('Введите длины сторон и диагонали');
   ReadLn(A,B,C,D,E);
   Stp(A,B,E,S1);  {вызов процедуры}
   Stp(C,D,E,S2);  {вызов процедуры}
   WriteLn('Площадь четырехугольника ',S1+S2:7:3);
End.
```

Использование параметров структурных типов, таких, как массивы, строки (кроме типа string), множества, записи, файлы, имеет одну общую особенность: тип таких параметров должен быть предварительно объявлен в инструкции описания типа type.

Например:

```
Type mas=array[1..10] of real;
     str80=string[80];
procedure A(M:mas; fout:str80);...
```

Пример 5.2. Разработать подпрограмму суммирования элементов массива размерности n, $n \le 10$.

Поскольку результат – скалярное значение, то будем использовать подпрограмму-функцию. Описание типа «массив из 10 целых чисел» должно быть выполнено отдельно в разделе описаний. Новый тип mas затем исполь-

зуется при объявлении массива a и при объявлении типа массива, передаваемого через параметр-значение. Полный текст программы представлен ниже.

```
Program ex;
 Type mas=array[1..10] of integer; {тип «массив из 10 целых чисел»}
 Var a:mas;
     i,n:integer;
 Function sum(b:mas; n:integer):integer;
  Var s:integer;      i:integer;
  Begin s:=0;
        for i:=1 to n do s:=s+b[i];
        sum:=s;
  End;
Begin
      ReadLn(n);
      for i:=1 to n do Read(a[i]);
      ReadLn;
      WriteLn('Сумма=',sum(a,n));
End.
```

5.2. Практикум. Выделение подпрограмм методом пошаговой детализации

Метод пошаговой детализации позволяет не только разрабатывать алгоритмы, но и выделять подпрограммы, соблюдая все требования, предъявляемые структурным программированием.

Пример 5.3. Разработать программу исследования элементарных функций, которая для функций y = sin x, y = cos x, y = tg x, y = ln x, y = e^x выполняет следующие действия:

- строит таблицу значений функции на заданном отрезке с заданным шагом;
- определяет корни функции на заданном отрезке;
- определяет максимум и минимум функции на заданном отрезке.

Программы такого рода обычно взаимодействуют с пользователем через меню. В данной программе будет использовано два меню: меню функций, в котором пользователь будет выбирать функцию, и меню операций, в котором пользователь будет выбирать вид обработки.

Меню функций (рис. 5.6, *а*) должно выводиться на экран при запуске программы. После ввода номера функции на экране должно появиться меню операций (рис. 5.6, *б*). Введя номер операции, пользователь должен увидеть на экране запрос на ввод данных, необходимых для выполнения этой операции (рис. 5.6, *в-д*). Задав данные, пользователь должен получить на экране

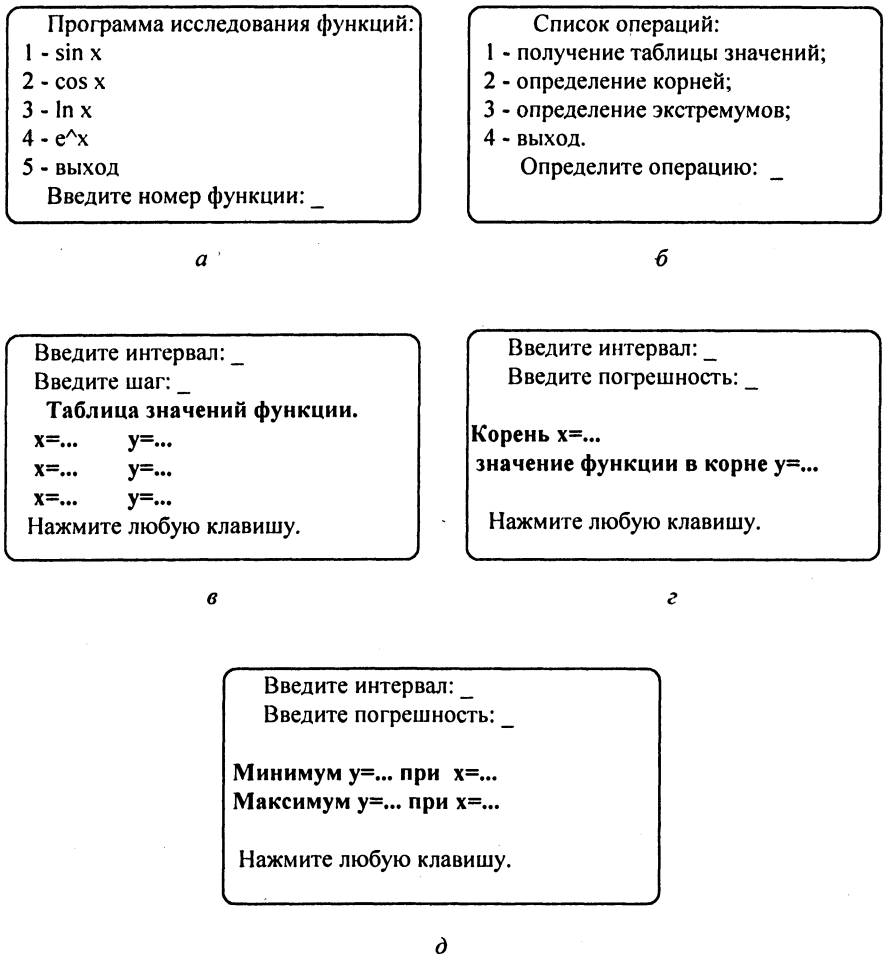


Рис. 5.6. Состояния интерфейса программы исследования функций:
а – меню функций; *б* – меню операций; *в–д* – оформление операций

результаты. После анализа результатов пользователь должен нажать любую клавишу, чтобы вернуться в меню операций.

Далее пользователь может выбрать другую операцию, а может вернуться в меню функций, если выберет операцию 4 (выход). Вернувшись в меню функций, пользователь может выбрать другую функцию или выйти из программы.

На рис. 5.7 представлена диаграмма переходов состояний интерфейса пользователя, из которого видно, в какой последовательности и по нажатию каких клавиш осуществляется переключение форм (экранов) интерфейса.

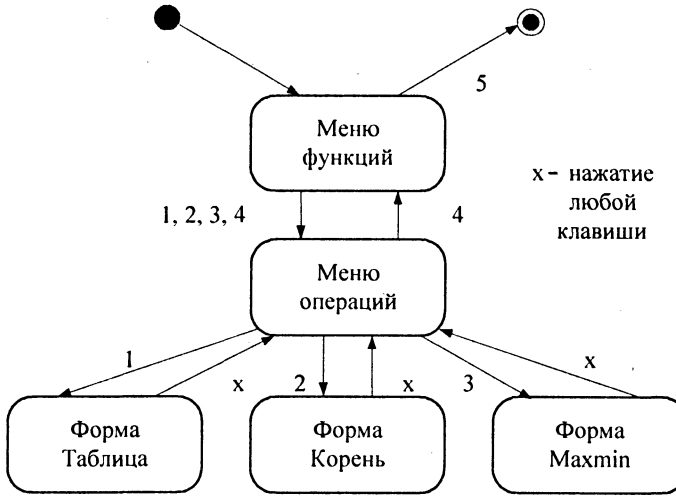


Рис. 5.7. Диаграмма переходов состояний интерфейса программы исследования функции

При разработке алгоритма методом пошаговой детализации будем использовать псевдокод. Начинаем разработку алгоритма «сверху», т.е. с реализации меню функций.

Меню функций работает следующим образом. При запуске программы выводим меню на экран. Затем вводим номер функции и, если номер функции не равен 5, передаем управление меню операций, сообщая ему номер выбранной функции. Когда меню операций завершит свою работу, то на экран вновь необходимо вывести меню функций и опять ввести номер функции:

Программа исследования элементарных функций:
ВЫВЕСТИ_МЕНЮ_ФУНКЦИЙ.
Ввести Номер_функции.
Цикл-пока Номер_функции <> 5
Вызвать МЕНЮ_ОПЕРАЦИЙ (Номер_функции)
ВЫВЕСТИ_МЕНЮ_ФУНКЦИЙ.
Ввести Номер_функции.
Все-цикл.
Конец.

На этом шаге определились две подпрограммы: ВЫВЕСТИ_МЕНЮ_ФУНКЦИЙ и МЕНЮ_ОПЕРАЦИЙ (Номер_функции). Подпрограмма ВЫВЕСТИ_МЕНЮ_ФУНКЦИЙ будет иметь линейную структуру:

Подпрограмма ВЫВЕСТИ_МЕНЮ_ФУНКЦИЙ:
ОЧИСТИТЬ_ЭКРАН.
Вывести «Программа исследования функций.»
Вывести «1 – sin x;».
Вывести «2 – cos x;».
Вывести «3 – ln x;».
Вывести «4 – e^x;».
Вывести «5 – Выход.».
Конец подпрограммы.

Подпрограмма МЕНЮ_ОПЕРАЦИЙ (Номер_функции) также должна реализовать меню:

Подпрограмма МЕНЮ_ОПЕРАЦИЙ (Номер_функции)**:**
ВЫВЕСТИ_МЕНЮ_ОПЕРАЦИЙ.
Ввести Номер_операции.
Цикл-пока Номер_ операции <> 4
Выбор Номер_ операции:
1: Вызвать ТАБЛИЦА (Номер_функции);
2: Вызвать КОРЕНЬ (Номер_функции);
3: Вызвать МАКСМИН (Номер_функции);
Все-выбор.
ВЫВЕСТИ_МЕНЮ_ОПЕРАЦИЙ. .
Ввести Номер_ операции.
Все-цикл.
Конец подпрограммы.

Подпрограмма ВЫВЕСТИ_МЕНЮ_ОПЕРАЦИЙ похожа на ВЫВЕСТИ_МЕНЮ_ФУНКЦИЙ:

Подпрограмма ВЫВЕСТИ_МЕНЮ_ОПЕРАЦИЙ:
ОЧИСТИТЬ_ЭКРАН.
Вывести «Список операций:».
Вывести «1 – получение таблицы значений;».
Вывести «2 – определение корней;».
Вывести «3 – определение экстремумов;».
Вывести «4 – выход.».
Вывести «Определите операцию: ».
Конец подпрограммы.

Подпрограмма ТАБЛИЦА(Номер_функции) должна вводить дополнительные данные о границах интервала и значении шага:

```
Подпрограмма ТАБЛИЦА(Номер_функции):
   ОЧИСТИТЬ_ЭКРАН.
   Вывести «Введите границы интервала»
   Ввести A, B.
   Вывести «Введите шаг».
   Ввести h.
   x=A
   Цикл-пока x ≤ B
       Если ФУНКЦИЯ(Номер_функции, x, y)
          то Вывести «x=», x, « y=», y
          иначе «x=», x, «значение функции не определено»
       Все-если
       x=x+h
   Все-цикл.
   ОЖИДАТЬ_НАЖАНИЯ_КЛАВИШИ.
Конец подпрограммы.
```

Подпрограммы КОРЕНЬ(Номер_функции) и МАКСМИН(Номер_функции) определяются аналогично. Естественно, они также обращаются к подпрограмме ФУНКЦИЯ для получения конкретного значения функции в точке.

Подпрограмма ФУНКЦИЯ будет возвращать логическое значение: true – если значение функции определено, и false – в противном случае. Значение исследуемой функции в точке x подпрограмма будет возвращать через параметр-переменную y:

```
Подпрограмма ФУНКЦИЯ(Номер_функции, x, y):
   ФУНКЦИЯ=true
   Выбор Номер_функции:
       1: y=sin(x);
       2: y=cos(x);
       3: Если x>0
             то y=ln(x)
             иначе ФУНКЦИЯ=false
          Все-если
       4: y=exp(x)
   Все-выбор
Конец подпрограммы.
```

Таким образом, выполняя пошаговую детализацию программы, мы осуществили ее декомпозицию на основную программу и семь подпрограмм (подпрограммы ОЧИСТИТЬ_ЭКРАН и ОЖИДАТЬ_НАЖАНИЯ_КЛАВИШИ являются стандартными процедурами библиотеки управления экраном в текстовом режиме и их можно не учитывать).

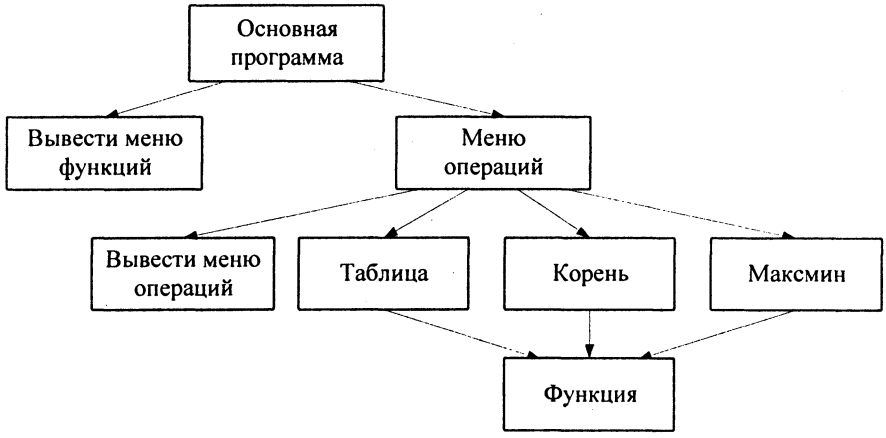


Рис. 5.8. Иерархия вызовов подпрограмм программы исследования функций

Результат процедурной декомпозиции обычно представляют в виде схемы, которая показывает иерархию вызовов подпрограмм (рис. 5.8).

В соответствии с рекомендациями структурного программирования реализация и отладка программы также должны выполняться поэтапно. Сначала реализуют основную программу, используя вместо подпрограмм следующего уровня «заглушки» (подпрограммы, у которых отсутствуют операторы между begin и end). Затем реализуют эти подпрограммы, используя заглушки вместо подпрограмм следующего уровня и т.д.

Задания для самопроверки

Задание 1. Дана матрица B(n,m), n, m ≤ 20, m > n. Разработайте программу, которая приведет матрицу к квадратному виду B(n,n) путем последовательного вычеркивания из каждой строки матрицы ее m-n неположительных элементов. Если таковых нет или их не хватает для выравнивания матрицы, то вычеркнуть из начала строки столько элементов, сколько необходимо для завершения процесса. Вывести на печать исходную и сформированную матрицы.

Задание 2. Дана символьная матрица SIM(n,m), n,m£20). Разработайте программу, которая вводит значения элементов матрицы с клавиатуры и упорядочивает матрицу, разместив строки в алфавитном порядке в соответствии с символами первого столбца. Для каждой строки матрицы SIM определить, каких букв в ней больше: гласных или согласных. Вывести на печать исходную и отсортированную матрицы, а также информацию о соотношении букв в каждой строке.

5.3. Модули

При разработке больших программ целесообразно часть подпрограмм и других ресурсов, таких, как переменные, константы, описания типов, собирать вместе и компилировать отдельно от основной программы в виде библиотек ресурсов или модулей.

Модуль – это автономно компилируемая коллекция программных ресурсов, предназначенная для использования другими модулями и программами.

Все ресурсы модуля делятся на две группы: *внешние* – предназначенные для использования другими программными единицами, и *внутренние* – рабочие ресурсы данного модуля.

Структура модуля выглядит следующим образом:

```
Unit <имя модуля>;
Interface
     <интерфейсная секция>
Implementation
     <секция реализации>
[Begin
     <секция инициализации>]
End.
```

Имя модуля *должно совпадать с именем файла*, в котором он содержится. Результат компиляции модуля помещается в файл с тем же именем и расширением .tpu.

Примечание. Среда языка Borland Pascal предусматривает три режима компиляции программы, использующей модули:

- Compile – компилируется только основная программа, все модули должны быть предварительно откомпилированы в файлы <имя модуля>.tpu и размещены либо в текущем каталоге, либо в одном из каталогов, указанных как источники файлов .tpu в настройках среды (Options/Directories);
- Make – модули, для которых не обнаружены файлы .tpu, компилируются из соответствующих файлов .pas, которые должны находиться в текущем каталоге или в каталогах, указанных в настройках среды в качестве источников исходных файлов модулей;
- Build – все ранее откомпилированные модули .tpu игнорируются и все модули компилируются из своих исходных файлов заново.

В процессе отладки модулей целесообразно использовать режим Build, а при отладке программы – режим Compile.

Интерфейсная секция содержит объявление ресурсов (в том числе заголовки подпрограмм), к которым возможны обращения извне.

Секция реализации содержит описание подпрограмм, объявленных в интерфейсной секции, и описание внутренних ресурсов модуля (локальных пе-

ременных, типов, подпрограмм). Обращение к этим ресурсам возможно только из подпрограмм, описанных в том же модуле.

Секция инициализации содержит операторы, которые выполняют некоторые действия, необходимые для нормальной работы процедур модуля (например, открывают файлы, инициализируют некоторые переменные и т.п.). Операторы секции инициализации выполняются один раз (при подключении модуля) до начала выполнения основной программы. Эта секция в модуле может отсутствовать.

Программа, которая использует ресурсы нескольких модулей, должна в области описаний содержать спецификацию используемых модулей:

Uses <имя модуля1>, <имя модуля2>, ...;

В спецификации uses необходимо указывать только те модули, ресурсы которых данная программная единица (программа или модуль) использует непосредственно. Если подключаемый модуль использует другие модули, то их подключение уже описано в нем. Секции инициализации подсоединяемых модулей выполняются в порядке их подключения.

Пример 5.4. Разработать модуль, содержащий подпрограмму суммирования элементов массива.

Разбиваем текст программы примера 5.2 на две части: подпрограмму размещаем в модуле, а тестирующую программу оставляем в качестве основной программы. Так как все структурные типы параметров должны быть предварительно объявлены, описываем тип массива в модуле.

```
{Модуль должен размещаться в файле Summa.pas}
Unit Summa;
 Interface          {объявление внешних ресурсов}
   Type mas=array[1..10] of integer;
   Function sum(b:mas;n:integer):integer;
 Implementation
  Function sum;   {описание функции}
    Var     s:integer;i:integer;
    begin s:=0;
        for i:=1 to n do s:=s+b[i];
        sum:=s;
    end;
End.
```

Программа использует из модуля два ресурса: описание типа mas для объявления массива A и функцию Sum.

```
Program ex;
Uses Summa;  {указание используемого модуля}
Var a:mas;   {используем ресурс mas}
    i,n:integer;
```

```
Begin   readln(n);
        for i:=1 to n do read(a[i]);
        ReadLn;
        WriteLn('Сумма=',sum(a,n));   {используем ресурс sum}
End.
```

Все ресурсы, объявленные в интерфейсной части модуля, доступны в основной программе. В случаях перекрытия имен, когда основная программа и подключенный модуль содержат ресурсы с одинаковыми именами, при обращении к ресурсам модуля используют составные имена

<имя модуля>.<имя ресурса>.

Например, в модуле описана переменная X:

```
Unit A;
  Interface
    Var X:real; ...
End.
```

А в основной программе, которая использует этот модуль, объявлена собственная переменная X:

```
Program ex;
  Uses A;
  Var X:integer;
Begin
  X:=10;          {переменная программы}
  A.X:=0.45; ...  {переменная модуля A}
```

В виде модулей в Borland Pascal реализованы библиотеки подпрограмм, использование которых существенно упрощает разработку программ.

Вместе с системой программирования на Borland Pascal поставляются следующие библиотеки.

System – основная библиотека – содержит описания всех стандартных процедур и функций, таких, как математические функции, функции преобразований, процедуры и функции обработки строк и т.п. Ресурсы данной библиотеки доступны любой программе без специального указания.

Crt – библиотека управления экраном в текстовом режиме – содержит описание переменных, констант и процедур и функций, обеспечивающих управление экраном, клавиатурой и динамиком.

Graph – библиотека управления экраном в графическом режиме – содержит описание переменных, констант и процедур и функций, обеспечивающих управление экраном в графическом режиме.

Dos – библиотека организации взаимодействия с операционной системой MS DOS – содержит описание процедур и функций, обеспечивающих обращение к функциям операционной системы.

Поставляемые вместе с описанными модули Turbo3, Printer, Graph3, Overlay устарели и практически не используются.

Примечание. Следует иметь в виду, что модули System, Crt, Dos, Printer объединены в файл Turbo.tpl, а модуль Graph поставляется отдельно в виде файла Graph.tpu.

При разработке собственных библиотек программисты стремятся создавать подпрограммы, имеющие широкую область применения, для чего используют специальные средства объявления параметров: открытые массивы и строки, нетипизированные параметры и параметры процедурного типа.

5.4. Открытые массивы и строки

По правилам Borland Pascal размерность любого массива, передаваемого в качестве параметра, должна быть определена. Следовательно, без использования специальных средств применимость процедур и функций, имеющих параметры-массивы, существенно ограничивается. Чтобы снять ограничение размерности для параметров – одномерных массивов, можно использовать открытые массивы.

Открытый массив – это конструкция описания типа массива без указания типа индексов, например:

```
array of real;
array of integer;
```

Такое определение возможно только при описании формальных параметров подпрограммы. Применяя открытые массивы, следует помнить, что индексы параметров, описанных как открытые массивы, *всегда начинаются с нуля*. Реальное количество элементов фактического параметра массива можно определить двумя способами. Во-первых – передать через дополнительные параметры, во-вторых – использовать специальные функции.

High(<идентификатор массива>***)*** – для обычного массива возвращает верхнюю границу индекса массива, для открытого – максимальное значение индекса.

Low(<идентификатор массива>***)*** – для обычного массива возвращает нижнюю границу индекса массива, для открытого – ноль.

Проиллюстрируем оба способа.

Пример 5.5. Вернемся к примеру 5.4. Пусть по-прежнему требуется разработать модуль, содержащий подпрограмму суммирования элементов массива, но при условии отсутствия *ограничения размерности массива.*

В решении, предложенном в примере 5.4, размерность массива ограничена: функция может работать только с массивами типа mas. Описание параметра как открытого массива позволит разработать функцию, которую можно использовать для любого массива целых чисел. Реальное количество элементов массива в этом случае следует передавать через дополнительный параметр n.

```
Unit Summa2;
Interface
      Function sum(b:array of integer; n:integer):integer;
Implementation function sum;
     Var s:integer;  i:integer;
       begin s:=0;
      {при вычислении суммы учитываем,
              что индексы изменяются от 0 до n-1,
              всего n элементов}
            for i:=0 to n-1 do s:=s+b[i];
            sum:=s;
      end;
End.
```

Теперь основная программа уже может описывать массив любой размерности и использовать из модуля только функцию определения суммы.

```
Program ex;
  Uses Summa2;
  Var a:array[1..10] of integer;
      i,n:integer;
 Begin   ReadLn(n);
         for i:=1 to n do Read(a[i]);
         ReadLn;
         WriteLn('Сумма=',sum(a,n));
 End.
```

Пример 5.6. Разработать программу суммирования элементов матрицы по строкам, считая, что матрица заполнена целиком.

Функцию суммирования элементов строки матрицы поместим в модуль. Поскольку она суммирует все элементы строки, для получения верхнего значения индекса будем использовать функцию High.

```
Unit Summa3;
Interface
   Function sum(b:array of integer):integer;
```

```
Implementation
  Function sum;
    Var s:integer;i:integer;
    begin s:=0;
          for i:=0 to High(b) do s:=s+b[i];
          sum:=s;
    end;
End.
```

Тестирующая программа для работы с индексами также может использовать функции Low и High, которые вернут соответственно -5 и -3.

```
Program ex;
Uses Summa3;
 Const a:array[-5..-3,3..7] of integer=((1,1,1,1,1),
                                        (2,2,2,2,2),
                                        (3,3,3,3,3));
 Var i:integer;
Begin
  for i:=Low(a) to High(a) do WriteLn(sum(a[i]));
 end.
```

Примечание. Обратите внимание, что допустимо передавать в подпрограмму строку матрицы как одномерный массив, указав ее номер – a[i].

В том случае, если необходимо разработать универсальную подпрограмму с параметрами – многомерными массивами, следует применять *нетипизированные* параметры (см. параграф 5.5).

Проблема разработки универсальных подпрограмм существует и для строк.

По правилам Borland Pascal, если строка передается в подпрограмму как параметр-значение, то длина ее не контролируется. Контроль длины осуществляется только для строк, передаваемых в подпрограмму как параметр-переменная. Поэтому при написании универсальных подпрограмм, работающих со строками произвольного размера, необходимо включать режим *открытых строк* {$P+} или объявлять параметры-строки как *openstring*.

Пример 5.7. Разработать программу, которая формирует строку, содержащую буквы латинского алфавита.

Для решения задачи будем использовать процедуру, которая добавляет к строке символ, номер которого на единицу превышает номер последнего символа:

```
Unit Stroka;
Interface
  Procedure Add(var s:openstring);
```

```
Implementation
 Procedure Add;
   begin
     s:=s+chr(succ(Ord(s[length(s)])));
   end;
End.
```

Параметр s должен передаваться как открытая строка, так как в противном случае при попытке использовать процедуру в программе мы получим сообщение о том, что типы формального и фактического параметров не совпадают (Error 26: Type mismatch).

```
Program ex;
Uses Stroka;
 Var S:string[26];    i:integer;
 Begin
   s:='A';
   for i:=2 to 26 do Add(s);
   WriteLn(s);
 End.
```

Увеличить область применения подпрограмм позволяет также использование нетипизированных параметров и параметров процедурного типа.

Задания для самопроверки

Задание 1. Разработайте универсальную подпрограмму, удаляющую из массива элемент, номер которого передается в списке параметров. Поместите подпрограмму в модуль. Разработайте тестирующую программу.

Задание 2. Разработайте универсальную подпрограмму, которая сортирует слова строки по алфавиту. Слова в строке разделены одним пробелом. Поместите подпрограмму в модуль. Разработайте тестирующую программу.

5.5. Нетипизированные параметры

В Borland Pascal допускается использовать параметры, тип которых не указан. Такие параметры могут передаваться в подпрограмму только по ссылке (как параметры-переменные), так как в этом случае в подпрограмму реально передается адрес параметра.

Безусловно, для того чтобы подпрограмма могла выполнять какие-либо действия с этим параметром, она должна как-то назначить ему тип.

Для приведения нетипизированного параметра к определенному типу можно использовать:

- автоопределенное преобразование типов;
- наложенное описание переменной определенного типа.

При автоопределенном преобразовании типов тип выражения указывают явно (см. параграф 2.5), например:

```
Procedure Proc(Var:a); ...
  ...b:= Integer(a)+10; ...
```

Для наложения переменной определенного типа используют описание с absolute (см. параграф 2.3), например:

```
Procedure Proc(Var:a); ...
  Var r:real absolute a;...
```

При этом переменная r оказывается в памяти размещенной в том же месте, что и нетипизированный параметр a, и, соответственно, любое изменение r приведет к изменению a.

Пример 5.8. Разработать подпрограмму, которая может суммировать как элементы массива целых чисел, так и элементы массива вещественных чисел.

Тип массива подпрограмма будет определять по значению третьего параметра, для которого объявим специальный перечисляемый тип. В разделе описаний подпрограммы определим шаблоны для каждого случая. Шаблон представляет собой описание массива соответствующего типа максимально возможного размера 64 Кб/<размер элемента>. Оба шаблона наложены по адресу нетипизированного параметра. Если значение третьего параметра подпрограммы treal, то используется шаблон mr, а если tinteger – шаблон mi.

```
Unit Summa4;
Interface
    Type ttype=(treal,tinteger);   {описание типа третьего параметра}
    Function sum(var x;n:integer;t:ttype):real;
Implementation
 Function sum;
  Var  mr:array[1..maxint*2 div sizeof(real)] of real absolute x;
       mi:array[1..maxint*2 div sizeof(integer)] of integer absolute x;
       s:real;i:integer;
  begin s:=0;
      if t=treal then
         for i:=1 to n do s:=s+mr[i]
      else for i:=1 to n do s:=s+mi[i];
      sum:=s;
  end;
End.
```

Тестирующая программа вызывает одну и ту же функцию для суммирования массивов с разным типом элементов.

```
Program ex;
Uses Summa4;
 Var a:array[1..10] of integer;
     b:array[1..15] of real;
     i,n:integer;
 Begin
      for i:=1 to 10 do Read(a[i]); ReadLn;
      WriteLn('Сумма=',sum(a,10,tinteger):8:1);
      for i:=1 to 15 do Read(b[i]); ReadLn;
      WriteLn('Сумма=',sum(b,15,treal):8:1);
end.
```

Примечание. Вместо описания массива максимально возможного размера в подпрограмме можно описать массив длиной 1 элемент, но при работе с таким шаблоном необходимо отключать контроль индексов {$R-}.

Нетипизированные параметры можно применить для написания универсальных подпрограмм, использующих в качестве параметров многомерные массивы. Поскольку невозможно в подпрограмме определить универсальный шаблон многомерного массива, приходится обрабатывать многомерный массив как одномерный, учитывая то, что *элементы многомерных массивов в памяти располагаются так, что чем правее индекс, тем быстрее он возрастает*. Так, элементы трехмерного массива B(2,3,2) будут расположены в памяти в следующем порядке:

$b_{1,1,1}$, $b_{1,1,2}$, $b_{1,2,1}$, $b_{1,2,2}$, $b_{1,3,1}$, $b_{1,3,2}$, $b_{2,1,1}$, $b_{2,1,2}$, $b_{2,2,1}$, $b_{2,2,2}$, $b_{2,3,1}$, $b_{2,3,2}$.

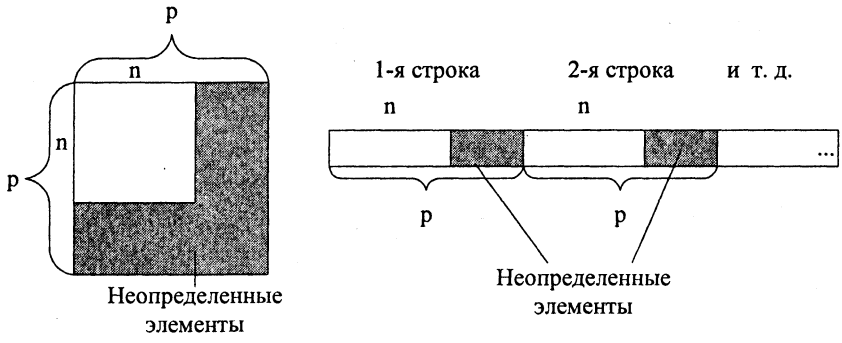
Пример 5.9. Разработать универсальную подпрограмму транспонирования матрицы.

На рис. 5.9 показано, как в матрице A(P,P) расположены элементы реальной матрицы размером n×n, где n<P, и, соответственно, как те же элементы расположены в памяти.

Таким образом, в одномерном массиве элемент с индексами i и j имеет индекс (i-1)*P+j, т.е. индекс этого элемента зависит от размера строки зарезервированной матрицы. В подпрограмму это значение передается через параметр P.

Чтобы транспонировать матрицу, меняем местами элементы, расположенные ниже и выше главной диагонали.

Примечание. При разработке алгоритма процедуры учтено, что при транспонировании матрицы по аналогии с переворотом строки количество перестановок равно n*n div 2. Если выполнить n*n перестановок, то матрица примет исходный вид, так как будет транспонирована дважды.



а *б*

Рис. 5.9. Матрица A(P,P) (*а*) и ее внутреннее представление (*б*)

```
Unit Matrica;
  Interface
     Procedure tran(Var x;n,P:integer);
  Implementation
    Procedure tran;
    Var a:array[1..2*maxint div sizeof(real)] of real absolute x;
       i,j:integer;
       t:real;
    begin
       for i:=2 to n do
         for j:=1 to i-1 do
           begin t:=a[(i-1)*P+j];
                 a[(i-1)*P+j]:=a[(j-1)*P+i];
                 a[(j-1)*P+i]:=t;
           end;
   end;
End.
```

Тестирующая программа выглядит следующим образом:

```
Program ex;
Uses Matrica;
Var   a:array[1..10,1..10] of real;
      i,j:integer;
Begin
     WriteLn('Введите матрицу a(5*5):');
     for i:=1 to 5 do
       begin
          for j:=1 to 5 do Read(a[i,j]);        ReadLn;
       end;
```

```
    tran(a,5,10);
    WriteLn('Результат:');
    for i:=1 to 5 do
      begin
        for j:=1 to 5 do Write(a[i,j]:6:2);
        WriteLn;
      end;
End.
```

Задания для самопроверки

Задание 1. Разработайте универсальную подпрограмму, определяющую среднее арифметическое элементов матрицы размером n×m элементов. Поместите подпрограмму в модуль. Разработайте тестирующую программу.

Задание 2. Разработайте универсальную подпрограмму, удаляющую из матрицы размером n×m заданную строку и столбец. Поместите подпрограмму в модуль. Разработайте тестирующую программу.

5.6. Параметры процедурного типа

Параметры процедурного типа используют тогда, когда нужно передать в подпрограмму имена процедур и функций.

Для объявления процедурного типа используют заголовок подпрограммы, в котором отсутствует имя, например:

```
Type  proc=procedure (a,b,c:real;Var d:real);
      func=function(x:real):real;
```

Значениями переменных процедурных типов являются идентификаторы процедур и функций с соответствующими заголовками:

```
Var f:func;
...
  f:=fun1;...
```

Процедуры или функции, идентификаторы которых будут передаваться в качестве параметров процедурного типа, по правилам языка необходимо компилировать в режиме *дальнего вызова* (с указанием директивы компилятора {$F+}или служебного слова far). В этом режиме при вызове подпрограмм используются длинные 4-байтовые адреса (см. параграф 7.1) в отличие от коротких 2-байтовых адресов, которые применяются для адресации подпрограмм, объявленных в основной программе или ее подпрограммах.

Примечания: 1. Если процедуры или функции описываются в интерфейсной части модуля (см. параграф 5.2), то по умолчанию они вызываются в режиме дальнего вызова, поэтому специально это оговаривать не надо.

2. Стандартные процедуры и функции, описанные в модуле System, по умолчанию являются функциями ближнего вызова. Если имена этих подпрограмм необходимо передавать через параметры, то надо описать собственную подпрограмму с указанием режима дальнего вызова, в теле которой будет находиться вызов стандартной подпрограммы, и использовать ее вместо стандартной подпрограммы.

Пример 5.10. Разработать подпрограмму, которая возвращает массив значений произвольной функции при заданных интервале изменения аргумента [a,b] и количестве точек n.

Имя функции будем передавать в подпрограмму через параметр процедурного типа. Массив значений будем описывать как открытый:

```
Unit SFun;
 Interface
    Type func=function(x:real):real;
    Procedure TabFun(f:func;a,b:real;n:integer;
                                        Var Masf:array of real);
 Implementation
   Procedure TabFun;
     Var h:real; i:integer;
     Begin
         h:=(b-a)/(n-1);
         for i:=0 to n-1 do Masf[i]:=f(a+h*i);
     End;
 End.
```

Основная программа должна описать конкретную функцию как функцию дальнего вызова. Для функции sin создается специальная функция дальнего вызова.

```
Program ex;
Uses SFun;
Var masF1:array[1..10] of real;  masF2:array[1..20] of real;
      i:integer;
function F1(x:real):real; far;
      Begin F1:=sin(x); end;
function F2(x:real):real; far;
      Begin F2:=exp(x)+cos(x); end;
 Begin
   WriteLn('Таблица значений функции sin x:');
   TabFun(F1,0,2,10,masF1);
```

```
  for i:=1 to 10 do Write(masF1[i]:7:1);
  Writeln;
  WriteLn('Таблица значений функции exp x+cos x:');
  TabFun(F2,0,2,20,masF2);
  for i:=1 to 20 do Write(masF2[i]:7:1);
  Writeln;
End.
```

Задания для самопроверки

Задание 1. Разработайте процедуру вычисления производных функции y(x) по формулам Лагранжа в трех соседних точках, отстоящих на величину шага h:

$$y_0' = (-3y_0 + 4y_1 - y_2)/2h; \quad y_1' = (-y_0 + y_2)/2h; \quad y_2' = (y_0 - 4y_1 + 3y_2)/2h.$$

Поместите процедуру в модуль. Разработайте тестирующую программу.

Задание 2. Разработайте процедуру определения корня функции на заданном отрезке. Поместите процедуру в модуль. Разработайте тестирующую программу.

5.7. Рекурсия

В математике строгое определение некоторых понятий может опираться на то же понятие. Такие определения называют *рекурсивными* или *индуктивными*. Например, рекурсивно определяется факториал:

$$N! = \begin{cases} 1, & \text{если } N=0; \\ N\times(N-1)!, & \text{если } N>0. \end{cases}$$

Данное определение состоит из двух утверждений. Первое утверждение носит название *базисного*. Базисное утверждение нерекурсивно. Второе утверждение носит название *рекурсивного* или *индуктивного*. Оно строится так, чтобы полученная в результате повторных применений цепочка определений сходилась к базисному утверждению.

В программировании *рекурсивной* называется подпрограмма, которая в процессе выполнения вызывает сама себя.

Различают два вида рекурсии подпрограмм:

- *прямая* или *явная* рекурсия – характеризуется существованием в теле подпрограммы оператора обращения к самой себе;
- *косвенная* или *неявная* рекурсия – образуется при наличии цепочки вызовов других подпрограмм, которые в конечном итоге приведут к вызову исходной.

Каждое обращение к рекурсивной подпрограмме вызывает независимую *активацию* этой подпрограммы. Совокупность данных, необходимых для *одной* активации рекурсивной подпрограммы, называется *фреймом активации.*

Фрейм активации включает:

- копии всех локальных переменных подпрограммы;
- копии параметров-значений;
- четырехбайтовые адреса параметров-переменных и параметров-констант;
- копию строки результата (для функций типа string);
- служебную информацию – около 12 байт, точный размер этой области зависит от способа вызова (ближний или дальний) и внутренней организации подпрограмм.

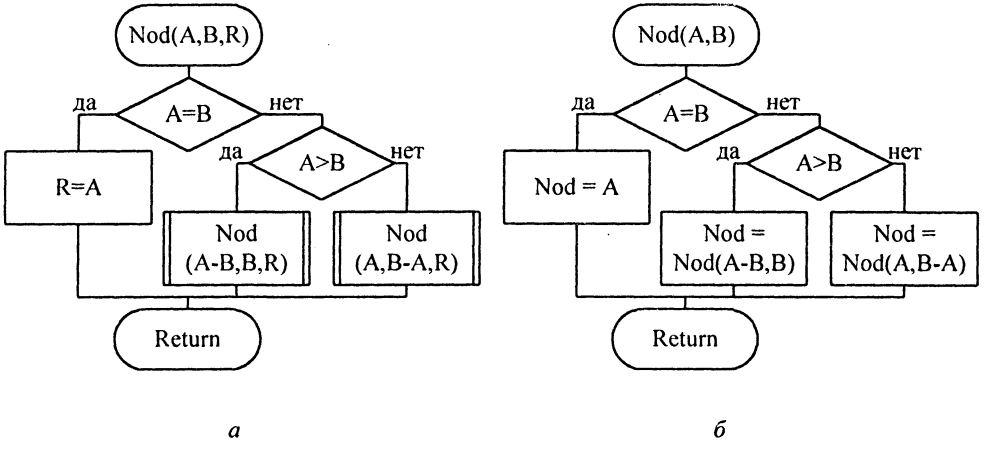
Пример 5.11. Разработать программу определения наибольшего общего делителя двух чисел, используя алгоритм Евклида.

Нерекурсивный вариант этого алгоритма уже был рассмотрен в примере 1.2. Рекурсивный вариант будем строить исходя из двух утверждений.

Базисное утверждение: если два числа равны, то их наибольший общий делитель равен этим числам.

Рекурсивное утверждение: наибольший общий делитель двух чисел равен наибольшему общему делителю их разности и меньшего из чисел.

Рекурсивная подпрограмма может быть реализована и как процедура, и как функция. При реализации в виде процедуры список параметров кроме чисел A и B включает параметр-переменную R. Через этот параметр процедура возвращает результат (рис. 5.10, *а*). Текст программы, реализующий вариант с рекурсивной процедурой, представлен ниже.



а *б*

Рис. 5.10. Схема алгоритма Евклида:

а – с использованием рекурсивной процедуры; *б* – с использованием рекурсивной функции

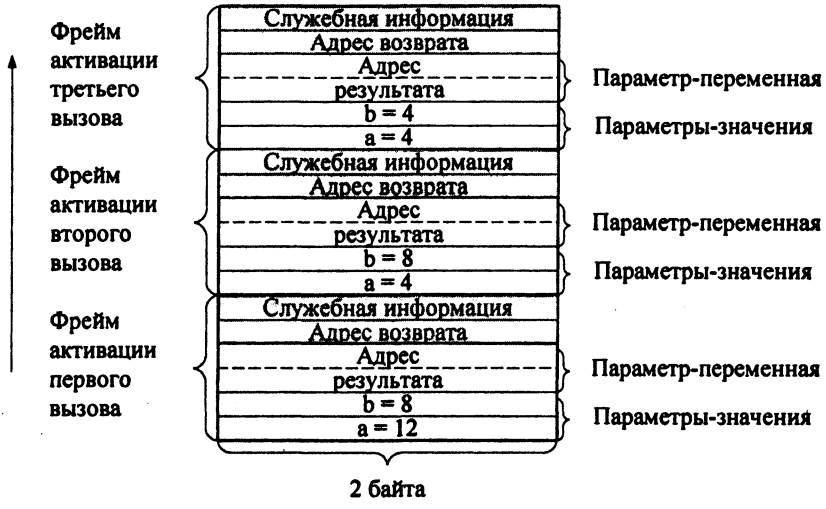


Рис. 5.11. Заполнение стека при выполнении рекурсивной подпрограммы

```
Program ex;
Var a,b,r:integer;
Procedure nod(a,b:integer; var r:integer);
  Begin if a=b then r:=a    {базис}
                else if a>b then nod(a-b,b,r)
                            else nod(a,b-a,r)
   End;
Begin ReadLn(a,b);
      nod(a,b,r);
      WriteLn(r);
End.
```

При выполнении программы фреймы активации будут размещаться в *стеке* – специальным образом организованной памяти. Например, для a=12 и b=8 в момент завершения нерекурсивной третьей активации стек будет выглядеть, как показано на рис. 5.11 (запись в стек идет словами, т. е. по 2 байта, как и изображено на рисунке).

Если реализовать вариант с рекурсивной функцией (рис. 5.10, *б*), то фрейм активации уменьшится, так как сократится список параметров. Соответствующая программа будет выглядеть следующим образом:

```
Program ex;
   Var  a,b,r:integer;
   Function nod(a,b:integer):integer;
       begin if a=b then nod:=a   {базис}
```

```
        else      {рекурсивный вызов}
          if a>b then
                nod:=nod(a-b,b)
          else  nod:=nod(a,b-a)
    end;
Begin
    ReadLn(a,b);
    r:=nod(a,b);
    WriteLn(r);
End.
```

Итак, если подпрограмма обращается к себе несколько раз, образуется несколько одновременно существующих активаций и, соответственно, несколько фреймов активации. Все фреймы размещаются в *стеке*, и при большом количестве вызовов возможно переполнение стека. Поэтому необходимо стремиться к уменьшению фрейма активации.

Примечание. Размер стека устанавливается в настройках среды, по умолчанию он принимает размер 16 кб, и его размер не может превышать 64 кб.

Пример 5.12. Разработать рекурсивную подпрограмму «переворота» строки (первая буква должна стать последней, вторая – предпоследней и т.д.).

Можно предложить два способа разворота строки.

Первый способ заключается в последовательном отсечении начального элемента и добавлении его в конец результирующей строки.

Второй – базируется на последовательной перестановке элементов: первого с последним, второго – с предпоследним и т. д. Перестановки должны прекратиться, когда дойдем до середины строки, иначе вновь поставим элементы на исходные места.

Вариант с отсечением и дописыванием:

```
Function reverse1(const st:string):string;
 Begin if length(st)=0 then reverser1:=''
       else
         reverser1:= reverse1(copy(st,2,length(st)-1))+st[1];
 End;
```

Определим размер фрейма активации:

V = 4 (адрес параметра) + 256 (результат-строка) +
+ <размер служебной области > ≈270 (байт).

Вариант с использованием перестановок:

```
Procedure reverse2(var ss:string; n:integer);
Var temp:char;
  Begin if n<=length(ss) div 2 then
          begin temp:=ss[n];
                ss[n]:=ss[length(ss)-n+1];
                ss[length(ss)-n+1]:=temp;
               reverse2(ss,n+1);
          end;
  End;
```

Размер фрейма активации для этого варианта:

V=4(адрес 1-го параметра)+2(копия 2-го параметра) +
+ 1(локальная переменная)+<размер служебной области>≈17 (байт).

Следовательно, второй вариант предпочтительнее.

Одной из наиболее часто встречающихся ошибок при создании рекурсивных процедур является «зациклившаяся», или бесконечная, рекурсия, при которой базисное утверждение не достигается, и, соответственно, вновь и вновь вызывается рекурсивная подпрограмма. От бесконечного цикла бесконечная рекурсия отличается тем, что каждая активация требует дополнительной памяти для размещения фрейма активации, и, следовательно, программа, содержащая бесконечную рекурсию, обычно завершается аварийно при переполнении стека.

Причиной бесконечной рекурсии может быть не только ошибка в программе, но и обработка некорректных данных.

Пример 5.13. Разработать программу определения корней уравнения $y=x^2-2$ на заданном отрезке методом половинного деления.

Метод половинного деления для непрерывных функций, так же как и метод хорд, рассмотренный в примере 3.7, позволяет находить корень функции только в случае, если функция имеет на концах заданного отрезка разные знаки, т.е. пересекает ось абсцисс. Этот метод базируется на следующих утверждениях.

Базисное утверждение: если абсолютная величина функции в середине отрезка не превышает заданного значения погрешности, то координата середины отрезка и есть корень.

Рекурсивное утверждение: корень расположен между серединой отрезка и тем концом, значение функции в котором по знаку не совпадает со значением функции в середине отрезка (рис. 5.12).

```
Program ex;
Var a,b,eps,x:real;
```

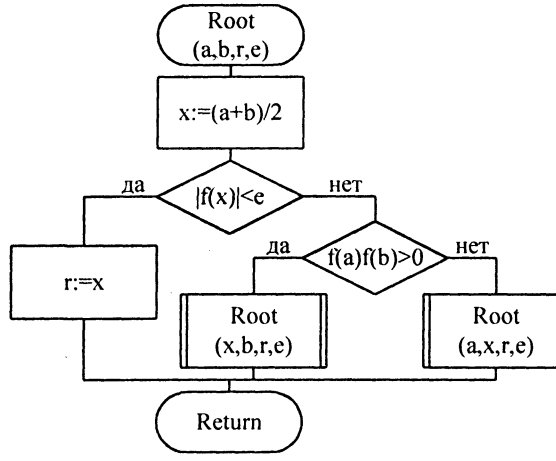


Рис. 5.12. Рекурсивный алгоритм определения корней уравнения

```
Procedure root(a,b,eps:real;var r:real);
  Var f,x:real;
  Begin  x:=(a+b)/2;
         f:=x*x-1;
         if abs(f)<=eps then r:=x
         else
              if (a*a-1)*f>0 then root(x,b,eps,r)
                             else root(a,x,eps,r)
  End;
Begin    ReadLn(a,b,eps);
         root(a,b,eps,x);
         WriteLn('Корень x=',x:9:7);
End.
```

Если для данной программы задать отрезок, не содержащий корня, то произойдет «зацикливание», что вызовет аварийное завершение программы по переполнению стека. Для исключения подобных ситуаций необходимо в основной программе проверить приведенное выше условие наличия корней на отрезке.

При объявлении косвенно рекурсивных подпрограмм возникает проблема: их описания принципиально не удается расположить так, чтобы избежать обращения к еще не описанным ресурсам, как того требуют правила объявления ресурсов Borland Pascal. В этом случае используют специальную директиву forward, которая позволяет выполнять *предописание*: сначала запи-

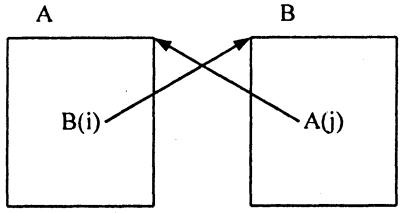


Рис. 5.13. Пример взаимно рекурсивных подпрограмм

сывают заголовки подпрограмм с параметрами, за которыми вместо тела подпрограммы следует forward; затем описывают тела этих подпрограмм (причем параметры второй раз можно уже не описывать).

Например, описание процедур A и B, которые рекурсивно вызывают друг друга (рис. 5.13), можно выполнить следующим образом:

```
procedure B(j:byte); forward;
procedure A(j:byte);
      begin ...  B(i); ...  end;
procedure B;
      begin ...  A(j); ...  end;
```

Пример 5.14. Разработать программу проверки чередования букв и цифр в заданной строке (первый символ – обязательно буква).

Базисные утверждения:

- строка допустима, если она пустая;
- строка не допустима, если в ней обнаружен недопустимый символ.

Рекурсивное утверждение: если текущий символ строки допустим, то допустимость строки определяется оставшейся частью строки.

Для написания данной программы используем взаимно рекурсивные подпрограммы. Первая будет проверять, является ли текущий символ буквой, и если является, будет удалять этот символ и вызывать вторую. Вторая – будет проверять, является ли текущий символ цифрой, и если является, будет удалять эту цифру и вызывать первую. Выход из рекурсии – по достижении конца строки. Параметры: строка передается по ссылке, а номер символа – по значению.

```
Program ex;
Var s:string;
Function f_char(var st:string;i:word):boolean; forward;
Function f_value(var st:string;i:word):boolean;
  Begin
    if i>length(st) then f_value:=true
    else if (st[i] in ['0'..'9'])then f_value:=f_char(st,i+1)
        else f_value:=false;
  End;
```

```
Function f_char;
  Begin
    if i>length(st) then  f_char:=true
    else
        if (st[i] in ['A'.. 'Z', 'a'.. 'z']) then
              f_char:=f_value(st,i+1)
        else     f_char:=false;
  End;
Begin
      WriteLn('Введите строку:');
      Readln(s);
      if f_char(s,1) then
          WriteLn('Строка корректна')
      else WriteLn('Строка не корректна');
End.
```

Во всех рассмотренных выше случаях рекурсивные подпрограммы имели структуру, представленную на рис. 5.14, которая характеризуется тем, что каждая рекурсивная подпрограмма вызывает саму себя один раз. Такая организация рекурсии названа *линейной*.

Операторы, которые на схеме помечены как «операторы после вызова», выполняются после возврата управления из рекурсивно вызванной подпрограммы. Если попытаться изобразить последовательность действий при линейной рекурсии, то она будет выглядеть, как это показано на рис. 5.15.

Таким образом, сначала в каждой активации выполняются операторы, расположенные до рекурсивного вызова, затем (при достижении условия выхода) – нерекурсивная часть очередной активации, а затем – операторы, записанные после рекурсивного вызова. На этом построены многие рекурсивные алгоритмы.

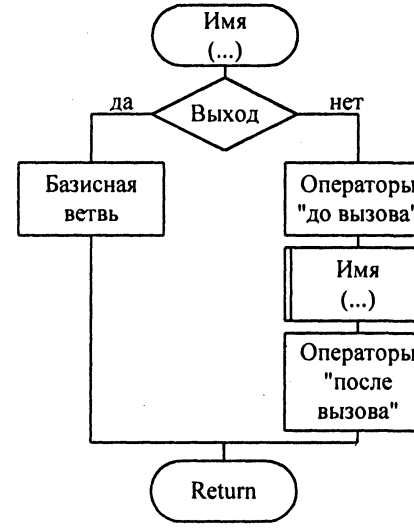


Рис. 5.14. Структура подпрограммы с линейной рекурсией

Пример 5.15. Разработать программу, которая выводит из заданного массива, завершающегося нулем, сначала положительные значения, а затем – отрицательные в любом порядке. Между положительными и отрицательными элементами массива вывести три звездочки.

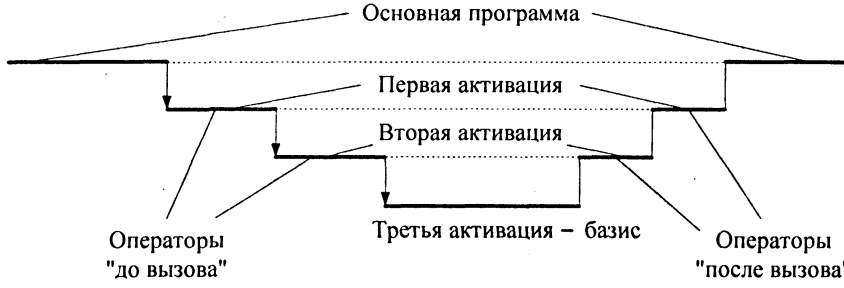


Рис. 5.15. Рекурсивное погружение и рекурсивный выход при линейной рекурсии

Нерекурсивная программа для решения данной задачи должна содержать, по крайней мере, два цикла, не считая циклов ввода и вывода. Рекурсивный вариант может использовать рекурсивный возврат для вывода оставшихся элементов.

Построим решение следующим образом: в области до вызова запрограммируем печать положительных чисел, в области после вызова – печать отрицательных чисел, а нерекурсивная часть будет содержать вывод звездочек.

```
Program ex;
Type mas=array[1..10] of real;
Var x:mas;
    i:integer;
Procedure print(var x:mas;i:integer);
 Begin
       if x[i]=0 then WriteLn('***')
       else
          begin
            if x[i]>0 then WriteLn(i,' ', x[i]);
            print(x,i+1);      {рекурсивный вызов}
            if x[i]<0 then WriteLn(i,' ', x[i]);
          end
 End;
Begin
      i:=0;
      repeat i:=i+1; Read(x[i]) until x[i]=0;
      ReadLn;
      print(x,1);
End.
```

Примечание. Обратите внимание, что значение i в каждой активации свое, и оно не меняется во время выполнения подпрограммы: одно и то же как до рекурсивного вызова, так и после него.

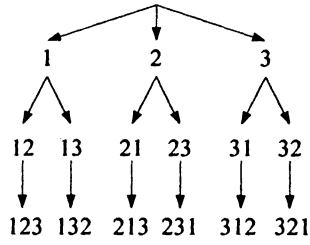


Рис. 5.16. Дерево формирования перестановок при m=3

Кроме линейной достаточно часто встречается рекурсия, получившая название *древовидной*. При древовидной рекурсии подпрограмма в течение одной активации вызывает саму себя более одного раза. Полученная в данном случае последовательность вызовов имеет форму дерева, с чем и связано название данного вида организации процесса вычислений.

Пример 5.16. Разработать программу, которая формирует все перестановки без повторений некоторого массива значений. Например, если задан массив, содержащий символы ABC, то необходимо сформировать следующие комбинации: ABC, ACB, BAC, BCA, CAB, CBA.

Будем исходить из того, что в комбинации на первое место можно поставить любой из имеющихся m символов. На второе место в каждом варианте можно поставить любое из оставшихся m-1 значений. На третье место – любое из m-2 значений и т.д. На последнее m-е место – единственное оставшееся значение. На следующем m+1-м шаге перестановку можно выводить. На рис. 5.16 представлено дерево формирования вариантов перестановок для m=3.

Таким образом, каждая активация подпрограммы уровня n должна вызывать саму себя n-m+1 раз, каждый раз передавая следующей активации массив еще не расставленных значений r1. Окончательный алгоритм подпрограммы формирования перестановок представлен на рис. 5.17.

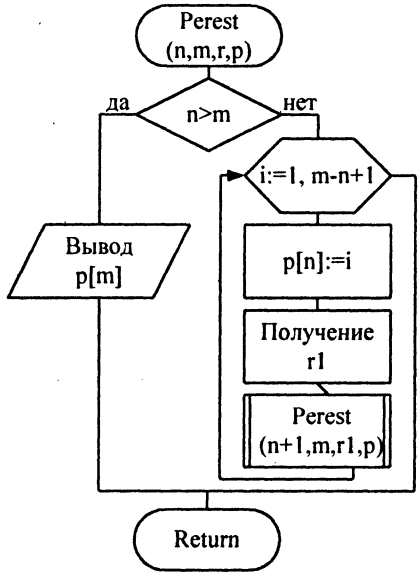


Рис. 5.17. Схема алгоритма подпрограммы формирования перестановок

```
Program ex;
Type
   mas=array[1..5] of char;
Const a:mas='ABCDE';
Var  pole:mas;
```

```
procedure Perest(n,m:integer; Const r:mas; Var pole:mas);
  Var   r1:mas;
        k,j,i:integer;
  Begin
     if n>m then
       begin
         for i:=1 to m do Write(pole[i]);
         Write(' ');
       end
     else
      for i:=1 to m-n+1 do
        begin
          pole[n]:=r[i];
          k:=1;
          for j:=1 to m-n+1 do
            if j<>i then
                    begin
                       r1[k]:=r[j];
                       k:=k+1;
                    end;
           Perest(n+1,m,r1,pole);
        end;
  End;
Begin
   Perest(1,5,a,pole);
End.
```

Определим размер фрейма активации.
Параметры: $V_1 = 2*2 + 4 + 4$;
локальные переменные: $V_2 = m + 3*2$;
служебная информация: $V_3 \approx 12$.
Откуда

$$V = V_1 + V_2 + V_3 = 12 + m + 6 + 12 \approx 30 + m.$$

Максимальное количество одновременно существующих активаций равно m+1. Соответственно максимальный объем используемой памяти стека

$$V_{max} = (30 + m)(m + 1) = m^2 + 31m + 30.$$

Так, для m=5 $V_{max} = 210$ (байт).

Примечание. Интересно, что при развертывании древовидной рекурсии в каждый момент времени в стеке хранятся фреймы активации *одной ветви дерева*, и, следовательно, существенного увеличения объема хранимой информации не происходит.

Задания для самопроверки

Задание 1. Разработайте рекурсивную подпрограмму, формирующую последовательность строк:

A
BB
CCC
DDDD
EEEEE
FFFFFF и т. д. Всего 26 строк. Разработайте тестирующую программу.

Задание 2. Разработайте рекурсивную подпрограмму вычисления биномиальных коэффициентов:

$$C\{m,n\} = \begin{cases} 0, & \text{если } m>n\geq 0; \\ 1, & \text{если } (m=0 \text{ и } n>0) \text{ или } (m=n=0); \\ C\{m-1,n-1\} + C\{m,n-1\} & \text{в остальных случаях.} \end{cases}$$

Задание 3. Разработайте рекурсивную подпрограмму быстрой сортировки элементов массива (сортировка Хоора [3, 6]). Быстрая сортировка выполняется следующим образом. Выбирают любой, например, первый элемент массива, и затем элементы переставляют так, чтобы слева располагались элементы меньше выбранного, а справа – больше. Для этого массив просматривают с двух сторон и меняют местами элементы, стоящие не в своей части массива. Тем самым выбранный элемент оказывается на своем месте. После этого описанный алгоритм применяют к левой и правой частям подмассива и т. д., пока очередной подмассив не окажется состоящим из одного элемента. Корректная реализация данного метода обеспечивает вычислительную сложность $O_{cp}(n \log_2 n)$.

Задание 4. Разработайте рекурсивную подпрограмму «Ханойская башня». Имеется три колышка. На первом нанизаны m пронумерованных колец. Необходимо расположить кольца в том же порядке на любом из оставшихся колышков. При этом кольца можно переносить только по одному, причем не разрешается класть кольцо с большим номером на кольцо с меньшим номером.

5.8. Практикум. Полный и ограниченный перебор. Реализация ограниченного перебора с использованием рекурсии

Существует класс задач, в которых из некоторого количества вариантов необходимо выбрать наиболее подходящий (оптимальный). Для таких задач далеко не всегда удается найти алгоритм, который позволил бы получить решение без анализа всех или большого количества комбинаций исходных данных, т.е. без осуществления *перебора*. Осуществление полного перебора требует много времени. Вычислительная сложность решения задач с использованием перебора обычно оценивается как O(n!) или даже $O(n^n)$.

Для решения задач данного типа применяют две стратегии:

- формируют сразу всю комбинацию исходных данных и выполняют ее анализ в целом;
- генерацию и анализ комбинации осуществляют по частям.

Если имеется возможность, анализируя часть комбинации исходных данных, оценить ее перспективность с точки зрения получения решения, то вторая стратегия позволит получить результат за меньшее время за счет исключения из рассмотрения всех комбинаций, в которые входит данная часть. Исключение бесперспективных комбинаций (*отсечение* вариантов) позволяет осуществить не полный, а *ограниченный* перебор.

Если по части комбинации исходных данных нельзя сделать заключение о ее перспективности, то предпочтительнее первая стратегия, так как обычно она требует меньше времени на анализ варианта.

Общий алгоритм решения задачи с использованием полного перебора по первой стратегии может быть представлен следующим образом.

Цикл-пока еще есть варианты
  Генерировать комбинацию исходных данных
  если вариант удовлетворяет условию,
    то Обработать вариант
  все-если
Все-цикл

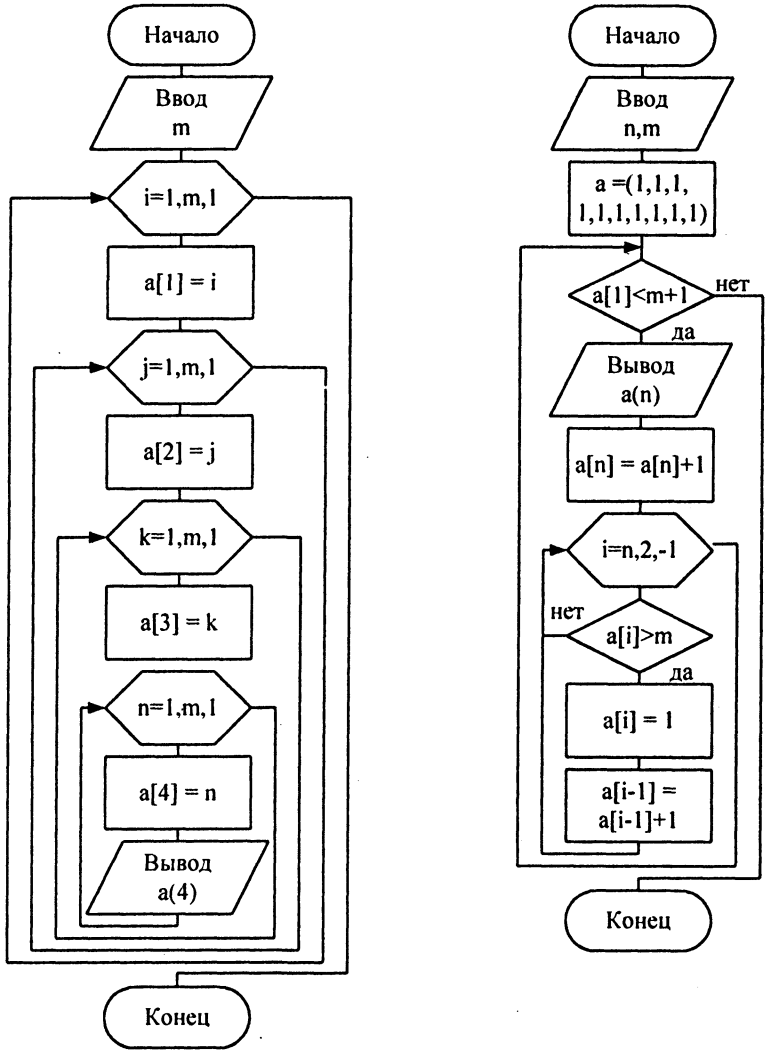
Для генерации всех комбинаций исходных данных можно использовать вложенные циклы или программно реализуемый счетчик (по типу, например, электросчетчика). Второй способ позволяет получить более общее решение.

Пример 5.17. Разработать программу генерации следующих комбинаций: 1111, 1112, 1113, 1121, 1122, 1123, 1131, 1132, 1133, 1211, ..., 3333.

Вариант 1. Для генерации комбинаций используем вложенные циклы. Количество разрядов – 4. Следовательно, для генерации всех комбинаций понадобится 4 цикла, вложенных один в другой (рис. 5.18, *а*). Если ввести параметр m – максимальное значение в каждом разряде, то эта же программа будет генерировать комбинации от 1111 до mmmm.

Если количество разрядов генерируемой комбинации изменить, то придется переписывать программу, увеличивая или уменьшая количество вложенных циклов.

Вариант 2. С точки зрения увеличения универсальности программы для генерации всех комбинаций лучше использовать программно реализуемый счетчик в общем случае на n разрядов, который в каждом разряде считает от 1 до m. Каждое состояние такого счетчика соответствует комбинации. Для генерации следующего варианта в младший разряд счетчика добавляют 1 и осуществляют межразрядные переносы в соответствии с характеристиками каждого разряда (рис. 5.18, *б*). Ниже представлена программа, реализующая данный алгоритм для $n \le 10$.



а *б*

Рис. 5.18. Два варианта алгоритма генерации комбинаций:
а – с использованием вложенных циклов; *б* – программно-реализуемый счетчик

```
Program ex;
  Const
      a:array[1..10] of byte=(1,1,1,1,1,1,1,1,1,1);
  Var i,m,n:integer;
```

```
Begin
 ReadLn(n,m);
 while a[1]<m+1 do    {условие «сброса» счетчика}
  begin
    for i:=1 to n do Write(a[i]); {вывод комбинации}
    Write(' ');
    a[n]:=a[n]+1;{добавление 1 в последний разряд}
    for i:=n downto 2 do{анализ и осуществление переносов}
       if a[i]>m then
            begin
              a[i]:=1;
              a[i-1]:=a[i-1]+1;
            end;
  end
End.
```

Ассоциируя комбинации счетчика с вариантами данных и проверяя полученные комбинации по критериям конкретной задачи, можно осуществить полный перебор вариантов.

Как уже упоминалось выше, для реализации ограниченного перебора используется вторая стратегия, при которой генерацию и анализ комбинаций исходных данных выполняют поэтапно. С этой целью удобно использовать рекурсию, при которой процесс генерации комбинации продолжается, пока полученная часть комбинации перспективна, или не исчерпаны все варианты.

Пример 5.18. Задача о расстановке ферзей. Разработать программу, которая формирует все возможные варианты расстановки m (m>3) ферзей на шахматной доске m×m клеток, при которых ферзи не «бьют» друг друга.

Начнем с выбора способа представления данных задачи. С первого взгляда кажется, что доска должна представляться матрицей, а местоположение ферзей – размещением символов «*» в матрице. Однако заметим, что гораздо удобнее вариант, при котором доска представлена вектором. Индекс вектора в таком представлении соответствует номеру столбца доски, а значение – номеру строки, в которой расположен ферзь (рис. 5.19).

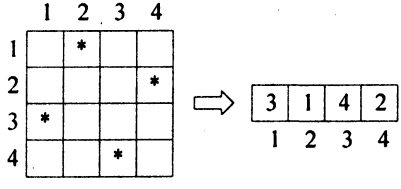


Рис. 5.19. Представление данных для задачи о ферзях

Определим, что для векторного представления означает «бьет». Ферзь j «бьет» ферзь i, если они расположены на одной диагонали, вертикали или горизонтали. При векторном представлении шахматной доски расположение на одной вертикали невозможно, следовательно, необходимо проверять два условия:

pole[j] = pole[i] – одна горизонталь;
| pole[j] - pole[i] | =| j - i | – одна диагональ.

Для определения всех вариантов расстановок ферзей, удовлетворяющих условию задачи, необходимо проверить все возможные варианты расстановок.

Полный перебор. Полный перебор будем реализовывать в соответствии с алгоритмом, описанным в начале данного раздела. Ниже приведена программа, реализующая стратегию полного перебора для данной задачи. Она включает две функции: функцию генерации вариантов расстановки и функцию проверки комбинации.

Функция генерации вариантов возвращает в точку вызова булевское значение: true – если еще есть варианты, false – если просмотрены все варианты. Следующий вариант расстановки возвращается через параметр-переменную pole.

Функция проверки комбинаций для каждых двух ферзей на доске проверяет условие «бьет». Если хотя бы один раз это условие выполняется, то данный вариант не является решением.

```
Program ex;
Type p=array[1..100] of integer;
Var   pole:p;
      i,m:integer;
{функция проверки комбинации}
Function new_r(m:integer; pole:p):boolean;
Var i,j:integer;
Begin
    new_r:=false;
   for i:=1 to m-1 do
      for j:=i+1 to m do
         if (pole[i]=pole[j]) or(abs(pole[j]-pole[i])=j-i) then exit;
   new_r:=true;
End;
{функция генерации вариантов}
Function Variant(m:integer; Var pole:p):boolean;
Var i:integer;
Begin
   pole[m]:=pole[m]+1;       {добавление единицы в младший разряд
                                  счетчика}
   for i:=m downto 2 do      {обработка переносов}
     if pole[i] > m then
       begin  pole[i]:=1;
              pole[i-1]:=pole[i-1]+1;
       end;
```

```
    if pole[1] <=m then {если есть еще варианты}
        Variant:=true
    else Variant:=false;
End;
{основная программа}
Begin
      Writeln('Введите размер доски');
      ReadLn(m);
      for i:=1 to m do pole[i]:=1;    {исходная комбинация}
      repeat
        if New_r(m,pole) then          {проверяем расстановку}
          begin
            for i:=1 to m do Write(pole[i]:2);
            Writeln;
          end;
      until not Variant(m,pole);    {если есть варианты, то генерируем новый вариант}
End.
```

Недостаток полного перебора, как уже говорилось выше, заключается в том, что вычислительная сложность данного решения составляет $O(m^m)$.

Ограниченный перебор, реализованный с использованием рекурсии. Для генерации комбинаций будем использовать древовидную рекурсию, на каждом уровне которой ставится один ферзь m различными способами. При этом появляется возможность проверки перспективности уже имеющейся комбинации. Поясним сказанное на примере доски 4×4.

На первом шаге первого ферзя можно поставить на одно из четырех полей первого вертикального ряда, что соответствует комбинациям

1..., 2..., 3..., 4... .

На втором шаге второго ферзя можно поставить на одно из четырех полей второго вертикального ряда. При этом мы получим 4 комбинации для каждого варианта установки первого ферзя:

11.., 12.., 13.., 14..
21.., 22.., 23.., 24..
31.., 32.., 33.., 34..
41.., 42.., 43.., 44..

На этом этапе уже можно исключить все варианты, для которых первые два ферзя «бьют» друг друга: 11.., 12.., 21.., 22.., 23.., 32.., 33.., 34.., 43.., 44.. Генерацию остальных комбинаций необходимо продолжить, установив третьего ферзя одним из четырех способов и исключив неперспективные решения. На последнем этапе необходимо установить четвертого ферзя и опять

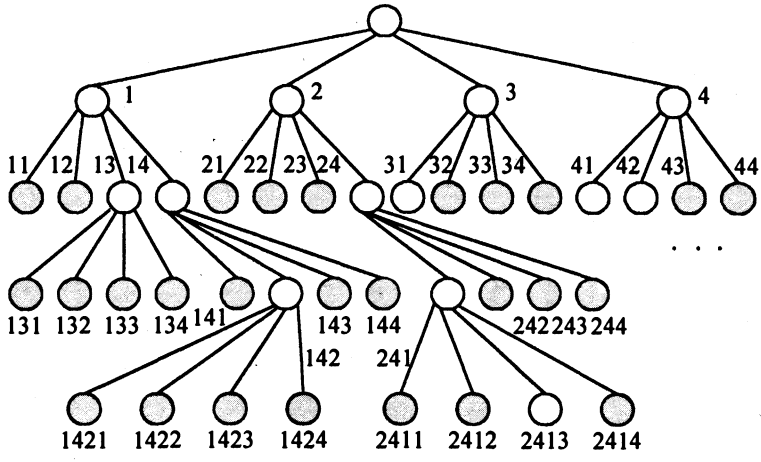


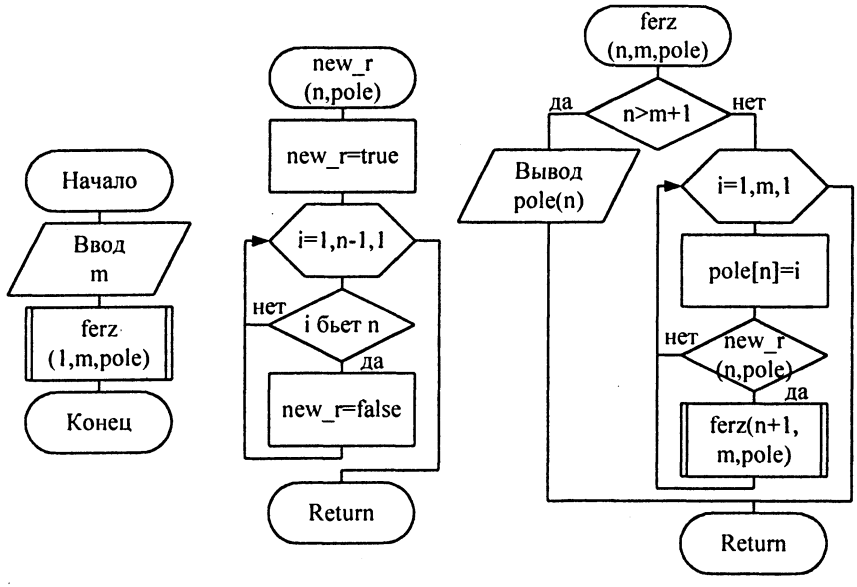
Рис. 5.20. Дерево вариантов с отсечением неперспективных ветвей

исключить варианты, в которых ферзи бьют друг друга. Оставшиеся варианты являются решениями задачи.

На рис. 5.20 показан фрагмент дерева генерации вариантов для m = 4 с отсечением неперспективных ветвей. Первое найденное решение – комбинация 2413.

На рис. 5.21 показаны алгоритм основной программы, рекурсивной подпрограммы добавления ферзя ferz и функции проверки перспективности полученной комбинации new_r, а ниже приведена соответствующая программа.

```
Program ex;
Type p=array[1..100] of integer;
Var pole:p;
    k,m:integer;
{функция проверки перспективности комбинации}
Function new_r(n:integer;pole:p):boolean;
  Var j:integer;
   Begin
     new_r:=false;
     for j:=1 to n-1 do
         if (pole[j]=pole[n])or(abs(pole[j]-pole[n])=n-j) then exit;
     new_r:=true;
   End;
{рекурсивная функция генерации комбинации}
Procedure ferz(n,m:integer; var pole:p);
  Var i:integer;
```



а *б* *в*

Рис. 5.21. Алгоритмы основной программы (*а*), функции проверки перспективности полученной комбинации (*б*) и рекурсивной процедуры добавления ферзя (*в*)

```
Begin
  if n=m+1 then    {если установлено m ферзей, то вывести решение}
    begin
      for i:=1 to m do Write(pole[i]:2);
      WriteLn;
    end
  else      {иначе – пытаемся установить следующего ферзя}
    for i:=1 to m do   {m способами}
      begin
        pole[n]:=i;   {установка n-го ферзя}
        if new_r(n,pole)       {проверка перспективности комбинации}
            then ferz(n+1,m,pole);  {рекурсивный вызов установки
                                      следующего ферзя}
    end;
End;
{основная программа}
Begin
   WriteLn('Введите размер доски:');
   ReadLn(m);
```

Таблица 5.1

Размер доски	Количество вариантов, которое проверяется при полном переборе	Количество вариантов, рассмотренных при ограниченном переборе	Количество полученных решений
4×4	$4^4 = 256$	17	2
5×5	$5^5 = 3125$	54	10
6×6	$6^6 = 46656$	153	4
7×7	$7^7 = 823543$	552	40
8×8	$8^8 = 16777216$	2057	92

```
    k:=0;
    ferz(1,m,pole);
End.
```

Процедуру new_r используют для проверки уже сгенерированной комбинации (уровень n соответствует попытке установить n-го ферзя). Процедура ferz рекурсивна. На каждом уровне она может породить до m рекурсивных вызовов (в соответствии с деревом генерации вариантов). Однако общее количество рассматриваемых вариантов резко уменьшается, так как неперспективные комбинации отсекаются, что наглядно представлено в табл. 5.1.

Задания для самопроверки

Задание 1. Разработайте рекурсивную подпрограмму, осуществляющую поиск комбинации для отпирания кодового замка по методу перебора с отсечением неперспективных комбинаций. Замок представляет собой набор из n переключателей, каждый из которых может находиться в положении «включено» или «выключено». Замок открывается при одном положении переключателей, причем в положении «включено» может находиться не более половины переключателей.

Задание 2. Разработайте рекурсивную подпрограмму, которая формирует из заданного списка предметов определенной стоимости и веса набор, вес которого не превышает заданного, а стоимость максимальна.

6. ФАЙЛОВАЯ СИСТЕМА. ФАЙЛЫ

Файлом называют именованную последовательность элементов данных (компонент файла), расположенных, как правило, во внешней памяти: на дискетах, винчестере, CD или других устройствах хранения информации, также устройствах ввода-вывода. В файле может храниться текст, программа, числовые данные, графическое изображение и т.д. Для организации работы с файлами программа на Borland Pascal взаимодействует с операционной системой MS DOS.

6.1. Файловая система MS DOS

Как сказано выше, каждый файл обязательно имеет имя. Имена файлов в MS DOS подчиняются определенным правилам:

- имя файла должно содержать не менее одного и не более восьми символов;
- имя файла может иметь расширение, которое отделяется от имени точкой и содержит не более трех символов;
- для записи имен и расширений могут использоваться строчные и прописные буквы латинского алфавита a-z, A-Z, арабские цифры и некоторые специальные символы, например, символ подчеркивания «_» или знак доллара «$»;
- в качестве имен запрещается использовать некоторые буквенные сочетания, которые зарезервированы операционной системой для обозначения устройств, например: PRN, CON, NUL, COM1, COM2, AUX, LPT1, LPT2, LPT3.

В операционных системах типа Windows некоторые из этих правил отменяются, например, имя файла может содержать больше восьми символов и включать символы русского алфавита. Однако при работе с файлами из Borland Pascal лучше придерживаться правил MS DOS.

Независимо от используемой операционной системы имена обычно составляют так, чтобы они указывали на содержимое файла. Расширение обычно определяет тип хранящихся данных.

Существуют стандартные расширения, используемые операционной системой, например:

COM, EXE – исполняемые файлы (загрузочные файлы программ);
PAS, BAS, CPP – файлы исходных текстов программ на алгоритмических языках ПАСКАЛЬ, БЭЙСИК и С++ соответственно.

Для удобства работы с группами файлов применяют групповые имена файлов с использованием символов «*» и «?», где «*» соответствует любой последовательности символов, а «?» – одному любому символу, например:

***. EXE** – все файлы с расширением EXE;
A*. COM – все файлы типа COM с именами на букву «А»;
??B. PAS – все файлы типа PAS, имена которых содержат три символа, последний из которых «В»;
PRG1.* – файлы любых типов с именем PRG1;
. – все файлы.

Для того чтобы MS DOS могла размещать файлы на дисках, последние должны быть специальным образом размечены (форматированы). Разметка осуществляется средствами используемой операционной системы. Если форматируется диск, бывший в употреблении, то вся хранившаяся на нем информация уничтожается и восстановлению не подлежит.

Как правило, диски хранят большое количество файлов (количество их на жестких дисках обычно исчисляется тысячами). Для удобства и ускорения работы с таким количеством файлов применяется древовидная структура каталогов, аналогичная библиотечной.

Главным является корневой каталог, не имеющий имени и создаваемый в процессе форматизации диска системой. Файл корневого каталога состоит из записей, содержащих информацию о файлах, хранящихся на диске. В качестве файлов главного каталога могут фигурировать *пользовательские каталоги*, т.е. каталоги второго уровня (подкаталоги), каждый из которых может содержать подкаталоги следующего уровня. Таким образом, получается дерево каталогов (рис. 6.1). Подкаталоги создаются и уничтожаются пользователем с помощью специальных команд. Все каталоги, кроме корневого, имеют имена, образованные по общим правилам операционной системы.

Чтобы найти файл, системе требуется просмотреть всю цепочку каталогов на пути от корневого каталога до подкаталога, хранящего сведения о требуемом файле. Таким образом, чтобы сослаться на файл, нужно указать не только его имя, но и перечислить все предшествующие каталоги. Перечень имен каталогов на пути к файлу называется *маршрутом* или *путем*.

Перечисляемые в маршруте каталоги разделяются символом «\», причем перечень начинается с символа «\», так как корневой каталог не имеет имени. Например:

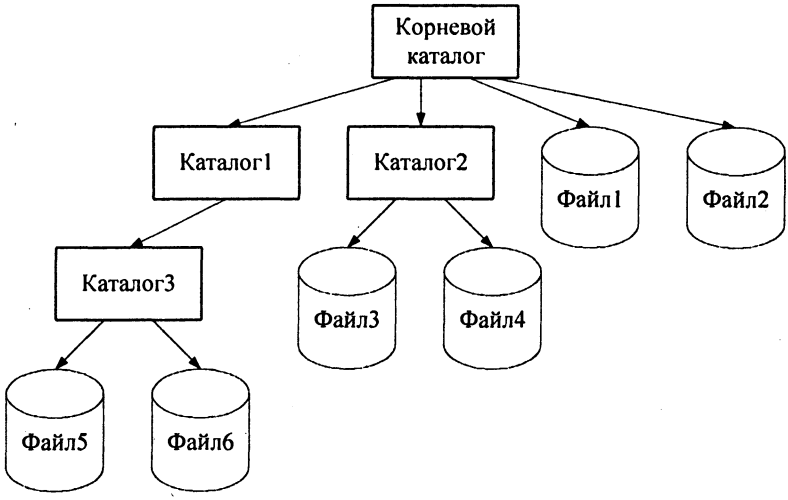


Рис. 6.1. Пример дерева каталогов

\kat1\kat3

Полное имя файла содержит также имя диска, на котором расположен файл. Например:

c:\kat1\ kat3\file5.dat

Такая организация позволяет в разных подкаталогах создавать файлы с одинаковыми именами. Подкаталоги тоже могут иметь одинаковые имена, если они подчинены разным подкаталогам более высокого уровня.

6.2. Файлы Borland Pascal

В Borland Pascal файл определяется как последовательность компонентов, относящихся к одному типу: файл записей, файл целых чисел, файл строк и т. п. Особенностью файлов по сравнению с другими структурными типами данных является то, что в любой момент доступен только один компонент. Количество компонентов файла заранее не определяется. Максимальный размер файла, размещенного во внешней памяти, ограничивается лишь техническими возможностями вычислительной системы.

Различают дисковые файлы и логические устройства.

Дисковый файл представляет собой поименованную область внешней памяти на устройстве хранения информации, например, дискете или винчестере. Физически операции ввода-вывода с файлами выполняются с использованием специального буфера. Так, выводимые записи вначале помещают-

ся в буфер, откуда переписываются в файл по мере заполнения буфера, а вводимые читаются из буфера, куда они были предварительно помещены. Использование буферов позволяет существенно повысить скорость выполнения операций ввода-вывода с файлом, так как на одну операцию ввода-вывода с дисководом, которая выполняется сравнительно медленно, обычно приходятся десятки операций чтения из буфера. Для дисковых файлов принципиально возможен не только последовательный, но и произвольный доступ, при котором чтение информации осуществляется из указанного места.

Логические устройства используют для организации обмена информацией с основными устройствами ввода-вывода, такими как дисплей, клавиатура и т. п. Логические устройства имеют стандартные имена, например:

CON – консоль: при выводе данных соответствует экрану, при вводе – клавиатуре;

PRN – принтер;

NUL – «пустое устройство», обычно заменяет устройство вывода отладочной информации после завершения отладки программы.

В отличие от дисковых файлов с логическими устройствами операции ввода-вывода осуществляют только последовательно, так как при выполнении операций вывода данные передаются на устройство покомпонентно, а при выполнении операций ввода – покомпонентно запрашиваются с него.

Доступ к компоненту файла осуществляется через *указатель файла*. При выполнении операции чтения или записи указатель автоматически перемещается на следующий компонент (рис. 6.2).

Для идентификации файлов в Borland Pascal используют *файловые переменные*. В зависимости от способа представления информации различают три типа файлов, соответственно различаются и способы описания файловых переменных (рис. 6.3).

Типизированные файлы. Файловая переменная типизированного файла описывается как

Type <идентификатор файловой переменной> =
file of <тип компонента>;...

где <тип компонента> – любой тип данных, кроме файлового.

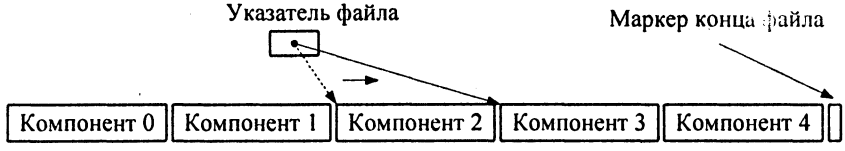


Рис. 6.2. Организация файла

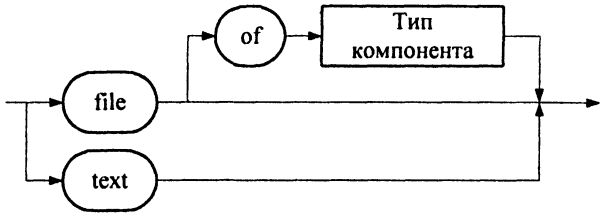


Рис. 6.3. Синтаксическая диаграмма
<Файловый тип>

Типизированные файлы используют, когда обрабатывают хранящуюся в файле или передаваемую с устройства/на устройство последовательность компонентов одинаковой длины (чисел, записей и т.п.).

Текстовые файлы – тип файловой переменной описывается так:

Type <идентификатор файловой переменной> = ***text; ...***

Текстовые файлы используют для работы с текстами, представленными в виде строк переменной длины.

Нетипизированные файлы:

Type <идентификатор файловой переменной> = ***file; ...***

Нетипизированные файлы применяют для организации скоростного обмена между внешней и оперативной памятью физическими записями указанной длины без преобразования и обработки.

Как и любая переменная языка Borland Pascal, файловая переменная может быть описана в инструкции объявления переменных, например:

```
Var     F1: file of real;
        F2: file;
        F3: text; ...
```

или с предварительным объявлением типа:

```
Type FF = file of integer;
Var F1:FF; ...
```

При необходимости файловую переменную допускается передавать в подпрограмму через параметры. Однако следует помнить, что с этой целью можно использовать только параметры-переменные, например:

```
Type FF = file of integer;
Procedure Print(Var F1:FF); ...
```

Работа с файлом включает:

- *инициализацию файловой переменной* – установление связи файловой переменной с файлом;
- *открытие файла* – подготовку файла для выполнения операций ввода/вывода;
- *обработку компонентов файла* – выполнение операций ввода-вывода;
- *закрытие файла* (при повторном открытии файл закрывается автоматически).

Инициализация файловой переменной. Связь между физическим устройством (дисководом или внешним устройством) и файловой переменной устанавливается специальной процедурой.

Процедура ***Assign (Var f; st:string)*** – инициализирует файловую переменную f, связывая ее с файлом или логическим устройством, определенным строкой st.

Если файл находится в текущем каталоге, то достаточно указать имя файла и его расширение. В противном случае необходимо указать полное имя файла, например:

```
Type  FI1 = text;
Var  f1,f2,f3: FI1;
...
Assign (f1, 'F1.dat'); {связывание файловой переменной с файлом
                                    в текущем каталоге}
Assign (f2, 'd:\iva\a.dat'); {связывание файловой переменной с файлом
                                    в указанном каталоге}
Assign(f3,'CON');     {связывание файловой переменной с консолью}
```

Открытие файла. Открытие файла предполагает указание направления передачи данных. В Borland Pascal файл можно открыть для *чтения* и для *записи*. Текстовый файл можно открыть также для добавления строк. В типизированный файл, открытый для чтения, можно дописывать новые записи или писать в нем новые записи на место старых.

1. Процедура ***ReSet(Var f)*** – открывает файл, определенный файловой переменной f для чтения. При выполнении этой процедуры указатель файла устанавливается на первый компонент файла (физически первый блок записей считывается в буфер). Логическое устройство в этом случае готовится к выполнению операций ввода.

При открытии для чтения несуществующего файла регистрируется ошибка выполнения, а функция IOResult типа Word возвращает значение, отличное от 0 (см. далее описание функции). Отключив контроль операций ввода-вывода и используя функцию IOResult, можно организовать проверку наличия файла с указанным именем на диске:

```
Var f: file of char;
Begin
  Assign( f, 'a. dat ' );  {инициализация файловой переменной}
  {$ I-} {отмена контроля ошибок ввода-вывода}
  ReSet (f); {открытие файла для чтения}
  {$ I+} {включение контроля ошибок}
   if IOResult <>0 then  WriteLn ('Файл не существует');
                  else   WriteLn('Файл существует '); ...
```

2. Процедура ***ReWrite(Var f)*** – открывает файл, определенный файловой переменной f, для записи. При открытии для записи существующего файла старый файл *уничтожается без предварительной проверки и выдачи предупреждения пользователю*. Если файла с таким именем не существовало, то он создается и подготавливается к записи (физически – очищается буфер). Логическое устройство при этом подготавливается к приему информации.

3. Процедура ***AppEnd(Var f:text)*** – открывает текстовый файл, определенный файловой переменной f, для добавления строк.

При открытии для добавления строк указатель файла устанавливается на конец файла, и, соответственно, все строки, выводимые в файл, дописываются к уже существующему файлу.

Любой программе без объявления, инициализации файловой переменной и открытия доступны два файла со стандартными файловыми переменными:

INPUT – чтение со стандартного устройства ввода;
OUTPUT – вывод на стандартное устройство вывода.

Это текстовые файлы, используемые для выполнения элементарных операций ввода-вывода. В операторах ввода-вывода файловые переменные этих файлов обычно не указывают (см. параграф 2.6). Остальные файлы становятся доступными только после связывания файловой переменной с файлом или логическим устройством и открытия файла.

Стандартным устройством ввода MS DOS по умолчанию является клавиатура. Стандартным устройством вывода – экран дисплея.

Примечание. При необходимости эти устройства можно *переназначить* средствами операционной системы. Так, для организации ввода данных из файла вместо ввода с клавиатуры необходимо запустить программу из командной строки MS DOS, указав после имени программы символ «<» и имя файла, а для организации вывода в файл вместо вывода на экран – символ «>» и имя файла. Можно перенаправить только ввод или только вывод или и то и другое сразу.

Например:

A:\>example.exe <a.dat >a.res – ввод из файла a.dat, а вывод в файл a.rez.

Такое переназначение будет выполнено, если в программе не используется модуль crt (см. параграф 8.1), который организует операции ввода-вывода напрямую, непосредственно взаимодействуя с устройством.

Обработка компонентов файла. Основные операции над компонентами – это операции записи и чтения. На базе этих операций выполняют более сложные операции:

- *создание файла* – занесение в файл требуемых записей;
- *модификация файла* – изменение всех или нескольких записей, добавление и удаление записей;
- *поиск* нужной информации в файле.

Выполнение этих операций осуществляется по-своему для каждого типа файла (см. параграфы 6.3 – 6.5).

Закрытие файла. Закрытие файла, открытого для записи или чтения, осуществляется процедурой

Close(Var f).

При этом вновь созданный файл регистрируется в каталоге. Поскольку любое обращение к диску осуществляется через буферную память, часть данных, выводимых в файл, может остаться в буфере. Процедура закрытия файла обеспечивает вывод оставшихся компонентов из буфера в файл. Связь файловой переменной с файлом при закрытии сохраняется, и при повторном использовании этого же файла процедуру Assign применять еще раз не требуется.

Стандартные процедуры и функции обслуживания файлов. Для взаимодействия с файловой системой MS DOS в Borland Pascal определены стандартные процедуры и функции, которые применимы к файлам любых типов.

1. Процедура ***ReName(Var f; name:string)*** – осуществляет переименование файла, определенного файловой переменной f. Новое имя файла задается параметром name. Если в процессе работы программы требуется переименовать файл, открытый для чтения или записи, необходимо предварительно закрыть этот файл. При совпадении нового имени файла с каким-либо уже существующим выдается сообщение об ошибке.

2. Процедура ***Erase(Var f)*** – осуществляет удаление созданного или находящегося в процессе формирования файла. Перед уничтожением файл должен быть закрыт, так как разрешается удалять только закрытые файлы.

3. Функция ***EOF(Var f):boolean*** – определяет конец файла. Как было отмечено выше, размер файла при его создании не фиксируется. Поэтому в процессе работы требуется проверка достижения конца файла. Функция принимает значение TRUE, если указатель стоит в конце файла (после последней записи). При этом, если производится чтение, то это означает, что файл исчерпан, а если идет запись, то новая запись дописывается в конец файла. Функция принимает значение FALSE, если конец файла еще не достигнут.

Примечание. Функция EOF по-разному работает с дисковыми файлами и логическими устройствами. Для логического устройства невозможно предвидеть, каким будет результат чтения очередного символа. Поэтому при работе с логическим устройством функция EOF возвращает TRUE, если *последним символом был* маркер конца файла, а при чтении с диска – если *следующим считываемым символом будет* маркер конца файла. Физически это выражается в том, что при выполнении функции EOF запрашивается ввод информации с клавиатуры.

В качестве маркера конца файла используется символ ASCII с кодом 26 (#26). При работе с клавиатурой этот код формируется при вводе комбинации CTRL-Z. Считается, что признак конца файла физически присутствует в файле, однако, как правило, такой символ в конце дискового файла отсутствует, и конец файла в системе определяется другим способом.

4. Функция ***IOResult(Var f): word*** – возвращает код правильности выполнения операций ввода/вывода. Если ошибок не зафиксировано, то функция возвращает 0. Информация об ошибках может быть получена и обработана в режиме компилятора {$I-} – отключение контроля ошибок ввода/вывода.

5. Процедура ***Truncate(Var f)*** – обрезает файл, оставляя компоненты до того, на который ссылается указатель файла (кроме текстовых файлов).

6. Процедура ***ChDir(path:string)*** – изменяет текущий каталог: назначает текущим каталог, указанный параметром path.

7. Процедура ***GetDir(drive:word; Var dir:string)*** – возвращает имя текущего каталога на указанном устройстве, где устройство drive: 0 – устройство по умолчанию; 1 – диск A; 2 – диск B и т.д.

8. Процедура ***MkDir(dir:string)*** – создает новый каталог. Строка dir определяет путь и новое имя.

9. Процедура ***RmDir (dir:string)*** – удаляет каталог с указанным именем. Каталог должен быть пустым.

6.3. Текстовые файлы

Текстовый файл – это файл, компонентами которого являются символьные строки переменной длины, заканчивающиеся специальным маркером конца строки (рис. 6.4).

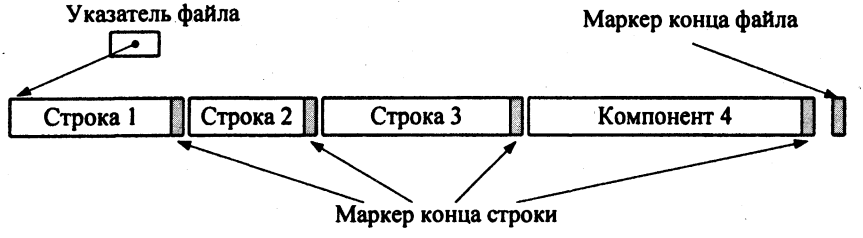


Рис. 6.4. Структура текстового файла

Примечание. Маркер конца строки – это последовательность из двух специальных символов по таблице ASCII «#13, #10». Символ с кодом 13 интерпретируется в компьютере как команда установки курсора в начало строки, а символ с кодом 10 – как команда перехода на следующую строку. Как уже упоминалось ранее, такая комбинация кодов вводится при нажатии клавиши ENTER.

Текстовый файл можно открыть для записи, чтения и добавления записей в конец (см. параграф 6.2). Файл, открытый для записи, не может использоваться для чтения и наоборот. При открытии файла для добавления система проверяет, не был ли файл открыт для чтения или записи, и если такое открытие имело место, то производится сначала закрытие файла, а затем уже открытие для добавления.

Текстовые файлы используют для хранения и обработки текстовой информации: символов, строк, символьных массивов. Логические и числовые данные при записи в текстовые файлы должны преобразовываться в символьные строки.

Следует иметь в виду, что при необходимости текстовый файл может быть создан или прочитан любым текстовым редактором, в том числе и текстовым редактором, входящим в состав среды Borland Pascal.

Для работы с текстовыми файлами используют специальные процедуры и функции.

1. Функция ***EOLn([Var f]): boolean*** – возвращает TRUE, если во входном текстовом файле достигнут маркер конца строки; при отсутствии файловой переменной проверяется стандартный файл INPUT, который обычно связан с клавиатурой.

Примечание. Функция EOLn, как и EOF, по-разному работает с дисковыми файлами и логическими устройствами. Для логического устройства невозможно предвидеть, каким будет результат чтения очередного символа. Поэтому при работе с логическим устройством функция EOLN возвращает TRUE, если *последним считанным символом был* символ #13. При работе с диском функция EOLN возвращает TRUE, если *следующим считанным символом будет* символ #13.

2. Процедура ***Read([Var f:text;] v1, v2,... vn)*** – обеспечивает ввод символов, строк и чисел. Список ввода представляет собой последовательность из одной или более переменных типа CHAR, STRING, а также любого целого и вещественного типа. При отсутствии файловой переменной ввод осуществляется из стандартного файла INPUT.

При вводе значений переменных типа CHAR выполняется чтение одного символа из файла, считанное значение присваивается очередной переменной из списка ввода. Как уже упоминалось в параграфе 2.6, символы вводятся подряд, а не через пробел, как числа. Если перед выполнением чтения указатель файла достиг конца очередной строки, то результатом чтения будет символ #13, а если был достигнут конец файла, то – символ #26.

При вводе переменных типа STRING количество считанных процедурой и помещенных в строку символов равно максимальной длине строки, если раньше не встретились маркеры конца строки или конца файла, которые в строку не включаются. Символы, выходящие за размер максимальной длины строки, отбрасываются. Новое обращение к процедуре Read вернет пустую строку (см. также параграф 2.6). Следовательно, процедура Read не в состоянии читать последовательность строк, так как первая строка будет прочитана правильно, а все последующие окажутся пустыми.

При вводе числовых данных процедура Read пропускает все пробелы, знаки табуляции и маркеры до первого значащего символа и читает строку до пробела, знака табуляции или маркера. Полученная подстрока преобразуется из символьного во внутреннее представление в соответствии с типом значения и присваивается следующей переменной из списка. Если нарушен формат, то фиксируется ошибка ввода-вывода. Если достигнут маркер конца файла, то переменной присваивается значение 0, причем никаких сообщений в этом случае не выдается.

Ввод логических констант процедурами Read и ReadLn не предусмотрен.

Чтение с клавиатуры осуществляется через буфер, который передается процедуре при нажатии клавиши ENTER и имеет размер 127 байт, поэтому ввести с клавиатуры строку большего размера нельзя.

3. Процедура ***ReadLn(*** [*Var f;*] ***v1,v2, ...,vn)*** – также обеспечивает ввод символов, строк и чисел. Процедура использует те же правила ввода, что и процедура Read, но после чтения последней переменной оставшаяся часть строки до маркера конца строки пропускается, так что следующее обращение к ReadLn или Read начнется с первого символа новой строки. Процедура может быть вызвана без указания списка ввода, что приведет к пропуску всех символов текущей строки до маркера конца строки.

Процедуры Read и ReadLn могут использоваться без указания файловой переменной. Тогда операция чтения осуществляется из стандартного файла INPUT. Использование процедуры ReadLn без параметров после процедуры Read приведет к очистке буфера ввода. Применение этой же процедуры без предшествующей ей процедуры Read переводит программу в состояние ввода, т.е. выполнение программы приостанавливается до нажатия клавиши ENTER, что может использоваться для организации паузы на время просмотра содержимого экрана.

4. Процедура ***Write(*** [*Var f;*] ***v1,v2, ...,vn)*** – обеспечивает вывод данных в текстовый файл или передачу их на логическое устройство. Список вывода – последовательность из одного или более выражений типа CHAR, STRING, BOOLEAN, а также целого или вещественного типов. При выводе числовых значений последние преобразуются в символьное представление. При отсутствии файловой переменной вывод осуществляется в стандартный файл OUTPUT, который обычно назначен на экран.

Любой параметр из списка вывода может иметь формат:

<параметр> [: <целое1> [: < целое2>]],

где <целое1> и <целое2> интерпретируются в соответствии с правилами, описанными в параграфе 2.6.

5. Процедура ***WriteLn([Var f;] v1,v2, ...,vn)*** – обеспечивает вывод информации в текстовый файл или ее передачу на логическое устройство вывода. При отсутствии файловой переменной вывод осуществляется в стандартный файл OUTPUT, который обычно связан с дисплеем.

Процедура полностью идентична процедуре Write, за исключением того, что выводимая строка символов завершается символами #13 и #10. При вызове WriteLn допускается опускать список вывода, в этом случае в файл передается маркер конца строки (при выводе на экран это приведет к переводу курсора в начало следующей строки).

6. Функция ***SeekEOLn([Var f]):boolean*** – пропускает все пробелы и знаки табуляции до маркера конца строки или до первого значащего символа и возвращает TRUE при обнаружении маркера. Если файловая переменная не указана, то функция проверяет стандартный файл INPUT.

7. Функция ***SeekEOF([Var f]):boolean*** – пропускает все пробелы, знаки табуляции и маркеры конца строки до маркера конца файла или до первого значащего символа и возвращает TRUE при обнаружении маркера. Если файловая переменная отсутствует, то функция проверяет стандартный файл INPUT.

Рассмотрим несколько примеров.

Пример 6.1. Разработать программу, которая формирует текстовый файл из 26 строк, содержащих случайное количество соответствующих прописных букв латинского алфавита, например:

```
AAAAA
BBBBB
C
DDDDDDDDDDDDDDDDDDDDD
EEEEEEEEEEEEE и т.д.
```

```
Program form_text_file;
Var
   f:text;      {файловая переменная для текстового файла}
  a:char;  n,i:integer;  fname,st:string[30];
Begin
    WriteLn('Введите имя файла');  ReadLn(fname);
    Assign(f,fname);   {инициализируем файловую переменную}
    ReWrite(f);        {открываем файл для записи}
    Randomize;         {инициализируем датчик случайных чисел}
```

```
  for a:='A' to 'Z' do    {формируем строки}
      begin
        st:='';
        n:=Random(30)+1;
       for i:=1 to n do   st:=st+a;
        WriteLn(f,st);  {записываем строку в текстовый файл}
        WriteLn(st);      {для контроля – выводим ее на экран}
      end;
  Close(f);   {закрываем файл}
End.
```

Поскольку компоненты текстового файла могут иметь различную длину, возможна только последовательная их обработка (запись, чтение и поиск). Любой вид модификации файла, кроме добавления записей в конец, выполняется с перезаписью информации в другой файл. Так, для того чтобы исключить некоторую запись, необходимо переписать все строки, кроме подлежащей исключению, в другой текстовый файл. При этом обычно старый файл удаляют, новый – переименовывают, присваивая ему имя исходного файла, и файловую переменную связывают с измененным файлом.

Пример 6.2. Разработать программу, которая удаляет из текстового файла «пустые» строки: строки, не содержащие символов, и строки, содержащие только пробелы и знаки табуляции.

Поскольку в результате обработки часть строк текстового файла будет удалена, нам потребуется создать специальный файл, куда будут помещены непустые строки файла.

```
Program ex;
 Var f1,f2:text;   {файловые переменные текстовых файлов}
    st,name:string;
 Begin
    WriteLn('Введите имя файла:'); ReadLn(name);
    Assign(f1,name);  {инициализируем файловую переменную}
    {$I-}    {проверяем существование файла}
    Reset(f1);
    {$I+}
    if IOResult=0 then    {если файл с заданным именем существует}
   begin
    Assign(f2,'temp.dat');  {инициализируем новый файл}
    Rewrite(f2);              {открываем новый файл для записи}
    while not EOF(f1) do    {пока не достигнут конец файла}
      begin
        if SeekEOLn(f1) then ReadLn(f1,st)  {если строка пустая,
                                              то пропускаем ее}
```

```
      else
        begin
          ReadLn(f1,st);   {читаем строку}
          WriteLn(f2,st);  {записываем ее в новый файл}
        end;
      end;
    Close(f1);    {закрываем старый файл}
    Close(f2);    {закрываем новый файл}
    Erase(f1);    {удаляем старый файл}
    ReName(f2,name);  {переименовываем новый файл}
  end
 else   WriteLn('Файл с таким именем не найден.');
End.
```

Задания для самопроверки

Задание 1. Дан текстовый файл, состоящий из символьных строк, представляющих собой совокупность слов, разделенных пробелами. Разработайте программу, которая переформирует файл, заменяя в каждой строке сочетание «это» на сочетание «то», и удаляя слова, начинающиеся с символа #. Вывести на экран скорректированный файл.

Задание 2. Разработайте программу, которая осуществляет поиск в текстовом файле заданных слов. Слова последовательно вводятся с клавиатуры. Для каждого слова должно определяться количество вхождений и номера строк текста. Если указанные слова в файле отсутствуют, то программа должна выводить соответствующее сообщение.

6.4. Типизированные файлы

Типизированный файл – это файл, все компоненты которого одного типа, заданного при объявлении файловой переменной (рис. 6.5). Компоненты файла хранятся на диске во *внутреннем* (двоичном) формате и нумеруются с 0. Если посмотреть такой файл любым текстовым редактором, то можно распознать только символьную информацию, на месте же чисел в файле будут располагаться пробелы или символы псевдографики.

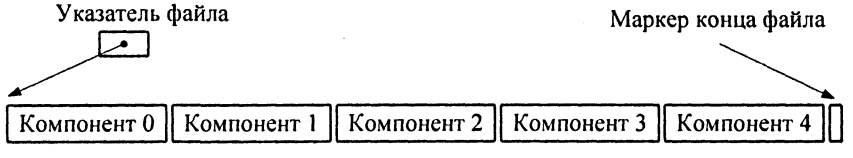


Рис. 6.5. Типизированный файл

Для работы с типизированными файлами используют специальные процедуры и функции.

1. Процедура ***Read(Var f; c1, c2, ...,cn)*** – осуществляет чтение очередных компонентов типизированного файла. Список переменных ввода содержит одну или несколько переменных того же типа, что и компоненты файла, разделенных запятыми. Если файл исчерпан, обращение к процедуре вызывает ошибку ввода-вывода.

2. Процедура ***Write(Var f; c1, c2, ...,cn)*** – осуществляет запись данных в типизированный файл. Список вывода содержит одно или более выражений того же типа, что и компоненты файла, разделенных запятыми.

3. Процедура ***Seek(Var f; numcomp:word)*** – осуществляет установку указателя файла на компонент файла с номером numcomp.

4. Функция ***FileSize(Var f):longint*** – возвращает количество компонент файла, указанного файловой переменной. Может использоваться для установки на конец файла совместно с Seek():

Seek(f, FileSize(f));...

5. Функция ***FilePos(Var f):longint*** – возвращает порядковый номер компонента, который будет обрабатываться следующей операцией ввода-вывода.

После открытия файла для чтения или записи указатель файла стоит в его начале и указывает на первый компонент, имеющий номер 0. После каждого чтения или записи указатель сдвигается к следующему компоненту файла. Поскольку длина каждой компоненты файла строго постоянна, помимо последовательного возможно осуществление *прямого доступа* к компонентам файла. По той же причине, если требуется изменить компонент файла, то не обязательно переписывать компоненты в другой файл, а достаточно установить указатель файла на изменяемый компонент и записать новый компонент на место старого.

Добавление компонентов в конец файла выполняется в режиме чтения. Для этого указатель файла устанавливается на его конец (как показано выше), после чего все выводимые компоненты дописываются в конец файла.

Добавление компонентов в середину или начало файла может выполняться следующим образом: определяем место, в которое должны быть добавлены элементы, все последующие компоненты переписываем во временный файл, вставляем новые компоненты и, наконец, дописываем в файл компоненты, переписанные во временный файл.

Удаление компонент обычно требует перезаписи файла.

Пример 6.3. Разработать программу, создающую файл, компонентами которого являются символы, введенные с клавиатуры. Затем эта программа должна изменять символы, записанные в файл, организовывать чтение символов из файла попеременно с начала и с конца (прямой доступ), находить указанный символ в файле и удалять его из файла.

```
Program ex;
Var f, f1:file of char;  { две файловые переменные}
    ch,i:char;
    j:longint;
    name:string[8];
Begin
   WriteLn('Введите имя файла:');   ReadLn(name);
   {создание и открытие файла}
   Assign(f, name+'.dat'); {связываем файл с файловой переменной}
   ReWrite(f);             {открываем файл для записи (создаем)}
   WriteLn('Вводите символы или CTRL-Z:');
   {занесение записей в файл}
   while not EOF do {пока не введено CTRL-Z с клавиатуры}
      begin
         ReadLn(ch);  {вводим символ с клавиатуры}
         Write(f,ch);  {записываем символ в файл}
      end;
   WriteLn;
   {последовательное чтение записей из файла}
   ReSet(f);                {открываем файл для чтения}
   while not EOF(f) do      {пока не достигнут конец файла}
    begin
      Read(f,ch);           {читаем символ из файла}
      Write(ch,' ');        {выводим символ на экран}
    end;
   WriteLn;
   {изменение записей в файле}
   ReSet(f);                {открываем файл для чтения}
   while not EOF(f) do      {пока не достигнут конец файла}
     begin
         Read(f,i);            {читаем символ из файла}
         Write(i,' ');         {выводим символ на экран}
         i:=chr(ord(i)+10);  {изменяем символ}
         WriteLn(i);           {выводим на экран измененный символ}
         Seek(f,FilePos(f)-1);  {возвращаемся на один компонент}
         Write(f,i);           {перезаписываем символ}
     end;
   WriteLn;
   {попеременное чтение записей с начала и конца файла}
   ReSet(f);        {открываем файл для чтения}
  j:=0;             {устанавливаем номер компонента равным 0}
   while not EOF(f) do      {пока не достигнут конец файла}
     begin
```

```
            Read(f,i);    {читаем символ из начала файла}
            Write(i);     {выводим символ на экран}
            Seek(f,FileSize(f)-FilePos(f)); {устанавливаем указатель
                                  для чтения из конца файла}
            Read(f,i);  {читаем символ из конца файла}
            Write(i);     {выводим символ на экран}
            j:=j+1;     {увеличиваем номер компонента}
            Seek(f,j);  {устанавливаем указатель на следующий от
                                  начала компонент}
        end;
    WriteLn;
    WriteLn('Введите символ для удаления:'); ReadLn(ch);
    {подготовка к удалению записей: переименование исходного
                файла и открытие нового файла с тем же именем}
    Close(f);                    {закрываем файл}
    ReName(f,name+'.bak'); {переименовываем файл}
    ReSet(f);                    {открываем файл для чтения}
    Assign(f1,name+'.dat');  {связываем новый файл с переменной}
    ReWrite(f1);                 {открываем новый файл для записи}
    {удаление записей – перепись остающихся записей в другой файл}
    while not EOF(f) do
      begin
        Read(f,i);               {читаем символ из файла}
      if i<>ch then Write(f1,i); {если символ не подлежит удалению, то
                записываем его в новый файл}
      end;
    Erase(f);    {удаляем старый файл, после закрытия в нем ничего не
                изменилось, поэтому повторно его можно не закрывать}
    {последовательное чтение записей из нового файла}
    ReSet(f1);    {открываем новый файл для чтения}
    while not EOF(f1) do
      begin
        Read(f1,ch);  {читаем из файла}
        Write(ch,' ');
      end;
    WriteLn;
End.
```

Пример 6.4. Разработать программу, которая создает файл, содержащий список фамилий и даты рождения. Осуществить поиск в этом файле даты рождения по заданной фамилии.

```
Program ex;
 Type fam=record      {тип запись «сведения о сотрудниках»}
                ff:string[20];  {фамилия}
                year:word;      {год рождения}
                month:1..12;    {месяц рождения}
                day:1..31       {день рождения}
              end;
 Var f:file of fam;  {файловая переменная «файл сотрудников»}
     fb:fam;
     n,i:integer;
     fff:string;
     key:boolean;
 Begin
     Assign(f,'a.dat');  {связываем файловую переменную с файлом}
     ReWrite(f);         {открываем файл для записи}
     WriteLn('Введите данные или CTRL-Z');
     while not EOF do  {цикл, пока не введено CTRL-Z}
       begin
         ReadLn(fb.ff, fb.year, fb.month, fb.day); {вводим данные по полям,
                        фамилию вводим в отдельной строке, так как ввод
                        строки завершается нажатием клавиши Enter}
         Write(f,fb);  {заносим запись в файл как один компонент}
       end;
     Close(f);         {закрываем файл}
     WriteLn('Введите фамилию');
     Readln(fff);
     key:=false;     {устанавливаем признак «запись не найдена»}
     ReSet(f);         {открываем файл для чтения}
     while (not EOF(f)) and (not key) do  {пока не обнаружен конец
                                           файла и не найдена запись}
       begin
        Read(f,fb);      {читаем запись из файла}
        if fb.ff=fff then  {если фамилии совпадают, то}
          begin  {выводим данные}
             WriteLn('Дата:',fb.year,fb.month:3,fb.day:3);
             key:=true;  {устанавливаем признак «запись найдена»}
          end;
       end;
     if not key then                    {если признак не установлен}
                 WriteLn('Нет данных');  {то выводим «нет данных»}
     Close(f);  {закрываем файл}
 end.
```

Следует отметить, что любой текстовый файл может быть прочитан как типизированный файл с компонентами типа CHAR. В этом случае необходимо учитывать, что маркер конца строки текстового файла рассматривается в символьном представлении как последовательность из двух символов #13 и #10.

Пример 6.5. Разработать программу, которая открывает текстовый файл как типизированный с компонентом типа CHAR и читает его по символу.

```
Program char_text_file;
Type  ff=file of char;  {новый тип - символьный файл}
Var
     f:ff;          {файловая переменная типа файл символов}
    a:char;
    n,i:integer;
    fname,st:string[30];
Begin
   WriteLn('Введите имя файла');    ReadLn(fname);
   Assign(f,fname);  {связываем файловую переменную с файлом}
   ReSet(f);  {открыть текстовый файл как типизированный на чтение}
    while not EOF(f) do   {пока не достигнут конец файла}
   begin
      st:='';
     Read(f,a);      {читаем символ}
          while (a<>#13) and not EOF(f) do   {до маркера конца строки
                                              или конца файла}
        begin
          st:=st+a;   {добавляем считанный символ в строку}
          Read(f,a);     {читаем очередной символ}
        end;
     if not EOF(f) then Read(f,a); {пропускаем символ #10}
    WriteLn(st);                  {выводим сформированную строку}
   end;
   Close(f);
 End.
```

Задания для самопроверки

Задание 1. Разработайте программу, которая создает типизированный файл, содержащий сведения об импортируемых в Россию товарах: наименование товара, страна, поставляющая товар, и объем поставляемой партии. В сформированном файле определить товары, импортируемые из страны P (вводимой с клавиатуры в процессе выполнения программы), а также объем партий. Если импорт из страны отсутствует, вывести соответствующее сообщение.

Задание 2. Разработайте программу, которая формирует типизированный файл из К целых чисел в диапазоне –50 +120, используя датчик случайных чисел как для задания К (К находится в диапазоне от 1 до 100), так и для задания значений компонент файла. Для сформированного файла определите сумму его четных отрицательных компонент и поместите эту сумму вместо максимального по абсолютной величине компонента этого же файла.

Задание 3. Дан типизированный файл вещественных чисел. Разработайте программу, которая определяет среднее арифметическое значение компонент файла. Удалите из файла все компоненты, меньшие найденного среднего арифметического. Выведите на экран исходный и переформированный файлы и значение среднего арифметического.

6.5. Нетипизированные файлы

Нетипизированными называют файлы, объявленные без указания типа его компонентов. Операции чтения и записи с такими файлами осуществляются блоками. Отсутствие типа компонента делает эти файлы совместимыми с любыми другими, а выполнение ввода/вывода блоками позволяет организовать высокоскоростной обмен данными между диском и памятью. Нетипизированные файлы, как и типизированные, допускают организацию прямого доступа.

Нетипизированный файл можно открыть для записи и чтения, используя процедуры ReSet и ReWrite. При открытии нетипизированного файла этими процедурами вторым параметром recsize можно указать длину записи файла в байтах. Если длина записи не указана, она принимается равной 128 байтам:

ReSet (Var f; [recsize:word]);
ReWrite(Var f; [recsize:word]).

Длина записи recsize – положительное число, не превышающее 65535 байт. Для обеспечения максимальной скорости обмена данными следует задавать длину, которая была бы кратна размеру сектора диска (512 байт), например: 1024, 2048.

При работе с нетипизированными файлами можно использовать все процедуры и функции, предназначенные для работы с типизированными файлами, за исключением процедур Read и Write. Эти процедуры заменяются высокоскоростными процедурами BlockRead и BlockWrite.

1. Процедура ***BlockRead(Var f:file;Var buf; Count:word*** [***;res:word***]) – осуществляет чтение блока записей из файла в буфер.

Параметр buf определяет буфер, который будет участвовать в обмене данными. Размер буфера должен быть достаточен для размещения Count записей указанной в процедуре ReSet длины.

Параметр res будет содержать количество фактически обработанных записей. *Если последняя запись – неполная, т.е. ее длина меньше указанной длины записи, то значение параметра res не будет ее включать.*

2. Процедура ***BlockWrite(Var f:file; Var buf; Count:word*** [***;res:word***] ***)*** – осуществляет запись блока из буфера buf в файл.

Пример 6.6. Разработать программу копирования файлов. При создании данной программы нам безразлично, что именно хранится в файле, поэтому используем нетипизированные файлы. При этом длину записи установим равной 1, буфер предусмотрим на 2048 байт. Так мы гарантируем отсутствие неполных записей и в то же время за одну операцию будем обрабатывать блок большого размера.

```
Program copir;
 Const recs=1024;  {размер записи}
 Var  fi, fo:file;{нетипизированные файлы}
      buf : array [1..2*recs] of byte;     {буфер на 2048 байт}
      i:word;
      namein,nameout: string;
 Begin
     WriteLn('Введите имя файла – источника:');
     ReadLn(namein);
     {проверка наличия файла с указанным  именем}
     Assign(fi,namein);
     {$I-}
      ReSet(fi,1);  {открываем файл для чтения}
     {$I+}
      if IOResult <> 0 then
           begin
              WriteLn(#7,' Не существует файла с именем ',namein);
              Halt
           end;
      WriteLn('Введите имя файла - приемника ');
      ReadLn(nameout);
      Assign(fo,nameout);
      ReWrite(fo,1);    {открываем файл для записи}
       while not EOF(fi) do
          begin
           BlockRead(fi,buf,sizeof(buf),i);{читаем блок из входного файла}
            BlockWrite(fo,buf,i);{пишем блок из буфера в выходной файл}
          end;
      Close(fi);
      Close(fo)
End.
```

6.6. Процедуры и функции библиотеки DOS для работы с файлами

Borland Pascal предоставляет программисту возможность обращения к некоторым функциям MS DOS, обеспечивающим функционирование файловой системы. Для этого необходимо в начале программы подключить библиотеку Dos с помощью директивы Uses Dos, чтобы сделать доступными ее ресурсы, описанные в этом разделе.

Для работы с файлами библиотека Dos содержит следующие процедуры и функции.

1. Функция ***DiskFree(drive:byte):LongInt*** – определяет и возвращает объем свободного места на диске ***drive*** в байтах. Параметр drive может принимать значения: 0 – устройство по умолчанию, 1 – диск A, 2 – диск B и т.д.

2. Функция ***DiskSize(drive:byte):LongInt*** – возвращает полный объем указанного диска в байтах. Параметр drive принимает те же значения, как и в предыдущей функции.

3. Процедура ***GetFTime(Var f; Var time:LongInt)*** – возвращает время создания или последнего обновления указанного файла f. Время возвращается в упакованном формате, который необходимо распаковать.

4. Процедура ***UnPackTime (time:LongInt; Var DT:DateTime)*** – распаковывает значение параметра time, который содержит время в упакованном формате, в специальный тип DateTime, описанный в модуле Dos следующим образом:

```
Type DateTime = record
       year :   word;    {год}
       month:   word;    {месяц: 1..12}
       day:     word;    {число: 1..31}
       hair:    word;    {часы: 0..23}
       min:     word;    {минуты: 0..59}
       sec:     word;    {секунды: 0..59}
                end;
```

5. Процедура ***PackTime(DT:DateTime;Var time:LongInt)*** – позволяет упаковать параметр DT типа DateTime в целое число time.

6. Процедура ***SetFTime(Var f; time:LongInt)*** – используется для установки даты создания или обновления файла; time – время и дата в упакованном формате.

7. Процедура ***GetFAttr (Var f; Var Attr:word)*** – применяется для получения атрибутов файла, указанного файловой переменной. Атрибуты файла («только чтение», «скрытый», «системный» и т.д.) кодируются битами байта атрибутов файла. Комбинация битов в байте может указывать самые разные варианты. Байт атрибутов возвращается в младшем байте параметра Attr.

Для расшифровки байта атрибутов в модуле Dos описаны специальные маски длиной 1 байт:

```
Const   Readonly  = $01;    {только чтение}
        Hidden    = $02;    {скрытый файл}
        Sysfile   = $04;    {системный файл}
        VolumeID  = $08;    {идентификатор тома}
        Directory = $10;    {имя подкаталога}
        Archive   = $20;    {архивный файл}
        Anyfile   = $3F;    {любой файл}
```

Для того чтобы определить, установлен или нет соответствующий бит, используют фрагменты, аналогичные следующему:

```
GetFAttr(f,AttrF);
if Lo(AttrF) && ReadOnly <>0 then
      WriteLn('Файл имеет атрибут "только чтение"');
```

8. Процедура ***SetFAttr(Var f; Attr:word)*** – служит для установки атрибутов файла указанного файловой переменной.

9. Функция ***FSearch(path:PathStr; DirList:string):PathStr*** – возвращает путь к файлу, заданному в строке path, который ищется в указанном списке каталогов; имена каталогов в списке должны быть разделены точкой с запятой «;». Если файл не найден, то возвращается пустая строка.

10. Процедура ***FSplit(path:PathStr; Var Dir:DirStr; Var name:NameStr; Var Ext:ExtStr)*** – осуществляет расщепления имени файла, заданного в path, т. е. возвращает в качестве отдельных строк путь к файлу dir, его имя name и расширение ext. Процедура не проверяет наличие на диске указанного файла.

11. Функция ***FExpand(path:PathSrt):PathStr*** – возвращает полное имя указанного файла. Функция не проверяет наличие файла на диске, а просто дополняет имя файла недостающими параметрами.

Пример 6.7. Разработать программу, которая обращается к функциям библиотеки Dos для:

- определения и изменения даты создания файла;
- определения атрибутов файла;
- определения количества свободного места на диске.

```
Program fundos; { проверка функций библиотеки dos}
 Uses dos,crt;
 Var fi,fo:text;    {текстовые файлы}
    k,i:word;          time,size:longint;      date:DateTime;
    pathf:DirStr;      namef:NameStr;          extf:ExtStr;
    atr:byte; {переменные для хранения байта атрибутов файла}
    name:string;
```

```
Begin
  ClrScr;
  WriteLn('Введите имя файла:');
  ReadLn(name);
  Assign(fi,name);
  {$I-} ReSet(fi);{$I+}  {проверяем наличие файла с именем name}
  if IOResult <> 0 then
     begin
       WriteLn(#7,' нет файла с именем ',name);
       Halt;
     end;
  GetFTime(fi,time);     {определяем дату создания файла}
  UnPackTime(time,date);  {распаковываем дату}
  WriteLn('Дата создания файла=',date.year:5,date.month:3, date.day:3);
  WriteLn('Время создания файла =',date.hour:3,date.min:3,date.sec:3);
  with date do
    begin  year:=2001;  month:=3;  day:=8    end;
  PackTime(date,time);  {упаковываем дату}
  SetFTime(fi,time);    {меняем дату}
  WriteLn('После изменения даты: ');
  WriteLn('дата создания файла =',date.year:5,date.month:3,date.day:3);
  WriteLn('время создания файла =',date.hour:3,date.min:3,date.sec:3);
  FSplit(name,pathf,namef,extf);  {расщепляем имя файла}
  WriteLn('Полное имя файла =',pathf:25,namef:12,extf:8);
  GetFAttr(fi,k);          {определяем атрибуты файла}
  atr:=lo(k);
  WriteLn(' байт атрибутов ',atr);
  size:=DiskFree(1);
  WriteLn('Свободное место на диске A ',size:10,' байт ');
  Close(fi);
End.
```

Результат работы программы:

```
Введите имя файла:
c:\iva\primer.pas\file\primer.txt
Дата создания файла = 2001 3 7
Время создания файла = 10 47 22
После изменения даты
дата создания файла = 2001 3 8
время создания файла = 10 47 22
Полное имя файла =  c:\iva\primer.pas\file\    primer .txt
байт атрибутов 32
Свободное место на диске A   9104 байт
```

7. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ДИНАМИЧЕСКОЙ ПАМЯТИ

До настоящего момента мы имели дело с переменными, которые размещаются в памяти согласно вполне определенным правилам. Так, память под глобальные переменные программы выделяется в процессе компиляции, и эти переменные существуют в течение всего времени работы программы. Для локальных переменных, описанных в подпрограмме, память отводится при вызове подпрограммы, при выходе из нее эта память освобождается, а сами переменные прекращают свое существование. Иными словами, распределение памяти во всех случаях осуществляется полностью автоматически. Переменные, память под которые выделяется описанным образом, называют *статическими*. Под эту категорию подпадают все переменные, объявленные в области описаний программных блоков. Однако Borland Pascal предоставляет возможность создавать новые переменные во время работы программы, сообразуясь с потребностями решаемой задачи, и уничтожать их, когда надобность в них отпадает.

Переменные, созданием и уничтожением которых может явно управлять программист, называют *динамическими*. Для более полного понимания механизма работы с динамическими переменными следует сначала разобраться в механизме адресации оперативной памяти MS DOS.

7.1. Указатели и операции над ними

Наименьшей адресуемой единицей памяти персонального компьютера, построенного на базе микропроцессоров фирмы Intel и их аналогов, является байт. Таким образом, память представляет собой последовательность нумерованных байтов. Для обращения к конкретному байту необходимо знать его номер, который называют его *физическим адресом*.

Память принято делить на слова, двойные слова и параграфы. Слово имеет длину 2 байта, двойное слово – 4 байта, а параграф – 16 байт.

При работе с памятью используется адресация по схеме «база + смещение» (рис. 7.1). При этом адрес конкретного байта M определяется как адрес некоторого заданного байта $A_б$ (*адрес базы*) + расстояние до требуемого байта $A_{см}$ (*смещение*).

В микропроцессорах фирмы Intel (начиная с i8086) в качестве адреса базы используют адреса, кратные 16. Четыре последних бита такого адреса равны 0, и их не хранят, а аппаратно добавляют при вычислении физического адреса.

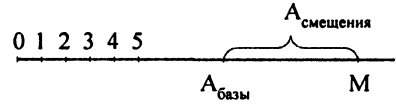


Рис. 7.1. Адресация по схеме «База + смещение»

Непрерывный участок памяти, имеющий длину не более 64 КБ и начинающийся с адреса, кратного 16 (0,16,32,.....), называют *сегментом*. Адрес начала сегмента принимают за базу для всего сегмента. Адрес базы сегмента без последних четырех бит называют *сегментным*.

Сегментный адрес и смещение имеют размер по 16 бит (слово). Физический адрес, получаемый при их сложении с учетом отброшенных четырех бит (рис. 7.2), имеет размер 20 бит и может адресовать память объемом 2^{20} байт или 1 МБ.

Максимальное смещение равно 2^{16}-1, что соответствует 64 КБ памяти. Таким образом, относительно одной базы можно адресовать не более 64 КБ памяти, что ограничивает размер сегмента.

Примечание. Современные модели микропроцессоров используют адреса большей длины с отличающейся схемой получения физического адреса, что учитывается версиями Pascal, предназначенным для работы «под Windows», но принцип адресации по схеме «база+смещение» используется и там.

Программа и данные хранятся в памяти фрагментами, каждый из которых расположен в своем сегменте. Различают три вида сегментов: кодов, данных и стека. В сегментах кодов хранится собственно программа. В сегментах данных размещаются глобальные переменные и константы. Сегмент стека интенсивно используется в процессе выполнения программы: при вызове подпрограмм в стек записывается адрес возврата, в нем размещаются локальные переменные, копии параметров-значений, адреса параметров-переменных и параметров-констант и т.п. (см. фрейм активации в параграфе 5.6).

В процессе работы сегментные адреса хранятся в специальных сегментных регистрах:

CS – адрес базы сегмента кодов;

DS – адрес базы сегмента данных;

SS – адрес базы сегмента стека.

Доступ к конкретным участкам сегмента осуществляется через соответствующие смещения.

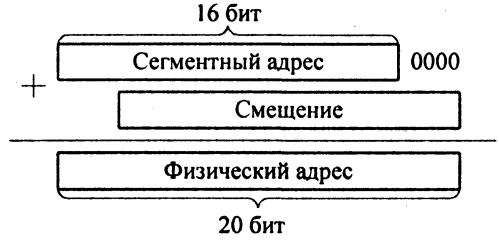


Рис. 7.2. Получение физического адреса

Рис. 7.3. Структура записи адреса в память

При записи адреса в память отдельно сохраняются сегментный адрес и смещение (рис. 7.3).

В Borland Pascal для работы с адресами используется специальный тип данных – *указатель*. Данные этого типа включают два поля типа word и хранят соответственно сегментный адрес и смещение.

Различают указатели двух типов: типизированные и нетипизированные.

Типизированные указатели содержат адреса, по которым в памяти размещаются *данные определенных типов*. Используя эти указатели с данными указанных типов, можно выполнять операции, предусмотренные базовым типом. Синтаксическая диаграмма объявления типизированного указателя приведена на рис. 7.4.

Например:

```
Type tpi=^integer;   {объявляем тип «указатель на целое»}
Var pi:tpi;          {объявляем переменную этого типа}
```

или без предварительного объявления типа:

```
Var pi: ^integer;  {объявляем переменную типа «указатель на целое»}
```

Нетипизированные указатели хранят просто адреса, которые не связаны с данными конкретных типов. Для их объявления используют зарезервированное слово pointer. Например:

```
Var p:pointer; ...
```

Указатели – единственное исключение из общего правила, согласно которому все ресурсы перед использованием должны быть описаны. Для них допускаются описания вида:

```
Type pp = ^percon;        {тип person еще не определен!}
     percon = record      {определение типа person}
                name: string;
                next: pp;
              end; ...
```

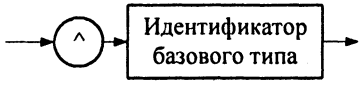


Рис. 7.4. Синтаксическая диаграмма <Объявление типизированного указателя>

Для указателей, которые не хранят никаких адресов, введена константа «нулевой адрес» с именем nil. Константу nil можно присваивать указателю любого типа.

Инициализация указателей. Для объявления инициализированных указателей используют типизированные константы, но

единственное значение, которое может быть присвоено указателю при инициализации – это значение nil. Например:

```
Const p:^real=nil; ...
```

Операции над указателями. Над значениями указателей возможны следующие операции.

Присваивание. При выполнении этой операции указателю присваивается значение другого указателя или nil. Допускается присваивать указателю только значение того же или неопределенного типа.

Например:

```
Var    p1, p2: ^integer;
       p3: ^real;
       p: pointer;
      ...
{допустимые операции}
 p1:=p2;   p:=p3;  p1:=p;  p1:=nil;  p:=nil;
 ...
{недопустимые операции}
 p3:=p2;   p1:=p3; ...
```

Получение адреса. Это унарная операция, которая строится из знака операции – символа @ (коммерческое а) и одного операнда – переменной любого типа. Результат операции – указатель типа pointer, который можно присвоить любому указателю.

Например:

```
Var i:integer;
    pi: ^integer; ...
 pi:=@i;  {указатель pi будет содержать адрес переменной i}
```

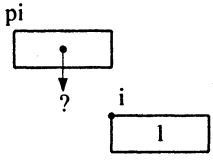
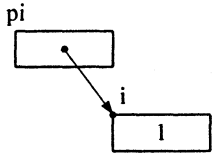
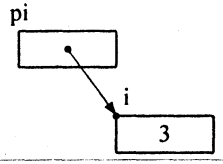
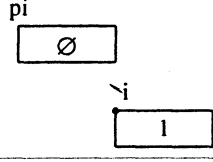
Доступ к данным по указателю (операция разыменования). Чтобы получить доступ к переменной по указателю, необходимо *после* переменной – типизированного указателя поставить знак «^». Полученное значение имеет тип, совпадающий с базовым типом указателя. Нетипизированные указатели разыменовывать нельзя.

Например:

```
j:=pi^; {переменной j присваивается значение целого, расположенного по адресу pi}
pi^:=pi^+2; {целое значение, расположенное по адресу pi, увеличивается на 2}
```

В табл. 7.1 показано, как выполняются операции с указателями.

Таблица 7.1

Фрагмент программы	Результат операции	Описание операции
Const i:integer=1; *Var pi: ^integer;*		Создается инициализированная переменная i и указатель на целое pi
pi:=@i;		Указателю pi присваивается адрес переменной i
pi^:=pi^+2;		Значение, адрес которого находится в pi, увеличивается на 2
pi:=nil;		Запись в pi константы «нулевой адрес»

Операции отношения. Из всех возможных операций отношения допускаются только операции проверки *равенства* (=) и *неравенства* (< >). Эти операции проверяют соответственно равенство и неравенство адресов. Например:

```
sign:=p1=p2; {переменная sign логического типа получает значение
              true или false в зависимости от значений указателей}
```

или

```
if p1<>nil then ... {проверка адреса}
```

Поскольку в качестве базового типа типизированного указателя может быть использован любой тип, допустимо определять «указатель на указатель». Например, если переменную ppi описать и инициализировать следующим образом:

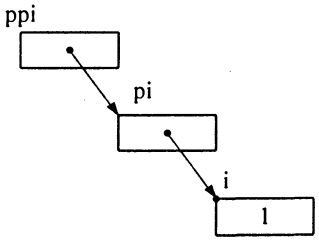


Рис. 7.5. Указатель на указатель

```
Const i:integer=1;
Var pi: ^integer;
    ppi: ^pi;
    ...
    pi:=@i;
    ppi:=@pi; ...
```

то будет реализована схема, изображенная на рис. 7.5.

Для получения значения переменной i необходимо дважды применить операцию разыменования. В нашем случае ppi^^ имеет тип integer и равно 1.

Процедуры и функции, работающие с указателями. Для работы с указателями в Паскале предусмотрены стандартные функции, облегчающие и упрощающие выполнение часто встречающихся операций.

1. Функция ***ADDR(x): pointer*** – возвращает адрес объекта x, в качестве которого может быть указано имя переменной, функции, процедуры. Выполняет те же действия, что и операция «@».

2. Функция ***SEG(x): word*** – возвращает сегментный адрес указанного объекта.

3. Функция ***OFS(x): word*** – возвращает смещение указанного объекта.

4. Функция ***CSEG: word*** – возвращает текущее значение сегментного регистра CS – сегментный адрес сегмента кодов.

5. Функция ***DSEG: word*** – возвращает текущее значение сегментного регистра DS – сегментный адрес сегмента данных.

6. Функция ***PTR(seg,ofs:word):pointer*** – возвращает значение указателя по заданным сегментному адресу seg и смещению ofs.

Преобразование типов данных с использованием типизированных указателей. Как отмечалось ранее, типизированный указатель связывается с некоторым типом данных и адресует вполне определенную область памяти, соответствующую длине внутреннего представления своего типа. Если указателям разного типа присвоить один и тот же адрес, то каждый из них будет рассматривать содержимое области в соответствии с внутренним представлением своего типа. Эта особенность указателей позволяет использовать их для неявного преобразования типа.

Необходимо помнить, что для присвоения разнотипным указателям одного и того же адреса следует использовать нетипизированные указатели, либо задавать абсолютное значение требуемого адреса.

Например:

```
Var L:longint;          {длинное целое число}
    P1:^array[1..4] of byte;  {указатель на область длиной 4 байта}
    k:byte;
```

```
Begin
  L:=123456789;
  P1:=@L; {операция @ возвращает нетипизированный указатель}
  k:=P1^[1];  {младший байт внутреннего представления числа L,
               младший потому, что числа в памяти для данного типа компьютеров хранятся с младшего байта}
```

Контроль корректности значений, полученных в результате выполненных действий, системой не осуществляется, а ложится целиком на программиста.

7.2. Управление динамической памятью

Определяемые в примерах предыдущего параграфа указатели для наглядности содержали адреса статически размещенных переменных. Однако основное назначение указателей – адресация динамических переменных. Такие переменные располагаются в свободной области, называемой *динамической памятью* или «кучей». Эта область расположена после программы, и ее объем составляет около 200 ... 300 кБ, как это представлено на рис. 7.6. (Соответственно, чем больше объем программы, тем меньше размер свободной области памяти.) На этом рисунке также показаны значения стандартных переменных Borland Pascal, используемых для управления динамической областью:

HeapOrg – указатель на начало динамической области;

HeapEnd – указатель на конец динамической области;

HeapPtr – указатель на текущее значение границы свободной динамической области.

Заказать и освободить память требуемого объема из динамической области можно, используя специальные процедуры и функции.

1. Процедура *New (Var* <типизированный указатель>*)* – возвращает адрес выделенного участка памяти через параметр-переменную. Размер участка памяти определяется базовым типом указателя.

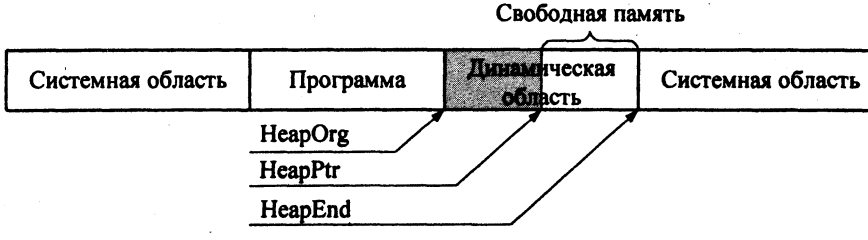


Рис. 7.6. Размещение динамической области

Например:

```
Var pi: ^integer; ...
New(pi); {теперь pi содержит адрес двух байт, выделенных из
          динамической памяти под размещение переменной целого типа}
```

2. Функция ***New (***<тип типизированного указателя>***):pointer*** – возвращает адрес выделенного участка памяти. Размер участка памяти также определяется базовым типом указателя.

Например:

```
Type tpi: ^integer;
Var pi:tpi; ...
pi:= New(tpi); {pi также содержит адрес двух байт, выделенных из
          динамической памяти под размещение переменной целого типа}
```

Для размещения изученных нами типов переменных можно использовать как процедуру New, так и функцию New.

3. Процедура ***Dispose (***<типизированный указатель>***)*** – освобождает память по адресу, хранящемуся в указателе.

Например:

```
Dispose(pi);...
```

При попытке применить процедуру к указателю, имеющему значение nil, или освободить уже освобожденную память фиксируется ошибка и выдается соответствующее сообщение.

Серия последовательных обращений к New и Dispose обычно приводит к *фрагментации памяти* – память разбивается на небольшие фрагменты с чередованием свободных и занятых участков. В результате может возникнуть ситуация: свободной памяти для размещения новой переменной достаточно, но она не может быть размещена из-за отсутствия непрерывного участка требуемого размера.

Для уменьшения явления фрагментации используют специальные процедуры.

4. Процедура ***Mark (Var p:pointer)*** – запоминает значение HeapPtr в указателе p, полученном в качестве параметра.

5. Процедура ***Release (Var p:pointer)*** – освобождает весь фрагмент памяти, начиная с адреса p, зафиксированного в указателе процедуры Mark. Например:

```
  ...
new(p1);
new(p2);
mark(p);
new(p3);
```

```
new(p4);
release(p);...
```

Примечание. Совместное использование процедур Dispose и Release недопустимо, так как Release разрушает список освобожденных фрагментов, создаваемый при выполнении Dispose.

Динамическую память можно выделять фрагментами, указывая их размер.

6. Процедура ***GetMem (Var p:pointer; size:word)*** – запрашивает у системы память размера, указанного в параметре size (запрашиваемый объем не должен превышать 64КБ), и помещает адрес выделенного системой фрагмента в переменную типа pointer с именем p. Как правило, данный способ выделения памяти используется, если требуется память под размещение буферов, формат которых программисту не известен.

7. Функция ***SizeOf(x): word*** – возвращает длину указанного объекта x в байтах.

8. Процедура ***FreeMem (p:pointer; size:word)*** – освобождает область памяти, выделенную процедурой GetMem.

Однако каким бы способом не запрашивалась память, может возникнуть ситуация, когда оставшаяся свободная память меньше требуемой, и память выделить невозможно. В этом случае система по умолчанию выдает сообщение об ошибке выполнения и аварийно завершает задачу.

Избежать подобной ситуации можно несколькими способами.

Первый способ заключается в предварительной проверке наличия свободной памяти требуемого размера.

9. Функция ***Maxavail: longint*** – возвращает длину максимального непрерывного участка памяти.

10. Функция ***Memavail: longint*** – возвращает размер всей свободной памяти – сумму длин всех свободных фрагментов.

Второй способ базируется на возможности перехвата системной обработки ошибки выделения памяти. Для этого необходимо определить свою подпрограмму обработки ошибки, в которой вместо признака ошибки распределения динамической памяти 0, установленного по умолчанию, необходимо задать Heapfunc:=1, например:

```
Function HeapFunc(size: word) : integer; far;
  begin HeapFunc: =1;  end;
```

В программе необходимо определить адрес подпрограммы обработки ошибки HeapError, указав собственную программу HeapFunc:

```
HeapError:=@HeapFunc; ...
```

Использование такой подпрограммы приведет к тому, что процедуры New и GetMem при исчерпании памяти вернут указатели, установленные в

nil, и выполнение программы не будет прервано. Действия по обработке возникшей ситуации выполняет программист, который должен проверить указатели после возврата из процедур выделения памяти.

Пример 7.1. Разработать программу для определения суммы элементов массива большой размерности ($n \le 10000$, $m \le 10000$, $n \times m \le 50000$).

Для размещения массива n×m вещественных чисел (real) потребуется 6×n×m = 300000 байт памяти, что превышает 64 кб, следовательно, использовать стандартный тип «массив» нельзя.

Если создать массив указателей размерности n×m, то потребуется 4×n×m = 200000 байт памяти, что также превышает 64 кб.

Решением задачи является реализация массива в виде статически размещенного массива указателей ptrstr на n элементов (по числу строк), каждый указатель которого хранит адрес динамически размещенного массива – строки матрицы (рис. 7.7).

Тогда одного сегмента достаточно для размещения около 64000/4 = = 16000 указателей на строки матрицы. Так как указатель может адресовать целый сегмент, то в каждой строке можно разместить 64000/6 = 10677 элементов типа real, т.е. даже больше, чем требуется по условию задачи. Однако конкретный размер массива, который можно разместить, определяется размером доступной динамической памяти (кучи). Поэтому в программе осуществляется контроль выделения памяти методом «перехвата» системной ошибки с помощью собственной функции обработки ошибок.

Наибольшую сложность в данной программе представляет определение адреса элемента матрицы с индексами i, j. Этот адрес складывается из сегментного адреса, хранящегося в указателе – Seg(ptrstr[i]), и смещения, состоящего из смещения начала динамического массива, которое хранится в том же указателе – Ofs(ptrstr[i]), и смещения на j-1 элемент внутри динамически размещенного массива – (j-1)×SizeOf(real). Эти вычисления в программе целесообразно реализовать в виде подпрограммы-функции, которая возвращает результат типа ^real.

Рис. 7.7. Размещение матрицы в динамической памяти

```
Program ex_large_mas;
 Const  nn=16000; {максимальное количество строк}
 Var  i,j,n,m:word;  s:real;
      ptrstr:array[1..nn] of pointer; {статический массив указателей на
                                       строки матрицы}
 Type  tpreal=^real;
{функция формирования адреса элемента матрицы}
 Function AddrR(i,j:word):tpreal;
   begin
     AddrR:=Ptr(Seg(ptrstr[i]^),Ofs(ptrstr[i]^)+(j-1)*SizeOf(real))
  end;
{собственная функция обработки ошибок}
 function heapfunc(size:word):integer; far;
    begin heapfunc:=1; end;
{основная программа}
 begin
     Randomize;
     heaperror:=@heapfunc;  {подключаем собственную функцию обра-
                             ботки ошибок}
     WriteLn('Введите n,m');
     ReadLn(n,m);
     for i:=1 to n do
       begin
         GetMem(ptrstr[i],m*sizeof(real));{запрашиваем память под стро-
                                           ку матрицы}
         if ptrstr[i]=nil then        {если памяти не хватает,}
              begin                   {то завершаем программу}
                WriteLn(' Не хватает памяти под матрицу.');
                for j:=1 to i-1 do
                 FreeMem(ptrstr[j],m*sizeof(real)); {освобождаем уже
                                                     выделенную память}
                 Halt(2);
              end;
        for j:=1 to m do AddrR(i,j)^:=Random;  {если память есть, то за-
                         полняем строку случайными числами}
       end;
     s:=0;
     for i:=1 to n do  for j:=1 to m do  s:=s + AddrR(i,j)^;
     WriteLn('Значение суммы =',s:15:10);
     WriteLn('Среднее значение =',s/(n*m):15:10);
     for i:=1 to n do  FreeMem(ptrstr[i],m*SizeOf(real));  {освобождаем
                                  использованную память}
End.
```

7.3. Динамические структуры данных

Как упоминалось выше, переменные типа «указатель» обычно используют при реализации динамических переменных, в том числе и динамических структур данных.

Динамические структуры данных могут быть организованы линейно, в виде дерева и в виде сети.

Линейная динамическая структура представляет собой изменяемую последовательность элементов. Частными случаями таких структур являются:

- *стеки*, в которых разрешено добавлять элементы только в конец и удалять только последние элементы (рис. 7.8, *а*);
- *очереди*, в которых добавление элементов осуществляется в конец, а удаление – из начала (рис. 7.8, *б*);
- *деки*, которые допускают добавление и удаление элементов и с начала, и с конца (рис. 7.8, *в*).

В *древовидной структуре* каждый элемент (вершина) ссылается на один или более элементов следующего уровня (рис. 7.8, *г*).

В *сетевой структуре* никаких ограничений на связи элементов не накладывается (рис. 7.8, *д*).

Линейные динамические структуры, такие, как стеки, очереди и деки, при известном максимальном количестве элементов в них можно реализовать в виде динамических или статических одномерных массивов. В против-

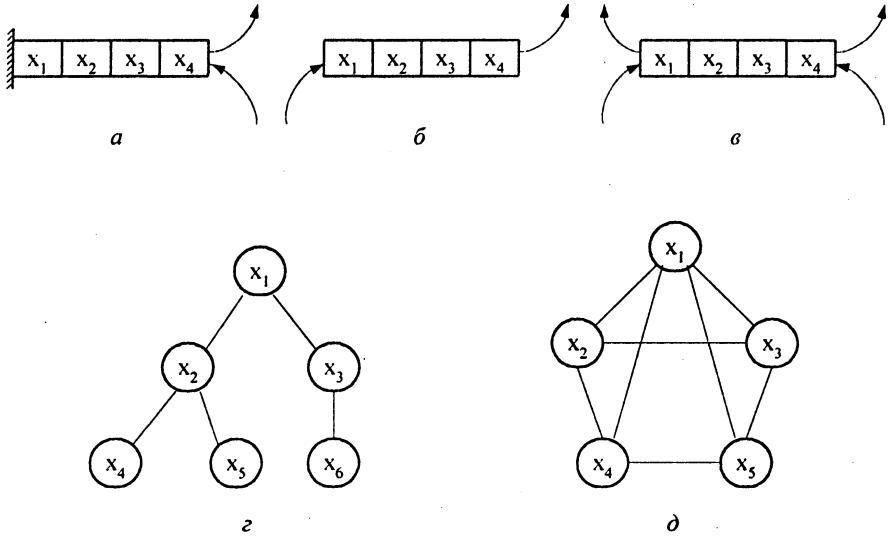


Рис. 7.8. Динамические структуры:
а – стек; *б* – очередь; *в* – дек; *г* – древовидная; *д* – сетевая

ном случае, а также для представления остальных структур (древовидной и сетевой) используют списки.

Списком называют структуру, в которой помимо данных хранятся также адреса элементов. Элемент списка состоит из двух частей: информационной, содержащей данные, и адресной, где хранятся указатели на следующие элементы. В зависимости от количества полей в адресной части и порядка связывания элементов различают:

- *линейные односвязные* списки – единственное адресное поле содержит адрес следующего элемента, если следующий элемент отсутствует, то в адресное поле заносят константу nil (рис. 7.9, *а*);
- *кольцевые односвязные* списки – единственное адресное поле содержит адрес следующего элемента, а последний элемент ссылается на первый (рис. 7.9, *б*);

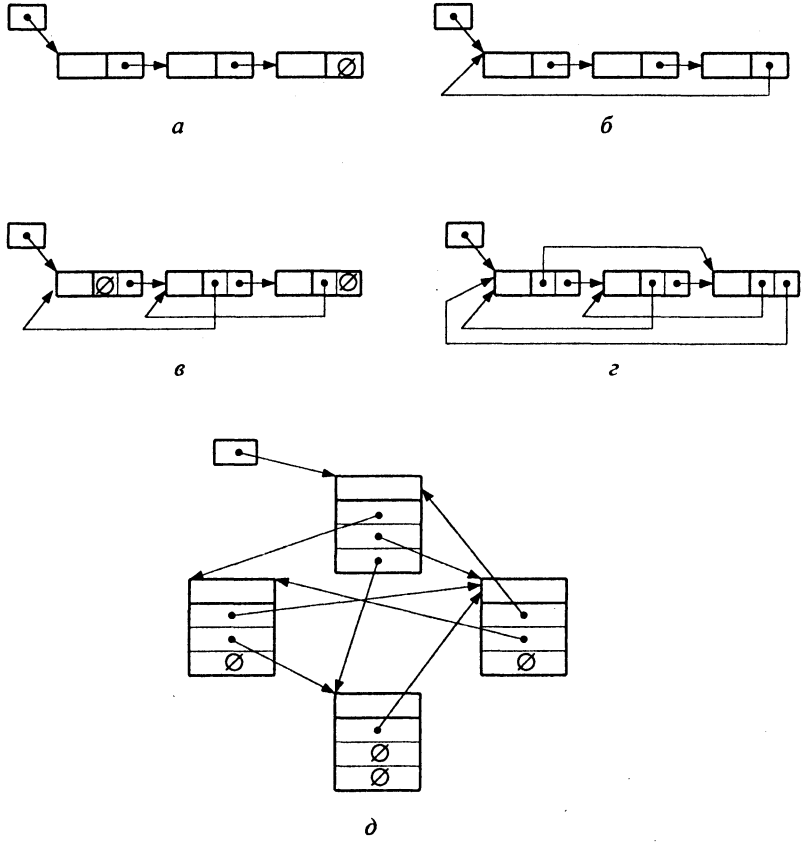


Рис. 7.9. Списки:

а – линейный односвязный; *б* – линейный односвязный кольцевой;
в – линейный двусвязный; *г* – двусвязный кольцевой; *д* – n-связный

• *линейные двусвязные* списки – каждый элемент содержит адреса предыдущего и последующих элементов, соответственно, первый элемент в качестве адреса предыдущего, а последний – в качестве адреса следующего элемента содержат nil (рис. 7.9, *в*);

• *кольцевые двусвязные* списки – каждый элемент содержит адреса предыдущего и последующих элементов, причем первый элемент в качестве предыдущего содержит адрес последнего элемента, а последний элемент в качестве следующего – адрес первого элемента (рис. 7.9, *г*);

• *n-связные списки* – каждый элемент включает несколько адресных полей, в которых записаны адреса других элементов или nil (рис. 7.9, *д*).

Для описания элементов списка используют записи, например, элемент односвязного списка с двумя информационными и одним адресным полями может быть описан следующим образом:

```
Type pe = ^element;        {тип указателя}
    element = record
                name: string[16];  {информационное поле 1}
                telefon:string[7]; {информационное поле 2}
                p: pe;              {адресное поле}
            end;
```

Элемент двусвязного списка описывается с двумя адресными полями, например:

```
Type pe = ^element;        {тип указателя}
    element = record
                name: string[16];  {информационное поле 1}
                telefon:string[7]; {информационное поле 2}
                prev: pe;          {адресное поле «предыдущий»}
                next: pe;          {адресное поле «следующий»}
              end;
```

Соответственно элемент n-связного списка содержит заданное количество адресных полей.

У любого списка имеется хотя бы один указатель, размещенный в статической памяти, который содержит адрес первого элемента списка или константу nil, если список пуст. Достаточно часто используют дополнительные указатели, в которых хранят адреса других элементов, например, адрес текущего элемента, адрес последнего элемента и т.п. Эти указатели также описываются как «указатели на элемент», например:

```
Var first, last, q: pe; ...
```

В процессе выполнения программы динамические структуры создаются, обрабатываются и уничтожаются.

Рассмотрим некоторые приемы работы со списками на примере линейных односвязных списков и бинарных деревьев.

7.4. Линейные односвязные списки

Линейные односвязные списки используют чаще других списковых структур, так как они сравнительно просты, но одновременно в отличие от одномерных массивов позволяют:

- работать с произвольным количеством элементов, добавляя и удаляя их по мере необходимости;
- осуществлять вставку и удаление записей, не перемещая остальных элементов последовательности.

Недостатком этой структуры является то, что при поиске элемента по номеру приходится просматривать все ранее расположенные элементы, в то время как в одномерном массиве возможен прямой доступ к элементу по индексу. К тому же реализация линейного односвязного списка требует дополнительной памяти для хранения адресной части элементов.

Рассмотрим более подробно, как выполняются основные операции с линейными односвязными списками.

Исходные установки. В начале программы необходимо описать элемент и его тип:

```
Type tpel=^element;   {тип «указатель на элемент»}
     element=record
                num:integer;  {число}
                p:tpel;       {указатель на следующий элемент}
             end;
```

В статической памяти описываем переменную-указатель списка и несколько переменных-указателей, используемых при выполнении операций со списком:

```
Var first,     {указатель списка – адрес первого элемента списка}
    n,f,q:tpel;  {вспомогательные указатели}
```

Исходное состояние «список пуст»:

```
first:=nil;
```

Добавление нового элемента к списку. Добавление элемента к списку включает запрос памяти для размещения элемента и заполнение его информационной части. Построенный таким образом элемент добавляется к уже существующей части списка.

В общем случае при добавлении элемента к списку возможны следующие варианты:

- список пуст, добавляемый элемент станет единственным элементом списка;
- элемент необходимо вставить перед первым элементом списка;
- элемент необходимо вставить перед заданным (не первым) элементом списка;
- элемент необходимо дописать в конец списка.

Добавление элемента к пустому списку состоит из записи адреса элемента в указатель списка, причем в поле «адрес следующего» добавляемого элемента необходимо поместить nil:

```
new(first);          {запрашиваем память под элемент}
first ^.num:=5;      {заносим число в информационное поле}
first ^.p:=nil;      {записываем nil в поле «адрес следующего»}
```

На рис. 7.10 показана последовательность операций при добавлении элемента к пустому списку.

Добавление элемента перед первым элементом списка. При выполнении этой операции необходимо в поле «адрес следующего» переписать адрес первого элемента списка, а в указатель списка занести адрес добавляемого элемента (рис. 7.11):

```
new(q);          {запрашиваем память под элемент}
q^.num:=4;       {заносим число в информационное поле}
q^.p:=first;     {в поле «адрес следующего» переписываем адрес
                                          первого элемента}
first:=q; ...    {в указатель списка заносим адрес нового элемента}
```

Добавление элемента перед заданным (не первым). Для выполнения операции необходимо знать адреса элементов, между которыми вставляется элемент, так как адресные части этих элементов при выполнении операции

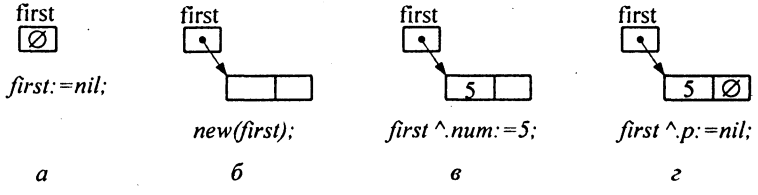


Рис. 7.10. Последовательность операций при добавлении элемента к пустому списку:

а – исходное состояние; *б* – запрос памяти под элемент;

в – заполнение элемента; *г* – занесение nil в поле адреса следующего элемента

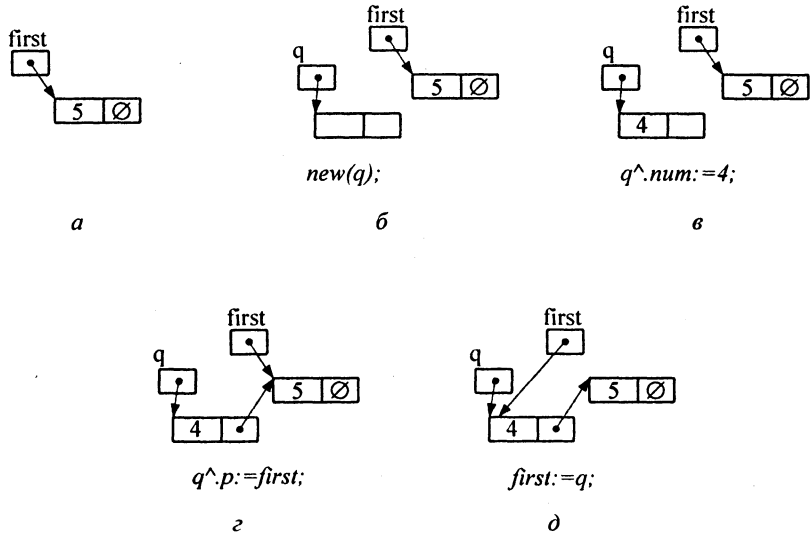


Рис. 7.11. Последовательность операций при добавлении элемента перед первым:

а – исходное состояние; *б* – запрос памяти под элемент; *в* – заполнение элемента; *г, д* – шаги включения элемента в список

будут корректироваться (рис. 7.12). Пусть f – адрес предыдущего элемента, а n – адрес следующего элемента, тогда:

```
new(q);        {запрашиваем память под элемент}
q^.num:=3;     {заносим число в информационное поле}
q^.p:=n;       {в поле «адрес следующего» нового элемента переписываем адрес следующего элемента}
f^.p:=q; ...   {в поле «адрес следующего» предыдущего элемента заносим адрес нового элемента}
```

Минимально для вставки элемента в линейный односвязный список необходимо знать только адрес предыдущего элемента, так как тогда адрес следующего элемента известен – $n \Leftrightarrow f^{\wedge}.p$:

```
new(q);        {запрашиваем память под элемент}
q^.num:=3;     {заносим число в информационное поле}
q^.p:=f^.p;    {в поле «адрес следующего» нового элемента переписываем адрес следующего элемента}
f^.p:=q; ...   {в поле «адрес следующего» предыдущего элемента заносим адрес нового элемента}
```

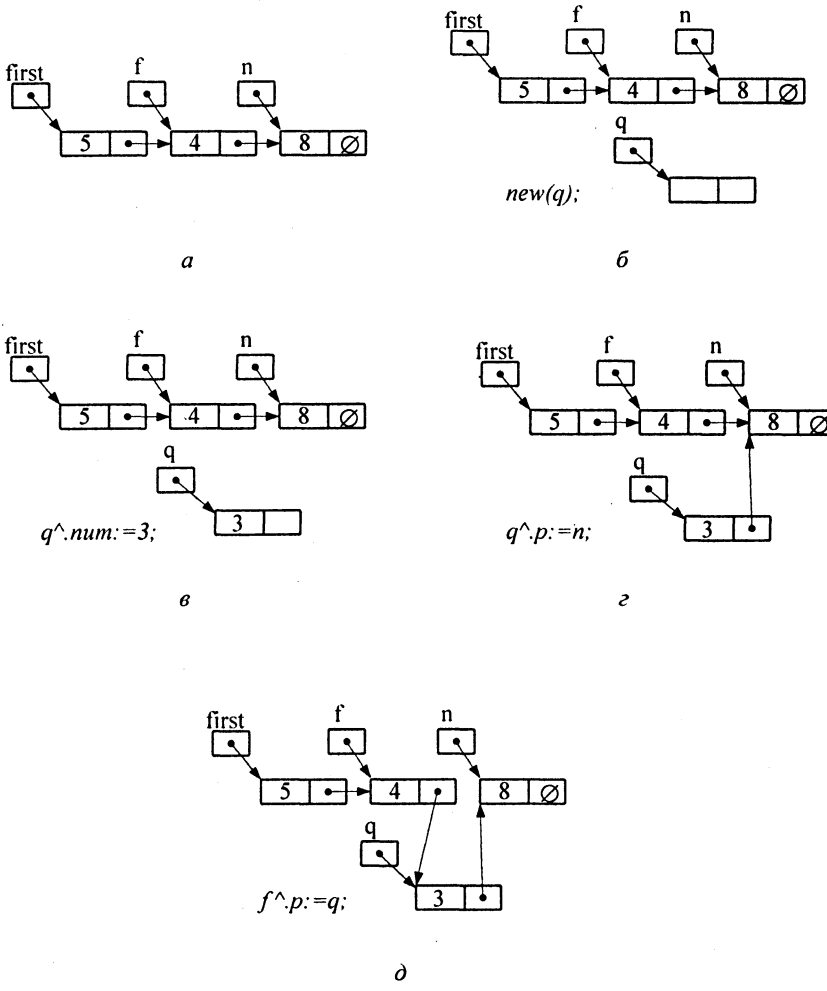


Рис. 7.12. Добавление элемента перед заданным (не первым):
а – исходное состояние; *б* – запрос памяти под элемент;
в – заполнение элемента; *г, д* – шаги включения элемента в список

Добавление элемента в конец списка. В этом случае должен быть известен адрес элемента, после которого добавляется новый элемент (рис. 7.13):

```
new(q);          {запрашиваем память под элемент}
q^.num:=7;  {заносим число в информационное поле}
q^.p:=nil;       {в поле «адрес следующего» элемента записываем nil}
f^.p:=q; ...  {в поле «адрес следующего» предыдущего элемента заносим адрес нового элемента}
```

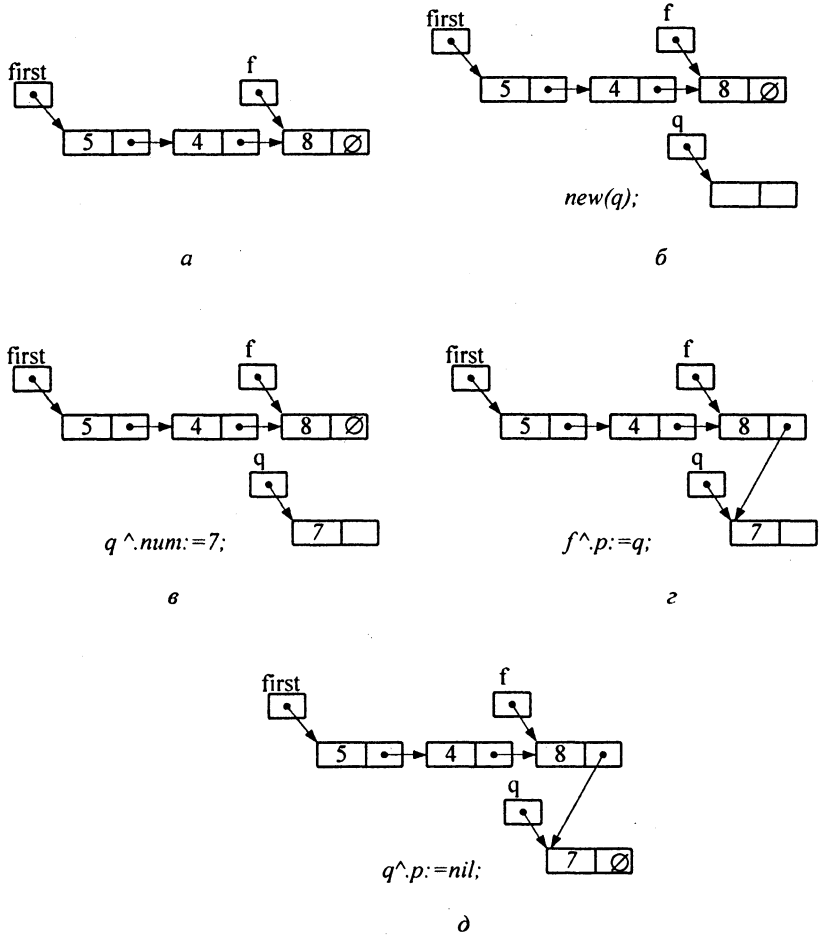


Рис. 7.13. Добавление элемента в конец списка:
а – исходное состояние; *б* – запрос памяти под элемент;
в – заполнение элемента; *г*, *д* – включение элемента в список

Анализ показывает, что этот случай можно свести к предыдущему, так как адресная часть последнего элемента содержит nil:

```
new(q);          {запрашиваем память под элемент}
q^.num:=7;  {заносим число в информационное поле}
q^.p:=f^.p;  {в поле «адрес следующего» нового элемента записываем nil}
f^.p:=q; ...  {в поле «адрес следующего» предыдущего элемента заносим адрес нового элемента}
```

Комбинируя эти фрагменты, можно организовать построение списка с любой дисциплиной добавления элементов.

Пример 7.2. Разработать программу, которая строит список по типу стека из целых чисел, вводимых с клавиатуры. Количество чисел не известно, но отлично от нуля. Конец ввода – по комбинации клавиш CTRL-Z (конец файла на устройстве ввода).

Обычно построение списка по типу стека выполняется в два этапа: в список заносится первый элемент, а затем организуется цикл добавления элементов перед первым:

```
ReadLn(a);
new(first);          {запрашиваем память под элемент}
first ^.num:=a;      {заносим число в информационное поле}
first ^.p:=nil;      {записываем nil в поле «адрес следующего»}
while not EOF do
  begin
    ReadLn(a);
    new(q);          {запрашиваем память под элемент}
    q^.num:=a;       {заносим число в информационное поле}
    q^.p:=first;     {в поле «адрес следующего» переписываем адрес
                      первого элемента}
    first:=q;        {в указатель списка заносим адрес нового элемента}
  end; ...
```

Пример 7.3. Разработать программу, которая строит список по типу очереди из целых чисел, вводимых с клавиатуры. Количество чисел не известно, но отлично от нуля. Конец ввода – по комбинации CTRL-Z (конец файла на устройстве ввода).

При построении списка по типу очереди сначала мы заносим в стек первый элемент, а затем организуем цикл добавления элементов после последнего, приняв во внимание, что nil необходимо разместить в адресном поле только последнего элемента:

```
ReadLn(a);
new(first);          {запрашиваем память под элемент}
first ^.num:=a;      {заносим число в информационное поле}
f:=first;            {f – текущий элемент, после которого добавляется
                      следующий}
while not EOF do
  begin
    ReadLn(a);
    new(q);          {запрашиваем память под элемент}
    q^.num:=a;       {заносим число в информационное поле}
```

```
        f ^.p:=q;   {в поле «адрес следующего» предыдущего элемента
                        заносим адрес нового элемента}
        f:=f ^.p;   {теперь новый элемент стал последним}
    end;
q^.p:=nil;      {в поле «адрес следующего» последнего элемента запи-
                    сываем nil}
```

Этот фрагмент можно упростить, если заметить, что новый элемент к списку можно добавлять сразу при запросе памяти:

```
ReadLn(a);
new(first);          {запрашиваем память под элемент}
first ^.num:=a;  {заносим число в информационное поле}
f:=first;            {f – текущий элемент, после которого добавляется
                        следующий}
while not EOF do
  begin
    ReadLn(a);
    new(f ^.p);  {запрашиваем память под элемент}
    f:=f ^.p;    {теперь новый элемент стал последним}
    f^.num:=a;   {заносим число в информационное поле}
  end;
f^.p:=nil; ...   {в поле «адрес следующего» последнего элемента за-
                    писываем nil}
```

Пример 7.4. Разработать программу, которая строит список, сортированный по возрастанию элементов, из целых чисел, вводимых с клавиатуры. Количество чисел не известно, но отлично от нуля. Конец ввода – по комбинации CTRL-Z (конец файла на устройстве ввода).

Список сортирован, соответственно, добавляя элемент, необходимо сохранить закон сортировки: элемент должен вставляться перед первым элементом, который больше, чем добавляемый. В зависимости от конкретных данных он может вставляться в начало, середину и конец списка.

```
new(first);          {запрашиваем память под первый элемент}
ReadLn(first ^.num);  {заносим число в информационное поле}
first^.p:=nil;
while not EOF do
  begin
    new(q);              {создаем элемент}
    ReadLn(q^.num);  {заносим значение}
    if q^.num<first^.num then {если элемент меньше первого
                                  элемента списка, то}
```

```
        begin              {вставляем перед первым}
          q^.p:=first;
          first:=q;
        end
      else                 {иначе вставляем в середину или конец}
        begin
          n:=first;        {указатель на текущий элемент}
          f:=first;        {указатель на предыдущий элемент}
          flag:=false;       {"элемент не вставлен"}
          {цикл поиска места вставки}
          while (n^.p<>nil) and (not flag) do
              begin
                n:=n^.p;  {переходим к следующему элементу}
                if q^.num<n^.num  then   {место найдено}
                   begin        {вставляем в середину}
                      q^.p:=f^.p;
                      f^.p:=q;
                      flag:=true; {"элемент вставлен"}
                   end
                else f:=n;       {сохраняем адрес текущего элемента}
              end;
            if not flag then   {если элемент не вставлен, то}
              begin            {вставляем после последнего}
                 q^.p:=nil;
                 f^.p:=q;
              end;
        end;
  end;
```

Просмотр и обработка элементов списка. Просмотр и обработка элементов списка выполняется последовательно с использованием дополнительного указателя:

```
f:=first;
while f<>nil do
   begin
     <обработка элемента по адресу f>
     f:=f ^.p;
   end;   ...
```

В качестве примера рассмотрим вывод на экран элементов списка:

```
f:=first;
while f<>nil do
    begin
      WriteLn(f^.num,' ');
      f:=f^.p;
    end;   ...
```

Поиск элемента в списке. Поиск элементов в списке также выполняется последовательно, но при этом, как это и положено в поисковом цикле, обычно организуют выход из цикла, если нужный элемент найден, и осуществляют проверку после цикла, был ли найден элемент:

```
f:=first;
flag:=false;
while (f<>nil) and (not flag) do
   begin
     if f^.num=k then flag:=not flag
                 else f:=f^.p;
   end;
 if flag then <элемент найден>
         else <элемент не найден>; ...
```

Удаление элемента из списка. При выполнении операции удаления также возможно четыре случая:

- удаление единственного элемента;
- удаление первого (не единственного) элемента списка;
- удаление элемента, следующего за данным;
- удаление последнего элемента.

Удаление единственного элемента. После удаления единственного элемента список становится пустым, следовательно при выполнении этой операции необходимо не только освободить память, выделенную для размещения элемента, но и занести nil в указатель списка first (рис. 7.14):

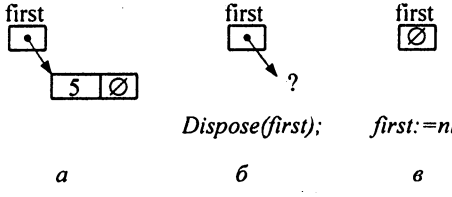


Рис. 7.14. Удаление единственного элемента списка:
а – исходное состояние; *б* – освобождение памяти; *в* – занесение константы nil в указатель списка

```
Dispose(first);
first:=nil; ...
```

Удаление первого (не единственного) элемента списка. Удаление первого элемента состоит из сохранения адреса следующего элемента в рабочей переменной f, освобождения памяти элемента и записи в указатель списка сохраненного адреса следующего элемента (рис. 7.15):

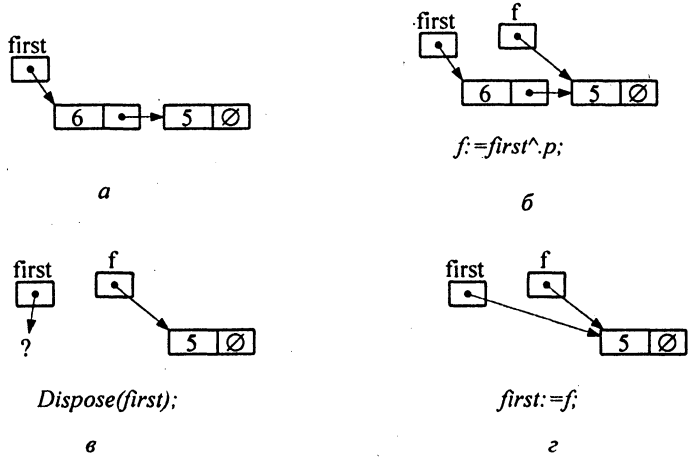


Рис. 7.15. Удаление первого (не единственного) элемента списка:

а – исходное состояние; *б* – сохранение адреса следующего элемента в специальном указателе; *в* – освобождение памяти; *г* – запись в указатель списка адреса следующего элемента

```
f:=first^.p;      {сохраняем адрес следующего элемента}
Dispose(first); {освобождаем память}
first:=f; ...     {заносим в указатель списка адрес следующего
                                                  элемента}
```

Удаление единственного элемента и первого элемента в программе можно объединить:

```
q:=first;
first:=first^.p;
Dispose(q); ...
```

Удаление элемента, следующего за данным (не последнего). Удаление элемента, следующего за данным, требует запоминания адреса удаляемого элемента, изменения адресной части данного элемента и освобождения памяти (рис. 7.16).

```
n:=f^.p;          {сохраняем адрес удаляемого элемента}
f^.p:=n^.p;       {заносим в адресное поле предыдущего элемента ад-
                     рес следующего элемента}
Dispose(n); ... {освобождаем память}
```

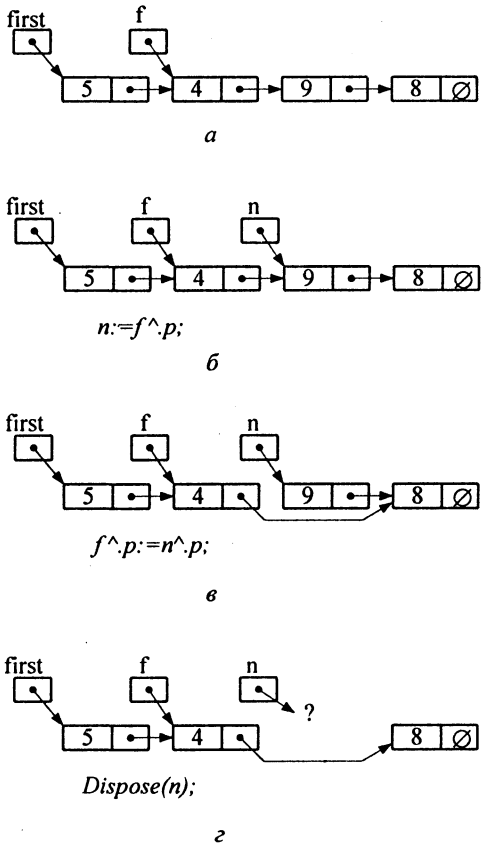


Рис. 7.16. Удаление элемента, следующего за данным (не последнего):
а – исходное состояние; *б* – сохранение адреса удаляемого элемента; *в* – исключение удаляемого элемента из списка; *г* – освобождение памяти

Удаление последнего элемента. Удаление последнего элемента отличается только тем, что в поле «адрес следующего» заданного элемента записывается константа nil:

```
n:=f^.p;
f^.p:=nil;
Dispose(n); ...
```

Удаление последнего элемента можно свести к удалению элемента, следующего за данным, так как адресная часть удаляемого элемента равна nil.

Комбинируя приемы удаления, мы также можем организовать любую дисциплину удаления.

Пример 7.5. Разработать программу, которая удаляет из списка все элементы меньше заданного значения k.

Удаляемые значения могут располагаться в списке на любом месте, следовательно, возможны все четыре варианта удаления элемента, которые сводятся к двум случаям:

- удаление единственного элемента и удаление записей из начала списка – удаление из начала списка;
- удаление средних и последнего элементов – удаление не из начала списка.

Для разделения этих двух случаев введем специальный признак «удаление из начала», который в начале установим равным true, а затем, как только в списке будет оставлен хотя бы один элемент – изменим на false.

```
n:=first;
nft:=true;     {признак «удаление из начала списка»}
repeat
    if n^.num<k then
      begin
```

```
      if nft then   {если «удаление из начала списка»}
        begin   {удаление из начала списка}
          q:=first;
          first:=first^.p;
          Dispose(q);
          n:=first;   {переходим к следующему элементу}
        end
      else      {иначе}
        begin     {удаление из середины и конца}
          q:=n;
          n:=n^.p; {переходим к следующему элементу}
          Dispose(q);
          f^.p:=n;
        end
    end
  else  {оставляем элемент в списке}
    begin
      f:=n;          {устанавливаем адрес предыдущего элемента}
      n:=n^.p;       {переходим к следующему элементу}
      nft:=not nft   {«удаление не из начала списка»}
    end;
until n=nil; ...     {до завершения списка}
```

Задания для самопроверки

Задание 1. Разработайте программу, которая вводит с клавиатуры последовательность чисел до символа «#», а затем удаляет из нее все числа, превышающие среднее арифметическое чисел введенной последовательности. Оставшиеся значения выведите в обратном порядке.

Задание 2. Разработайте программу, которая вводит с клавиатуры последовательность чисел до символа «#», а затем определяет следующие суммы:

$$x_1 + x_n;\ \ x_2 + x_{n-1};\ \ x_3 + x_{n-2};\ \ \ldots x_n + x_1.$$

Указание. Используйте двусвязный список.

Задание 3. Разработайте программу, которая определяет «водящего» в детской игре. Водящий определяется с помощью «считалки» следующим образом. Все играющие встают в круг и начинают «считаться». Каждый раз тот, на ком закончилась считалка, выбывает из круга. Водит оставшийся. Исходное количество играющих n. Количество слов считалки m.

Указание. Используйте кольцевой список.

7.5. Бинарные деревья

В математике *бинарным* (двоичным) *деревом* называют конечное множество вершин, которое либо пусто, либо состоит из корня и не более чем двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня.

Таким образом, каждая вершина бинарного дерева может включать одно или два поддерева или не включать поддеревьев вовсе. Первое поддерево обычно называют левым, а второе – правым. Соответственно ветвь, исходящую из вершины и ведущую в корень левого поддерева, называют *левой*, а ветвь, ведущую в корень правого поддерева – *правой*. Вершины, из которых не выходит ни одной ветви, называют *листьями* (рис. 7.17).

В программах для реализации бинарных деревьев используют n-связные списки. С вершинами бинарного дерева обычно связывают записи, хранящие некоторую информацию.

Построение дерева выполняется следующим образом. Если дерево пусто, то первая же вершина становится корнем дерева. Добавление остальных вершин регламентируется в зависимости от условия задачи: в соответствии с заданной дисциплиной построения дерева отыскивается подходящая вершина, к которой и подсоединяется новая вершина.

Достаточно часто используют регулярные бинарные деревья с разными законами построения. Примером могут служить сортированные бинарные деревья, построение которых осуществляется по правилу: *ключевое поле левого поддерева всегда должно содержать значение меньше, чем в корне, а ключевое поле правого поддерева – значение больше или равное значению в корне*.

Рассмотрим основные операции с сортированными бинарными деревьями.

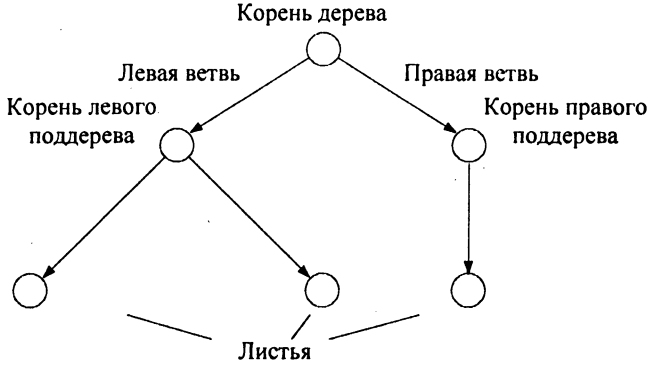


Рис. 7.17. Пример бинарного дерева

Исходные установки. В начале программы необходимо описать элемент и его тип:

```
Type top_ptr=^top;   {тип «указатель на вершину»}
     top=record
           value:integer;  {число}
           left,           {указатель на левое поддерево}
           right:top_ptr;  {указатель на правое поддерево}
         end;
```

В статической памяти описываем указатель корня дерева и несколько указателей, используемых при выполнении операций со списком:

```
Var root,     {указатель структуры – адрес корня дерева}
    pass, next, q:top_ptr;  {вспомогательные указатели}
```

Исходное состояние – «пустое дерево»:

```
root:=nil;
```

Построение дерева. Дерево строится в соответствии с главным правилом. Например, пусть дана последовательность целых чисел {5, 2, 8, 7, 2, 9, 1, 5}. Первое число 5 будет записано в корень дерева (рис. 7.18, *а*). Второе число 2 меньше значения в корне дерева, следовательно, оно будет записано в левое поддерево (рис. 7.18, *б*). Следующее число 8 больше значения в корне, соответственно оно будет записано в правое поддерево (рис. 7.18, *в*). Следующее число 7 больше, чем значение в корне дерева, значит, оно должно быть записано в правое поддерево, но правое поддерево уже построено. Сравниваем 7 со значением в корне правого поддерева – числом 8. Так как добавляемое значение меньше значения в корне правого поддерева, то добав-

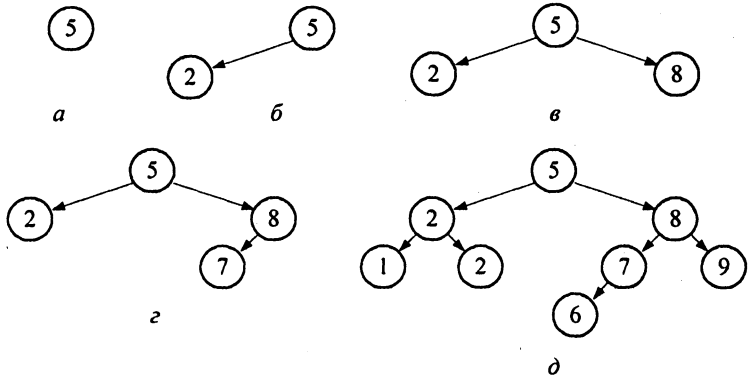


Рис. 7.18. Построение сортированного бинарного дерева:
первые шаги (*а* – *г*) и окончательный вариант (*д*)

ляем левое поддерево уже к этому корню (рис. 7.18, *г*). Полностью сформированное бинарное дерево представлено на рис. 7.18, *д*.

Фрагмент программы, реализующий добавление вершины к дереву, состоит из трех частей: создания вершины, поиска корня, к которому можно добавить поддерево, придерживаясь основного правила, и, непосредственно, добавления вершины:

```
{создание новой вершины}
new(q);   {выделяем память для нового элемента}
   with q^ do  {заносим значения}
        begin value:=n;
              left:=nil;
             right:=nil;
        end;
{поиск корня для добавляемой вершины}
pass:=root;              {начинаем с корня бинарного дерева}
 while pass<>nil do   {пока не найдено свободное место}
    begin next:=pass;   {сохраняем адрес корня-кандидата}
          if q^.value<pass^.value then pass:=pass^.left    {влево}
                                   else pass:=pass^.right; {вправо}
    end;
 {добавление вершины}
    if q^.value<next^.value then     {если значение меньше корня}
                 next^.left:=q         {добавляем левое поддерево}
          else next^.right:=q;      {добавляем правое поддерево}
```

Используя рекурсивность определения дерева, можно построить рекурсивную процедуру добавления вершин к дереву. В качестве параметров эта процедура будет получать указатель на корень дерева и указатель на добавляемый элемент.

```
Procedure Add(Var r:top_ptr; pass:top_ptr);
  begin
     if r=nil then r:=pass  {если место свободно, то добавляем}
     else    {иначе идем налево или направо}
        if (pass^.value<r^.value) then Add(r^.left,pass)
                                  else Add(r^.right,pass);
  end; ...
```

Поиск вершины в сортированном бинарном дереве. Поиск в сортированном бинарном дереве осуществляется следующим образом: вначале значение ключа поиска сравнивается со значением в корне. Если значение ключа в искомой вершине меньше, чем в корневой, то поиск переходит в левую ветвь. Если больше или равно – то в правую ветвь. И так в каждой следующей вершине до тех пор, пока не отыщется искомая. Так, для предыду-

щего варианта последовательности поиск вершины с ключом 7 будет выполнен за три шага (рис. 7.19).

Если мы добрались до листа, а искомая вершина не обнаружена, то следует выдать соответствующее сообщение. В противном случае вершину помечают как найденную (запоминают ее адрес) и обрабатывают в соответствии с алгоритмом программы:

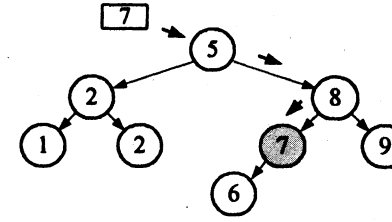


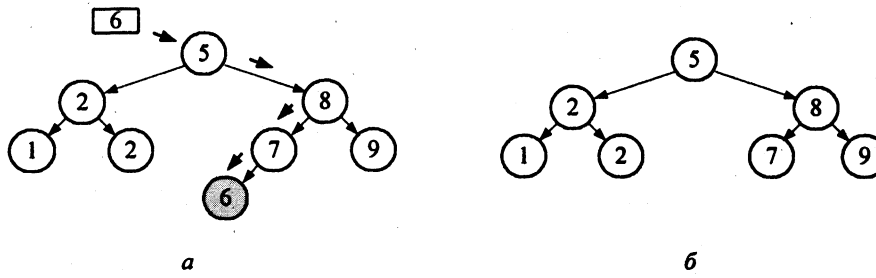
Рис. 7.19. Пример поиска в бинарном дереве

```
pass:=root;              {начинаем с корня бинарного дерева}
flag:=false;             {признак «вершина не найдена»}
   while (pass<>nil) and not flag do {пока не найден элемент или не дошли до листа}
    if n=pass^.value then flag:=true {значение найдено}
    else
     if n<pass^.value then pass:=pass^.left    {влево}
                      else pass:=pass^.right; {вправо}
if flag then <вершина найдена>
       else <вершина не найдена> ...
```

Поиск вершины также можно осуществлять, используя рекурсию. Для удобства использования построим рекурсивную функцию, которая будет возвращать true, если элемент найден, и false – в противном случае. Адрес найденного элемента будем возвращать через параметр pass:

```
Function Find(r:top_ptr; Var pass:top_ptr; n:integer):boolean;
Begin
   if r=nil then Find:=false   {значение не найдено}
   else
       if n=r^.value then
           begin
              Find:=true;  {значение найдено}
              pass:=r;     {запомнили адрес}
           end
       else
          if n<r^.value then
                        Find:=Find(r^.left,n)    {влево}
          else Find:=Find(r^.right,n);   {вправо}
End;
```

Вызывать такую функцию можно непосредственно из условия оператора условной передачи управления или циклов, например:



а *б*

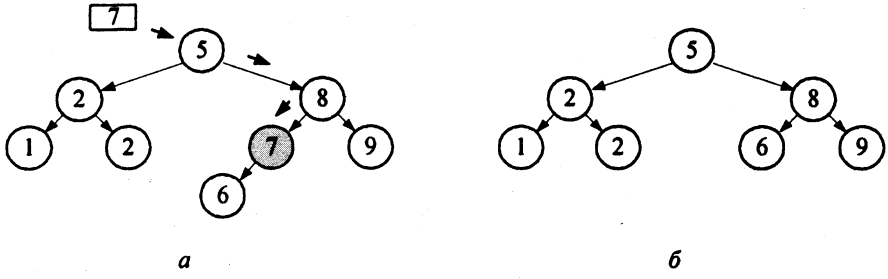
Рис. 7.20. Удаление листа:
а – поиск удаляемой вершины; *б* – удаление вершины

```
if Find(r,pass,n) then   <вершина найдена, адрес в pass>
     else                <вершина не найдена> ...
```

Удаление вершины с указанным ключом. Удалению вершины с указанным ключом предшествует ее поиск (см. выше). Непосредственное удаление вершины реализуется в зависимости от того, какая вершина удаляется:

- удаляемая вершина не содержит поддеревьев (лист) – удаляем ссылку на вершину из корня соответствующего поддерева (рис. 7.20);
- удаляемая вершина содержит одну ветвь (рис. 7.21, *а*): для удаления необходимо скорректировать соответствующую ссылку в корне, заменив адрес удаляемой вершины адресом вершины, из нее выходящей (рис. 7.21, *б*);
- удаляемая вершина содержит две ветви (рис. 7.22, *а*): в этом случае нужно найти подходящую вершину, которую можно вставить на место удаляемой, причем эта подходящая вершина должна легко перемещаться. Такая вершина всегда существует: это либо *самый правый* элемент *левого* поддерева, либо *самый левый* элемент *правого* поддерева удаляемой вершины (рис. 7.22, *б*).

Ниже представлена рекурсивная процедура удаления вершины с указанным значением (параметры: r – адрес корня дерева, k – значение).



а *б*

Рис. 7.21. Удаление корня с одним поддеревом:
а – поиск удаляемой вершины; *б* – удаление вершины

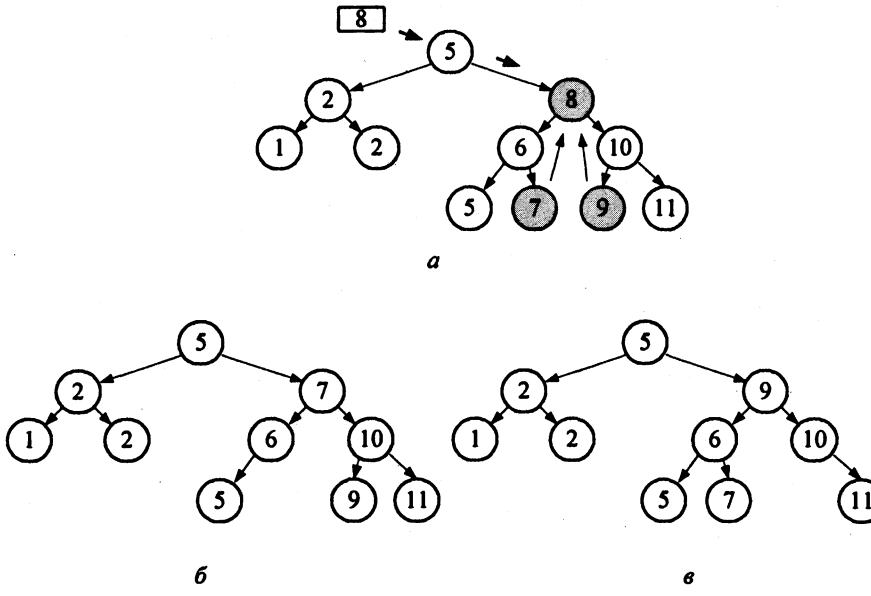


Рис. 7.22. Удаление корня с двумя поддеревьями:

а – поиск удаляемой вершины и вершин-кандидатов на замещение;

б – замена вершины самой правой вершиной левого поддерева;

в – замена вершины самой левой вершиной правого поддерева

```
Procedure Delete(var r:top_ptr;k:integer);
  {Внутренняя рекурсивная процедура поиска заменяющей
   вершины в левом  поддереве. Параметры: r – адрес корня
   левого поддерева, q – адрес заменяемой вершины}
   Procedure del(var r:top_ptr; q:top_ptr);
   Var q1:top_ptr;
    begin
      if r^.right=nil then      {заменяющая вершина найдена}
        begin
          q^.value:=r^.value; {копируем значение}
          q1:=r;
          r:=r^.left;           {удаляем заменяющую вершину}
          Dispose(q1);          {освобождаем память}
        end
      else  del(r^.right,q)     {идем по поддереву направо}
     End;
Var q:top_ptr;
begin
 if r=nil  then WriteLn('Элемент не найден или дерево пустое')
 else      {поиск элемента с заданным ключом}
```

```
if k<r^.value then        {если меньше, то налево}
                  Delete(r^.left,k)
else
 if k>r^.value then       {если больше, то направо}
                  Delete(r^.right,k)
 else
  begin {элемент найден, его необходимо удалить}
   {удаление листа или корня с одним поддеревом}
   if r^.right=nil then {нет правого поддерева}
        begin
          q:=r;
          r:=r^.left;
          Dispose(q);
        end
   else
     if r^.left=nil then {нет левого поддерева}
          begin
              q:=r;
              r:=r^.right;
              Dispose(q);
          end
     else {удаление корня с двумя поддеревьями}
          del(r^.left,r);
  end
End; ...
```

Сортировка с использованием дерева. Так как дерево формируется по определенным выше правилам, то сортировка по возрастанию осуществляется обходом дерева «слева направо». Обход начинается с самого нижнего левого листа или, если такого листа нет, корня. Вывод значений осуществляется в следующем порядке: сначала выводится значение самого нижнего левого поддерева, затем корня, затем самого нижнего левого поддерева правого поддерева и т.д. (рис. 7.23).

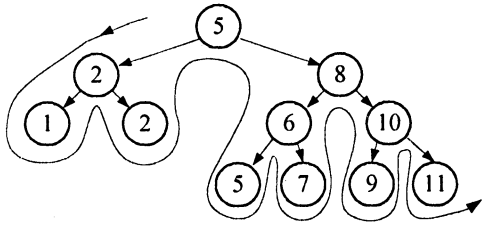


Рис. 7.23. Обход дерева «слева направо»

Пример 7.6. Разработать программу сортировки заданной последовательности целых чисел с использованием сортированного бинарного дерева.

Программа должна строить бинарное дерево из вводимых с клавиатуры целых чисел, а затем осуществлять обход дерева для вывода отсортированных данных.

Построение дерева реализуем отдельной подпрограммой, которая будет получать адрес добавляемой вершины и адрес корня бинарного дерева. Поиск корня для добавляемой вершины, как показано выше, будем осуществлять рекурсивно.

Обход дерева «слева направо» также будем осуществлять рекурсивно. Нерекурсивный вариант достаточно сложен, и его использование нецелесообразно.

Полностью текст программы приведен ниже.

```
Program Sort4;
Type top_ptr=^top;  {тип "указатель на вершину дерева"}
   top=record              {тип вершины дерева}
       value:integer;      {целое число}
       left, right:top_ptr; {указатели на левое и правое поддеревья}
      end;
Var next_number:integer;
   r, pass:top_ptr;    {корень бинарного дерева}
 {процедура добавления вершины к дереву}
Procedure Add(Var r:top_ptr; pass:top_ptr);
  begin
      if r=nil then r:=pass  {если место свободно, то добавляем}
      else    {иначе идем налево или направо}
        if (pass^.value<r^.value) then Add(r^.left,pass)
                                  else Add(r^.right,pass);
  end;
{процедура сортировки – обход дерева}
procedure Tree(r:top_ptr);
  begin
     if r<>nil then
          begin    {если есть поддерево}
              Tree(r^.left);        {обход левого поддерева}
              Write(r^.value:4); {вывод значения из корня}
              Tree(r^.right);       {обход правого поддерева}
           end;
  end;
 {основная программа}
begin
   {формирование исходного дерева}
   WriteLn('Вводите числа');
   r:=nil;
   Read(next_number);
   while not EOF do
```

```
        begin
            new(pass);   {выделяем память для нового элемента}
            with pass^ do  {заносим значения}
                begin
                    value:=next_number;
                    left:=nil;        right:=nil;
                end;
            Add(r, pass);  {добавляем элемент к дереву}
            Read(next_number)
        end;
    ReadLn;
    WriteLn('Сортированная последовательность:');
    Tree(r);
End.
```

Оценка вычислительной сложности операций с сортированными бинарными деревьями. Использование сортированных бинарных деревьев достаточно эффективно с точки зрения временных оценок. Оценим вычислительную сложность основных операций с данной структурой.

Операция поиска вершины. Худшим случаем при поиске вершины является случай, когда дерево имеет вид последовательности вершин (рис. 7.24, *а*, *б*). В этом случае для поиска вершины в среднем потребуется (n+1)/2 проверок, как при последовательном поиске, например, в массиве, или $O_x(n)$.

Оценку в среднем выполним для сбалансированного дерева (рис. 7.24, *в*). Поиск вершины в этом случае потребует $(\log_2 n+1)/2$ проверок, т. е. получаем вычислительную сложность в среднем $O_{ср}(\log_2 n)$.

Операция построения дерева. Худшим случаем при построении дерева является дерево, представляющее собой последовательность вершин, так как при этом поиск вершины, к которой необходимо добавить новую, потребует максимального числа проверок. Количество проверок в этом случае: n-1 – для последнего элемента, 1 – для второго (для первого элемента проверки не

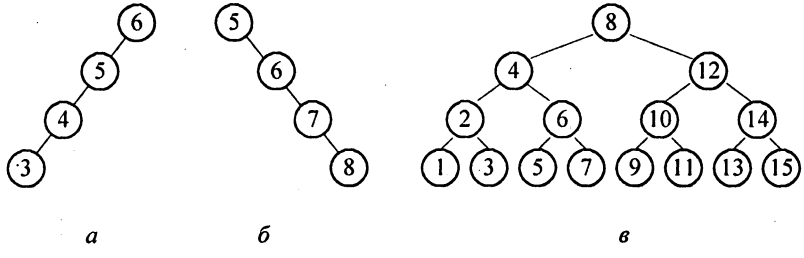


Рис. 7.24. Частные случаи бинарных деревьев:
а, *б* – деревья в виде последовательности вершин; *в* – сбалансированное дерево

требуется). В среднем необходимо [(n-1)+1]/2 проверок; умножив на n-1 элемент, получаем n(n-1)/2 проверок, или $O_x(n^2)$.

Оценку в среднем выполним также на сбалансированном дереве. Количество проверок для поиска родительской вершины составит $(\log_2 (n-1)+1)/2$; соответственно, умножив на (n-1), получим $O_{cp}(n\log_2 n)$.

Операция обхода дерева в процессе получения сортированной последовательности. Для деревьев обоих видов сложность одинакова, так как для каждой вершины при обходе проверяется наличие левого и правого поддеревьев, т.е. суммарное количество проверок равно 2n. Следовательно, вычислительная сложность операции обхода равна O(n). С учетом построения дерева вычислительная сложность в среднем составит $O_{cp}(n \log_2 n)$.

Задания для самопроверки

Задание 1. Разработайте программу, которая обеспечивает быстрый поиск информации о сотрудниках фирмы с использованием бинарных деревьев: имя, фамилия, отчество, отдел, должность, служебный телефон, домашний адрес и телефон. Определите, во сколько раз быстрее в среднем будет выполняться поиск информации по сравнению с последовательным поиском (см. параграф 4.6), если количество сотрудников фирмы 10, 100, 1000 человек.

Задание 2. Для программы предыдущего задания оцените объем оперативной памяти, необходимой для размещения информации о 300 сотрудниках фирмы. Определите долю памяти, отводимой для хранения адресов элементов.

7.6. Практикум. Разбор арифметических выражений с использованием бинарных деревьев

Проблема вычисления арифметических выражений, вводимых пользователем, а потому представленных символьной строкой, возникает при решении многих практических задач, например, при разработке универсальных программ, решающих задачи вычислительной математики (поиск корней функции, вычисление определенного интеграла и т.п.).

Одним из способов разбора выражения является представление его в виде дерева, каждое поддерево которого отображает одну операцию выражения в порядке убывания приоритета операции. В корнях такого дерева хранится знак операции, а каждое поддерево представляет собой операнд. Так, например, выражение

$$(x+1,5)(x-10)^{(x+5)/(x-3,1)}$$

может быть представлено бинарным деревом, изображенным на рис. 7.25 (возведение в степень обозначено символом «^»).

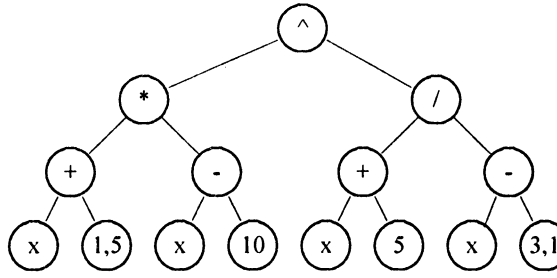


Рис. 7.25. Представление выражения в виде бинарного дерева

Построение дерева выражения выполняется следующим образом:

а) в символьной строке, содержащей запись выражения, определяется положение операции с минимальным приоритетом, записанной вне круглых скобок (если строка содержит несколько операций одинакового приоритета, то выбираем любую из них);

б) операция записывается в корень дерева, а строка делится на две: подстрока первого операнда и подстрока второго операнда, при этом по необходимости выполняется операция снятия скобок;

в) в подстроках операндов вновь определяется положение операции с минимальным приоритетом и т.д.

Если полученные на очередном шаге подстроки не содержат операций, то процесс заканчивается, а подстроки записываются в вершины очередного уровня в качестве листьев.

Листья такого дерева могут быть двух типов: лист, содержащий имя переменной, и лист, содержащий значение константы. Вычисление выражения выполняется рекурсивно:

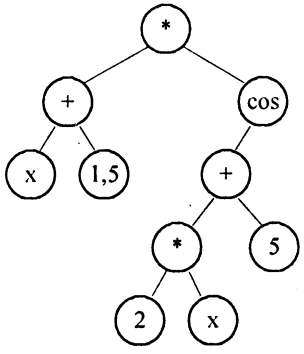


Рис. 7.26. Представление в виде бинарного дерева выражения, содержащего элементарные функции

- если вершина – лист, то в качестве результата подставляется значение константы или переменной,
- если вершина – операция, то вычисляется левый операнд, потом – правый операнд, а затем над ними выполняется операция.

Правила работы с деревом выражения легко можно дополнить так, чтобы обеспечить вычисление элементарных функций, таких, как sin x или ln x. Например, имя функции будем записывать в соответствующей вершине дерева, а аргумент представлять в виде левого поддерева. Выражение (x+1,5)*cos(2*x+5), таким образом, будет представлено деревом, изображенным на рис. 7.26.

Пример 7.7. Разработать программу построения таблицы значений функции одного аргумента, определенной пользователем. (Чтобы не усложнять и без того сложную программу, не будем предусматривать операцию «унарный минус».)

В первую очередь определим структуру информационной части записи, соответствующей вершине дерева: каждая вершина должна хранить знак операции, адреса операндов и для констант – значение константы, причем для листьев в качестве знака операции будет храниться признак типа листа: «o» – константа или «x» – переменная.

Процедура Constr_Tree конструирования дерева из выражения по сути рекурсивна: она последовательно должна разбивать строку выражения на отдельные операции и строить соответствующие поддеревья. Параметры процедуры: адрес корня дерева r и строка выражения, из которой необходимо построить дерево. Процедура должна многократно осуществлять поиск разделяющего знака операции: сначала нижнего уровня приоритета, затем все более высокого. Этот поиск целесообразно реализовать как внутреннюю функцию, которая будет получать множество знаков операции Set_Of и строку st.

Вычисление выражения по дереву также будет выполняться рекурсивной функцией Count. Эта функция в качестве параметров будет получать значение адрес корня дерева и значение x. При вычислении учтем, что выполнение некоторых операций, например деления на ноль, получение логарифма отрицательного числа и т.п., невозможно; в этом случае функция Count будет возвращать значение параметра Key=false.

```
Program ex;
Type setChar=set of char;  {тип «множество символов»}
   str80=string[80];   {тип «строка длиной 80 символов»}
   pTop=^Top;          {тип «указатель на вершину»}
   Top=record              {тип «вершина»}
      operator:string[5];        {знак операции}
      value:single;              {значение константы}
      left,right:pTop;    {указатели на левое и правое поддерево}
    end;
Var st:str80;              {строка – запись выражения}
    Root:pTop;             {корень дерева выражения}
    key:boolean;  {признак существования значения в заданной точке}
    x,xn,xe,dx,y:single; {начальное, конечное значения и шаг
                          аргумента, значение аргумента и функции}
     n,i:word;       {количество точек и номер текущей точки}
{рекурсивная функция конструирования поддерева
  выражения с корнем r из строки st}
Procedure Constr_Tree(r:pTop;st:str80);
 Var next:pTop; SetOp:setChar; po,code:integer;
       stl,stri:str80; c:single;
```

```
{внутренняя функция поиска разделительного знака в строке st:
 SetOp – множество знаков; функция возвращает позицию разде-
 лительного знака или 0}
   Function PosOp(st:str80;SetOp:setChar):byte;
    Var i,j,k,p:byte;
    begin
      j:=0; k:=0; p:=0; i:=1;
      while (i<= length(st)) and ( p=0)  do
        begin
            if st[i]='('  then inc(j)  {считаем количество
                                        открывающихся скобок}
            else  if st[i]=')' then inc(k) {считаем количество
                                        закрывающихся скобок}
                 else  if (j=k) and (st[i] in SetOp) then p:=i;
          inc(i);
        end;
    PosOp:=p;
  end;
{раздел операторов функции конструирования дерева выражения}
Begin
 po:=PosOp(st,['+','-']);  {ищем разделительный знак операции + или -}
 if po=0  then po:=PosOp(st,['*','/']); {ищем разделительный знак
                                        операции * или /}
 if po=0  then po:=PosOp(st,['^']);{ищем разделительный знак операции ^}
 if po<>0 then                {разделяющий знак найден}
   begin
     r^.operator:=st[po];  {записываем знак операции в вершину}
     stl:=copy(st,1,po-1);          {копируем подстроку первого операнда}
     if (stl[1]='(') and (PosOp(stl,['*','/','+','-','^'])=0) then
                  stl:=copy(stl,2,length(stl)-2);       {убираем скобки}
     stri:=copy(st,po+1,length(st)-po);    {копируем подстроку второго опе-
                                 ранда}
     if (stri[1]='(') and (PosOp(stri,['*','/','+','-','^'])=0) then
                 stri:=copy(stri,2,length(stri)-2);  {убираем скобки}
     new(r^.left);                  {создаем левое поддерево}
     Constr_Tree(r^.left,stl);    {конструируем левый операнд}
     new(r^.right);               {создаем правое поддерево}
     Constr_Tree(r^.right,stri); {конструируем правый операнд}
   end
 else
      if st[1]='x' then {аргумент}
           begin
             r^.operator:='x';
```

```
                r^.left:=nil;
                r^.right:=nil;
              end
          else
            begin
              val(st,c,code);   {пытаемся получить число}
              if code=0 then    {константа}
                 begin
                   r^.operator:='o';
                   r^.left:=nil;
                   r^.right:=nil;
                   r^.Value:=c;
                 end
              else          {функция}
                begin
                   po:=Pos('(',st);
                   r^.operator:=copy(st,1,po-1); {выделяем имя функции}
                   r^.right:=nil;
                   stl:=copy(st,po+1,length(st)-po-1);   {выделяем подстроку
                                                          параметра}
            new(r^.left);
            Constr_Tree(r^.left,stl); {конструируем параметр}
          end;
      end;
  end;
{рекурсивное вычисление значения функции:
  если Key=false, то значение не существует}
 Function Count(r:pTop;x:single;Var key:boolean):single;
   Var s,s1:single;
   begin
     if not key then     {значение функции не существует}
          begin
             Count:=0;
             exit;
          end;
     if r^.operator='o'  then
                              Count:=r^.Value  {константа}
    else
         if r^.operator='x' then
                              Count:=x      {переменная x}
         else
          case r^.operator[1] of
          '+': Count:=Count(r^.left,x,key)+Count(r^.right,x,key);
```

```
        '-': Count:=Count(r^.left,x,key) - Count(r^.right,x,key);
        '*': Count:=Count(r^.left,x,key)*Count(r^.right,x,key);
        '/': begin
               s:=Count(r^.right,x,key);
               if abs(s)<1e-10  then  {практический ноль}
                  begin
                    Count:=0;   key:=false;
                  end
               else Count:=Count(r^.left,x,key)/s;
             end;
      '^': begin
             s:=Count(r^.left,x,key);
             s1:=Count(r^.right,x,key);
             if s<>0 then
                      Count:=exp(s1*ln(abs(s)))
             else
                 if s1=0 then Count:=1
                         else Count:=0;
           end;
      's': Count:=sin(Count(r^.left,x,key));
      'c': Count:=cos(Count(r^.left,x,key));
      else       {неопределенная операция}
       begin
          Count:=0;
          Key:=false;
       end
    end
end;
          {основная программа}
    Begin
      WriteLn('Введите выражение:');
      ReadLn(st);
      Write('Введите xn, xe, n: ');
      ReadLn(xn,xe,n);
      new (Root);
      Constr_Tree(Root,st);
      dx:=(xe-xn)/(n-1);
      Writeln('  x   ','          y');
      x:=xn;
      for i:=1 to n do
        begin
          key:=true;
          y:=Count(Root,x,key);
```

```
        if key then
                WriteLn(x:6:3,y:20:3)
        else    WriteLn(x:6:3,' не существует');
        x:=x+dx;
    end;
End.
```

Задания для самопроверки

Задание 1. Дополните программу разбора выражений, чтобы с ее помощью можно было обрабатывать выражения, содержащие основные математические функции: cos, tg, ctg, e^x и т. д.

Задание 2. Дополните программу разбора выражений возможностью обрабатывать унарный минус.

Задание 3. Модернизируйте программу разбора выражений так, чтобы она могла разбирать выражения с двумя переменными.

8. УПРАВЛЕНИЕ ТЕХНИЧЕСКИМИ СРЕДСТВАМИ И ВЗАИМОДЕЙСТВИЕ С MS DOS

Управление техническими и многими программными средствами в операционной системе MS DOS осуществляется через систему прерываний.

Прерывание – это событие в системе, которое требует специальной обработки. К таким событиям относятся: требования обработки от внешних устройств, например, часов или устройств ввода-вывода, и требования программ на выполнение некоторых действий, например, операций ввода-вывода. Каждое прерывание имеет свой номер. Необходимая обработка осуществляется специальными программами – *обработчиками прерываний*, большая часть которых входит в состав BIOS (базовой системы ввода-вывода) или MS DOS.

В Borland Pascal для взаимодействия с техническими средствами и MS DOS существует набор специальных процедур и функций, которые входят в состав модулей Crt, Graph и Dos.

8.1. Управление экраном в текстовом режиме

Выше рассматривались программы, которые выводили результаты на экран в так называемом *консольном режиме*. В этом режиме вывод на экран происходит построчно, доступ возможен только к последней выводимой строке. По мере заполнения экрана осуществляется его «прокрутка», при которой строки перемещаются по экрану вверх, причем верхние строки безвозвратно теряются, а внизу появляются новые строки. В таком режиме программист почти не может управлять формой представления выводимой информации.

В текстовом режиме программист имеет доступ ко всему экрану. Экран при этом поделен на строки и столбцы. На пересечении строки и столбца находится область, в которою возможен вывод одного знака. Такие области получили название *знакоместо*. Обычно программа на Borland Pascal использует тот же текстовый режим, что и MS DOS, т.е. режим, при котором на экране выделяется 25 строк и 80 столбцов.

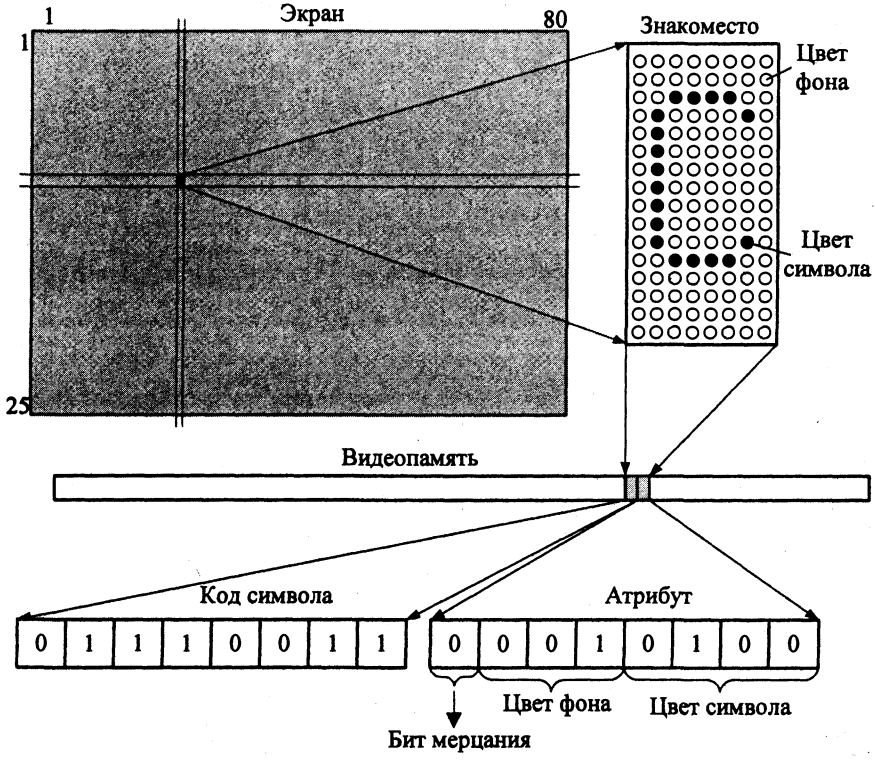


Рис. 8.1. Схема управления экраном в текстовом режиме

Каждому знакоместу экрана в специальной памяти, называемой *видеобуфером*, соответствует 2 байта, в которых хранится информация о высвечиваемом символе:

- код символа по таблице ASCII, которому соответствует матрица изображения символа в специальной таблице знакогенератора;
- байт атрибут, в котором хранится информация о цвете символа и цвете фона данного знакоместа (рис. 8.1).

Изображение на экране получают с помощью электронного луча, который обходит экран слева направо и сверху вниз с заданной частотой развертки. Каждая точка при этом высвечивается цветом символа или цветом фона знакоместа, которому она принадлежит. Информация о знакоместе выбирается из видеобуфера. Таким образом, если изменить информацию в видеобуфере, то изменится и изображение на экране.

Цвета в текстовом режиме формируются следующим образом: три бита управляют включением и выключением трех основных цветов (синего, зеленого и красного) и один бит – яркостью (табл. 8.1).

Таблица 8.1

Значения бит, кодирующих цвет				Десятичная константа	Цвет
Яркость	r (красный)	g (зеленый)	b (голубой)		
0	0	0	0	0	Черный
0	0	0	1	1	Синий
0	0	1	0	2	Зеленый
0	0	1	1	3	Голубой
0	1	0	0	4	Красный
0	1	0	1	5	Фиолетовый
0	1	1	0	6	Коричневый
0	1	1	1	7	Светло-серый
1	0	0	0	8	Темно-серый
1	0	0	1	9	Светло-синий
1	0	1	0	10	Светло-зеленый
1	0	1	1	11	Светло-голубой
1	1	0	0	12	Розовый
1	1	0	1	13	Сиреневый
1	1	1	0	14	Желтый
1	1	1	1	15	Белый

Ресурсы модуля crt для управления экраном. Управление экраном с помощью ресурсов модуля crt базируется на понятии «окно».

Окно – часть экрана прямоугольной формы. В момент получения программой управления весь экран считается окном 25×80 знакомест. Программист может определять на экране новые окна и управлять как цветом символов и фона окна, так и размещением информации в окне.

Вывод информации в текстовом режиме осуществляется стандартными процедурами ***Write*** и ***WriteLn*** *текущими цветом символа* и *цветом фона.* При выводе четыре символа интерпретируются особым образом:

#7 – звуковой сигнал;

#8 – перемещение курсора влево на один символ;

#10 – перемещение курсора на строку вниз (если курсор находился в последней строке, то содержимое экран «прокручивается» на строку вверх);

#13 – перемещение курсора в начало текущей строки.

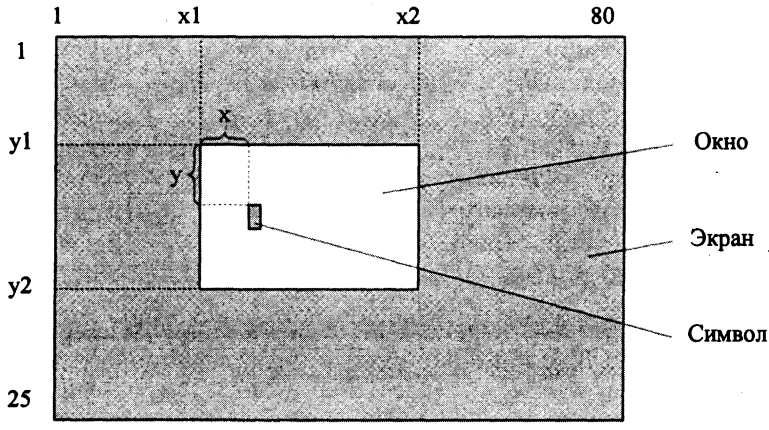


Рис. 8.2. Текущее окно на экране и относительная адресация символа в окне

Процедуры начинают вывод с того места, где стоит курсор. Координаты курсора определяются относительно верхнего левого угла текущего окна (рис. 8.2).

Для управления окнами и размещения в них информации модуль crt содержит следующие процедуры и функции.

1. Процедура ***Window(x1, y1, x2, y2:word)*** – определяет на экране окно. Местоположение и размеры окна определяются координатами верхнего левого (x1, y1) и нижнего правого (x2, y2) углов прямоугольника. Координаты текущего окна модуль crt хранит в специальных переменных:

***WindMin, WindMax: word*.**

Откуда координаты текущего окна можно определить, применив функции lo и hi, которые выделяют из слова младший и старший байты соответственно:

x1=Lo(WindMin) – координата x верхнего левого угла;
y1=Hi(WindMin) – координата y верхнего левого угла;
x2=Lo(WindMax) – координата x нижнего правого угла;
y2=Hi(WindMax) – координата y нижнего правого угла.

После объявления окна курсор устанавливается в верхний левый угол окна.

2. Процедура ***TextColor(color:byte)*** – устанавливает текущий цвет вывода символов в окне.

3. Процедура ***TextBackGround(color:byte)*** – устанавливает текущий цвет фона окна.

Цвета для процедур TextColor и TextBackGround можно задавать, используя специальные константы модуля crt:

Black = 0;	{черный}	***DarkGrey = 8;***	{темно-серый}
Blue = 1;	{синий}	***LightBlue = 9;***	{светло-синий}
Green= 2;	{зеленый}	***LightGreen = 10;***	{светло-зеленый}
Cyan = 3;	{голубой}	***LightCyan = 11;***	{светло-голубой}
Red = 4;	{красный}	***LightRed = 12;***	{розовый}
Magenta = 5;	{фиолетовый}	***LightMagenta = 13;***	{сиреневый}
Brown = 6;	{коричневый}	***Yellow = 14;***	{желтый}
LightGrey =7;	{светло-серый}	***White = 15;***	{белый}
Blink = 128;	{мерцание}		

Текущие цвета символа и фона в виде байта атрибута хранятся в переменной ***TextAttr:word***. Текущий цвет фона из этой переменной определяется как

(TextAttr div 16) mod 8,

а текущий цвет символа как

TextAttr mod 16.

4. Процедура ***ClrScr*** – очищает окно, выводя в него пробелы с текущим атрибутом. После этого курсор устанавливается в верхний левый угол окна. Если окно не установлено, то очищается весь экран. Эту процедуру обычно используют после определения окна и атрибута его символов, чтобы обозначить окно на экране.

5. Функция ***WhereX:word*** – возвращает координату x текущего положения курсора в окне.

6. Функция ***WhereY:word*** – возвращает координату y текущего положения курсора в окне.

7. Процедура ***GotoXY(x,y:word)*** – перемещает курсор на знакоместо с координатами x и y.

Модуль ctr содержит также процедуры, работающие с *текущей* строкой (строкой, в которой стоит курсор).

8. Процедура ***DelLine*** – удаляет текущую строку.

9. Процедура ***InsLine*** – вставляет строку, сдвигая остальные строки вниз.

10. Процедура ***ClrEol*** – стирает часть строки справа от курсора.

Пример 8.1. Разработать программу вычисления среднего арифметического заданного количества чисел n, где $n \le 10$. Реализовать оконный интерфейс, представленный на рис. 8.3.

Верхнее окно интерфейса не зависит от исходных данных. Его нужно определить, задать цвета символов и фона, очистить и вывести в него текст. Ввод ответа пользователя будет выполняться с того места, где окажется курсор. Количество окон для ввода чисел зависит от введенного значения. Их положение на экране необходимо рассчитать.

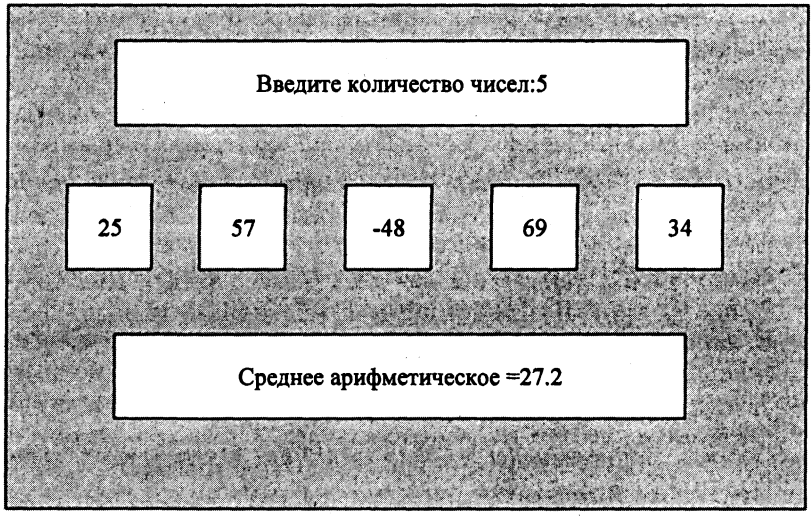


Рис. 8.3. Интерфейс программы вычисления среднего арифметического

Нижнее окно также не зависит от исходных данных. Его нужно определить и вывести в него результат. Ниже представлена соответствующая программа с комментариями.

```
Program ex;
  Uses Crt;
  Var a:array[1..10] of integer;
     x,dx,n,i:integer;       s:real;
  Begin
     ClrScr;                    {очищаем экран}
     Window(20,3,50,5); {выделяем окно ввода количества чисел}
     TextAttr:=16+7;{определяем светло-серые символы на синем фоне}
     ClrScr;                    {выделяем окно ввода}
     GotoXY(2,2);           {устанавливаем курсор для вывода запроса}
     Write('Введите количество чисел: ');     {выводим запрос}
     ReadLn(n);                              {вводим ответ пользователя}
     dx:=(80-(n+1)*3) div n;
     x:=0;
     s:=0;
     TextAttr:=2*16+14; {желтые символы на зеленом фоне}
     for i:=1 to n do
       begin x:=x+3;
             Window(x,7,x+dx,9);  {устанавливаем окно ввода данных}
             ClrScr;          {выделяем это окно}
```

```
        GotoXY(2,2); {устанавливаем курсор для ввода данных}
        ReadLn(a[i]); {вводим число}
        s:=s+a[i];
        x:=x+dx;
    end;
  TextAttr:=4*16+14;  {желтые символы на красном фоне}
  Window(18,11,53,13); {устанавливаем окно результата}
  ClrScr;              {выделяем окно результата}
  GotoXY(3,2);  {устанавливаем курсор для вывода результата}
  Write('Среднее арифметическое =',s/n:5:2);
  ReadLn;              {ожидаем нажатия клавиши Enter}
  Window(1,1,80,25);   {восстанавливаем окно на весь экран}
  TextAttr:=7;         {восстанавливаем стандартные цвета}
  ClrScr;               {очищаем экран}
End.
```

8.2. Управление клавиатурой

Клавиатура – достаточно сложное устройство, в состав которого входит микропроцессор и память – буфер клавиатуры. При нажатии и отпускании любой клавиши в буфер клавиатуры записываются так называемые *коды нажатия/отпускания*, при этом микропроцессор клавиатуры отсекает *дребезг* клавиш – сигналы, полученные при неполном контакте в процессе нажатия и отпускания клавиши.

Клавиши клавиатуры делят на буквенно-цифровые, специальные и клавиши смещения.

К *буквенно-цифровым* относят клавиши букв, цифр, специальных знаков и пробела. Их используют для ввода информации.

Специальные клавиши – это клавиши управления курсором (←, ↑, →, ↓, Home, End, Tab, Page up, Page down), удаления (Del, Backspace), клавиши переключения режимов (Ins, Caps lock, Num lock, Scroollock), функциональные клавиши (Esc, Break, F1, F2, F3, ..., F12) и т.д. Эти клавиши используют для выполнения вспомогательных операций во время работы с компьютером.

Клавиши смещения – это клавиши Shift, Ctrl и Alt. Их используют совместно с другими клавишами для изменения вводимых кодов. Так, если при нажатии клавиши «а» формируется код строчной буквы а латинского алфавита, то нажатие Shift-а приведет к вводу кода заглавной буквы А латинского алфавита.

Всего выделяют около 400 различных комбинаций, которые могут обрабатываться программой. Эти комбинации формируются на основании кодов нажатия/отпускания специальной программой *BIOS* (базовая система обработки ввода-вывода) и записываются в *буфер BIOS клавиатуры*.

Изначально считалось, что количество различных комбинаций не превысит 256, и, соответственно, для представления этой информации будет достаточно 1 байта (см. приложение 2), но со временем количество комбинаций возросло, и потребовалось использование второго байта.

В настоящее время для представления комбинаций, не вошедших в таблицу ASCII, используют *расширенные коды*, состоящие из двух байт: первый байт равен 0, а второй – содержит *расширенный scan-код* (см. приложение 3).

Ввод буквенно-цифровых данных с клавиатуры осуществляется процедурами ***Read*** и ***ReadLn***, при этом реально происходит чтение кодов ASCII из буфера BIOS клавиатуры. Считанные символьные коды преобразуются во внутренний формат в соответствии с типом переменной.

Процедуры Read и ReadLn обрабатывают только комбинации, соответствующие буквам и цифрам, а также некоторые специальные комбинации, например, маркер конца строки (комбинация символов #13, #10).

Модуль crt содержит специальные функции управления клавиатурой, которые позволяют работать с расширенными кодами.

1. Функция ***KeyPressed:boolean*** – возвращает true, если нажата любая клавиша, false – если буфер BIOS клавиатуры пуст; функция не извлекает символы из буфера, не ожидает ввода;

2. Функция ***ReadKey:char*** – извлекает очередной код из буфера BIOS клавиатуры и возвращает его как результат операции, ожидает ввода, но не высвечивает вводимого символа.

Для чтения расширенного кода функцию ReadKey необходимо вызывать дважды: первый раз она вернет 0, а второй – расширенный scan-код:

```
ch1:=ReadKey;  {читаем код}
if ch1=#0 then ch2:=ReadKey;  {если код=0, то читаем второй байт}
```

Пример 8.2. Разработать программу определения кодов клавиш и их комбинаций. Выход из цикла осуществлять по нажатию клавиши Esc.

```
Program ex;
 Uses crt;
 Var c1,c2:char;
 Begin
  repeat    c1:=ReadKey;     {вводим код}
    if c1=#0 then      {если расширенный код}
       begin
          c2:=ReadKey;        {то читаем расширенный scan-код}
          WriteLn(ord(c1):5, ord(c2):5)   {выводим расширенный код}
       end
    else WriteLn(ord(c1):5)     {выводим код ASCII}
  until c1=#27;{до нажатия Esc}
End.
```

Таблица 8.2

Номер бита	Содержимое
1	1 – включен режим вставки
2	1 – включен режим Caps Lock
3	1 – включен режим Num Lock
4	1 – включен режим Scroll Lock
5	1 – нажата клавиша Alt
6	1 – нажата клавиша Ctrl
7	1 – нажата клавиша левый Shift
8	1 – нажата клавиша правый Shift

Примечание. Функция ReadKey обрабатывает коды из буфера BIOS клавиатуры, поэтому с ее помощью нельзя получить коды нажатия/отпускания отдельных клавиш, не преобразуемых в расширенные scan-коды, например, клавиш смещения, клавиш переключения режимов.

Состояния клавиш смещения и клавиш переключения режимов BIOS фиксирует в *байте состояния клавиатуры* (табл. 8.2), который расположен в оперативной памяти по адресу $0:$417.

Для прямого обращения к этому байту можно использовать стандартно объявленный массив Mem:array of byte, например: *Mem[$0:$417]*, или наложить некоторую переменную на интересующий нас байт оперативной памяти:

Var KeyState:byte ***absolute $0:$417;*** *... .*

8.3. Управление динамиком

Модуль crt также содержит процедуры, обеспечивающие управление динамиком.

1. Процедура ***Sound (f:word)*** – генерирует звук указанной частоты в Гц. Для справки, основной гамме соответствуют следующие частоты: нота «до» основной октавы – 330 Гц, далее – 349 Гц, 370 Гц, 392 Гц, 415 Гц, 440 Гц, 466 Гц, 494 Гц, 523 Гц, 554 Гц, 588 Гц, 622 Гц и, наконец, нота «до» следующей октавы – 660 Гц. Частоты нот других октав кратны частотам основной.

2. Процедура ***NoSound*** – выключает динамик.

3. Процедура ***Delay (t:word)*** – обеспечивает задержку на заданный интервал времени, мс.

Поскольку к настоящему моменту времени быстродействие компьютеров существенно возросло и изменились некоторые принципы их построения, процедура Delay не всегда обеспечивает корректную задержку, как видно из последующей программы. В этих случаях для организации задержки целесообразно использовать процедуру, которая читает реальное время.

4. Процедура ***GetTime (Var Hour, Minute, Second, Sec100:word)*** – возвращает текущее время суток. Определена в модуле Dos.

Пример 8.3. Разработать программу проигрывания основной октавы.

Проигрывание гаммы осуществляется включением и выключением динамика с разными частотами.

```
Program ex;
 Uses Crt;
 Const f: array[1..13] of word = (330, 349, 370, 392, 415, 440,
                                  466, 494, 523, 554, 588, 622, 660);
 Var i:byte;
 Begin for i:=1 to 13 do
            begin
              Sound(f[i]);
              for j:=1 to 5000 do Delay(1000);   {задержка ?!}
              NoSound;
            end;
 End.
```

Чтобы не подбирать время задержки для конкретного компьютера, построим свою процедуру обеспечения требуемой задержки, использующую процедуру GetTime:

```
Program ex;
Uses Crt, Dos;
Procedure NewDelay(dTime:word);
Var key:boolean;
  Hour, Min, Sec, Sec100, MyHour, MyMin, MySec, MySec100: Word;
Begin
 GetTime(Hour, Min, Sec, Sec100);  {узнаем текущее время}
 {определяем время завершения задержки}
 MySec100:=Sec100+dTime;  MySec:=Sec+MySec100 div 100;
 MySec100:=MySec100 mod 100;
 MyMin:=Min+MySec div 60;      MySec:=MySec mod 60;
 MyHour:=Hour+MyMin div 60;    MyMin:=MyMin mod 60;
 key:=false;
 while not key do    {цикл задержки}
   begin
     GetTime(Hour, Min, Sec, Sec100); {узнаем текущее время}
     {проверяем, наступил ли заданный момент}
     if (Hour>MyHour) or ((Hour=MyHour) and ((Min>MyMin) or
        ((Min=MyMin) and ((Sec>MySec) or
         ((Sec=MySec) and ((Sec100>=MySec100)))))))
             then key:=true;
   end
End;
Const f: array[1..13] of word = (330, 349, 370, 392, 415, 440,
                                 466, 494, 523, 554, 588, 622, 660);
 Var i:byte;   j:integer;
```

```
 {описываем массив окон пунктов меню}
Const menu:array[1..4] of win=
                    ((x1:5;y1:4;x2:15;y2:4;text:'new '),
                     (x1:5;y1:5;x2:15;y2:5;text:'open'),
                     (x1:5;y1:6;x2:15;y2:6;text:'save'),
                     (x1:5;y1:7;x2:15;y2:7;text:'exit'));
 {процедура рисования пункта меню}
Procedure DrawWin(w:win;attr:byte);
  Begin
   with w do
     begin
        TextAttr:=attr;          {устанавливаем атрибут окна пункта}
        Window(x1,y1,x2,y2);     {устанавливаем окно пункта}
        Clrscr;                  {высвечиваем окно пункта}
        GotoXY(2,1);             {устанавливаем курсор}
        Write(text);             {выводим название пункта}
     end;
  End;
 {процедура рисования меню с выделенным пунктом npos}
Procedure DrawMenu(npos:integer);
  Begin
    Clrscr;
    for i:=1 to 4 do
      if i=npos then DrawWin(menu[i],94)  {выводим выделенный пункт}
      else DrawWin(menu[i],30);           {выводим невыделенный пункт}
  End;
 {основная программа}
Begin
  npos:=1; {выделенный пункт меню}
  DrawMenu(npos);  {выводим меню}
  repeat
    ch1:=ReadKey;     if ch1=#0 then ch2:=ReadKey;
    case ch1 of
    #0: case ch2 of
        #72: begin     {стрелка вверх}
               if npos>1 then  {если не верхний пункт}
                 begin
                   DrawWin(menu[npos],30); {убираем выделение текущего пункта меню}
                   npos:=npos-1; {переходим к предыдущему пункту}
                   DrawWin(menu[npos],94); {выделяем новый пункт}
                 end;
             end;
```

```
Begin for i:=1 to 13 do
          begin
            Sound(f[i]);
            NewDelay(50);
            NoSound;
          end;
End.
```

8.4. Практикум. Создание меню

Как уже упоминалось ранее, современные программы взаимодействуют с пользователями через специальный интерфейс типа «меню», в котором пользователю предлагается выбрать один из пунктов. Для реализации меню необходимо обрабатывать коды нажатия клавиш управления курсором (↓, ↑, →, ←), по которым обычно осуществляется переход на следующий пункт. Активизация пункта обычно выполняется нажатием клавиши Enter. После выполнения нужного пункта программа должна вновь выводить меню и продолжать работу с ним. Выход из программы осуществляется по выбору специального пункта «выход» или по нажатию клавиши Esc.

Пример 8.4. Разработать вертикальное меню из четырех пунктов: new, open, save, exit. Окно меню синего цвета, названия пунктов должны быть выведены желтым цветом. Выделение пункта выполнить фиолетовым цветом фона (рис. 8.4). Реализовать выход по выбору пункта «exit» или по нажатию клавиши Esc.

При разработке алгоритма программы выделим две подпрограммы:

- подпрограмму рисования пункта меню как окна с текстом внутри;
- подпрограмму рисования меню с выделенным пунктом.

В процессе работы основная программа вводит коды клавиш и организует работу с меню. После завершения работы такая программа должна восстановить стандартное окно 25×80 и стандартные цвета символа и фона MS DOS.

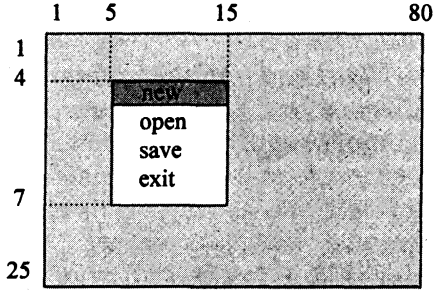


Рис. 8.4. Внешний вид меню

```
Program ex;
 Uses crt;
 Var npos,i:integer;
     ch1,ch2:char;
 Type
  win=record  {описываем тип
               окон пунктов меню}
    x1,y1,x2,y2:word; {координаты
                       окна}
    text:string[8]; {название пункта
                     меню}
  end;
```

```
    #80: begin
           if npos<4 then  {если не нижний пункт}
             begin
               DrawWin(menu[npos],30);  {убираем выделение текущего пункта}
               npos:=npos+1;     {переходим к следующему пункту}
               DrawWin(menu[npos],94);  {выделяем новый пункт}
             end;
         end;
      end; {case интерпретации расширенного кода}
  #13: begin
         Window(1,1,80,25);
         TextAttr:=7;
         ClrScr; {очищаем экран}
         case npos of
         1: begin
              Write('Выполнен пункт ', menu[npos].text);
              ReadLn;
            end;
         2: begin
              Write('Выполнен пункт ', menu[npos].text);
              ReadLn;
            end;
         3: begin
              Write('Выполнен пункт ', menu[npos].text);
              ReadLn;
            end;
         end;  {case}
       DrawMenu(npos);   {выводим меню}
      end;
    end; {case}
  until (ch1=#27) or ((ch1=#13) and (npos=4));
  Window(1,1,80,26);
  TextAttr:=7;
  ClrScr; {очищаем экран}
End.
```

Задания для самопроверки

Задание 1. Разработайте программу исследования элементарных функций, задаваемых пользователем (см. параграф 5.2). Обеспечьте для каждой функции вывод таблицы значений на заданном интервале с заданным шагом, поиск корней и определение максимума и минимума. Взаимодействие пользователя и программы организуйте с использованием меню.

Задание 2. Разработайте программу тестирования обучающихся по теме «системы счисления». Тестируемому должны предлагаться 6 вопросов по данной теме, включая обычные вопросы с выбором ответа из нескольких и задачи на выполнение арифметических операций, когда необходимо ввести результат указанной операции. Вопросы должны случайным образом выбираться из списка, хранящегося в файле, и не повторяться. Для ответа на каждый вопрос дается две попытки. Предусмотреть, чтобы тестирующийся мог по желанию отказаться отвечать на данный вопрос и получить правильный ответ. Оценку проводить по соотношению правильных и неправильных ответов.

8.5. Управление экраном в графическом режиме

В графическом режиме программист получает возможность управлять каждой точкой (*пикселем*) экрана. Координаты точки определяются относительно верхнего левого угла. Каждая точка экрана при этом может высвечиваться одним из доступных цветов. Информация о цвете точки хранится в видеобуфере.

Количество цветов зависит от количества бит, отведенных в видеобуфере под одну точку. Рассмотрим основные варианты.

1. «1 точка – 1 бит» – монохромный режим: каждая точка высвечивается либо основным цветом, если в видеобуфере для точки записана 1, либо цветом фона, если в видеобуфере записан 0.

2. «1 точка – 2 бита» – режим с двумя трехцветными палитрами:

Палитра 0:	Палитра 1:
01 – зеленый;	01 – светло-голубой;
10 – красный;	10 – сиреневый;
11 – коричневый.	11 – белый.

Если в буфере записана комбинация 00, то точка высвечивается цветом фона.

3. «1 точка – 4 бита» – режим, использующий 16-цветную палитру. В этом режиме в отличие от предыдущих *в видеобуфер заносится не цвет точки, а номер регистра палитры*, в котором записан нужный цвет (рис. 8.5).

Для записи цвета используется 6 бит по схеме RGBrgb, причем первые три бита кодируют 2/3 яркости цвета, а вторые три бита – оставшуюся 1/3 яркости. Так, максимально яркий красный цвет будет кодироваться двумя единицами в первом и четвертом битах:

R	G	B	r	g	b
1	0	0	1	0	0

Таким образом, каждый цвет имеет четыре градации яркости: 0, 1/3, 2/3, 1, что позволяет кодировать $4^3 = 64$ варианта цвета. На экране в этом режи-

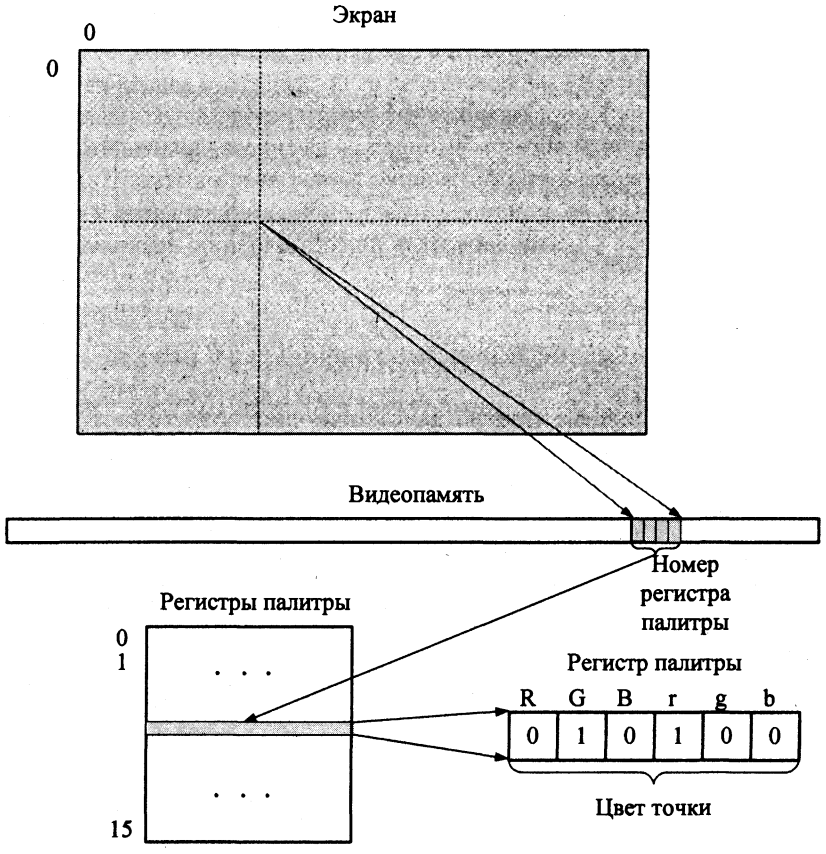


Рис. 8.5. Формирование цвета точки в 16-цветном режиме

ме одновременно может присутствовать не более 16 цветов, так как имеется всего 16 регистров палитры.

Примечание. То, что цвет точки определяется кодом, записанным в регистре палитры, позволяет получать интересный эффект: если во время работы программы изменить цвет в одном из регистров палитры, то все точки, связанные с этим регистром палитры, изменят цвет. Так можно реализовывать исчезновение и появление некоторого монохромного изображения.

4. «1 точка – 8 бит» – режим, использующий палитру на 256 цветов. В этом режиме используется та же идея записи кода цвета не в видеобуфер, а в регистры палитры, что и в предыдущем режиме, но используется 256 регистров палитры, в которых под запись цвета используется 18 бит. Из этих 18 бит под кодирование яркости каждого цвета используется 6 бит, обеспечивая 64 градации яркости каждого цвета. В этом режиме максимальная яркость красного цвета будет кодироваться так:

Red						Green						Blue					
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Таким образом, на экране можно одновременно видеть 256 цветов из $64^3 = 262144$ возможных.

В настоящее время в основном используются режимы, в которых цвет кодируется еще большим количеством бит. К сожалению, работа в этих режимах и даже в режиме 256 цветов не обеспечивается стандартными средствами Borland Pascal 7.0.

Количество точек на экране, набор возможных цветов и количество страниц изображения, которые могут одновременно храниться в памяти, зависят от используемых технических средств (типа монитора и блока управления монитором – *адаптера*) и режимов их работы. Для каждого типа оборудования, существовавшего на момент разработки среды программирования, среда программирования Borland Pascal 7.0 включает свою программу управления дисплеем – драйвер, которая обеспечивает работу в нескольких доступных режимах, различающихся количеством точек на экране и количеством страниц, информация о которых может храниться в видеобуфере.

Последнее время из имеющихся драйверов практически используется только драйвер VGA, который обеспечивает несколько различных режимов работы. Этот драйвер находится в файле EGAVGA.BGI и ему в модуле Graph соответствует поименованная константа VGA = 9. Режимы работы этого драйвера и соответствующие константы Borland Pascal приведены в табл. 8.3.

Набор программных ресурсов, используемых для управления экраном в графическом режиме, содержится в модуле Graph.

Процедуры и функции переключения режимов управления экраном. Модуль Graph содержит средства, обеспечивающие различные варианты переключения текстового и графического режимов.

1. Процедура ***InitGraph(Var driver, mode:integer; path:string)*** – переключает экран в графический режим. При вызове процедуры следует объявить специальные переменные, куда занести константу драйвера и константу режима, и указать эти переменные в качестве параметров процедуры. Существует возможность запросить автоматическое определение драйвера и режи-

Таблица 8.3

Идентификатор константы и номер режима	Количество цветов на экране	Количество точек на экране	Количество страниц в видеобуфере
VGALo = 0	16	640×200	4
VGAMed = 1	16	640×350	2
VGAHi = 2	16	640×480	2

ма: для этого необходимо вместо константы драйвера в переменную, используемую в качестве параметра, записать константу detect = 0.

Параметр path должен описывать путь, определяющий местоположение файла, который содержит требуемый драйвер (обычно все драйверы находятся в подкаталоге BGI основного каталога среды Borland Pascal). Если драйвер находится в текущем каталоге, то в качестве параметра передается пустая строка – ''.

2. Функция ***GraphResult:integer*** – возвращает номер ошибки, обнаруженной при инициализации графического режима.

3. Функция ***GraphErrorMSG(ErrNum:integer):string*** – позволяет по номеру ошибки определить ее причину.

Полностью инициализация графического режима может быть выполнена следующим образом:

```
Var driver,mode,error:integer;
Begin
  driver:=detect;   {или driver:=0;}
  InitGraph(driver,mode,'d:\BP\BGI');
  error:=GraphResult;
  if error<>0 then  {если обнаружены ошибки}
    begin
      WriteLn('Ошибка инициализации графического режима',
                                        GraphErrorMSG(error));
      Halt(1);  {аварийное завершение программы}
    end;
  {работа в графическом режиме}...
```

4. Процедура ***CloseGraph*** – завершает работу в графическом режиме: выгружает драйвер и восстанавливает текстовый режим. Если завершить программу, не выходя из графического режима, то нормальная работа MS DOS нарушается, так как MS DOS функционирует в текстовом режиме.

Примечание. Если программа выполняется в среде программирования Borland Pascal, то среда сама восстановит текстовый режим после завершения работы программы.

5. Процедура ***RestoreCrtMode*** – осуществляет временный возврат в текстовый режим с сохранением графического изображения в видеобуфере.

6. Процедура ***SetGraphMode(mode:integer)*** – осуществляет возврат в графический режим после временного выхода, осуществленного процедурой RestoreCrtMode.

7. Функция ***GetGraphMode:integer*** – возвращает номер активного графического режима.

Таким образом, временный выход в текстовый режим обычно оформляется следующим образом:

```
RestoreCrtMode;  {переход в текстовый режим}
  ... {выполнение обработки в текстовом режиме}
SetGraphMode(GetGraphMode); {возврат в графический режим}
```

Процедуры и функции управления цветом. Модуль Graph содержит специальные процедуры и функции управления цветом изображения.

1. Процедура ***GetPalette(Var Palette:PaletteType)*** – возвращает размер и цвета текущей палитры. Палитра при этом считывается в специальную переменную типа PaletteType, также определенного в модуле Graph:

```
Type
 PaletteType = record
       size:byte;  {размер палитры}
       Colors:array[0..MaxColors] of shortint; {цвета палитры}
 end;
```

Цвета палитры кодируются десятичными числами от 0 до 63.

2. Процедура ***SetAllPalette(Palette: PaletteType)*** – осуществляет установку палитры. Новая палитра при этом записывается в переменную типа PaletteType, а цвета можно кодировать десятичными числами или использовать для их определения специальные константы:

EGABlack = 0;	{черный}	***EGADarkGrey=56;***	{темно-серый}
EGABlue = 1;	{синий}	***EGALightBlue= 57;***	{светло-синий}
EGAGreen= 2;	{зеленый}	***EGALightGreen = 58;***	{светло-зеленый}
EGACyan = 3;	{голубой}	***EGALightCyan = 59;***	{светло-голубой}
EGARed = 4;	{красный}	***EGALightRed = 60;***	{розовый}
EGAMagenta=5;	{фиолетовый}	***EGALightMagenta = 61;***	{сиреневый}
EGABrown=20;	{коричневый}	***EGAYellow = 62;***	{желтый}
EGALightGrey=7;	{светло-серый}	***EGAWhite = 63;***	{белый}

3. Процедура ***SetPalette(ColorNum,Color:word)*** – заменяет цвет в регистре палитры с номером ColorNum на цвет Color.

4. Процедура ***SetBkColor(Color:word)*** – устанавливает новый цвет фона. При инициализации графического режима видеобуфер очищается (обнуляется). При этом все точки оказываются связаны с 0 регистром палитры, который и считается регистром, хранящим цвет фона. Процедура SetBkColor записывает в 0 регистр палитры цвет, указанный в параметре Color. Результат вызова этой процедуры аналогичен результату вызова процедуры SetPalette(0,Color).

5. Процедура ***SetColor(ColorNum:word)*** – объявляет цвет в регистре с номером ColorNum текущим цветом рисования.

6. Процедура ***GetDefaultPalette(var Palette: PaletteType)*** – возвращает стандартную палитру.

В качестве примера рассмотрим фрагмент программы, осуществляющий установку цвета фона и цвета рисования, а также чтение текущей палитры и вывод значений, содержащихся в регистрах палитры:

```
. . .
SetPalette(5,18); {записываем в 5-й регистр палитры
                                    цвет 010010 – ярко-зеленый}
SetColor(5); {текущий цвет рисования – цвет 5-го регистра палитры}
SetBkColor(5); {записываем в 0-й регистр палитры
                цвет фона 000101 – фиолетовый: фон сразу меняет цвет}
GetPalette(p); {читаем палитру в переменную p типа PaletteType}
for i:=0 to p.size do Write(p.colors[i]:5); {выводим цвета палитры
                                                на экран}
. . .
```

Процедуры и функции управления типом, толщиной линии и видом штриховки. Модуль Graph содержит средства, позволяющие управлять типом и толщиной линии, а также образцом закраски замкнутых контуров.

1. Процедура ***SetLineStyle(style, pattern, thickness: word)*** – устанавливает стиль style или образец pattern линии и ее толщину thickness.

Для установки стиля используют следующие константы:

```
SolidLn=0;   {сплошная}
DottedLn=1;  {точечная}
CenterLn=2;  {штрихпунктирная}
DashedLn=3;  {пунктирная}
UserBitLn=4; {определенная программистом в образце}
```

Если указаны стили 0..3, то образец игнорируется. Если установлен стиль 4, то вид линии определяется образцом.

Образец линии – 16 или 48 бит, заданных шестнадцатеричным числом, единицы и нули кодируют точки и пропуски, например: комбинация 1010 1010 1010 1010, которая соответствует шестнадцатеричному числу $AAAA, означает линию из частых точек. Образцы из 48 бит используют для определения вида линии тройной толщины.

Толщину линии можно установить одинарной и тройной, соответствующие константы выглядят следующим образом:

```
NormWidth=1; {нормальная толщина линии}
ThickWidth=3; {тройная толщина линии }
```

Например:

```
SetLineStyle(UserBitLn,$AAAA,NormWidth); {устанавливает текущий
стиль линии: частые точки, линия одинарной толщины}
```

2. Процедура ***SetFillStyle(fillstyle, color: word)*** – устанавливает образец fillstyle и цвет color штриховки замкнутых контуров. Для определения образцов штриховки можно использовать следующие константы:

EmptyFill=0;	{без штриховки – прозрачный контур}
SolidFill=1;	{сплошная заливка}
LineFill=2;	{—— – горизонтальные линии}
LtSlashFill=3;	{//// – тонкие линии}
SlashFill=4;	{//// – толстые линии}
BkSlashFill=5;	{\\\\ – толстые линии}
LtBkSlashFill=6;	{\\\\ – тонкие линии}
HatchFill=7;	{++++ – клетка горизонтальная}
XHatchFill=8;	{×××× – клетка диагональная}
InterLeaveFill=9;	{+ + + – клетка редкая}
WideDotFill=10;	{. . . . – точки редкие}
CloseDotFill=11;	{..... – точки частые}
UserFill=12;	{стиль, определенный программистом}

3. Процедура ***SetFillPattern (:FillPatternType; color:word)*** – устанавливает пользовательский образец fillPatern и цвет color штриховки. Образец штриховки задают с помощью переменной специального типа:

Type ***FillPatternType*** *= array[1..8] of byte; ...*

Переменная этого типа определяет квадрат 8×8 точек, где биту, установленному в 1, соответствует точка заданного цвета, а биту, установленному в 0 – точка цвета фона.

Например:

Const P1:FillPatternType = ($AA, $55, $AA, $55, $AA, $55, $AA, $55);

P1 определяет квадрат 8×8 точек следующего вида:

1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1

На экране при штриховке данным стилем мы будем видеть частые ромбы.

4. Процедура ***FloodFill(x, y, color: word)*** – закрашивает текущим образцом и цветом закраски, установленными процедурой SetFillStyle, область,

ограниченную контуром цвета color, внутри которой находится точка с координатами (x,y). Если контур не замкнут, то будет закрашен весь экран.

Процедуры и функции рисования. Модуль Graph предоставляет программисту большой набор процедур и функций, рисующих различные *примитивы* (простейшие фигуры, размеры и местоположение которых определяются параметрами).

1. Процедура ***PutPixel(x, y, color: word)*** – рисует точку цветом, определенным в регистре палитры с номером color, в точке экрана с координатами (x, y).

2. Функция ***GetPixel(x,y:word):word*** – возвращает номер регистра палитры, определяющего цвет точки с указанными координатами.

3. Функция ***MoveTo(x,y:word):word*** – задает положение *текущей* точки – точки, используемой в качестве исходной при рисовании линий или выводе текста.

4. Процедура ***MoveRel(dx,dy:word)*** – задает положение текущей точки относительно ее предыдущего положения.

5. Функции ***GetX:word*** и ***GetY:word*** – возвращают соответственно координаты x и y текущей точки.

6. Функции ***GetMaxX:word*** и ***GetMaxY:word*** – возвращают соответственно максимальные размеры экрана в точках для текущего режима.

7. Процедура ***Line(x1,y1,x2,y2:word)*** – рисует линию из точки (x1,y1) в точку (x2,y2).

8. Процедура ***LineTo(x,y:word)*** – рисует линию из текущей точки в точку (x,y).

9. Процедура ***LineRel(dx,dy:word)*** – рисует линию из текущей точки в точку, отстоящую от текущей на dx,dy.

10. Процедура ***Rectangle(x1,y1,x2,y2:word)*** – рисует прямоугольник по координатам левого верхнего и правого нижнего углов.

11. Процедура ***Bar(x1,y1,x2,y2:word)*** – рисует заштрихованный прямоугольник с заданными координатами текущим цветом и закрашивает его текущим образцом закраски.

12. Процедура ***Bar3D(x1,y1,x2,y2,depth:word; top: boolean)*** – рисует параллелепипед, у которого: размер передней грани определяется координатами x1, y1, x2, y2; глубина depth обычно задается равной 25% ширины, а параметр top определяет, рисовать ли верхнюю грань (да, если true, и нет, если false). Верхнюю грань не рисуют, если необходимо изобразить несколько параллелепипедов, стоящих друг на друге. Передняя грань закрашивается текущим образцом и цветом закраски.

13. Процедура ***DrawPoly(numPoints:word; var PolyPoints)*** – рисует ломаную линию. Первый параметр определяет количество вершин ломаной, а второй – координаты этих вершин в массиве элементов специального типа PointType:

```
Type PointType = record
                   x,y:word;
                 end; ...
```

Например:

```
Var MP:array[5] of PointType; {объявление массива координат точек}
...
DrawPoly(5,MP);   {рисуем ломаную линию}
```

14. Процедура ***FillPoly (numPoints:word; Var PolyPoints)*** – рисует закрашенный многоугольник с указанными вершинами.

15. Процедура ***Circle (x, y, radius:word)*** – рисует окружность заданного радиуса radius с центром в точке (x, y).

16. Процедура ***Arc (x, y, stangle, endangle, radius: word)*** – рисует дугу окружности указанного радиуса radius от угла stangle до угла endangle. Углы отсчитываются от положительного направления оси абсцисс против часовой стрелки и указываются в градусах.

Существует специальная процедура, которая позволяет определить координаты концов дуги.

17. Процедура ***GetArcCoord (Var ArcCoo:ArcCoordType)*** – возвращает координаты концов дуги через параметр-переменную специального типа ArcCoordType, который описан следующим образом:

```
Type ArcCoordType = record
          x, y,              {центр дуги}
          xstart, ystart,    {начало дуги}
          xend, yend:word    {конец дуги}
        end;
```

Например, нарисуем дугу и соединим ее концы:

```
Var ArcCoords: ArcCoordType;
  ...
Arc(100, 100, 0, 270, 30); {рисуем дугу}
GetArcCoords(ArcCoords); {определяет координаты концов дуги}
with ArcCoords do
     Line(Xstart, Ystart, Xend, Yend); {рисуем соединяющую линию}
```

18. Процедура ***Pieslice (x, y, stangle, endangle, radius:word)*** – рисует заштрихованный сектор или окружность (если конечный угол равен начальному). Углы отсчитываются от положительного направления оси абсцисс против часовой стрелки и указываются в градусах.

19. Процедура ***Ellipse (x, y, stangle, endangle, radiusX, radiusY: word)*** – рисует эллипс или его дугу.

20. Процедура ***Sector (x, y, stangle, endangle, radiusX, radiusY: word)*** – рисует заштрихованный эллипс или его сектор.

21. Процедура ***FillEllipse (x, y, radiusX, radiusY: word)*** – рисует заштрихованный эллипс.

Процедуры и функции управления текстом. Текст в графическом режиме может быть выведен процедурами ***Write*** и ***WriteLn***, но при этом управление выполняется аналогично текстовому режиму. Специально для работы с текстом в графическом режиме модуль Graph предлагает следующие процедуры и функции.

1. Процедура ***SetTextStyle(font,direction,charsize:word)*** – устанавливает текущий шрифт font, направление вывода строки direction и размер шрифта charsize.

Для установки шрифта используют специальные константы:

```
DefaultFont=0;      {обычный}
TriplexFont=1;      {полужирный}
SmallFont=2;        {мелкий}
SanSerifFont=3;     {прямой}
GothicFont=4;       {готический}
```

Направление вывода может быть либо горизонтальным, либо вертикальным:

```
HorizDir=0;         {горизонтальное направление}
VertDir=1;          {вертикальное направление}
```

Размер шрифта указывается значением, на которое необходимо умножить исходный размер шрифта (8×8 точек). Таким образом, шрифт можно увеличивать в целое число раз.

2. Процедура ***SetTextJustify(horiz, vert :word)*** – устанавливает способ выравнивания для горизонтального goriz и вертикального vert направлений вывода. Способ выравнивания определяется относительно некоторой точки с помощью специальных констант:

```
LeftText=0;     {точка слева от текста}
CenterText=1;   {точка по центру текста при горизонтальном направлении вывода}
RightText=2;    {точка справа от текста}
BottomText=0;   {точка под текстом}
CenterText=1;   {точка по центру текста при вертикальном направлении вывода}
TopText=2;      {точка над текстом}
```

3. Функция ***TextHeight(st:string):word*** – возвращает высоту строки st в точках.

4. Функция ***TextWidth(st:string):word*** – возвращает ширину строки st в точках.

5. Процедура ***OutText (st:string)*** – выводит строку, выравнивая ее заданным способом относительно текущей точки.

6. Процедура ***OutTextXY(x,y:word; st:string)*** – выводит строку st, выравнивая ее в соответствии с установленными режимами вертикального и горизонтального выравнивания относительно точки (x, y).

Процедуры управления окнами, страницами. Помимо управления цветом, рисования примитивов и вывода текстов модуль Graph содержит специальные процедуры управления окнами и страницами.

1. Процедура ***ClearDevice*** – очищает экран.

2. Процедура ***SetViewPort (x1, y1, x2, y2:word; clip:boolean***) – устанавливает окно. Параметры x1, y1, x2, y2 определяют размеры окна, а параметр clip – будут ли видимы фрагменты рисунка, вышедшие за границы окна.

3. Процедура ***GetViewSettings(Var win:ViewPortType)*** – возвращает параметры последнего окна через параметр-переменную win типа ViewPortType:

```
Type ViewPortType= record
                     x1,y1,x2,y2:word; {координаты}
                     Clip:boolean;  {отсечение}
                   end; ...
```

4. Процедура ***ClearViewPort*** – очищает окно, связывая все точки этого окна с 0 регистром палитры.

5. Процедура ***SetActivePage(pageNumber:word)*** – переключает активную страницу – делает активной страницу с указанным номером pageNumber. На активной странице строится нужное изображение, в то время как высвечиваться продолжает прежняя страница. Когда изображение готово, осуществляют переключение видимости страниц и появляется новое изображение.

6. Процедура ***SetVisualPage(pageNumber:word)*** – переключает видимость страниц.

Например:

```
. . .
SetVisualPage(0);  {установили видимой 0 страницу}
SetActivePage(1);  {установили активной 1 страницу}
Rectangle(10, 60, 30, 80);  {вывели прямоугольник}
Readln;                     {убедились, что он не видим}
SetVisualPage(1);  {установили видимой 1 страницу –
                            прямоугольник стал видимым}
. . .
```

Процедуры и функции создания движущихся изображений. Модуль Graph предоставляет некоторые ресурсы для создания движущихся изображений.

1. Процедура ***GetImage(x1,y1,x2,y2:word;var p:pointer)*** – сохраняет в памяти прямоугольный фрагмент изображения. Параметры x1,y1,x2,y2 определяют прямоугольник. Параметр p должен содержать адрес буфера в памяти, куда копируется изображение. Размер этого буфера определяется с помощью специальной функции ImageSize.

2. Функция ***ImageSize(x1,y1,x2,y2:word):word*** – возвращает размер буфера, необходимого для сохранения прямоугольного фрагмента изображения.

3. Процедура ***PutImage(x,y:word; var p:pointer; bitblt:word)*** – выводит сохраненное изображение на экран в прямоугольную область с координатами верхнего левого угла (x, y). Параметр p должен содержать адрес буфера. Способ наложения bitblt определяется специальными константами:

```
NormalPut=0;    {наложение со стиранием}
XorPut=1;       {побитное «исключающее или»}
OrPut=2;        {побитное «или»}
AndPut=3;       {побитное «и»}
NotPut=4;       {побитное «не»}
```

Так, если цвет точки был розовым ($12 = 1100_2$) и на нее накладывается точка того же цвета ($12 = 1100_2$), то при разных способах наложения и стандартной палитре мы получим

NormalPut – 1100 – точка розового цвета;
XorPut – 0000 – точка черного цвета (цвета фона);
OrPut – 1100 – точка розового цвета;
AndPut – 1100 – точка розового цвета;
NotPut – 0011 – точка голубого цвета.

Буфер под образ обычно размещается в динамической памяти. Например:

```
. . .
Size:= ImageSize(0,0,99,99);
GetMem(p,Size); {выделяем память под буфер для сохранения изображения}
x:=0;
y:=0;
Pieslice(50,50,0,360,47); {рисуем закрашенный круг}
GetImage(0,0,99,99,p^); {сохраняем изображение в памяти}
repeat
    PutImage(x,y,p^,0); {выводим изображение на экран, стирание
                         выполняется чистым краем образа}
    delay(100);         {задержка}
    x:=x+3;
    y:=y+1;
```

```
until x>getmaxx-50; {выход по достижении конца экрана}
FreeMem(p, Size);          {освобождаем динамический буфер}
...
```

8.6. Практикум. Построение графиков и диаграмм

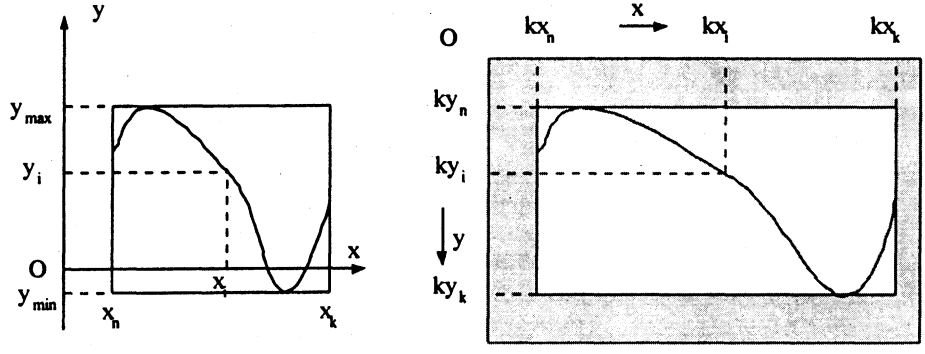
Результаты решения многих задач целесообразно представлять в виде графиков. Задачи построения графиков существенно различаются по исходной постановке. Так, может потребоваться график одной функции или сразу нескольких функций в одном или различных масштабах. Функции могут быть заданы аналитически или таблично, причем при табличном задании может потребоваться интерполяция функции. График может строиться в обычном или логарифмическом масштабе, включать или нет координатную сетку, включать или нет оси координат и т.п. Однако основные принципы построения графиков при этом остаются неизменными.

Для построения графика функции необходимо каждой точке графика поставить в соответствие точку на экране. Координаты точки на экране определяются относительно координат области графика с учетом масштаба и того, что ось Oy экрана направлена вниз (рис. 8.6).

Масштабы по осям x и y рассчитывают исходя из интервалов возможных значений и размеров области графика на экране:

$$m_x = \frac{kx_k - kx_n}{x_k - x_1}, \qquad m_y = \frac{ky_k - ky_n}{y_{max} - y_{min}},$$

где kx_n, ky_n, kx_k, ky_k – координаты области окна графика на экране; y_{max},



а *б*

Рис. 8.6. Определение координат точек графика:
а – график; *б* – его отображение в окне на экране

y_{min} – максимальное и минимальное значения функции на заданном интервале; x_1, x_k – первое и последнее значения аргумента, входящие в заданный интервал.

Координаты точек графика рассчитывают следующим образом:

$$kx_i = \lceil (x_i - x_1) \times m_x \rceil + kx_n,$$
$$ky_i = \lceil (y_{max} - y_i) \times m_y \rceil + ky_n,$$

где $\lceil \rceil$ – означает взятие целой части числа.

График строится как набор линий или ломаная линия, проходящая через все рассчитанные точки.

Расчет координат для построения координатной сетки выполняется аналогично.

Рассмотрим построение графика на конкретном примере.

Пример 8.5. Разработать программу, которая строит график функции y=cos(x+2)/2 в заданном окне с определенным количеством точек на указанном интервале и наносит на график координатную сетку, состоящую из заданного количества вертикальных и горизонтальных линий.

Ниже представлен текст программы с подробными комментариями. Для сокращения размера программы все редко изменяемые значения параметров графика заданы константами.

```
Program Gr;
 Uses Crt,Graph;
 Const
      n=5;    {количество позиций на число}
      m=2;    {размер мантиссы при выводе значений}
      k=100;          {количество точек просчета}
      nx=5; ny=5;     {количество линий сетки по x и y}
      kxn=60;  kxk=600;
      kyn=45; kyk=350;  {параметры окна}
 Type  arr = array[1..100 ] of real;
       ari = array[1..100] of integer;
 Var
   gd,gm,i:integer;  {тип и режим адаптера}
   x, y:arr;         {массивы для значений функции и аргумента}
   kx, ky:ari;       {массивы для координат точек по x и y}
   ymin, ymax:real;  {экстремальные значения y}
   dx,dy:real;       {шаг сетки по x и y на графике}
   dkx,dky:integer;  {шаг сетки по x и y на экране}
   mx,my:real;       {масштабные коэффициенты}
   st:string[5];     {рабочая строка}
   h, xn, xk: real;  {шаг, интервал по оси x}
```

```
Begin
 ClrScr;
 Write('Введите начало и конец интервала:');
 ReadLn(xn,xk);
 h:=(xk-xn)/(k-1);          {определяем шаг по оси x}
 x[1]:=xn;
 ymin:=1e30;
 ymax:=-1e10;
 for i:=1 to k do   {табулируем функцию и ищем экстремумы}
   begin
     y[i]:=cos(x[i]+2)/2;
     if y[i]>ymax  then  ymax:=y[i];
     if y[i]<ymin  then  ymin:=y[i];
     if i<>100  then x[i+1]:=x[i]+h;
   end;
 mx:=(kxk-kxn)/(x[k]-x[1]);          {определяем масштаб по оси x}
 my:=(kyk-kyn)/(ymax-ymin); {определяем масштаб по оси y}
 for i:=1 to k do                    {определяем координаты точек}
    begin
       kx[i]:=round((x[i]-x[1])*mx)+kxn;
       ky[i]:=round((ymax-y[i])*my)+kyn;
    end;
 gd:=detect;
 InitGraph(gd,gm,'');     {инициализируем графический режим}
 SetColor(4);             {текущий цвет – красный}
 OutTextXY(180,20,'Y=cos(x+2)/2'); {выводим заголовок}
 SetColor(17);            {цвет рисования – голубой}
 SetBKColor(7);           {на сером фоне}
 Rectangle(kxn,kyn,kxk,kyk);      {рисуем прямоугольник для вывода
                                        графика}
 SetColor(4);                     {текущий цвет – красный}
 for i:=1 to k-1 do               {выводим график}
        Line (kx[i],ky[i],kx[i+1],ky[i+1]);
 dkx:=round((kxk-kxn)/nx);     {определяем шаг сетки по x}
 dky:=round((kyk-kyn)/ny);     {определяем шаг сетки по y}
 SetColor(17);                 {текущий цвет – светло-синий}
 for i:=1 to nx do        {рисуем сетку, параллельную оси x}
        Line(kxn+dkx*i,kyn,kxn+dkx*i,kyk);
for i:=1 to ny do         {рисуем сетку, параллельную оси y}
   Line(kxn,kyk-dky*i,kxk,kyk-dky*i);
 dx:=(x[k]-x[1])/nx;     {определяем шаг для сетки по x}
 dy:=(ymax-ymin)/ny;     {определяем шаг для сетки по y}
 SetTextJustify(1,2);    {выравнивание «по центру снизу»}
```

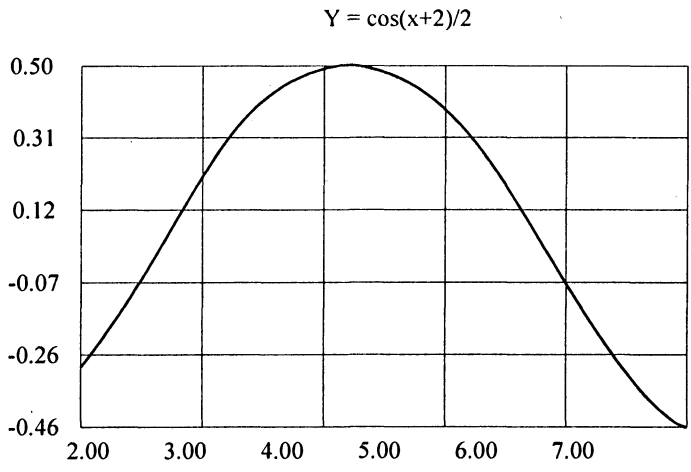


Рис. 8.7. Результат работы программы

```
  for i:=1 to nx+1 do      {выводим значения аргумента}
    begin
      Str((x[1]+dx*(i-1)):n:m,st);{преобразуем число в строку}
      OutTextXY(kxn+dkx*(i-1), kyk+6,st);  {вывод значений под линией сетки}
    end;
  SetTextJustify(2,1);      {выравнивание «слева по центру»}
  for i:=1 to ny+1 do       {выводим значения функции}
    begin
      Str((ymin+dy*(i-1)):n:m,st);{преобразуем число в строку}
      OutTextXY(kxn-6,kyk-dky*(i-1),st); {вывод слева от оси y}
    end;
  ReadLn; {ожидаем нажатия ENTER}
  Closegraph;
End.
```

Результатом работы программы является график функции на заданном интервале изменения аргумента, представленный на рис. 8.7.

Помимо графиков для представления результатов также могут использоваться диаграммы и гистограммы различных видов. Рассмотрим еще один пример.

Пример 8.6. Разработать программу построения круговой диаграммы по заданным значениям не более 12, которые должны выводиться рядом с соответствующим сектором диаграммы.

Круговая диаграмма рисуется как совокупность закрашенных секторов круга. Угол сектора пропорционален доле соответствующего значения в об-

щей сумме значений. Надпись должна выводиться напротив биссектрисы угла сектора соответственно слева или справа диаграммы.

Ниже приведена программа.

```
Program difgramma;
 Uses Graph;
 Const
    kmax=12;   {максимальное количество значений функции}
    r=150;     {радиус диаграммы}
    n=5;       {ширина поля вывода}
    m=2;       {размер дробной части числа}
 Type
      mas=array[1..kmax+1] of integer;
      mas1=array[1..kmax] of real;
 Var
    f:mas1;        {массив значений функции}
    alfa:mas;      {массив значений углов диаграммы}
    driver,err,    {тип и режим работы адаптера}
     k,            {фактическое количество значений функции}
    bet,  {величина угла, образованного радиусом и биссектрисой сек-
                              тора диаграммы}
    y,x,       {координаты точки, являющейся центром дуги сектора}
    x1,y1,     {координаты начальной точки вывода надписи}
    xn,yn,     {координаты центра диаграммы}
    i:integer;
    st:string[5]; {строка для хранения выводимой надписи}
    s:real;        {сумма значений функции}
begin
   WriteLn('Введите количество точек (от 1 до',kmax:3,')');
   ReadLn(k);
   {вводим значения функций и определяем их сумму}
   s:=0;
   for i:=1 to k do
    begin
      WriteLn('Введите',i:3,'-е значение функции');
      ReadLn(f[i]);
      while f[i]<0 do
        begin
          WriteLn('Недопустимое значение, повторите ввод');
          WriteLn('Введите',i:3,'-е значение функции');
           ReadLn(f[i]);
        end;
```

```
      s:=s+f[i];
    end;
   if  s=0  then
      begin
         WriteLn('Все значения нулевые.');
         ReadLn;
         halt(1);              {выход по ошибке}
      end;
  {инициализируем графический режим}
  driver:=detect;
  InitGraph(driver,err,'');
  SetBkColor(15); {белый цвет фона}
  SetPalette(1,0);
  SetColor(1);  {черный цвет рисования}
  {вычисляем координаты центра диаграммы}
  xn:=GetMaxX div 2;
  yn:=GetMaxY div 2;
  {рассчитываем значения углов секторов диаграммы}
  alfa[1]:=0;
  for i:= 2 to k+1 do
   begin
     if i<>k+1 then  alfa[i]:=alfa[i-1]+round(f[i-1]/s*360)
               else  alfa[k+1]:=360;
     SetFillStyle(i mod 10, i); {установим тип и цвет закраски сектора}
     Pieslice(xn,yn,alfa[i-1],alfa[i],r); {изобразим сектор}
     {вычисляем начальные координаты вывода надписей}
      bet:=(alfa[i-1]+alfa[i]) div 2;
      x:=xn+round(r*cos(bet*pi/180));
      y:=yn-round(r*sin(bet*pi/180));
      if ((bet>=0)and(bet<=90))or((bet>=270)and(bet<=360))
            then x1:=x+10  else x1:=x-8*n-10;
      if ((bet>=0)and(bet<=180))   then y1:=y-15
                                   else  y1:=y+7;
     Str(f[i-1]:n:m,st);      {преобразуем значение в строку}
     OutTextXY(x1,y1,st);     {выведем надпись}
   end;
  ReadLn;
  CloseGraph;
End.
```

Результатом работы программы является изображение круговой диаграммы, приведенное на рис. 8.8.

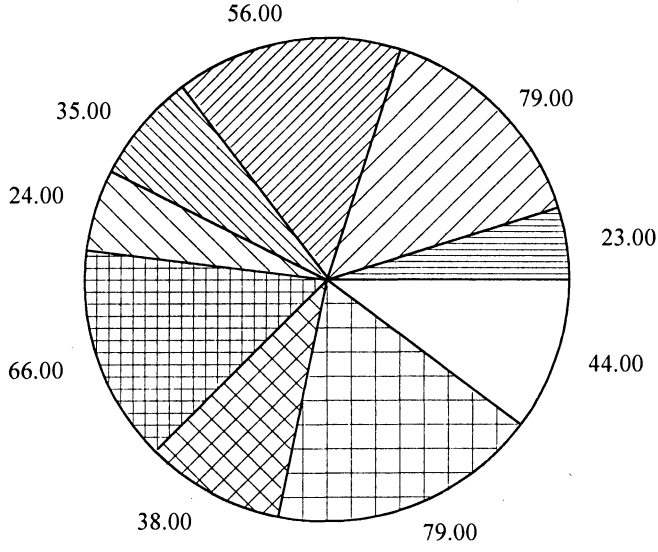


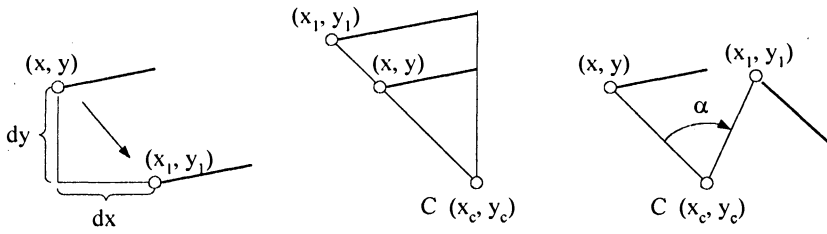
Рис. 8.8. Результат работы программы построения круговой диаграммы

8.7. Практикум. Создание движущихся изображений

Движение на экране создается по принципу мультипликации: на экран с соответствующей задержкой выводят последовательность кадров с небольшими изменениями положения «движущихся» объектов или объектов «фона», если эффект движения достигается изменением фона.

Сам «перемещаемый» объект может быть двумерным (плоским) и трехмерным (пространственным), причем движение может осуществляться по двум типам траектории: лежащей в плоскости экрана или выходящей за нее. Из аналитической геометрии известны формулы, по которым можно, зная закон движения, определить изменения положения каждой точки изображения движущегося объекта на экране. Проекции трехмерных объектов при движении изменяются достаточно сложным образом, и в то же время ничего принципиально нового в программировании движения трехмерных объектов по сравнению с двумерными не существует, поэтому в настоящем учебнике эти вопросы рассматриваться не будут.

Движение плоских объектов. Любое сложное движение плоских объектов на экране складывается из базовых: перемещения, масштабирования и поворота. Формулы пересчета координат изображений для базовых видов движения следующие.



а *б* *в*

Рис. 8.9. Элементарные изменения изображения:
а – перемещение; *б* – масштабирование; *в* – поворот

Перемещение (рис. 8.9, *а*)

$$x_1 = x + dx,$$
$$y_1 = y + dy,$$

где x, y – исходные координаты точки; x_1, y_1 – координаты точки после перемещения; dx, dy – смещения по оси x и y соответственно.

Масштабирование относительно точки C (x_c, y_c) (рис. 8.9, *б*):

$$x_1 = (x - x_c)\, M_x + x_c\,,$$
$$y_1 = (y - y_c)\, My + y_c\,,$$

где M_x, M_y – масштабы по x и y соответственно; x_c, y_c – координаты точки, относительно которой ведется масштабирование.

Поворот относительно точки C с координатами (x_c, y_c) (рис. 8.9, *в*):

$$x_1 = (x - x_c)\cos\alpha + (y - y_c)\sin\alpha + x_c\,,$$
$$y_1 = (y - y_c)\cos\alpha - (x - x_c)\sin\alpha + y_c\,,$$

где α – угол поворота.

Пример 8.7. Разработать программу, которая демонстрирует на экране движение прямоугольника: прямоугольник улетает от нас к некоторой точке горизонта, одновременно вращаясь вокруг своей оси.

Обобщенный алгоритм последовательного показа кадров данной задачи выглядит следующим образом.

Начало:
Установить точку отсчета координат (условное время).
Рассчитать координаты квадрата.
Цикл-пока не истекло время или не нажата любая клавиша
Установить цвет рисования.
Изобразить квадрат.
Приостановить выполнение программы на время просмотра кадра.

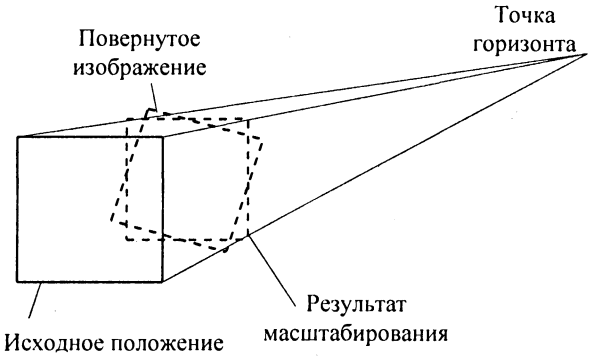


Рис. 8.10. Разложение движения

Установить в качестве цвета рисования цвет фона.
Изобразить квадрат цветом фона – стереть.
Изменить точку отсчета (условное время).
Пересчитать координаты квадрата.
все-цикл
Конец.

В а р и а н т 1. Координаты вершин квадрата будем хранить в специальных массивах x, y, а квадрат рисовать линиями, проводя их из одной вершины в другую. Изображение квадрата будет осуществлять специальная процедура Square.

Пересчет координат вершин реализуем через разложение «движения» прямоугольника на элементарные составляющие (рис. 8.10): эффект удаления от зрителя получим, используя масштабирование относительно точки горизонта, эффект вращения – за счет поворота вокруг геометрического центра.

Вначале определим координаты вершин и центра квадрата после масштабирования, а затем координаты вершин после поворота вокруг центра квадрата. Ниже приведена соответствующая программа:

```
Program ex;
  Uses Crt,Graph;
  Const
        r=100;
  Type
     mas=array[1..4] of integer;
  Var
     x, y, x1, y1:mas;
     gd,gm,xn,yn,xc,yc,i,j,k1:integer;
     t,k:real;
```

```
{изображение квадрата по координатам его вершин}
Procedure Square(x,y:mas);
   Begin
      Line(x[1],y[1],x[2],y[2]);
      Line(x[2],y[2],x[3],y[3]);
      Line(x[3],y[3],x[4],y[4]);
      Line(x[4],y[4],x[1],y[1]);
   End;
{определение координат повернутой точки}
Procedure Pow(xc,yc,x,y:integer;t:real;var x1,y1:integer);
   Begin
     x1:=xc+round((x-xc)*cos(t))+round((y-yc)*sin(t));
     y1:=yc+round((y-yc)*cos(t))-round((x-xc)*sin(t));
   End;
{определение координат точки после масштабирования}
Procedure Massch(xc,yc,x,y:integer;k:real;var x1,y1:integer);
 Begin
  x1:=round(x*k+(1-k)*xc);
  y1:=round(y*k+(1-k)*yc);
 End;
{основная программа}
Begin
  gd:=detect;
  InitGraph(gd,gm,'d:\bp\bgi');
  SetColor(2);
  xn:=GetMaxX div 4;
  yn:=GetMaxY div 3*2;
  xc:=GetMaxX-xn;
  yc:=GetMaxY-yn;
 {расчет начальных координат вершин квадрата}
 x[1]:=xn-r;   y[1]:=yn-r;
 x[2]:=xn+r;   y[2]:=yn-r;
 x[3]:=xn+r;   y[3]:=yn+r;
 x[4]:=xn-r;   y[4]:=yn+r;
 k:=0.99;
 t:=0;
 {покадровый вывод на экран}
 while (t<1) and not KeyPressed do
   begin
     SetColor(2);  {установим цвет рисования}
     Square(x,y);  {нарисуем квадрат}
     t:=t+0.001;   {увеличим угол поворота}
```

```
    {масштабирование}
    for j:=1 to 4 do   {определим координаты вершин}
          Massch(xc,yc,x[j],y[j],k,x1[j],y1[j]);
    Massch(xc,yc,xn,yn,k,xn,yn); {определим координаты центра}
    {поворот}
    for j:=1 to 4 do   {определим координаты вершин}
       Pow(xn,yn,x1[j],y1[j],-t,x1[j],y1[j]);
    for j:=1 to 1500 do Delay(1000); {или NewDelay см. параграф 8.3}
    SetColor(0);     {установим цвет рисования – цвет фона}
    Square(x,y);     {стираем квадрат}
    x:=x1;           {заменим координаты вершин на новые}
    y:=y1;
  end;
 CloseGraph;
End.
```

Недостатком данного способа является то, что квадрат на экране через несколько кадров уже выглядит не квадратным. Это происходит вследствие накопления ошибки округления при пересчете координат вершин. Избежать этого можно двумя способами:

1) все вычисления выполнять в вещественной арифметике и координаты также хранить как вещественные числа, преобразуя их в целые непосредственно перед использованием в процедуре рисования;

2) пересчитывать координаты не всех вершин, а какой-нибудь одной и центра квадрата, восстанавливая квадрат по одной вершине и положению центра квадрата.

Способы являются взаимодополняющими, поэтому используем оба.

В а р и а н т 2. Для упрощения вычислений вместо массивов, хранящих координаты вершин квадрата, будем использовать смещения этих вершин относительно центра (рис. 8.11). Соответственно процедура рисования квадрата Square1 должна использовать именно эти параметры. Также учтем, что при масштабировании изменяются размер диагонали и положение центра, а при повороте – смещения вершин относительно центра. Ниже представлен текст программы.

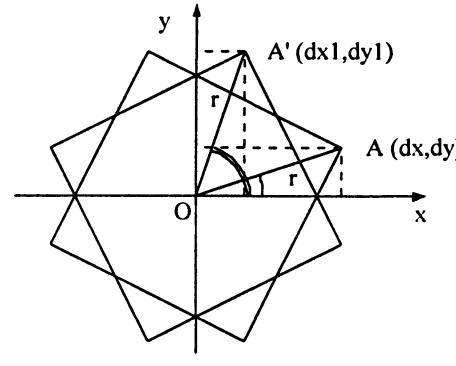


Рис. 8.11. Два соседних кадра при повороте

```
Program ex;
 Uses Crt,Graph;
 Const r:real=100;   {размер половины стороны квадрата}
 Var
    x, y, dx, dy, dx1, dy1, xn, yn, xc, yc, xn1, yn1:real;
    gd,gm,i,j:integer;
    t,k:real;   {угол поворота и масштаб}
 {изображение квадрата}
 Procedure Square1(x,y,dx,dy:integer);
  Begin
   Line(x+dx,y+dy,x-dy,y+dx);
   Line(x-dy,y+dx,x-dx,y-dy);
   Line(x-dx,y-dy,x+dy,y-dx);
   Line(x+dy,y-dx,x+dx,y+dy);
  End;
 {основная программа}
 Begin
   gd:=detect;
   InitGraph(gd,gm,'d:\bp\bgi');
   {устанавливаем начальную и конечную точки}
   xn:=GetMaxX div 4;
   yn:=GetMaxY div 3*2;
   xc:=GetMaxX-xn;
   yc:=GetMaxY-yn;
   {определяем начальные значения}
   dx:=r;
   dy:=0;
   k:=0.95;
   t:=0;
 {покадровый вывод на экран}
   while (t<100) and not KeyPressed do
    begin
      SetColor(2);   {выводим кадр}
      Square1(round(xn), round(yn), round(dx), round(dy));
      {масштабирование}
      xn1:= xn*k+(1-k)*xc;
      yn1:=yn*k+(1-k)*yc;
      r:= k*r;
      {поворот}
      t:=t+1;     {увеличиваем угол поворота}
      dx1:=r*cos(t);
      dy1:= r*sin(t);
      for j:=1 to 5000 do Delay(1000);  {приостановка}
```

```
    SetColor(0);  {стираем кадр}
    Square1(round(xn), round(yn), round(dx), round(dy));
    dx:=dx1;  {заменяем параметры квадрата}
    dy:=dy1;
    xn:=xn1;
    yn:=yn1;
  end;
 CloseGraph;
end.
```

Прямая запись в видеобуфер. При программировании на экране движения объектов критичным является время перезаписи изображения: именно из-за большого времени перезаписи движение получается прерывистым. Для уменьшения этого времени при программировании в MS DOS часто используют прямую запись информации в видеобуфер.

Как указывалось в параграфе 8.4, формат информации в видеобуфере зависит от используемого графического режима. При использовании младших режимов VGA, на которые рассчитан Borland Pascal, видеобуфер содержит 4 бита на каждую точку, причем биты расположены в параллельных битовых плоскостях и доступ к ним напрямую существенно затруднен (программирование таких операций обычно выполняется на ассемблере). Однако существует режим VGA (режим 19: 200*320 точек 256 цветов из палитры 262 144 цвета), при котором каждой точке соответствует байт (8 бит) в видеобуфере. Именно этот режим и используется, если возникает необходимость программировать сложное движение с использованием прямой записи в видеобуфер.

Пример 8.8. Разработать программу, обеспечивающую вывод «бегущей» строки. Направление движения строки по экрану – вверх-вниз.

Для создания изображения используем возможности модуля Graph, затем перепишем изображение из видеопамяти в буфер, расположенный в динамической памяти, и перейдя в режим 200*320, организуем циклический вывод изображения напрямую в видеобуфер. Стирание старого изображения будем выполнять чистой кромкой образа (образ «не прозрачный»).

Переход в другой, не поддерживаемый Borland Pascal графический режим, будем осуществлять, используя ресурсы модуля Dos, описанные далее.

```
Program ex;
 Uses Graph,Crt,Dos;
 Type
   ScreenType=array [0..199,0..319] of byte; {массив для хранения
                                образа экрана – формат видеобуфера}
   ImageType=array[0..999] of byte; {развертка изображения}
   ScrTypePtr=^ScreenType;  {указатель на массив образа экрана}
   ImageTypePtr=^ImageType;  {указатель на развертку изображения}
```

```
  {процедура установки режима с номером mode}
  Procedure SetBIOSMode(mode:byte);
    Var r:registers;
    Begin
        r.AL:=mode;        {запись номера режима в регистр AL}
        r.AH:=0;           {запись номера функции в регистр AH}
        intr($10,r);       {вызов 0-й функции 10-го прерывания}
    End;
{основная программа}
Var
     Driver,Mode:Integer;
     s:string;
     i,j,n,m,l,y,dy:integer;
     Active_Ptr:ScrTypePtr;  {указатель на тип "образ экрана"}
     Image:ImageTypePtr;     {указатель на развертку изображения}
Begin
 {формирование изображения и сохранение его в памяти}
  Driver:=Detect;  InitGraph(Driver,Mode,'');
  s:='ABCDEF';
  SetColor(4);  SetTextStyle(GothicFont,HorizDir,3);
  OutTextXY(2,2,s);
  n:=TextHeight(s)+3;
  m:=TextWidth(s)+3;
  GetMem(Image,(n+1)*(m+1));  {получение памяти для записи
                                                  изображения}
  l:=0;
  for i:=0 to n do
     for j:=0 to m do
         begin
           image^[l]:=Lo(GetPixel(j,i));  {запись изображения в буфер}
           l:=l+1;
         end;
  CloseGraph;
  {запись изображения «напрямую» в видеобуфер}
  SetBIOSMode($13);   {установка 19-го графического режима}
  Active_Ptr:=Ptr($A000,0);  {стандартный адрес видеобуфера}
  y:=0;
  dy:=1;
 {покадровый вывод изображения}
  repeat
    {побайтная запись изображения в видеобуфер}
    l:=0;
    for i:=0 to n do
```

```
      for j:=0 to m do
        begin
          Active_Ptr^[y+i+10,j+20]:=image^[l];
          l:=l+1;
        end;
    for i:=1 to 1000 do Delay(3000);   {задержка}
    Inc(y,dy);   {смещение изображения}
    if (y>120) or (y<0) then dy:=-dy; {организация  колебательного
                                                        движения}
  until KeyPressed;
  SetBIOSMode(3);      {возврат к стандартному текстовому режиму}
End.
```

8.8. Взаимодействие с драйвером мыши

Мышь – манипулятор, движение которого по столу или другой поверхности преобразуется в перемещение специального курсора мыши на экране. С ее помощью мы можем «указать» на какую-либо область экрана и, нажимая клавиши мыши, заказать некоторую обработку.

Для управления мышью программы, написанные на Borland Pascal, используют драйвер мыши, предоставляемый BIOS. Вызов этого драйвера осуществляется через инициацию прерывания с номером $33_{16} = 51_{10}$ (**int 33h** – на ассемблере).

Драйвер мыши реализует выполнение основных операций с мышью:

- инициализирует мышь (с проверкой наличия);
- устанавливает курсор мыши в заданное место экрана;
- определяет местоположение курсора мыши и состояния ее клавиш (нажаты или нет);
- управляет видимостью курсора мыши.

К сожалению, библиотеки Borland Pascal не включают функций обращения к драйверу мыши, и его вызов приходится осуществлять через процедуру активизации обработчиков прерываний, определенную в модуле Dos.

Процедура ***Intr(numInt:byte; Var Regs:Register)*** – активизирует обработчик прерывания с номером numInt. Через параметр Regs типа Registers организуется доступ к содержимому регистров (внутренней памяти) процессора:

```
Type Registers = record  case Integer of
                           0: (AX,BX,CX,DX,BP,SI,DI,DS,ES, Flags: word);
                           1: (AL,AH,BL,BH,CL,CH,DL,DH:byte);
                 end; ...
```

Обмен данными между программой и драйвером мыши выполняется через регистры, указанные в описании драйвера. Так, номер вызываемой функ-

ции помещается в регистр AX, а для передачи или получения дополнительной информации используются регистры CX, DX.

Ниже приводится текст модуля, содержащего ресурсы, которые обеспечивают доступ к драйверу мыши.

```
Unit Mouse;
Interface
  Uses Dos;
  Function ResetMouse:Boolean; {проверить наличие}
  Procedure ShowMouseCursor;  {показать курсор мыши}
  Procedure HideMouseCursor;  {спрятать курсор мыши}
  {прочитать состояние мыши}
  Procedure ReadMouseState(Var x, y:integer;{координаты мыши}
                    Var LeftButton,          {нажата левая клавиша}
                        MiddleButton,        {нажата средняя клавиша}
                        RightButton:boolean); {нажата правая клавиша}
  Procedure MoveMouseCursor(x,y:integer); {установить курсор мыши
                                    в точку с заданными координатами}
Implementation
  {проверить наличие}
  Function ReSetMouse:Boolean;
    Var r:Registers;
    Begin
        r.AX:=0;
        intr($33,r);
        ReSetMouse:=r.AX=$FFFF;
    End;
  {показать курсор мыши}
  Procedure ShowMouseCursor;
    Var r:Registers;
    Begin
       r.AX:=1;
       intr($33,r);
    End;
  {спрятать курсор мыши}
  Procedure HideMouseCursor;
    Var r:Registers;
    Begin
        r.AX:=2;
        intr($33,r);
    End;
```

```
    {прочитать состояние мыши}
  Procedure ReadMouseState(Var x,y:integer;{координаты мыши}
    Var LeftButton,                {нажата левая клавиша}
    MiddleButton,                  {нажата средняя клавиша}
    RightButton:boolean);          {нажата правая клавиша}
   Var r:Registers;
   Begin
      r.AX:=3;
      intr($33,r);
      x:=r.CX;
      y:=r.DX;
      LeftButton:=(r.BX AND 1)<>0;
      RightButton:=(r.BX AND 2)<>0;
      MiddleButton:=(r.BX AND 4)<>0;
   End;
     {установить курсор мыши}
  Procedure MoveMouseCursor(x,y:integer);
   Var r:Registers;
   Begin
        r.AX:=4;
        r.CX:=x;
        r.DX:=y;
        intr($33,r);
   End;
End.
```

Разработка программ, использующих мышь, имеет свои особенности. В отличие от клавиатуры, ввод информации с которой осуществляется пользователем по запросу программы, состояние мыши (перемещение и положение клавиш) может изменяться пользователем в любой момент времени, независимо от программы. Следовательно, программы, работающие с мышью, должны *циклически опрашивать* состояние мыши и при его изменении осуществлять требуемые операции.

На рис. 8.12 показано, как программа определяет моменты нажатия и отпускания клавиш (при выполнении опроса, если клавиша нажата, переменной Left присваивается значение true, а если – не нажата, то false).

Учитывать приходится также режим экрана, который использует программа, так как координаты курсора мыши на экране и в текстовом и в графическом режимах определяются в пикселях.

Взаимодействие с мышью в текстовом режиме. Координаты мыши при работе с ней в текстовом режиме необходимо пересчитывать, причем независимо от реального размера знакоместа считается, что оно имеет размер 8×8 пикселей.

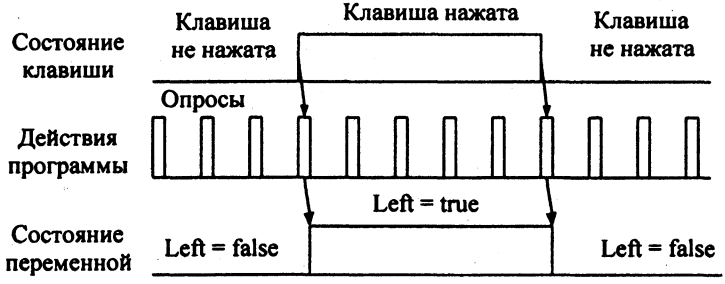


Рис. 8.12. Циклический опрос состояния мыши для фиксации моментов нажатия и отпускания клавиши мыши

Пример 8.9. Разработать программу, демонстрирующую особенности использования мыши в текстовом режиме. При нажатии левой клавиши мыши программа должна высвечивать координаты точки. Выход осуществить при нажатии левой клавиши мыши, когда ее курсор находится в окне «Конец» (рис. 8.13).

Управление разрабатываемой программой будет реализовано только посредством мыши. Вначале проверим наличие мыши, нарисуем окно Конец («кнопка»), установим курсор в левый верхний угол экрана. Затем будем отслеживать перемещение курсора мыши и нажатие ее левой клавиши. Если клавиша нажата, то определяем местоположение курсора, проверяем, не находится ли он над окном Конец, и если нет, то выводим его координаты. После этого ожидаем отпускания клавиши, чтобы повторно не выводить координаты курсора.

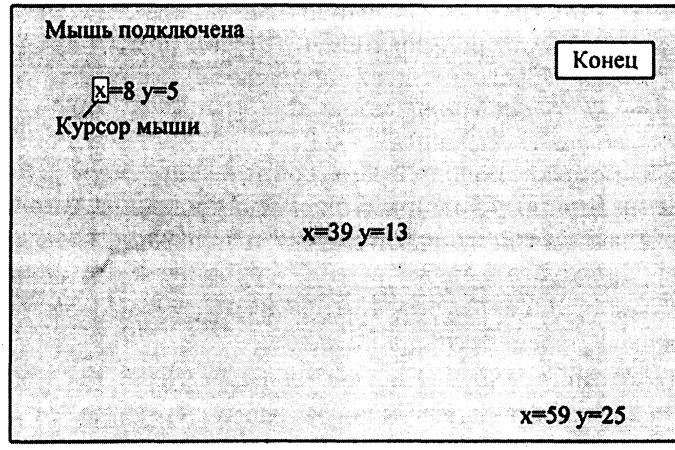


Рис. 8.13. Вид экрана в процессе работы программы

```
Program ex;
Uses Crt,Mouse;
Var x,y,xt,yt:integer;      l,m,r:boolean;
    exit_m:boolean;
Begin  Clrscr;
       if not ReSetMouse then   {проверка наличия мыши}
          begin
               WriteLn('Мышь не загружена');
               Halt(1);
           end
       else WriteLn('Мышь подключена.');
       ShowMouseCursor;  {покажем курсор: курсор устанавливается
                                                  в центр экрана}
       MoveMouseCursor(0,0);  {поместим курсор в левый
                                                  верхний угол экрана}
       Window(70,1,80,3);   {нарисуем окно-кнопку «Конец»}
       Textattr:=16*1+14;       Clrscr;
       Gotoxy(2,2);       Write('Конец');
       Window(1,1,80,25);     Gotoxy(1,3);
       Textattr:=5*16+9;
       repeat     {цикл обработки нажатий клавиши}
         ReadMouseState(x,y,l,m,r);
         if l then    {если нажата левая кнопка }
            begin
              {пересчет координат курсора для текстового  режима}
              xt:=x div 8+1;
              yt:=y div 8+1;
              exit_m:=(xt>=70) and (xt<=80) and (yt>=1) and (yt<=3);
              if not exit_m then
                 begin
                   Gotoxy(xt,yt);
                   HideMouseCursor; {если не убирать курсор,
                                  то курсор мыши будет затерт строкой}
                   Write('x=', xt:4, ' y=', yt:4);
                    ShowMouseCursor; {теперь символ под курсором
                                        «просвечивает»  другим цветом}
                 end;
              repeat ReadMouseState(x,y,l,m,r) until not l; {ждем
                                                  отпускания левой кнопки}
            end;
       until exit_m;   {до «нажатия кнопки» Конец}
       Textattr:=7;      Clrscr;
End.
```

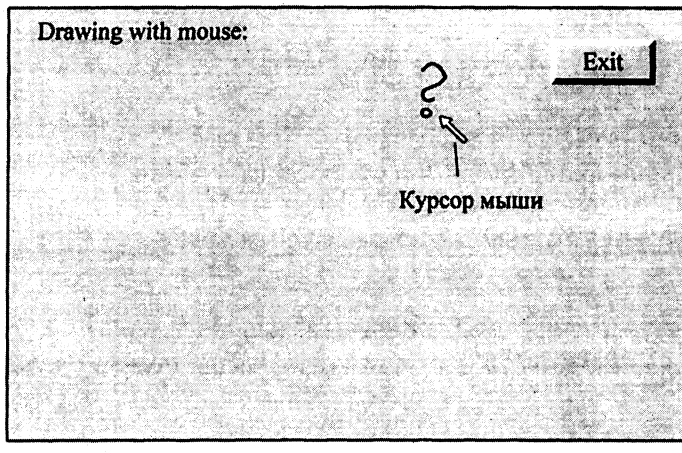


Рис. 8.14. Вид экрана при выполнении программы

Взаимодействие с мышью в графическом режиме. При использовании мыши в графическом режиме также приходится выполнять циклический опрос состояния мыши для определения момента нажатия клавиши.

Пример 8.10. Разработать программу рисования мышью при нажатой левой клавише: точка рисуется при каждом опросе состояния мыши, если клавиша остается нажатой. Реализовать двойное управление (с клавиатуры и мышью): выход – при нажатии мышью кнопки Exit на экране или клавиши Esc на клавиатуре (рис. 8.14).

```
Program ex;
Uses Mouse,Crt,Graph;
Var
   x,y:integer;
   l,m,r:boolean;
   exit_m:boolean;
   driver,mode:integer;
   ch:char;
   {процедура рисования кнопки}
  Procedure Button(x1,y1,x2,y2:integer;s:string);
  Begin
    SetColor(0);
    SetFillStyle(1,8);
    Bar(x1,y1,x2,y2);          {рисуем контур кнопки}
    SetFillStyle(1,7);
    Bar(x1+2,y1+2,x2-3,y2-3); {рисуем тени кнопки}
    SetFillStyle(1,15);
    Bar(x1,y1,x1+1,y2);
```

```
    Bar(x1+2,y1+1,x1+2,y2-1);
    Bar(x1,y1,x2,y1+1);
    Bar(x1+1,y1+1,x2-1,y1+2);
    SetColor(4);
    SetTextStyle(1,0,3);
    OutTextXY(x1+20,y1+3,s);
  End;
{основная программа}
Begin
   Clrscr;
   if not ReSetMouse then  {проверим наличие мыши}
            begin
              WriteLn('Mouse not found.');
              halt(1);
            end;
   driver:=detect;
   InitGraph(driver,mode,' ');
   SetColor(4);
   OuttextXY(10,10,'Drawing with mouse:');
   ShowMouseCursor;    {покажем курсор мыши}
   SetBkColor(3);
   Button (500,10,600,50,'Exit');
   repeat  {цикл опроса состояния клавиатуры и мыши}
       if keypressed then ch:=readkey {если нажата клавиша на
                                       клавиатуре, то введем код}
       else   {если клавиша не нажата, то}
         begin
           ReadMouseState(x,y,l,m,r);  {опросим состояние мыши}
           Exit_m:=(x>=500)and(x<=600)and(y>=10)and(y<=50);
           if not Exit_m and l then {если не «нажата» кнопка «Exit»
                                                    на экране}
              begin  {изобразим точку на экране}
                HideMouseCursor;
                PutPixel(x,y,4);
                ShowMouseCursor;
              end;
         end
   until (exit_m and l) or (ch=#27);
   repeat ReadMouseState(x,y,l,m,r) until not l; {ожидаем отпускания
                                                    клавиши}
  CloseGraph;
End.
```

Задание для самопроверки

Модернизировать программу из задания 1 к параграфу 8.4 так, чтобы реализовать двойное управление меню: с использованием клавиатуры и мыши.

8.9. Управление задачами. Вызов дочерних процессов

С точки зрения MS DOS каждая программа (задача) представляет собой *процесс*. При запуске процессу выделяется память и передаются окружение и параметры командной строки MS DOS.

Окружение – это специальная область памяти, в которой размещены в виде символьных строк некоторые параметры, установленные в DOS. Например:

COMSPEC=C:\COMMAND.COM {адрес интерпретатора команд MS DOS}
PATH=C:\QEMM;C:\DOS;C:\NC {каталоги автоматического поиска}
PROMPT=pq {вид запроса в командной строке MS DOS}

Пользователь может включить в окружение другие строки, используя команду SET.

Для работы с окружением модуль DOS содержит следующие ресурсы.

1. Функция ***EnvCount : integer*** – возвращает количество переменных окружения, содержащихся в среде MS DOS.
2. Функция ***EnvStr(Index:integer):string*** – возвращает переменную окружения MS DOS с указанным индексом.
3. Функция ***GetEnv(EnvVar:string):string*** – возвращает переменную окружения MS DOS с указанным именем.

Используя эти функции, можно, например, определить в системе местоположение каталога временных файлов, обычно заданного в MS DOS параметром work:

```
flag:=false;
i:=1;
while (i<EnvCount) and not flag  do
  begin
    if pos('work=',EnvStr[i])=1 then
       begin
          path:=copy(EnvStr[i],6,length(EnvStr[i])-5);
          flag:=true;
       end
    else inc(i);
```

```
  end;
if flag  then <каталог найден>
         else <каталог не определен> ...
```

Командная строка MS DOS – это символьная строка, которая вводится в командном режиме MS DOS при вызове той или иной программы (или команды). Помимо указания пути к исполняемому файлу она может содержать список параметров командной строки, например:

A:\>C:\My\copyfile A.dat B.dat 80

В модуле DOS имеются средства, обеспечивающие доступ к этим параметрам.

1. Функция ***ParamCount:integer*** – возвращает количество параметров.
2. Функция ***ParamStr(Index:integer):string*** – возвращает параметр с указанным индексом в виде символьной строки, например для программы, вызов которой приведен выше:

ParamCount = 3
ParamStr[0] = C:\My\copyfile
ParamStr[1] = 'A.dat'
ParamStr[2] = 'B.dat'
ParamStr[3] = '80'

Дочерний процесс – это самостоятельная программа, существующая в виде файла с расширением .com или .exe и вызываемая из другой программы. Необходимость организации дочерних процессов возникает, например, если требуется вставить в программу заставку, вывод на экран которой выполняется специальной программой, или разрабатывается среда, которая будет вызывать специальные программы обработки.

При вызове дочерних процессов обычно используются специальные ресурсы модуля DOS.

1. Процедура ***Exec (***<полное имя файла программы>,
<параметры командной строки>***:string)*** – осуществляет вызов дочернего процесса.
2. Функция ***DosExitCode:word*** – возвращает код завершения дочернего процесса.

Старший байт этого кода интерпретируется следующим образом:

0 – нормальное завершение;
1 – завершение по Ctrl - C;
2 – завершение по ошибке;
3 – завершение с сохранением в памяти.

Младший байт содержит код возврата дочернего процесса (параметр Halt).

3. Процедура ***SwapVectors*** – сохраняет в памяти настройки (например, адреса обработчиков прерываний) среды Borland Pascal или восстанавливает их, если они были сохранены:

DOS->Pascal → DOS(дочерний процесс) → Pascal->DOS
SwapVectors **SwapVectors**

Данная процедура должна выполняться, если дочерний процесс – программа, написанная на любом языке, кроме Borland Pascal.

При вызове дочерних процессов также используется специальная директива управления памятью, иначе выполняемая программа получает всю имеющуюся память и дочерний процесс некуда грузить:

{$M <размер стека>, <минимальный размер «кучи»>,
<максимальный размер «кучи»>}

где размер стека – от 1024 до 65520 байт; минимальный размер «кучи» – от 0 до 655360 байт; максимальный размер «кучи» – от минимального до 655360 байт.

Пример 8.11. Разработать программу, вызывающую специальную программу просмотра графических файлов типа .pcx. Эта программа требует указания имени просматриваемого файла в качестве параметра командной строки.

```
{$M $4000,0,0 }  {стек 16K, кучи нет}
Program ex;
Uses Dos;
begin
  WriteLn('Вызываем дочерний процесс ... ');
  SwapVectors;
  Exec('c:\utils\bitmap.exe', 'r24.pcx');
  SwapVectors;
  WriteLn('... вернулись в основную программу');
   if DosError <> 0 then {если есть ошибка при вызове дочернего
                                                    процесса}
          WriteLn('Ошибка DOS #', DosError)
  else
    WriteLn('Дочерний процесс вызван. Код завершения = ',
 DosExitCode);  {код завершения дочернего процесса}
End.
```

Часть 2. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

9. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Считается, что технология процедурного программирования применима, если размер программы не превышает 100 тыс. операторов. Программы, используемые в настоящее время, существенно длиннее. Поэтому современное программирование в основном базируется на технологии, позволившей снять это ограничение и получившей название «объектно-ориентированное программирование» (ООП). Именно ООП лежит в основе таких современных сред создания программного обеспечения «под Windows», как Delphi, Visual C++, C++ Builder.

В теории программирования ООП определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности *объектов*, каждый из которых является экземпляром определенного типа (*класса*), а классы образуют иерархию с *наследованием* свойств [2].

Как следует из определения, ООП в отличие от процедурного программирования, которое рассматривалось в первой части учебника, базируется не на процедурной, а на объектной декомпозиции предметной области программы.

9.1. Объектная декомпозиция

Объектной декомпозицией называют процесс представления предметной области задачи в виде совокупности функциональных элементов (*объектов*), обменивающихся в процессе выполнения программы входными воздействиями (*сообщениями*).

Каждый выделяемый объект предметной области отвечает за выполнение некоторых действий, зависящих от полученных сообщений и параметров самого объекта.

Совокупность значений параметров объекта называют его *состоянием*, а совокупность реакций на получаемые сообщения – *поведением*.

Параметры состояния и элементы поведения объектов определяются условием задачи.

В процессе решения задачи объект, получив некоторое сообщение, выполняет заранее определенные действия, например, может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и, в свою очередь, сформировать сообщения другим объектам. Таким образом, *процессом решения задачи управляет последовательность сообщений.* Передавая эти сообщения от объекта к объекту, программа выполняет необходимые действия.

Различие процедурной и объектной декомпозиции предметной области задачи продемонстрируем на примере разработки программы исследования элементарных функций, рассмотренной в параграфе 5.2.

Пример 9.1. Разработать программу исследования элементарных функций, которая для функций y=sin x, y=cos x, y=tg x, y=ln x, $y=e^x$ выполняет следующие действия:

- строит таблицу значений функции на заданном отрезке с заданным шагом;
- определяет корни функции на заданном отрезке;
- определяет максимум и минимум функции на заданном отрезке.

В основе объектной декомпозиции также лежит граф состояний интерфейса (см. рис. 5.6 – 5.7). Будем считать, что каждое состояние интерфейса – это состояние некоторого функционального элемента системы, т. е. объекта. Состояний интерфейса пять, соответственно, получаем пять объектов. Назовем эти объекты следующим образом: Главное меню, Меню операций, Табулятор, Определитель корней, Определитель экстремумов. Эти объекты передают управление друг другу, генерируя сообщение Активизировать. Результат объектной декомпозиции изображают в виде *диаграммы объектов* (рис. 9.1).

Кроме этого можно выделить еще один объект Функцию, который должен обеспечивать вычисление выбранной функции по заданному аргументу. Номер функции сообщается данному объекту Главным меню после того, как пользователь осуществит выбор.

Полная характеристика объекта включает идентифицирующее условное имя, а также перечень и описание параметров состояния и аспектов поведения.

Так, состояние объекта Функция характеризуется единственным параметром: номером функции, который передает ему Главное меню. Поведение же включает реакции на два типа сообщений: получив номер функции, объект должен сохранить его, изменив таким образом свое состояние, а получив запрос на вычисление значения функции, сопровождающийся определенным значением аргумента, – вернуть значение функции в заданной точке.

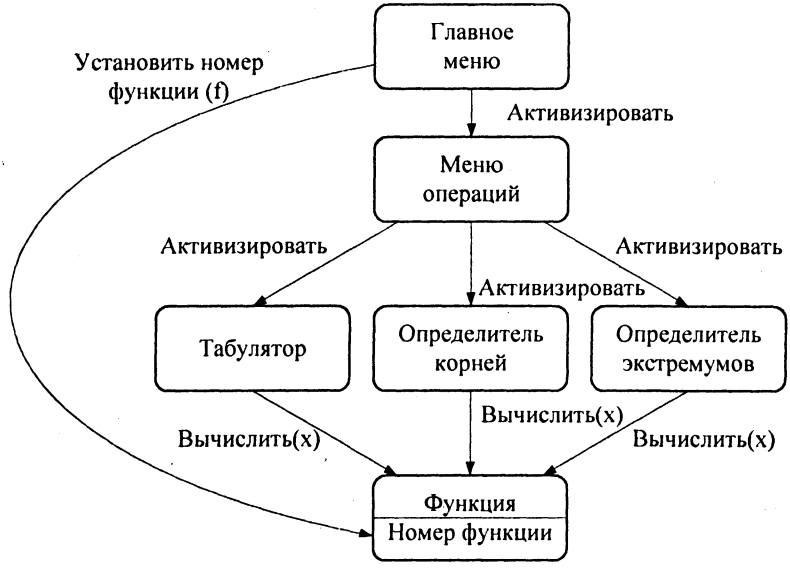


Рис. 9.1. Диаграмма объектов предметной области

Таким образом, при выполнении объектной декомпозиции определяют и описывают множество объектов предметной области и множество сообщений, которое формирует и получает каждый объект.

Задание для самопроверки

Выполнить объектную декомпозицию предметной области задания 2 к параграфу 8.4.

9.2. Классы и объекты-переменные

В программе для представления объектов предметной области используют переменные специальных типов – классов.

Класс – это структурный тип данных, который включает описание полей данных, а также процедур и функций, работающих с этими полями данных. Применительно к классам такие процедуры и функции получили название *методов*.

Поля, описанные в классе, используют для хранения составляющих состояния или *атрибутов* объекта. Например, если объект Функция должен хранить номер функции, то реализующий его класс должен содержать соответствующее поле.

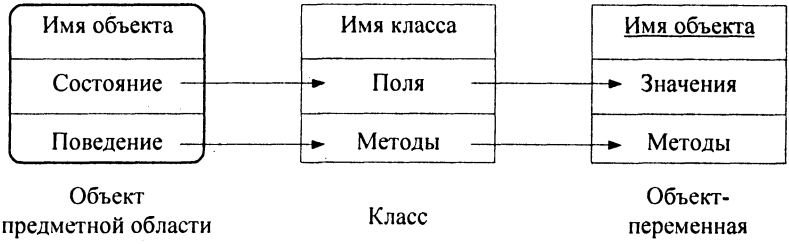


Рис. 9.2. Соответствие объектов предметной области, классам и объектам-переменным

Каждый метод определяет реакцию на некоторое внешнее или внутреннее сообщение. Например, объект Меню операций должен реагировать на сообщение Активизировать. Получив это сообщение, объект должен вывести меню операций и организовать работу с этим меню, т.е. при выборе некоторой операции формировать сообщение соответствующему объекту, передавая ему управление, а получив управление обратно, вновь вывести свое меню и ожидать ввода номера операции.

Переменные типа класса также обычно называют объектами. При необходимости в тексте данного учебника будем уточнять, что имеются в виду объекты-переменные или объекты предметной области. На рис. 9.2 показана связь объектов предметной области, классов и объектов-переменных.

Согласно общим правилам языка программирования объект-переменная должен быть:

- *создан* – для него должна быть выделена память;
- *инициализирован* – полям объекта должны быть присвоены значения;
- *уничтожен* – память, выделенная под объект, должна быть освобождена.

В зависимости от способа выделения памяти под объект-переменную различают *статические* объекты, память под которые выделяется при компиляции программы, и *динамические*, выделение памяти под которые производится в процессе выполнения программы.

9.3. Методы построения классов

Одним из наиболее значимых достоинств ООП является то, что большинство классов для реализации объектов не приходится разрабатывать «с нуля». Обычно классы строят на базе уже существующих, используя механизмы, реализующие определенное отношение существующего и строящего классов между собой: наследование, композицию, агрегацию и полиморфное наследование.

Наследованием или *обобщением* называют отношение между классами, при котором один класс строится на базе второго посредством добавления

полей и определения новых методов. При этом исходный класс, на базе которого выполняется построение, называют *родительским*, или *базовым*, или *супертипом*, а строящийся класс – *потомком*, или *производным* классом, или *подтипом*.

При наследовании поля и методы родительского класса повторно не определяют, специальный механизм наследования позволяет использовать эти компоненты класса, особо этого не оговаривая.

Примечание. В Borland Pascal реализовано только простое наследование, при котором класс может иметь всего одного родителя. В теории программирования определено также *множественное наследование*, предполагающее наличие у класса двух и более родителей. Такой вариант наследования реализован, например, в C++.

Наследование свойств в иерархии существенно упрощает работу программиста. В настоящее время созданы библиотеки наиболее часто встречающихся классов, которые можно использовать вновь и вновь, строя на их основе классы для решения различных задач.

Отношения между различными классами проекта принято иллюстрировать *диаграммой отношений классов*, или просто *диаграммой классов*. Если на диаграмме классов показано только отношение наследования, то такую диаграмму называют *иерархией* классов. На диаграмме классов наследование изображают линией с треугольной незакрашенной стрелкой на конце, направленном к классу-родителю (рис. 9.3, *а*, *б*). При необходимости допускается произвольное расположение классов родителей и потомков (рис. 9.3, *в*).

Кроме отношения между классами на диаграмме классов целесообразно указывать поля и методы каждого или только строящегося класса, так как это позволяет уточнить структуру разрабатываемых классов. Примеры диаграмм классов с уточняющим описанием приведены в главах 10–12.

Композицией называют такое отношение между классами, когда один является *неотъемлемой* частью второго. Физически композиция реализуется

Рис. 9.3. Примеры иерархий классов:
а – с одним потомком; *б* – с двумя потомками; *в* – с нестандартным расположением классов

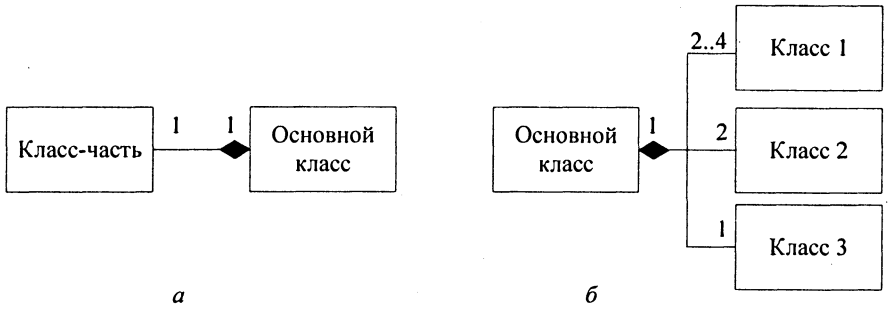


Рис. 9.4. Примеры диаграмм классов, изображающих композицию:
а – с одним объектным полем; *б* – с несколькими объектными полями различных типов

включением в класс фиксированного количества полей, являющихся объектами другого класса. Такие поля обычно называют *объектными*.

На диаграмме классов композицию изображают линией с закрашенным ромбом, указывающим на класс большей сложности, в который происходит включение объектных полей (рис. 9.4, *а*). Для большей информативности имеет смысл указывать над стрелкой количество объектов включаемого класса в каждый объект включающего класса. При этом допускается указывать точное значение или диапазон (рис. 9.4, *б*).

Наполнением или *агрегацией* называют такое отношение между классами, при котором точное количество объектов одного класса, включаемых в другой класс, не ограничено и может меняться от 0 до достаточно больших значений. Физически наполнение реализуется с использованием указателей на объекты. В отличие от объектного поля, которое включает в класс точно указанное количество объектов (1 или более – при использовании массива объектов или нескольких объектных полей) конкретного класса, использование указателей позволяет включить 0 или более объектов, если они собраны в массив или списковую (линейную или нелинейную) структуру.

На диаграмме классов наполнение изображают аналогично композиции, но ромб не закрашивают (рис. 9.5), обозначая таким образом менее жесткую связь между объектами соответствующих классов. Количество объектов указывают в виде диапазона, например, «0..*» или «1..*», или просто «*», что означает неопределенное множество объектов.

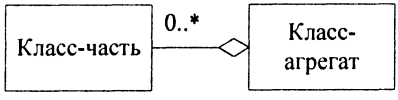


Рис. 9.5. Пример диаграммы классов, изображающей агрегацию или наполнение

Полиморфным наследованием называют наследование, при котором осуществляют *переопределение* методов класса-родителя потомком. Метод потомка в этом случае имеет то же имя, что и метод родителя, но выполняет другие действия.

Переопределение методов – частный случай реализации *полиморфизма* в программировании.

Примечание. Термин «полиморфизм» в соответствии со своим изначальным смыслом («многообразие») в программировании используют для обозначения возможности изменения кода программы в соответствии со значением некоторых параметров. Такая возможность существует не только в ООП.

Различают:

- *чистый* полиморфизм – возможность различной интерпретации кода функции в зависимости от типа аргументов; используется в языках высокого уровня абстракции, например, в языке LISP или SMALLTALK;
- *перегрузку (полиморфные имена) функций* – возможность определения нескольких функций с одним именем – одно и то же имя функции может многократно использоваться в разных местах программы; выбор нужной функции может определяться типами аргументов, областью видимости (внутри модуля, файла, класса и т.д.); если выбор определяется типом аргументов, то перегрузка называется *параметрической*; например, язык C++ позволяет разработчику выполнять параметрическую перегрузку функций вне классов, а в Borland Pascal возможно определение функций с одинаковыми именами в различных модулях или в модуле и основной программе;
- *переопределение методов* – в ООП возможность различных определений методов в классе-потомке и классе-родителе; конкретный метод определяется классом объекта, для которого он вызывается;
- *обобщенные функции*, или *шаблоны* – возможность описания параметризованных классов и шаблонов функций (реализована в C++), параметрами таких описаний являются *типы* аргументов методов или функций.

При переопределении методов различают простой и сложный полиморфизм.

Простой полиморфизм используют, если при вызове переопределенного метода тип объекта, для которого вызывается этот метод, точно известен, а, следовательно, и точно известно, какой метод должен быть подключен: метод родителя или метод потомка. В этом случае нужный метод определяется *на этапе компиляции* программы (*раннее связывание*).

Сложный полиморфизм используют, если при вызове переопределенного метода необходимо уточнить, какой метод должен быть подключен: метод родителя или метод потомка, так как объект, для которого вызывается переопределенный метод, может быть как объектом класса родителя, так и объектом класса потомка. В этом случае нужный метод определяется *на этапе выполнения* программы (*позднее связывание*), когда тип объекта точно известен. Позднее связывание обязательно должно реализовываться в конкретных случаях, которые будут перечислены в параграфе 11.5.

Для выявления отношения имеющегося и строящегося классов необходимо выполнить анализ структуры объектов предметной области, полученных в результате объектной декомпозиции.

Если объекты предметной области слишком сложны, чтобы ставить им в соответствие некий простой класс, то процесс декомпозиции можно продолжить, выделяя внутри сложных объектов более простые.

При этом возможны следующие варианты.

1. Внутри объекта можно выделить объект близкого назначения, но с более простой структурой и/или поведением – класс для реализации данного объекта следует строить на базе более простого, используя наследование. Если при этом объекты строящегося класса отличаются от объектов базового класса некоторыми аспектами поведения, то следует изменить поведение объектов строящегося класса при наследовании, используя полиморфное наследование с переопределением методов.

2. Внутри объекта можно выделить определенное количество объектов более простой структуры со своим поведением – класс строится объединением объектов других классов с добавлением полей и методов строящегося класса – композиция классов.

3. Внутри объекта можно выделить заранее не предсказуемое количество объектов более простой структуры со своим поведением – класс строится с возможностью динамического подключения объектов других классов (наполнение) и добавлением полей и методов строящегося класса.

Рассмотрим два примера.

Пример 9.2. Разработать класс для реализации объекта «Текст», который должен:

- для каждого слова некоторой последовательности слов хранить его атрибуты (тип шрифта и размер букв);
- определять количество слов в тексте;
- добавлять слова после слов с указанными номерами;
- удалять заданные слова из текста;
- менять местами слова с заданными номерами;
- заменять заданное слово на другое заданное во всем тексте;
- переопределять атрибуты слова с заданным номером.

Итак, реализуемый объект должен оперировать с некоторыми внутренними объектами «Словами», для которых можно определить собственное состояние и поведение. Естественно создать специальный класс TWord для реализации «Слов». Класс TText для реализации «Текста» может быть построен как с использованием композиции, так и с использованием наполнения.

В первом случае он должен включать массив объектов класса TWord. Максимальное количество элементов массива должно быть определено заранее и, следовательно, ограничено (рис. 9.6, *а*). При выполнении операций удаления и вставки придется сдвигать и раздвигать элементы массива.

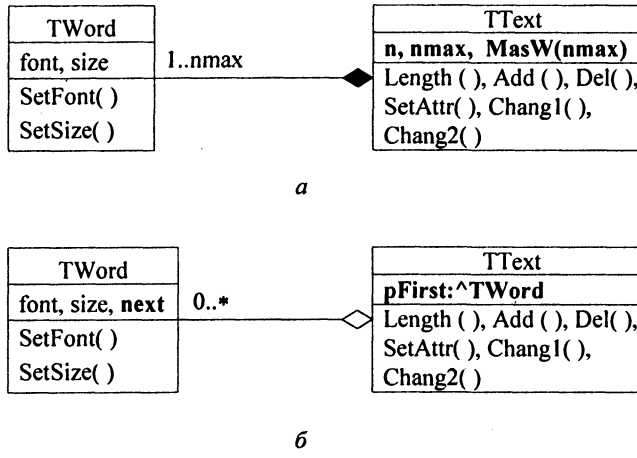


Рис. 9.6. Диаграммы классов для реализации объекта Текст:
а – с композицией; *б* – с наполнением

Во втором случае класс TText должен включать список объектов класса TWord (рис. 9.6, *б*). Ограничения предыдущей реализации будут сняты, но реализация со списком имеет несколько большую трудоемкость, и, кроме того, при обращении к слову по номеру придется каждый раз последовательно отсчитывать нужный элемент. Выбор конкретного варианта реализации зависит от условий решаемой задачи.

Пример 9.3. Разработать классы для реализации объектов Табулятор, Определитель корней и Определитель экстремумов из примера 9.1.

Объекты Табулятор, Определитель корней и Определитель экстремумов отвечают за реализацию методов решения некоторых частных задач при исследовании функций. Они имеют много общего. Попробуем определить это общее.

Любой объект, получив управление, должен ввести диапазон изменения аргумента [a, b], решить подзадачу, вывести результаты, а затем вернуть управление меню операций. Общее поведение и поля объектов опишем в классе TMetod. Основной метод этого класса Run должен реализовывать общее поведение и обеспечивать возможность изменения элементов этого поведения (решения конкретных подзадач) для объектов классов, которые будут от него наследоваться. Решение конкретной подзадачи реализуем как внутренний метод Task, вызываемый из метода Run. Этот внутренний метод для класса TMetod определять не будем (абстрактный метод).

Классы для реализации разрабатываемых объектов наследуем от TMetod, переопределяя метод Task и добавляя недостающие поля (рис. 9.7). На диаграмме курсивом показано, что класс TMetod и метод Task являются абстрактными. При необходимости это указывают явно, используя слово «abstruct».

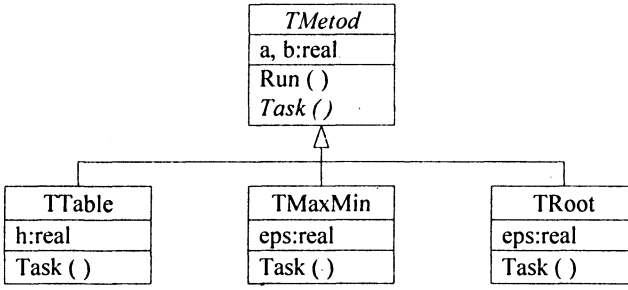


Рис. 9.7. Иерархия классов для реализации объектов примера

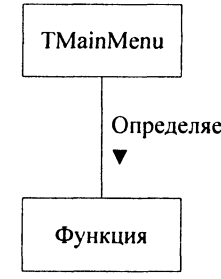


Рис. 9.8. Пример ассоциации

Помимо определенных выше, на диаграмме классов показывают также отношения *ассоциации* между классами, объекты которых обмениваются сообщениями, но не связаны отношениями композиции или агрегации. Они обозначаются линиями без стрелок. Если сообщения передаются в одну сторону, то направление ассоциации можно показать стрелкой над линией. Кроме того, ассоциации можно дать имя, которое в данном контексте называют *ролью* (рис. 9.8).

После разработки диаграммы классов переходят к их реализации в конкретном языке программирования. Особенности реализации классов будут обсуждаться в главах 10 – 11.

9.4. Этапы реализации объектно-ориентированного подхода

Процесс разработки программного обеспечения с использованием объектно-ориентированного подхода включает те же основные этапы, что и с использованием структурного подходы: анализ, проектирование, реализацию и модификацию. Однако процедуры, выполняемые на каждом этапе, несколько меняются. Кроме того, этап реализации при объектном подходе называют *эволюцией*, что также связано с его особенностями.

Рассмотрим эти этапы.

Анализ предметной области задачи. Цель анализа – максимально полное описание задачи. На этом этапе выполняют объектную декомпозицию разрабатываемой системы и определяют основные особенности поведения объектов. Результаты объектной декомпозиции представляют в виде диаграммы объектов, на которой показывают основные объекты и сообщения, передаваемые между ними.

Проектирование системы. Логическое проектирование при объектном подходе заключается в разработке структуры классов: определяют поля для хранения составляющих состояния объектов и алгоритмы методов, реализующих аспекты поведения объектов. При этом используют рассмотренные выше механизмы реализации отношений классов (наследование, компози-

ция, наполнение и полиморфизм). Результатом является иерархия или диаграмма классов, отражающая отношения между классами и включающая их описание.

Физическое проектирование заключается в объединении описаний классов в модули, определении способов взаимодействия с оборудованием, с операционной системой и/или другим программным обеспечением (например, базами данных, сетевыми программами), обеспечении синхронизации процессов для систем параллельной обработки и т.д. Результаты физического проектирования представляют в виде схемы композиции классов в модули, описания интерфейсов для взаимодействия с другими программами и т. п.

Эволюция системы. Эволюция системы – это процесс *поэтапной реализации* классов и подключения объектов к системе. Само название этапа подчеркивает поэтапный характер процесса, упрощающий сборку системы.

Реализацию начинают с создания основной программы или проекта будущего программного продукта. Затем описывают классы и подключают объекты этих классов так, чтобы создать грубый, но, по возможности, работающий *прототип* будущей системы, который тестируют и отлаживают.

Например, на первых этапах эволюции проект может включать только объекты, реализующие основной интерфейс программного продукта. На данном этапе передача сообщений в отсутствующую пока часть системы не выполняется. Отлаженный прототип системы может быть, например, показан заказчику для уточнения требований.

Далее к проекту подключают следующую группу классов, например, связанных с реализацией некоторого пункта меню. Полученный вариант также тестируется и отлаживается, и так далее, до реализации всех возможностей системы.

Модификация. Модификация – это процесс добавления новых функциональных возможностей или изменения существующих свойств системы. Как правило, изменения затрагивают реализацию класса, не трогая его интерфейс, что при использовании ООП обычно обходится без особых неприятностей, так как процесс изменений затрагивает локальную область.

Простота модификации позволяет сравнительно легко адаптировать программные системы к изменяющимся условиям эксплуатации, что увеличивает время жизни систем, на разработку которых затрачиваются огромные временные и материальные ресурсы.

Особенностью ООП является то, что объект или группа объектов могут разрабатываться отдельно, и, следовательно, их проектирование может находиться на различных этапах. Например, интерфейсные классы уже реализованы, а структура классов предметной области еще только уточняется. Обычно проектирование начинается, когда какой-либо фрагмент предметной области достаточно полно описан в процессе анализа.

10. КЛАССЫ И ОБЪЕКТЫ В BORLAND PASCAL

Объектная модель, реализованная в Borland Pascal, по современным меркам является упрощенной, но она позволяет изучить основные приемы объектно-ориентированного программирования и оценить его достоинства и недостатки.

В настоящей главе рассмотрены средства, используемые для объявления классов и объектов, и принципы создания «универсальных» классов.

10.1. Объявление класса. Поля и методы

С точки зрения синтаксиса класс представляет собой структурный тип данных, в котором помимо полей разрешается описывать *прототипы* (заголовки) процедур и функций, работающих с этими полями данных. По форме описание класса напоминает запись.

Как уже упоминалось ранее, процедуры и функции, заголовки которых описаны в классе, получили название *методов*.

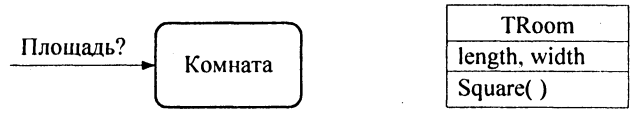
Описание типа класс выполняется следующим образом:

```
Type <имя класса> = object
                <описание полей класса>
                <прототипы методов>
            end; ...
```

Тела методов класса описываются после объявления класса. Причем в заголовке метода можно не повторять списка параметров, но перед именем метода необходимо указать имя класса, отделив его от имени метода точкой:

```
Procedure <имя класса>.<имя метода>;
 <локальные ресурсы процедуры>
   Begin
       <тело процедуры>
   End; ...
```

или



а *б*

Рис. 10.1. Объект Комната (*а*) и реализующий класс (*б*)

```
Function <имя класса>. <имя метода>;
 <локальные ресурсы функции>
   Begin
       <тело процедуры>
   End; ...
```

Пример 10.1. Разработать класс для реализации объекта Комната, который должен хранить длину и ширину комнаты и отвечать на запрос о площади комнаты (рис. 10.1, *а*).

Объект Комната характеризуется двумя параметрами: длиной и шириной, значит соответствующий класс должен включать два поля, в которых эти значения будут храниться (рис. 10.1, *б*). Единственное сообщение, на которое должен реагировать объект, – запрос о площади комнаты, следовательно, класс должен включать метод-функцию, которая должна возвращать значение площади комнаты.

```
Type TRoom = object
               length, width:real;{поля: длина и ширина комнаты}
               function Square:real;{метод определения площади}
             end;
Function TRoom.Square; {тело метода определения площади}
  Begin
     Square:= length* width;
  End; ...
```

Поскольку поля и методы описаны в одном классе, все поля класса доступны методам класса без дополнительного указания.

Физически это реализуется следующим образом. Каждая переменная – объект данного класса – при объявлении получает свой набор полей класса. Эти поля собраны в запись, внешнее имя которой совпадает с именем объекта. Любой метод класса, вызванный для конкретного объекта, *неявно* получает специальный параметр *Self*, значением которого является адрес записи, объединяющей все поля этого объекта. Этот параметр иногда называют *обобщенным внутренним именем объекта*. Реально обращение к полям конкретного объекта происходит через это обобщенное имя:

```
Function TRoom.Square;  {тело метода определения площади}
  Begin
     Square:= Self.length* Self.width;
  End; ...
```

При необходимости имя *Self* можно указывать явно, например *@Self* – адрес записи, содержащей поля объекта (естественно, такое обращение возможно только из методов, вызываемых для конкретного объекта, так как вне объекта это имя не определено).

10.2. Объявление объекта. Инициализация полей

Описав класс, мы можем объявить любое количество объектов этого класса, причем можно объявить отдельные объекты, массивы объектов и указатели на объекты данного класса.

Например:

```
Var A:TRoom;                    {объект А класса TRoom}
    B:array[1..5] of TRoom;     {массив объектов типа TRoom}
Type pTRoom=^TRoom; {тип указателя на объекты класса TRoom}
Var  pC: pTRoom;            {указатель на объекты класса TRoom}
```

Как и для любой другой динамической переменной, для динамического объекта необходимо выделить память, а после его использования – освободить память.

Выделение памяти осуществляют процедурой New или функцией New. Например:

```
New(pC);        или      pC:=New(pTRoom); ...
```

Для освобождения памяти используют процедуру Dispose. Например:

```
Dispose(pC); ...
```

Работа с динамическими объектами классов, построенных с использованием наследования со сложным полиморфизмом, имеет свои особенности, и мы вернемся к их рассмотрению в параграфе 11.7.

Обращение к полям и методам объекта. Обращение к полям и методам объекта выполняется так же, как к полям записей:

- с использованием точечной нотации:

  <имя объекта>.<имя поля> или <имя объекта>.<имя метода>;

- с использованием оператора with:

```
with <имя объекта> do
   begin
      ...<имя поля>...
      ...<имя метода>...
   end; ...
```

Например:

а) *v:=A.length;*
б) *s:= A.Square;*
в) *s:=s+B[i].Square;*
г) *pC^.length:=3; ...*

Инициализация полей объекта. Поля объекта должны инициализироваться. Инициализация полей объекта может осуществляться тремя способами:

- прямым занесением в поле, например:

```
Program ex;
Type TRoom = object
               length, width:real;{поля: длина и ширина комнаты}
               function Square:real;{метод определения площади}
             end;
Function TRoom.Square; {тело метода определения площади}
  Begin
     Square:= length* width;
  End;
Var A:TRoom;    {объявляем объект-переменную}
Begin
   A.length:=3.5;  {инициализируем поля объекта}
   A. width:=5.1;
   WriteLn('Площадь комнаты равна ',A.Square);
End.
```

- с использованием типизированных констант – синтаксис описания совпадает с синтаксисом типизированных констант типа «запись»:

```
Program ex;
Type TRoom = object
               length, width:real;{поля: длина и ширина комнаты}
               function Square:real;{метод определения площади}
             end;
```

```
Function TRoom.Square; {тело метода определения площади}
  Begin
     Square:= length* width;
  End;
Const
     A:TRoom = (length:3.5; width:5.1); {объявляем константу}
Begin
    WriteLn('Площадь комнаты равна ',A.Square);
End.
```

• посредством специального метода – очень часто в качестве такого метода используют специальную инициализирующую процедуру, которую рекомендуется называть Init:

```
Program ex;
Type TRoom = object
             length, width:real;{поля: длина и ширина комнаты}
             function Square:real;{метод определения площади}
             procedure Init(l,w:real);{инициализирующий метод}
           end;
{метод определения площади}
Function TRoom.Square;
  Begin
     Square:= length* width;
  End;
 {инициализирующий метод}
Procedure TRoom.Init;
  Begin
     length:=l; width:=w;
  End;
Var A:TRoom;    {объявляем объект-переменную}
{основная программа}
Begin
   A.Init(3.5,5.1);  {инициализируем поля объекта}
   WriteLn('Площадь комнаты равна ',A.Square);
End.
```

Операция присваивания объектов. Над объектами одного класса определена операция присваивания. Физически при этом происходит копирование полей одного объекта в другой методом «поле за полем»:

```
Const A:TRoom=(length:3.7;:5.2);
Var B:TRoom; ...
    B:=A;    {теперь B.length=3.7, а B.width=5.2}
```

Существуют некоторые особенности выполнения операции присваивания для объектов родственных классов и полиморфных объектов. Они описаны в параграфах 11.1 и 11.5.

10.3. Библиотеки классов. Ограничение доступа к полям и методам

Одним из достоинств ООП является возможность создания библиотек классов, на базе которых затем конструируют классы для реализации объектов реальной задачи. Библиотечные классы при этом описывают в интерфейсной части модуля, а тела методов – в разделе реализации.

Например:

```
Unit Room;
Interface
Type TRoom = object
          length, width:real; {поля: длина и ширина комнаты}
          function Square:real; {метод определения площади}
          procedure Init(l,w:real);{инициализирующий метод}
        end;
Implementation
Function TRoom.Square; {метод определения площади}
  Begin
     Square:= length* width;
  End;
Procedure TRoom.Init;      {инициализирующий метод}
  Begin
     length:=l;
     width:=w;
  End;
End.
```

В этом случае основная программа будет подключать соответствующий модуль и работать с классом, его полями и методами, как с ресурсами библиотеки:

```
Program ex;
Uses Room;{подключаем модуль с описанием класса TRoom}
Var A:TRoom;    {объявляем объект-переменную}
Begin
  A.Init(3.5,5.1);  {инициализируем поля объекта}
  WriteLn('Комната: длина= ', A.length,
                    '; ширина=', A.width);
```

```
    WriteLn('Площадь комнаты =',A.Square);
End.
```

В Borland Pascal можно ограничить доступ к полям и методам класса *в пределах модуля*. Для этого описание класса делится на специальные секции:

public – секция, содержащая описание общих или общедоступных полей и методов класса;

private – секция, содержащая описание внутренних или скрытых полей и методов класса.

В описании класса эти секции могут чередоваться, причем, если секции компонент не указаны, то по умолчанию принимается, что эти компоненты доступны как общие:

```
Unit <имя модуля>;
Interface
Type <имя класса>= object
            <описание общих полей и методов>
        private
            <описание внутренних полей и методов>
        public
            <описание общих полей и методов>
        private
            <описание внутренних полей и методов>
    end; ...
```

Например, в нашем случае, если объекты класса TRoom используются только для получения информации о площади комнаты, то можно поля описать в секции private, но тогда доступ к этим полям из программы станет невозможным:

```
Unit RoomHiden;
Interface
Type TRoom = object
        private    {скрытые компоненты класса}
          length, width:real; {поля: длина и ширина комнаты}
        public     {общие компоненты класса}
          function Square:real;{метод определения площади}
          procedure Init(l,w:real);{инициализирующий метод}
        end;
Implementation
Function TRoom.Square; {метод определения площади}
  Begin
    Square:= length* width;
  End;
```

```
Procedure TRoom.Init;      {инициализирующий метод}
  Begin
     length:=l;  width:=w;
  End;
End.
```

Сокрытие некоторых полей и методов класса упрощает интерфейс класса, т. е. программист, использующий библиотечный класс, не получает лишней для него информации о внутренних механизмах реализации состояния и поведения объектов данного класса. Одновременно с этим программист, занимающийся разработкой библиотечных классов, получает возможность вносить изменения в реализацию класса, не заботясь об изменении программ, использующих объекты данного класса.

10.4. Практикум. Создание универсальных объектов

Даже отдельные объекты, разработанные для выполнения наиболее часто встречающихся действий, могут существенно упростить программисту создание программных продуктов. В качестве примера такого «универсального» объекта разработаем класс, объектами которого будут графики функций.

Пример 10.2. Разработать класс, объекты которого представляют собой окно, содержащее график функции одной переменной. Размеры окна, интервал изменения аргумента, его шаг и сама функция должны задаваться основной программой (рис. 10.2).

Основные принципы построения графиков функций в программах обсуждались в параграфе 8.6. По сравнению с программой, рассмотренной там, внесем следующие изменения. Во-первых, обеспечим возможность построения графика функции, заданной в основной программе, для чего используем параметр процедурного типа. Во-вторых, в процессе разработки методов, обеспечивающих реакции на запросы, выделим внутренние процедуры. В-третьих, координаты точек графика запишем в динамический массив, а сам график будем выводить с использованием процедуры рисования ломаной линии DrawPoly.

Проектирование класса для реализации объекта начинаем с определения полей объекта. Прежде всего, переменные класса должны хранить всю исходную информацию: ссылку на функцию F, интервал изменения аргумента [xn, xk], количество точек n и параметры окна x1, y1, x2, y2, в которое выводится график. Кроме того, для построения графика необходимо знать минимальное Fmin и максимальные Fmax значения функции на заданном отрезке, масштабы mx, my по осям x и y, шаг изменения ар-

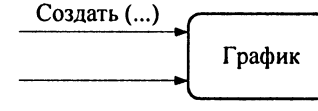


Рис. 10.2. Объект График

TWin_gr
x1, y1, x2, y2, F, xn, xk, n, dx, Fmin, Fmax, mx, my, xm, ym, xs, ys, s Init, Run, Fminmax, Rx, Ry

Рис. 10.3. Класс TWin_gr

гумента dx, координаты нижней левой точки графика xs, ys (они отличаются от координат окна, так как часть окна отводится под вывод координат сетки) и т. п.

Примечание. Целесообразно в скрытых полях переменной-объекта хранить все данные, используемые более чем в одном методе, за исключением тех, которые в каждом методе инициализируются заново. Последние имеет смысл объявлять локально в каждом методе. Например, Fmin, Fmax, xs и ys используются в методах Run, Rx, Ry, поэтому мы храним их в объекте и определяем в инициализирующей процедуре. В то же время динамический массив координат точек используется только в одном методе Run, следовательно, он объявляется в нем локально и там же создается, инициализируется и уничтожается.

Класс должен уметь обрабатывать сообщения Создать и Построить. Соответственно минимально он должен иметь два метода. Назовем их Init («Инициализировать») и Run («Выполнить»). Эти методы должны быть доступны из основной программы, а потому объявлены в общедоступной секции public. Разрабатывая алгоритмы этих методов, выделим из метода Init процедуру Fminmax, которая будет рассчитывать и инициализировать максимальное и минимальное значения функции на отрезке. Из метода Run выделим в отдельные процедуры последовательности действий по построению сетки графика Rx – по горизонтали, Ry – по вертикали. Эти методы внутренние, их и все поля класса будем описывать в разделе описаний метода Run, так как доступ к ним из программы не планируется. Окончательная структура класса TWin_gr показана на рис. 10.3.

Ниже приведен текст модуля, содержащего описание класса TWin_gr.

```
Unit WinGr;
Interface
  Uses crt,Graph;
  Type
    Fn=Function(X:real):real;
    arr=array[1..200] of pointtype;
    TWin_gr=object
      public
        procedure Init(axn,axk:real;an:integer; aF:Fn;
                       ax1,ay1,ax2,ay2:integer); {инициализация объектов}
        procedure Run;                  {вывод графика}
      private
        x1, y1, x2, y2:integer; {координаты окна}
        F:Fn;                   {адрес функции}
        n:integer;              {количество точек}
        xn, xk,                 {интервал изменения аргумента}
```

```
      dx, Fmin, Fmax,   {шаг аргумента и экстремумы функции}
      mx, my:real;      {масштабы по осям}
      xm, ym,           {размеры окна}
      xs, ys:integer;   {координаты нижней границы графика}
      s:string[10];     {рабочая строка для вывода разметки}
      procedure Fminmax; {определение минимума и максимума}
      procedure Rx;     {сетка, параллельная оси OX}
      procedure Ry;     {сетка, параллельная оси OY}
  end;
Implementation
{метод инициализации полей}
Procedure TWin_gr.Init;
 begin
  {параметры функции}
   xn:=axn;
   xk:=axk;
   n:=an;
   F:=aF;
  {координаты окна}
  x1:=ax1;
  y1:=ay1;
  x2:=ax2;
  y2:=ay2;
  {размер окна}
  xm:=x2-x1+1;
  ym:=y2-y1+1;
  dx:=(xk-xn)/n;    {устанавливаем шаг графика функции}
  Fminmax;{определяем минимальное и максимальное значения
                                        функции на заданном отрезке}
  {определяем нижнюю левую точку графика}
  xs:=60;
  ys:=ym-xs;
  {определяем масштабы по x и y}
  mx:=(xm-xs*2)/abs(xk-xn) ;
  my:=(ym-xs*2)/abs(Fmax-Fmin);
 end;
{метод определения экстремумов функции на отрезке}
Procedure TWin_gr.Fminmax;
Var x, y:real;
    i:integer;
 Begin
  x:=xn;
  y:=F(x);
```

```
  Fmin:=y;
  Fmax:=y;
  for i:=2 to n do
    begin
      x:=x+dx;
      y:=F(x);
      if y>Fmax then Fmax:=y;
      if y<Fmin then Fmin:=y;
    end;
 end;
```

{метод построения окна с графиком и координатной сеткой}

```
Procedure TWin_gr.Run;
Var  x:real;
     i:integer;
     din_arr:^arr; {указатель на динамический массив координат точек}
Begin
   SetViewPort(x1,y1,x2,y2,true);  {устанавливаем окно}
   ClearViewPort;                   {очищаем окно}
   SetColor(2);                     {устанавливаем цвет рисования}
   Rectangle(0,0,x2-x1,y2-y1);      {рисуем рамку}
   SetColor(14);              {устанавливаем цвет рисования}
   SetLineStyle(0,0,3);       {устанавливаем стиль линии: толщина 3}
   x:=xn;
   GetMem(din_arr,2*n);  {запрашиваем память под массив координат}
   for i:=1 to n do
       begin
          din_arr^[i].x:=round((x-xn)*mx+xs);
          din_arr^[i].y:=round((Fmin-f(x))*my+ys);
          x:=x+dx;
       end;
   DrawPoly(n,din_arr^);      {рисуем график}
   FreeMem(din_arr,2*n);      {освобождаем память}
   SetLineStyle(0,0,1);   {устанавливаем стиль линии: толщина 1}
   Rx;              {выводим сетку, параллельную оси x}
   Ry;              {выводим сетку, параллельную оси y}
   ReadKey;   {ожидаем ввода любого символа}
 end;
```

{метод вывода сетки, параллельной оси x}

```
Procedure TWin_gr.Rx;
 Var x, dxl:real;
 Begin
       x:=xn;
       dxl:=(xk-xn)/5;
```

```
        SetColor(11);
        SetTextStyle(0,1,1);
        repeat
          Str(x:6:3,s);
          OutTextXY(round((x-xn)*mx+xs-3),ys+5,s);
          Line(round((x-xn)*mx+xs),ym-20,round((x-xn)*mx+xs),20);
          x:=x+dxl;
        until x>xk+0.0000001;
end;
{метод вывода сетки, параллельной оси x}
Procedure TWin_gr.Ry;
 Var  y, dyl:real;
 Begin  y:=Fmin;
      SetTextStyle(0,0,1);
      dyl:=(fmax-fmin)/5;
      repeat
        Str(y:6:3,s);
        OutTextXY(5,round(-(y-Fmin)*my+ys-10),s);
        Line(10,  round(-(y-Fmin)*my+ys),
             xm-10,  round(-(y-Fmin)*my+ys));
        y:=y+dyl;
      until y>Fmax+0.0000001;
 end;
End.
```

Разработав модуль WinGr, мы можем в любой программе построить график. Для этого необходимо: подключить модуль WinGr, описать функцию, объявить соответствующий объект и вызвать методы Init и Run этого объекта.

Ниже приведен пример программы, строящей два графика с использованием WinGr.

```
Program Gr;
Uses WinGr, Graph;
{объявление переменных}
Var
      W1,W2:TWin_gr;
      dr, mode:integer;
{определение функции}
Function f(x:real):real; far;
   Begin
      f:=abs(cos(x*x)*x)
   End;
```

```
{основная программа}
Begin
    dr:=detect;
    InitGraph(dr,mode,'');
    W1.Init(-0.9,0.9,100,f,0,0,300,400);
    W1.Run;
    W2.Init(1,3,200,f,320,100,600,400);
    W2.Run;
End.
```

Задания для самопроверки

Задание 1. Спроектируйте универсальный класс для реализации объекта «Круговая диаграмма». Разработайте тестирующую программу.

Задание 2. Спроектируйте универсальный класс для реализации объекта «Графическая строка». Предусмотрите возможность изменения цвета символов строки и направления ее вывода: горизонтальное или вертикальное. Местоположение строки на экране задавайте координатами верхнего левого угла первого символа. Разработайте тестирующую программу.

11. ИЕРАРХИИ КЛАССОВ

Построение классов с использованием уже существующих – одно из основных достоинств ООП. Как уже упоминалось выше, в профессиональных средах программирования существуют мощные библиотеки классов, на базе которых строятся классы для решения конкретной задачи. При этом используют средства языка, реализующие основные отношения между классами: наследование (простое и полиморфное), композицию и наполнение.

11.1. Наследование

Наследованием называют конструирование новых более сложных производных классов из уже имеющихся базовых посредством добавления полей и методов.

При наследовании объекты класса-потомка получают возможность использования («наследуют») поля и методы класса-родителя, что позволяет повторно не определять эти компоненты класса.

При описании класса-потомка указывают класс-родитель и дополнительные, определенные только для класса-потомка, поля и методы:

Type <имя класса-потомка> = ***object(***<имя класса-родителя>***)***
<описание дополнительных полей и методов класса>
end; ...

Пример 11.1. Разработать класс для реализации объекта Трехмерная комната, который должен реагировать на запрос о площади и объеме комнаты (рис. 11.1).

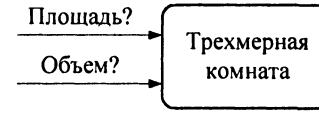


Рис. 11.1. Объект Трехмерная комната

В параграфе 10.3 уже был определен класс TRoom в модуле Room, который содержал поля для хранения длины и ширины ком-

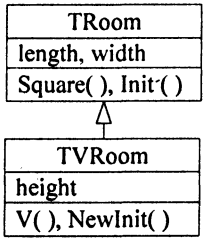


Рис. 11.2. Иерархия классов для TVRoom

наты, метод инициализации полей и метод определения площади. Построим класс для реализации объекта Трехмерная комната на базе TRoom, добавив поле для хранения высоты комнаты и метод определения объема комнаты V, который обращается к методу Square родительского класса (рис. 11.2). Класс TRoom также должен включать свой метод инициализации объектов для инициализации нового поля height.

Разрабатываемая программа должна содержать указание об использовании модуля Room, в котором описан родительский класс TRoom:

```
Program ex;
Uses Room;
Type TVRoom = object(TRoom)
        height:real;              {поле для хранения высоты}
        function V:real;          {метод определения объема}
        procedure NewInit(l,w,h:real); {инициализирующий метод}
     end;
Procedure TVRoom.NewInit;
 Begin
    Init(l,w);   {инициализируем наследуемые поля класса}
    height:=h; {инициализируем собственное поле класса}
 End;
Function TVRoom.V;
 Begin
   V:=Square*height;{обращаемся к методу базового класса}
 End;
```

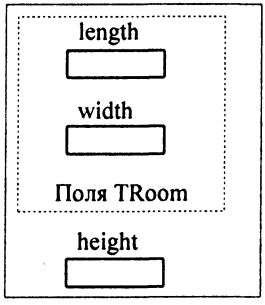


Рис. 11.3. Поля класса TVRoom

```
Var A:TVRoom;
Begin
    A.NewInit(3.4,5.1,2.8);
    WriteLn('Площадь комнаты = ',
                              A.Square:6:2);
    WriteLn('Объем комнаты = ', A.V:6:2);
End.
```

Класс TVRoom, таким образом, включает три поля: length, width и height (рис. 11.3) и четыре метода: Square, Init, NewInit и V. Все поля и методы доступны как из методов производного класса, так и непосредственно из программы.

Если программа, работая с объектами класса-потомка, не использует некоторых методов родителя, то в исполняемую программу они не включаются.

Примечание. По правилам Borland Pascal у класса может быть только один родитель, но сколько угодно потомков. Поскольку каждый производный класс добавляет при наследовании свои поля и методы, в иерархии классов по мере удаления от корня дерева иерархии сложность классов и соответственно объектов, которые эти классы реализуют, возрастает. Одновременно возрастает и специализация классов. Наиболее универсальные классы, таким образом, находятся у корня дерева классов.

Операция присваивания объектов родственных классов. В Borland Pascal допустимо присваивание объектам класса-родителя значений объектов класса-потомка. Обратное присваивание не разрешено, так как при его выполнении методом «поле за полем» собственные поля объекта класса-потомка окажутся не определенными.

Например:

```
Var A:TRoom;
    B:TVRoom; ...
A:=B;     {допустимо}
B:=A; ... {не допустимо: ошибка!}
```

Особенности работы с указателями на объекты родственных классов. По правилам Borland Pascal *допустимо указателям на объекты класса-родителя присваивать адреса объектов класса-потомка.* Однако при этом указатель на объект базового класса не обеспечивает возможности обращения к полям и методам, объявленным в производном классе (рис. 11.4).

Например:

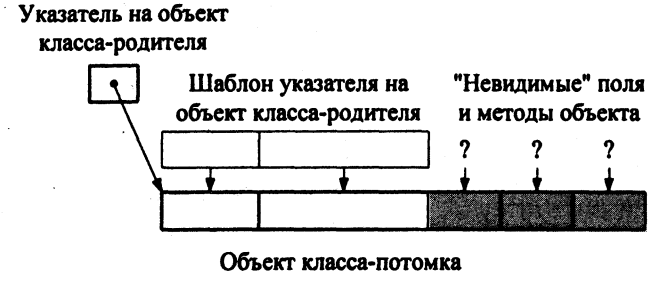


Рис. 11.4. Обращение к полям класса потомка через указатель на объекты класса-родителя

```
Var pC: ^TRoom;   {указатель на объекты базового класса}
    E:TVRoom; ...  {объект производного класса}

pC:=@E;        {присваиваем указателю на объекты базового класса адрес объекта производного класса}
pC^.height:=2.7; {ошибка, указатель на объекты типа класса-родителя не подозревает о существовании поля height класса TVRoom}
```

В этих случаях приходится *явно переопределять* тип указателя, используя имя типа указателя (см. параграф 2.5):

```
Type pTVRoom=^TVRoom;
Var pC: ^TRoom;   {указатель на объекты базового класса}
    E:TVRoom; ...  {объект производного класса}

pC:= @E;        {присваиваем указателю на объекты базового класса адрес объекта производного класса}
pTVRoom(pC)^.height:=2.7;  {явно переопределяем тип указателя}
```

Задания для самопроверки

Задание 1. Спроектируйте класс для реализации объекта «Окно», который, получив сообщение «Изобразить», должен выводить в определенное место экрана окно заданного размера и цвета. На базе класса, реализующего объект «Окно», спроектируйте класс для реализации объекта «Окно с текстом». Объект должен уметь реагировать на сообщение «Изобразить окно и вывести в него строку текста». Разработайте тестирующую программу.

Задание 2. Спроектируйте на базе класса, реализующего объект «Окно с текстом» предыдущего задания, класс, реализующий объект «Окно с текстом, меняющее цвет». Объекты данного класса должны уметь реагировать на сообщение «Изменить цвет окна».

11.2. Композиция

Композицией называют способ конструирования классов, при котором в строящийся класс включают объекты других классов. Включение реализовано посредством использования объектных полей – полей типа «класс».

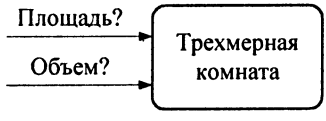


Рис. 11.5. Объект Квартира

Пример 11.2. Разработать класс для реализации объекта Квартира, который должен реагировать на запрос о жилой площади квартиры (рис. 11.5).

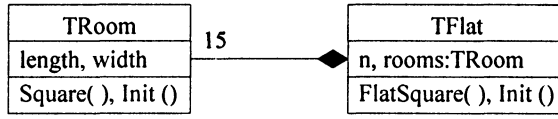


Рис. 11.6. Диаграмма классов примера

Физически любая квартира состоит из нескольких комнат. Ранее у нас уже был разработан класс TRoom, который хранил данные о длине и ширине комнаты и «умел» реагировать на запрос о площади комнаты. Количество комнат в квартире сравнительно не велико и всегда ограничено, для определенности будем считать, что реализуемый объект не может включать более 15 комнат. Тогда разрабатываемый класс TFlat должен включать массив из 15 объектов типа TRoom (рис. 11.6). Реальное количество комнат будем хранить в поле n. Для ответа на запрос о площади добавим метод FlatSquare, который будет обращаться к методам Square объектов-комнат TRoom.

Для инициализации объектных полей будем передавать в инициализирующий метод специальный массив значений параметров. Поскольку размерность этого массива будет определяться реальным количеством комнат, а значит будет отличаться от размерности поля rooms, необходимо описать параметр как открытый массив или нетипизированный параметр. Используем второй вариант.

```
Program ex;
 Uses Room;        {модуль Room определен в параграфе 10.3}
 Type TFlat=object   {описание класса}
       n:byte;          {количество комнат}
       rooms:array[1..15] of TRoom; {массив объектов TRoom}
       function FlatSquare:real;{метод определения площади}
       procedure Init(an:byte;Var arooms);{метод инициализации}
       end;
 Procedure TFlat.Init;  {тело метода инициализации}
 Var a:array[1..15] of TRoom absolute arooms; {переопределение типа
                                  массива наложением – см. параграф 5.5}
     i:byte;
 Begin
    n:=an;        {инициализируем поле количества комнат}
    for i:=1 to n do {инициализируем объектные поля, вызывая метод
                      инициализации TRoom для каждой комнаты
                      и передавая ему размеры комнат}
    rooms[i].Init(a[i].length, a[i].width);
 End;
```

```
Function TFlat.FlatSquare;  {тело метода определения площади}
  Var S:real;
      i:integer;
  Begin
     S:=0;
     for i:=1 to n do  {суммируем площади комнат}
                       S:=S+rooms[i].Square;
     FlatSquare:=S;
  End;
Const mas:array[1..3]of TRoom= ((length:2.5;   width:3.75),
                                (length:2.85; width:4.1),
                                (length:2.3;   width:2.8));
Var F:TFlat;   {объявляем объект-переменную}
Begin
   F.Init(3,mas);              {инициализируем объект}
  WriteLn('Площадь квартиры=',F.FlatSquare);{определяем площадь}
End.
```

Задания для самопроверки

Задание 1. Реализуйте класс, диаграмма которого изображена на рис. 9.6, *а*. Разработайте тестирующую программу.

Задание 2. Спроектируйте класс для реализации объекта «Меню», который, получив сообщение «Изобразить меню», выводит на экран окно, включающее окна меньшего размера, содержащие строки – названия пунктов (см. пример 8.4). Разработайте тестирующую программу.

11.3. Наполнение

Наполнением называют способ конструирования классов, при котором в строящийся класс включают неопределенное количество: от 0 до сравнительно больших значений (на практике обычно до нескольких десятков) объектов других классов.

Пример 11.3. Разработать класс для реализации объекта Комната с балконом, который должен реагировать на запрос об общей площади (рис. 11.7).

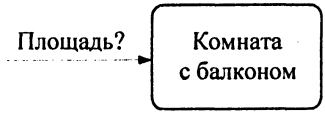


Рис. 11.7. Объект Комната с балконом

Балкон в комнате может существовать или нет. Для простоты будем считать, что в комнате может быть не более одного балкона прямоугольной формы. Следовательно, разрабатываемый класс можно наследовать от класса

TRoom и включить в него указатель на объект класса TRoom (рис. 11.8). Он должен добавлять свои методы определения площади и инициализации объектов, учитывающие наличие или отсутствие балкона.

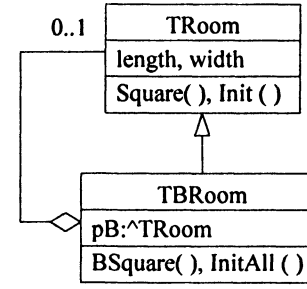


Рис. 11.8. Диаграмма классов для TBRoom

```
Program ex;
 Uses Room;
 Type
      TBRoom=object(TRoom)
         pB: ^TRoom;
         function BSquare:real;
         procedure InitAll(l,w:real; lb,wb:real);
       end;
 Procedure TBRoom. InitAll;
   Begin
      Init(l,w);
      if (lb=0) or (wb=0) then
                pB:=nil
         else
            begin
               New(pB);
               pB^.Init(lb,wb);
            end;
    End;
 Function TBRoom.BSquare;
   Begin
      if pB=nil then BSquare:= Square;
                else BSquare:= Square+pB^.Square;
   End;
Var B:TBRoom;   {объявляем объект-переменную}
 Begin
    B. InitAll(3.4,5.1,1.8,0.8); {инициализируем объект}
    WriteLn('Площадь комнаты с балконом =',B.BSquare);
 End.
```

Процесс выделения памяти под динамические поля объекта необходимо контролировать. С этой целью используют средства контроля выделения памяти, рассмотренные в параграфе 7.2. Кроме этих средств существуют специальные средства контроля выделения памяти под динамические поля, которые обычно используют при разработке конструкторов. Эти средства и приемы их использования будут рассмотрены в параграфе 11.7.

Задания для самопроверки

Задание 1. Реализуйте класс, диаграмма которого изображена на рис. 9.6, *б*. Разработайте тестирующую программу.

Задание 2. Спроектируйте класс для реализации объекта «Настраиваемое меню», который, получив сообщение «Изобразить меню», выводит на экран окно, включающее окна меньшего размера, содержащие строки – названия пунктов (см. пример 8.4). Объект должен создаваться в процессе работы программы. Количество пунктов меню и заголовки должны вводиться с клавиатуры. Разработайте тестирующую программу.

11.4. Простой полиморфизм

Как уже говорилось в параграфе 9.3, возможность переопределения методов при наследовании является частным случаем полиморфизма.

Пример 11.4. Разработать класс для реализации объекта Трехмерная комната 2, который должен реагировать на запрос о суммарной площади стен и потолка (рис. 11.9).

Класс, реализующий данный объект, будем наследовать от класса TRoom. Этот класс должен включать метод определения площади стен и потолка. Логично назвать этот метод Square, но метод с таким именем уже определен в базовом классе. Поскольку объект не должен реагировать на запрос о площади комнаты, метод Square базового класса можно переопределить (рис. 11.10).

Помимо переопределения метода Square класс TVRoom2 переопределяет метод инициализации полей объекта Init, добавляя определение значения нового поля height.

В Borland Pascal сохраняется возможность обращения к переопределенному родительскому методу из методов класса-потомка. С этой целью перед именем метода указывают служебное слово *inherited* или имя класса-родителя и точку:

inherited Square или *TRoom.Square*

Окончательно получаем следующую программу:

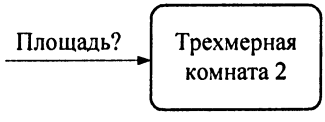


Рис. 11.9. Объект Трехмерная комната 2

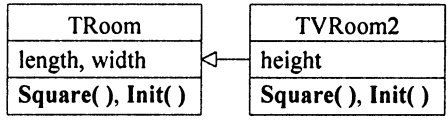


Рис. 11.10. Иерархия классов для класса TVRoom

```
Program ex;
Uses Room;
Type TVRoom2 = object(TRoom)
                height:real;          {дополнительное поле класса}
                function Square:real; {переопределенный метод класса}
                procedure Init(l,w,h:real);   {переопределенный
                                              инициализирующий метод}
              end;
Procedure TVRoom2.Init;
 Begin
    {инициализируем поля базового класса}
    inherited Init(l,w);    {или TRoom.Init(l,w) }
    height:=h; {инициализируем собственное поле класса}
 End;
Function TVRoom2.Square;
 Begin
     Square:=inherited Square+2*height*(length+width); {обращаемся
                              к переопределенному методу базового класса}
 End;
 Var A:TVRoom2;
 Begin
    A.Init(3.4,5.1,2.8);
    WriteLn('Площадь стен и потолка = ',A.Square);
 End.
```

Подключаемый метод в этом случае определяется типом объекта, для которого метод вызывают. Тип объекта известен уже на этапе компиляции программы: он совпадает с типом переменной-объекта, следовательно, и метод, вызываемый в программе, можно определить на этапе компиляции (раннее связывание). Данный вариант переопределения методов получил название простого полиморфизма, а сами методы были названы *статическими полиморфными.*

Списки параметров статических полиморфных методов могут различаться. Так, в рассмотренном выше примере метод Init класса TRoom имеет два параметра, а метод Init класса TVRoom2 – три параметра.

Обращение объекта производного класса к переопределенному методу базового класса из программы. Объект производного класса может обратиться к переопределенному методу базового класса из программы, но для этого необходимо явно переопределить тип объекта, используя имя базового класса как функцию:

<имя базового класса>(<имя объекта производного класса>).<имя метода>.

Например:

```
Var A: TVRoom2; ...
TRoom(A).Square; ...   {вызываем метод базового класса}
```

Такое переопределение типа в ООП называют *восходящим приведением* типа в отличие от *нисходящего*, которое используется, если требуемые методы или поля производного класса при обращении к объекту того же класса через указатель базового класса не видны (рис. 11.4). Восходящее приведение типа возможно всегда, в то время как при выполнении нисходящего приведения необходимо быть уверенным, что в данный момент времени указатель действительно содержит адрес объекта производного класса или его потомков.

11.5. Сложный полиморфизм. Конструкторы

Существуют три случая, в которых определение типа объекта на этапе компиляции программы невозможно, и, следовательно, невозможно правильное подключение переопределенного метода. Рассмотрим один из таких случаев.

Пример 11.5. Разработать классы для реализации объекта Комната П, который должен отвечать на запрос о площади пола, выводя результат сразу на экран, и объекта Трехмерная комната П, который должен отвечать на запрос о площади стен и потолка, также выводя результат на экран (рис. 11.11).

Класс TRoom2 строим аналогично классу TRoom, добавив метод вывода результата на экран Print. Класс TVRoomP наследуем от TRoomP, переопределив метод определения площади и метод инициализации полей объекта (рис. 11.11).

В результате классы будут описаны следующим образом.

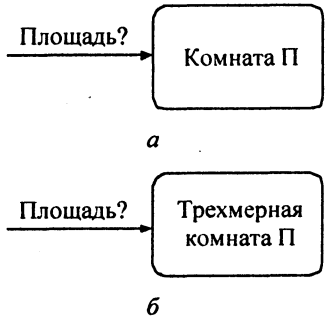


Рис. 11.11. Объекты: Комната П (*а*) и Трехмерная комната П (*б*)

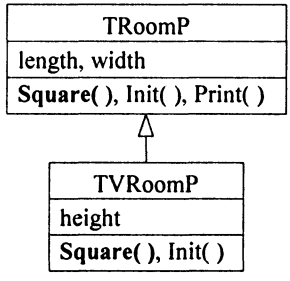


Рис. 11.12. Иерархия классов для TVRoomP

Вариант 1 – **с ошибкой!**

```
Program ex;
  Type TRoomP=object
            length, width:real;{поля: длина и ширина комнаты}
            function Square:real; {метод определения площади}
            procedure Print;{метод вывода результата на экран}
            procedure Init(l,w:real);{инициализирующий метод}
          end;
Function TRoomP.Square;   {метод определения площади}
  Begin
     Square:= length* width;
  End;
Procedure TRoomP.Print;    {метод вывода результатов}
  Begin
     WriteLn('Площадь =', Square:6:2);   {внутренний вызов метода}
  End;
Procedure TRoomP.Init;   {тело инициализирующего метода}
  Begin
     length:=l;        width:=w;
  End;
Type TVRoomP = object(TRoomP)
            height:real;          {дополнительное поле класса}
            function Square:real; {переопределенный метод класса}
            procedure Init(l,w,h:real);    {переопределенный
                                          инициализирующий метод}
     end;
Procedure TVRoomP.Init;
  Begin
     inherited Init(l,w);{инициализирует поля базового класса}
     height:=h;      {инициализируем собственное поле класса}
  End;
Function TVRoomP.Square;
  Begin   {обращаемся к методу базового класса}
     Square:=inherited Square + 2*height*(length + width);
  End;
  Var A:TRoomP; B:TVRoomP;  {объявляем объекты-переменные}
  Begin
     A.Init(3.5,5.1);      {инициализируем поля объекта А}
     A.Print;              {выведет «Площадь = 17.85»}
     B.Init(3.5,5.1,2.7); {инициализируем поля объекта В}
     B.Print;              {выведет «Площадь = 17.85» – ошибка!!!}
  End.
```

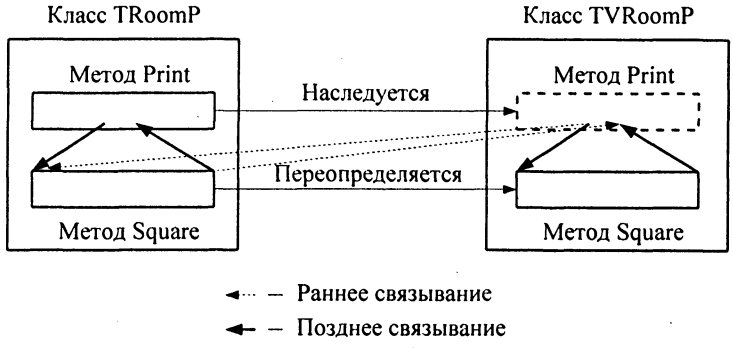


Рис. 11.13. Необходимость позднего связывания

Ошибка возникла из-за того, что метод Print, который наследуется классом TVRoomP, вызывает метод Square. Метод Square в производном классе переопределяется, но метод Print ничего об этом не «знает» и по-прежнему вызывает метод Square класса TRoomP (см. пунктирные стрелки на рис. 11.13).

Для того чтобы метод базового класса мог в зависимости от типа объекта, для которого он вызван, обращаться либо к методу базового класса, либо к переопределенному методу производного класса, необходимо тип объекта определять *на этапе выполнения программы* (*позднее связывание*). Переопределение методов в этом случае называют *сложным полиморфизмом*, а соответствующие методы – *виртуальными полиморфными*.

Для организации сложного полиморфизма необходимо:

1) переопределяемые методы описать служебным словом virtual;

2) к методам класса с виртуальными полиморфными методами добавить специальный метод-процедуру – *конструктор*, в котором служебное слово procedure заменено служебным словом constructor;

3) вызвать конструктор прежде, чем произойдет первое обращение к виртуальным полиморфным методам.

Методы, объявленные виртуальными полиморфными, на этапе компиляции подключаться не будут. Для каждого класса, содержащего виртуальные полиморфные методы, будет построена специальная внутренняя таблица виртуальных методов (ТВМ), в которой будут записаны адреса виртуальных полиморфных методов. Этой таблицей пользуются все объекты данного класса для определения адресов виртуальных полиморфных методов на этапе выполнения программы.

Адрес ТВМ хранится в объекте в специальном внутреннем, невидимом для программиста поле размером 2 байта (рис. 11.14). Запись адреса ТВМ в это поле происходит *неявно* при выполнении конструктора, поэтому попытки вызовов виртуальных полиморфных методов до выполнения конструктора приводят к ошибкам нарушения адресации и «зависанию» компьютера.

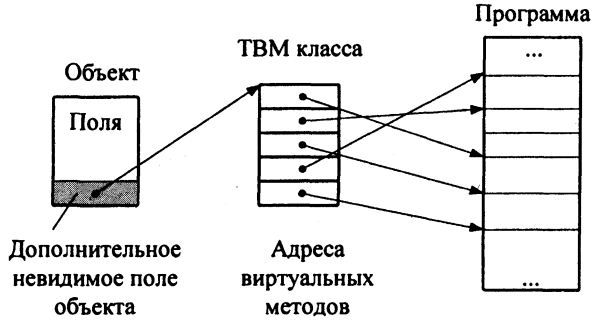


Рис. 11.14. Связь объекта с ТВМ

Как правило, в качестве конструктора используют метод инициализации полей, так как его обычно вызывают в начале работы с объектом. При повторном вызове конструктора никаких дополнительных действий не выполняется, метод работает как обычная процедура.

Пример 11.5. Продолжение. Исправим ошибку предыдущего варианта программы, объявив метод Square виртуальным полиморфным и используя метод Init в качестве конструктора.

Вариант 2 – правильный

```
Program case1;
  Type TRoomP=object
            length, width:real;{поля: длина и ширина комнаты}
            function Square:real; virtual; {метод определения площади}
            procedure Print;       {метод вывода результата на экран}
            constructor Init(l,w:real); {конструктор}
          end;
 Function TRoomP.Square;{тело метода определения площади}
  Begin
     Square:= length* width;
  End;
 Procedure TRoomP.Print; {тело метода вывода результатов}
  Begin
     WriteLn('Площадь =', Square:6:2); {теперь вызов метода
                                        происходит через ТВМ класса }
  End;
Constructor TRoomP.Init;      {тело конструктора}
  Begin
     length:=l;
     width:=w;
  End;
```

```
Type TVRoomP = object(TRoomP)
        height:real;          {дополнительное поле класса}
        function Square:real; virtual; {виртуальный полиморфный метод}
        constructor Init(l,w,h:real);    {конструктор}
    end;
Constructor TVRoomP.Init;
 Begin
     inherited Init(l,w);{инициализируем поля базового класса}
     height:=h;     {инициализируем собственное поле класса}
 End;
Function TVRoomP.Square;
 Begin
     Square:=inherited Square+2*height*(length+ width);
 End;
Var A:TRoomP; B:TVRoomP;  {объявляем объекты-переменные}
Begin
   A.Init(3.5,5.1);   {конструируем объект A}
   A.Print;      {выведет «Площадь = 17.85»}
   B.Init(3.5,5.1,2.7); {конструируем объект B}
   B.Print;      {выведет «Площадь = 94.64» – верно!!!}
End.
```

Определены три случая, когда использование позднего связывания обязательно:

1) наследуемый метод для объекта производного класса вызывает метод, переопределенный в производном классе – пример такой ситуации рассмотрен выше;

2) объект производного класса через указатель базового класса обращается к методу, переопределенному производным классом;

3) процедура вызывает переопределенный метод для объекта производного класса, переданного в процедуру через *параметр-переменную*, описанный как объект базового класса (данную ситуацию часто называют «процедурой с полиморфным объектом»).

Примечание. При использовании параметров-значений аргумент-объект копируется. При этом копируется объект типа параметра, т.е. базового. Следовательно, передать в процедуру объект производного класса не удастся.

Собственно, все три случая сводятся к одному: обращение к полиморфному методу выполняется через указатель базового класса, которому может быть присвоен адрес объекта не только базового, но и производного класса. В теории программирования объект, адресуемый подобным указателем, получил название *полиморфного*.

Рассмотрим более подробно второй и третий случаи и покажем, что действительно использование сложного полиморфизма применительно к ним обязательно.

Пример 11.6. Разработать классы для реализации двух динамических объектов: объект Комната Д должен отвечать на запрос о площади, объект Трехмерная комната Д должен отвечать на запрос о площади стен и потолка (рис. 11.15). *Предусмотреть возможность обращения к полям и методам производного класса через указатель на базовый класс.*

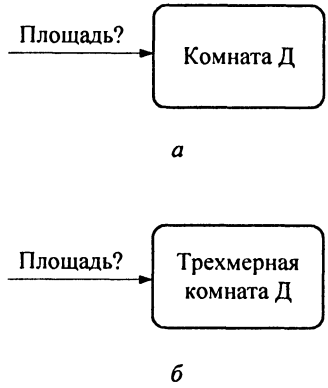


Рис. 11.15. Объекты Комната Д (*а*) и Трехмерная комната Д (*б*)

Между классами прослеживается отношение наследования, как в предыдущем примере, соответственно строим один на базе второго. Иерархия классов приведена на рис. 11.16.

Необходимость использования сложного полиморфизма для метода Square в разрабатываемой программе связана с тем, что при выполнении программы доступ к объекту производного класса выполняется через указатель базового класса. При компиляции программы тип объекта считается соответствующим типу указателя, что в данном случае становится неверным. Поэтому подключение переопределенного метода Square следует выполнять через ТВМ, чтобы требуемый метод определялся на этапе выполнения программы.

В настоящем примере не будем конструировать объект в динамической памяти и освобождать память после работы с ним, так как создание и уничтожение динамических полиморфных объектов имеет свои особенности, которые будут рассмотрены в параграфе 11.6. Вместо этого осуществим доступ через указатель к статически созданному объекту. Кроме того, поскольку следующий пример будет работать с теми же классами, поместим описание классов в модуль.

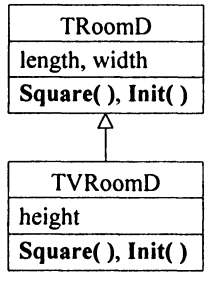


Рис. 11.16. Иерархия классов для реализации объектов

```
Unit RoomMod;
Interface
 Type
      TRoomD=object
        length,width:real;{поля: длина и ширина
                                        комнаты}
        function Square:real; virtual; {метод опреде-
                                        ления площади}
        constructor Init(l,w:real); {конструктор}
      end;
```

```
  Type TVRoomD = object(TRoomD)
        height:real;          {дополнительное поле класса}
        function Square:real;virtual; {виртуальный полиморфный метод}
        constructor Init(l,w,h:real);    {конструктор}
     end;
   Implementation
 Function TRoomD.Square;{тело метода определения площади}
  Begin
     Square:= length* width;
  End;
Constructor TRoomD.Init;      {тело конструктора}
  Begin
     length:=l;
     width:=w;
  End;
Constructor TVRoomD.Init;
  Begin
     inherited Init(l,w);{инициализирует поля базового класса}
     height:=h;     {инициализируем собственное поле класса}
  End;
Function TVRoomD.Square;
  Begin
    Square:=inherited Square+2*height*(length+ width);
  End;
End.
```

Тогда основная программа будет выглядеть следующим образом.

```
Program case2;
Uses RoomMod;
Var pA: ^TRoomD; {объявляем указатель на объекты класса}
    B:TVRoomD;  {объявляем объект класса}
Begin
  B.Init(3.5,5.1,2.7); {конструируем объект B}
  WriteLn('Площадь=', B.Square:6:2);      {выведет «Площадь= 94.64»}
  pA:=@B;     {присваиваем указателю базового класса адрес объекта
                                                производного класса}
  WriteLn('Площадь=', pA^.Square:6:2);  {выведет «Площадь= 94.64»}
End.
```

Пример 11.7. Разработать классы для реализации объекта «Комната Д», который должен отвечать на запрос о площади, и объекта «Трехмерная комната Д», который должен отвечать на запрос о площади стен и потолка. *Пре-*

дусмотреть внешнюю процедуру вывода результатов, которая получает адрес объекта через параметр-переменную.

Если при разработке классов не объявить метод Square виртуальным полиморфным, то из процедуры Print и для аргумента – объекта базового класса, и для аргумента – объекта производного класса будет вызываться метод Square базового класса.

```
Program case3;
Uses RoomMod;
Procedure Print(Var R:TRoomD); {процедура с полиморфным объектом}
 Begin
    WriteLn('Площадь=', R.Square:6:2);
 End;
Var A:TRoomD; B:TVRoomD;  {объявляем объекты-переменные}
Begin
   A.Init(3.5,5.1);      {конструируем объект A}
   B.Init(3.5,5.1,2.7); {конструируем объект B}
   Print(A);              {выведет «Площадь= 17.85»}
   Print(B);              {выведет «Площадь= 94.64»}
End.
```

При необходимости во всех трех случаях использования сложного полиморфизма можно определить конкретный тип полиморфного объекта, используя специальную функцию:

TypeOf(<имя класса или объекта>***):pointer*** – возвращает адрес ТВМ класса. Если адреса ТВМ объекта и класса совпадают, то объект является переменной данного класса. Например:

```
if TypeOf(Self) = TypeOf(<имя класса>)
        then <объект принадлежит классу>
        else  <объект не принадлежит классу>
```

При выборе механизма переопределения методов (с ранним или поздним связыванием) в тех случаях, когда это не вызвано необходимостью, следует помнить, что:

1) позднее связывание требует построения ТВМ, а следовательно больше памяти;

2) вызов виртуальных полиморфных методов происходит через ТВМ, а следовательно медленнее;

3) список параметров одноименных виртуальных полиморфных методов должен совпадать, а статических полиморфных – не обязательно;

4) статический полиморфный метод не может переопределить виртуальный полиморфный метод.

Операции присваивания полиморфных объектов. При выполнении операции присваивания для полиморфных объектов следует помнить, что объект, которому присваивается значение, должен быть сконструирован, т.е. для него должен быть вызван конструктор. Если объект сконструирован не был, то операция присваивания выполняется некорректно: поля объекта просто обнуляются, и никаких сообщений об ошибке не выдается. Программа же в этом случае естественно работает неправильно.

Задания для самопроверки

Задание 1. Реализуйте классы, диаграмма которых изображена на рис. 9.7. Разработайте тестирующую программу.

Задание 2. Спроектируйте классы для реализации объектов «Меню функций», «Меню операций» и «Функции» из примера 9.1. Используя объекты этих классов и объекты классов, разработанных в задании 1, реализуйте программу, функционирующую в соответствии с заданием примера 9.1.

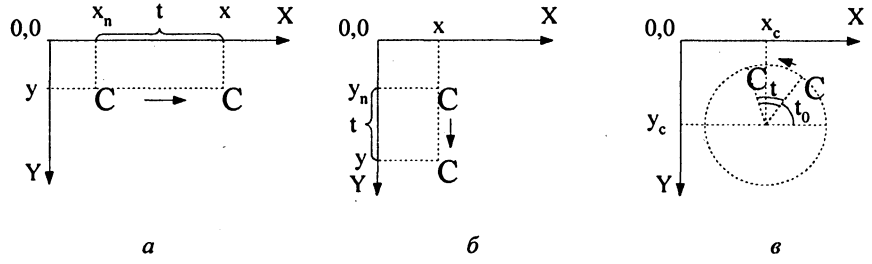
11.6. Практикум. Использование полиморфизма при создании движущихся изображений

Полиморфное наследование – очень мощное средство разработки классов. Переопределение методов при наследовании позволяет изменить поведение объектов, наследуя классы для их реализации от уже разработанных. При создании движущихся изображений полиморфизм позволяет определять разные законы движения объектов родственных классов.

Пример 11.8. Разработать программу, реализующую на экране движение символов по заданным траекториям: по горизонтали слева направо, по вертикали сверху вниз и по окружности, размещенной в центре экрана.

Объектная декомпозиция предметной области программы изображена на рис. 11.17.

Рис. 11.17. Объектная декомпозиция программы «Движение символов»



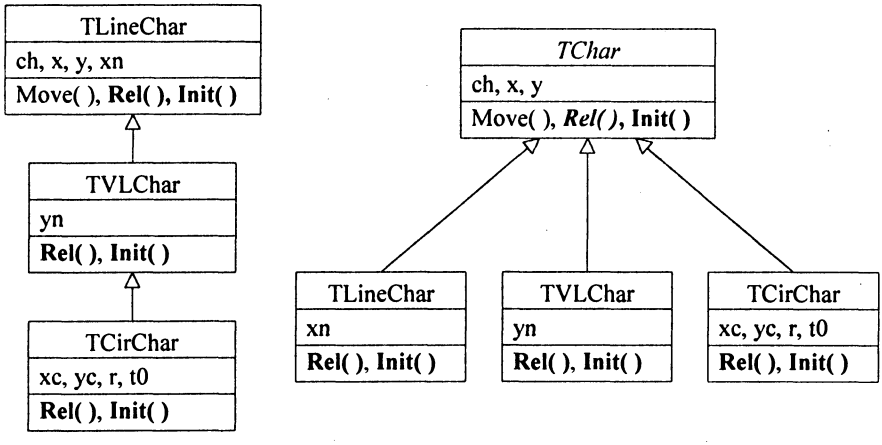
а *б* *в*

Рис. 11.18. Определение законов движения символов:
а – при движении по горизонтали; *б* – при движении по вертикали;
в – при движении по окружности

Каждому символу соответствует объект, движение которого происходит по своему закону (рис. 11.18). Инициализирует объекты и управляет их движением, посылая сообщение Перерисовать, основная программа, которая на данной декомпозиции представлена в виде объекта.

Классы для реализации объектов будут иметь много общего, различаясь только полями, определяющими траекторию движения, и методом, реализующим закон изменения положения объекта. В принципе они могут наследоваться один от другого, переопределяя метод Rel, конкретно реализующий закон пересчета координат для каждого вида движения (рис. 11.19, *а*). Однако в этом случае объекты будут включать лишние поля (объект класса TVLChar – поле xn и т.д.), что нежелательно.

В таких случаях целесообразно использовать *абстрактный* класс, реализующий некое обобщенное представление объекта, как символа, движу-



а *б*

Рис. 11.19. Два варианта иерархии классов:
а – обычная; *б* – с абстрактным классом

щегося по экрану (рис. 11.19, *б*). Конкретный закон изменения координат в данном классе определять не будем, но, поскольку соответствующий метод Rel будет вызываться из метода Move, реализующего движение, определим метод Rel пустым (*абстрактным*). Классы, наследуемые от абстрактного, должны переопределять этот метод, задавая свои законы изменения координат. Ниже приведен текст программы.

```
Program ex;
 Uses crt,Graph;
 {описание абстрактного класса}
 Type TChar=object
       ch:char;   {символ}
       x,y:integer;   {исходное положение}
       constructor Init(ach:char;ax,ay:integer);
       procedure Move(t:integer);
       procedure Rel(t:integer); virtual;
      End;
 Constructor TChar.Init;
  Begin
      ch:=ach;
      x:=ax;
      y:=ay;
   End;
 Procedure TChar.Rel;
  Begin  End;
 Procedure TChar.Move;
  Begin
     SetColor(GetBkColor);
     OuttextXY(x,y,ch);
     Rel(t);   {вызываем переопределяемый метод изменения координат}
     SetColor(ord(ch) mod 16);
     OutTextXY(x,y,ch);
  End;
{описание класса символа, перемещающегося по горизонтали}
 Type TLineChar=object(TChar)
           xn:integer;   {точка отсчета координат по горизонтали}
           constructor Init(ach:char;ax,ay:integer);
           procedure Rel(t:integer); virtual;
       End;
 Constructor TLineChar.Init;
  Begin
     inherited Init(ach,ax,ay);   xn:=ax;
  End;
```

```
Procedure TLineChar.Rel;
 Begin
     x:=(xn+t) mod GetMaxX;
 End;
{описание класса символа, перемещающегося по вертикали}
Type TVLineChar=object(TChar)
       yn:integer; {точка отсчета координат по вертикали}
       constructor Init(ach:char;ax,ay:integer);
        procedure Rel(t:integer); virtual;
    End;
Constructor TVLineChar.Init;
 Begin
    inherited Init(ach,ax,ay);
    yn:=ay;
 End;
Procedure TVLineChar.Rel;
 Begin
    y:=(yn+t) mod GetMaxY;
 End;
{описание класса символа, перемещающегося по окружности}
Type TCirChar=object(TChar)
      xc,yc,r:integer;  {параметры окружности}
      t0:real;          {исходное положение – начальный угол}
      constructor Init(ach:char;axc,ayc,ar:integer;at0:real);
      procedure Rel(t:integer); virtual;
    End;
Constructor TCirChar.Init;
 Begin
    inherited Init(ach,axc+round(ar*sin(at0)), ayc+round(ar*cos(at0)));
    xc:=axc;
    yc:=ayc;
    r:=ar;
    t0:=at0;
 End;
Procedure TCirChar.Rel;
 Begin
   x:=xc+Round(r*sin(t0+t*0.05));
   y:=yc+Round(r*cos(t0+t*0.05));
 End;
{объявление переменных}
Var A:TLineChar;
    B:TVLineChar;
    C:TCirChar;
```

```
    t:integer;
    i:integer;
    dr, md:integer;
{основная программа}
Begin
    dr:=detect;
    InitGraph(dr,md,'D:\BP\BGI');
    A.Init('a',0,25);
    B.Init('b',100,0);
    C.Init('c',GetMaxX div 2,GetMaxY div 2,80,0);
    t:=0;          {условное время движения}
    while not KeyPressed and (t<1000) do
      begin
        A.Move(t); {перерисовываем символы}
        B.Move(t);
        C.Move(t);
        t:=t+1;    {увеличиваем условное время движения}
        for i:=1 to 1000 do delay(1000);  {фиксируем кадр}
      end;
    CloseGraph;
End.
```

Задания для самопроверки

Задание 1. Разработайте программу, содержащую описание трех графических объектов: отрезка, правильного треугольника и квадрата. С использованием полиморфизма реализуйте вращение этих фигур вокруг их геометрических центров с разными скоростями и различными направлениями вращения.

Задание 2. Модифицируйте программу предыдущего задания, изменив закон движения объектов, например, пусть каждый из них движется сверху вниз. Оцените объем исправлений в программе.

Задание 3. Модифицируйте программу предыдущего задания, определив свой закон движения каждого объекта. Оцените объем исправлений в программе.

11.7. Динамические полиморфные объекты. Деструкторы

Borland Pascal содержит специальные средства, предназначенные для работы с динамическими полиморфными объектами и динамическими полями как динамических, так и статических объектов.

Конструирование и уничтожение динамических полиморфных объектов. Для выделения памяти под динамические полиморфные объекты целесообразно использовать функцию New, которая позволяет указателю базо-

вого класса присвоить адрес объекта производного класса и при необходимости сразу вызвать конструктор создаваемого объекта.

Функция ***New(***<тип указателя>[, <вызов конструктора>] ***)*** – возвращает адрес размещенного и, возможно, сконструированного объекта. Квадратные скобки означают, что второй параметр может быть опущен.

При уничтожении полиморфных объектов необходимо учитывать наличие у него дополнительного поля, содержащего адрес ТВМ. Для корректного освобождения памяти следует предварительно вызвать специальный метод класса – *деструктор*. Деструктором может служить любой, в том числе и пустой метод класса, при описании которого служебное слово procedure заменено служебным словом destructor. Если деструктор переопределяется, то его следует объявить виртуальным полиморфным, чтобы его вызов также выполнялся через ТВМ.

Обычно деструктору присваивается имя Done. Он может быть инициирован отдельным оператором или для его активизации можно использовать специальную форму процедуры освобождения памяти Dispose.

Процедура ***Dispose(***<указатель>[, <вызов деструктора>] ***)*** – выполняет вызов деструктора, если он указан, и освобождает память.

Пример 11.9. Разработать классы для реализации объекта «Комната Д2», который должен отвечать на запрос о площади пола, и объекта «Трехмерная комната Д2», который должен отвечать на запрос о площади стен и потолка. *Предусмотреть возможность создания динамических объектов разработанных классов и хранения адреса объекта производного класса в поле указателя базового класса (2-й случай обязательного использования сложного полиморфизма).*

Описание классов отличается от выполненных в программе примера 11.7 тем, что для обоих классов объявляется деструктор. Его придется объявить виртуальным полиморфным, так как возможен доступ через указатель базового класса к деструктору производного класса.

```
Program ex;
  Type
       pTRoomD2=^TRoomD2;
       TRoomD2=object
           length, width:real;{поля: длина и ширина комнаты}
           function Square:real; virtual; {метод определения площади}
           constructor Init(l,w:real);  {конструктор}
           destructor Done;             {деструктор}
         end;
Function TRoomD2.Square;   {тело метода определения площади}
  Begin
     Square:= length* width;
  End;
```

```
Constructor TRoomD2.Init;      {тело конструктора}
  Begin
     length:=l;
     width:=w;
  End;
Destructor TRoomD2.Done;
  Begin
  End;
Type pTVRoomD2=^ TVRoomD2;
     TVRoomD2 = object(TRoomD2)
          height:real;          {дополнительное поле класса}
          function Square:real; virtual; {виртуальный полиморфный
                                                          метод}
          constructor Init(l,w,h:real);   {конструктор}
        end;
Constructor TVRoomD2.Init;
  Begin
     inherited Init(l,w);{инициализирует поля базового класса}
     height:=h;    {инициализируем собственное поле класса}
  End;
Function TVRoomD2.Square;
  Begin
    Square:=inherited Square+2*height*(length+ width);
  End;
Var pA: pTRoomD2; pB:pTVRoomD2;  {объявляем указатели}
Begin
  {объект базового класса – указатель базового класса}
   pA:=New(pTRoomD2,Init(3.5,5.1));      {конструируем объект}
   WriteLn('Площадь=', pA^.Square:6:2);  {выведет «Площадь= 17.85»}
   Dispose(pA,Done);                     {уничтожаем объект}
  {объект производного класса – указатель производного класса}
   pB:=New(pTVRoomD2,Init(3.5,5.1,2.7));{конструируем объект}
   WriteLn('Площадь=', pB^.Square:6:2);  {выведет «Площадь= 94.64»}
   Dispose(pB,Done);                     {уничтожаем объект}
  {проявление полиморфных свойств:
   объект производного класса – указатель базового класса}
   pA:=New(pTVRoomD2,Init(3.5,5.1,2.7));{конструируем объект}
   WriteLn('Площадь=', pA^.Square:6:2);  {выведет «Площадь= 94.64»}
   Dispose(pA,Done);                     {уничтожаем объект}
End.
```

Динамические поля в статических и динамических полиморфных объектах. Если при разработке классов для реализации полиморфных объек-

тов используют динамические поля (объектные или нет), то запрос памяти под них обычно помещают в конструктор, а освобождение памяти – в деструктор. Тогда при уничтожении объекта автоматически освобождается память, отведенная для его полей.

При работе с динамическими объектами или с динамическими полями в статических объектах целесообразно использовать средства контроля выделения памяти, описанные в параграфе 7.2.

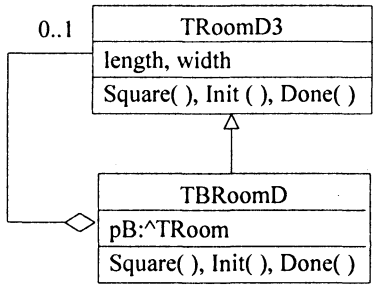


Рис. 11.20. Диаграмма классов

Возможна ситуация, когда в конструкторе запрашивается память под размещение нескольких динамических полей, но реально удовлетворяется только часть из них, например, только запрос памяти под размещение самого объекта. Естественно, такой объект в программе использоваться не может. В Borland Pascal существует специальный оператор fail, при выполнении которого все ранее удовлетворенные запросы на память аннулируются. При этом, если объект динамический, то указателю присваивается значение nil, а если объект статический, то конструктор, несмотря на то, что формально он является процедурой, возвращает значение false.

Пример 11.10. Разработать класс для реализации объекта «Комната с балконом Д», который должен реагировать на запрос об общей площади пола и о площади балкона. *Предусмотреть возможность создания динамических и статических объектов, возможность сохранения адреса объекта-потомка в указателе типа объекта-родителя и контроль выделения памяти.*

Для реализации объекта используем иерархию с наполнением (рис. 11.20). В качестве базового используем класс TRoomD3, аналогичный TRoomD2, разработанному в предыдущем примере. Единственное отличие этого класса заключается в том, что его деструктор объявлен виртуальным, так как он переопределяется классом TBRoomD, который использует деструктор для освобождения памяти, выделенной под динамическое поле.

```
Program ex;
 Type pTRoomD3=^TRoomD3;
      TRoomD3=object
          length, width:real;        {поля: длина и ширина комнаты}
          function Square:real; virtual; {метод определения площади}
          constructor Init(l,w:real);    {конструктор}
          destructor Done; virtual;      {деструктор}
        end;
```

```
Function TRoomD3.Square;   {метод определения площади}
  Begin
     Square:= length* width;
  End;
Constructor TRoomD3.Init;  {конструктор}
  Begin
     length:=l;
     width:=w;
  End;
Destructor TRoomD3.Done;
  Begin
  End;
Type pTBRoomD=^TBRoomD;
      TBRoomD=object(TRoomD3)
                    pB:pTRoomD3;
                    function Square:real; virtual;
                    function BSquare:real;   {площадь балкона}
                    constructor Init(l,w:real;  lb,wb:real);
                    destructor Done; virtual;
                  end;
  Constructor TBRoomD.Init;
     Begin
        inherited Init(l,w);
        if (lb=0)or(wb=0) then pB:=nil
          else
            begin
              New(pB);
              if pB=nil then
                 begin
                   WriteLn('Не хватает памяти для размещения поля');
                   Done; {завершение обработки; не нужно, если такой
                                          обработки нет и деструктор пустой}
                   fail; {отменяет все выполненные заказы на память}
                 end
              else  pB^.Init(lb,wb);
            end;
     End;
  Function TBRoomD.BSquare;
     Begin
        if pB<>nil then BSquare:=pB^.Square
                   else BSquare:=0;
     End;
```

```
  Function TBRoomD. Square;
     Begin
        Square:= inherited Square+BSquare;
     End;
  Destructor TBRoomD.Done;
    Begin  {освобождаем память, выделенную под динамическое поле}
       if pB<>nil then Dispose(pB);
    End;
Function HeapFunc(size:Word):integer; far;
 Begin HeapFunc:=1; end;
Var A:TBRoomD; pB1:pTBRoomD; pB2:pTRoomD3;
  Begin  {берем на себя обработку ошибок выделения памяти}
     HeapError:=@HeapFunc;
     {статический объект с динамическим полем}
     if A.Init(3.2,5.1,2.5,1) then {конструктор возвращает true или false}
       begin
         WriteLn(A.Square:6:2,A.BSquare:6:2);   {выводим площади}
         A.Done;    {вызываем деструктор как обычную процедуру}
       end
     else WriteLn('Не хватает памяти для размещения объекта.');
     {динамический объект с динамическим полем – указатель типа
      класса-потомка}
     pB1:=New(pTBRoomD,Init(3.2,5.1,2.5,1));{new вернет адрес или nil}
     if pB1<>nil then
       begin
         WriteLn(pB1^.Square:6:2,pB1^.BSquare:6:2);{выводим площади}
         Dispose(pB1,Done);    {уничтожаем объект}
       end
     else writeln('Не хватает памяти для размещения объекта.');
     {динамический объект с динамическим полем – указатель типа
      класса-родителя}
     pB2:=new(pTBRoomD,Init(3.2,5.1,2.5,1));{new вернет адрес или nil}
     if pB2<>nil then
       begin
         WriteLn(pB2^.Square:6:2,{необходимо позднее связывание –
                                                  случай 2}
           pTBRoomD(pB2)^.BSquare:6:2);  {явное переопределение типа
                 указателя, иначе метод класса-потомка для указателя
                 типа класса-родителя не "виден" (см. параграф 11.1)}
         Dispose(pB2,Done);{позднее связывание – случай 2}
       end
     else WriteLn('Не хватает памяти для размещения объекта.');
End.
```

11.8. Практикум. Создание контейнеров

Контейнером в ООП называют структуру, объединяющую объекты различных типов (классов). *Контейнерный класс* – это класс, реализующий контейнер. Как правило, такой класс включает массив или некую динамическую структуру (например, список или дерево), содержащую указатели на объекты базового класса. Все классы, объекты которых мы собираемся помещать в контейнер, должны наследоваться от данного базового класса. По правилам Borland Pascal указателям на объекты базового класса можно присваивать адреса объектов производных классов, соответственно контейнер может хранить объекты производных классов и манипулировать ими.

В функции контейнеров обычно входит создание объектов разных классов, их последовательная обработка и уничтожение.

В процессе создания объектов под них отводят память и выполняют инициализацию полей.

Для выполнения последовательной обработки обычно классы объектов строят таким образом, чтобы они включали методы с одинаковыми именами (полиморфные), возможно выполняющие различные действия для объектов различных классов. Однако при последовательной обработке возможна и проверка типа (класса) конкретного объекта (см. параграф 11.5), и выполнение для него специфических действий.

Уничтожение объектов требует освобождения выделенной памяти, размер которой при использовании виртуальных методов искусственно увеличивается на размер ссылки на ТВМ. Следовательно, базовый класс обязательно должен включать деструктор, возможно переопределяемый в производных классах. Причем с учетом возможного переопределения деструктор обязательно должен объявляться виртуальным.

Пример 11.11. Разработать программу, которая осуществляет движение строк по экрану: по горизонтали, по вертикали и по окружности.

В результате объектной декомпозиции получаем объекты четырех типов: управляющий объект и три объекта-строки, различающиеся законами движения (рис. 11.21).

Управляющий объект можно реализовать как основную программу, не строя соответствующий класс.

Чтобы реализовать заданные законы движения, необходимо каждую строку рассматривать как совокупность символов, перемещающихся по одной траектории, но с некоторым смещением. В примере, рассмотренном в параграфе 11.6, уже были разработаны классы, реализующие заданные законы перемещения символов. Используем эти классы для реализации объектов-символов, входящих в объекты-строки. Тогда каждая строка будет представлять собой контейнер, который управляет движением своих символов (рис. 11.22), вызывая их методы Move.

Рис. 11.21. Объектная декомпозиция предметной области программы «Перемещение строк»

Ниже приведен текст программы.

```
Program ex;
 Uses crt,Graph;
 Type pChar=^TChar;
      {описание абстрактного класса}
      TChar=object
          ch:char;
          x,y:integer;
          constructor Init(ach:char;ax,ay:integer);
          procedure Move(t:integer);
          procedure Rel(t:integer); virtual;
          destructor Done;virtual;  {деструктор обязателен, так как
                                     объекты динамические полиморфные}
      End;
```

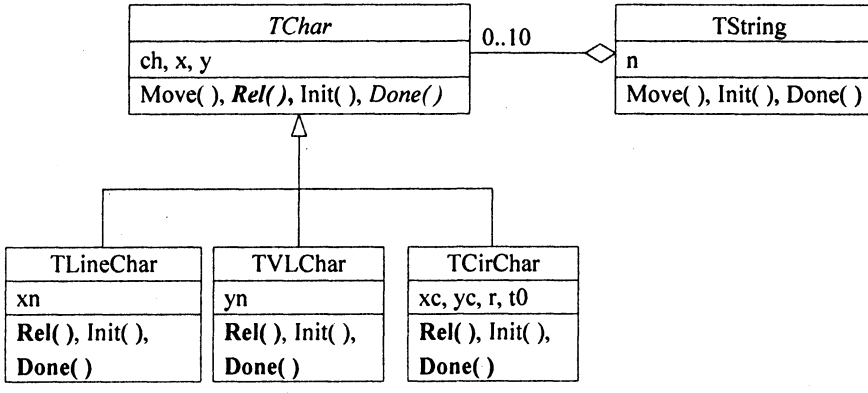


Рис. 11.22. Диаграмма классов программы «Перемещение строк»

```
Constructor TChar.Init;
   Begin
      ch:=ach;
      x:=ax;
      y:=ay;
   End;
Procedure TChar.Rel;
  Begin  End;
Procedure TChar.Move;
  Begin
    SetColor(GetBkColor);
    OuttextXY(x,y,ch);
    Rel(t);    {изменяем координаты}
    SetColor(ord(ch) mod 16);
    OutTextXY(x,y,ch);
  End;
Destructor TChar.Done;
  Begin  End;    {деструктор пустой, так как объект не содержит
                                                  объектных полей}
Type pLChar=^TLineChar;
 {описание класса символа, перемещающегося по горизонтали}
     TLineChar=object(TChar)
        xn:integer;
        constructor Init(ach:char;ax,ay:integer);
        procedure Rel(t:integer); virtual;
     End;
Constructor TLineChar.Init;
   Begin
      inherited Init(ach,ax,ay);
      xn:=ax;
   End;
Procedure TLineChar.Rel;
   Begin
      x:=(xn+t) mod GetMaxX;
   End;
Type pVChar=^TVLineChar;
   {описание класса символа, перемещающегося по вертикали}
    TVLineChar=object(TChar)
        yn:integer;
        constructor Init(ach:char;ax,ay:integer);
        procedure Rel(t:integer); virtual;
    End;
```

```
Constructor TVLineChar.Init;
    Begin
        inherited Init(ach,ax,ay);
        yn:=ay;
    End;
Procedure TVLineChar.Rel;
    Begin
        y:=(yn+t) mod GetMaxY;
    End;
Type pCChar=^TCirChar;
  {описание класса символа, перемещающегося по окружности}
    TCirChar=object(TChar)
        xc,yc,r:integer; t0:real;
        constructor Init(ach:char;axc,ayc,ar:integer;at0:real);
        procedure Rel(t:integer); virtual;
    End;
Constructor TCirChar.Init;
  Begin
      inherited  Init(ach,axc+round(ar*sin(at0)),ayc+ round(ar* cos(at0)));
        xc:=axc;
        yc:=ayc;
        r:=ar;
        t0:=at0;
  End;
Procedure TCirChar.Rel;
   Begin
      x:=xc+Round(r*sin(t0+t*0.05));
      y:=yc+Round(r*cos(t0+t*0.05));
   End;
Type TString=object
          mas:array[1..10] of pChar;  {массив указателей на объекты}
          n:integer;                  {реальное количество объектов}
          procedure Init(as:string; tmove:byte);  {создание объектов}
          procedure Move(t:integer);              {перемещение строк}
          procedure Done;                         {уничтожение объектов}
       End;
Procedure TString.Init;
  Var i:integer;
  Begin
      n:=length(as);
```

```
  for i:=1 to n do
    begin
      case tmove of
       1:mas[i]:=new(pLChar,Init(as[i], 9*i, GetMaxY div 2));
       2:mas[i]:=new(pVChar,Init(as[i], GetMaxX div 2-n*5+10*i, 0));
       3:mas[i]:=new(pCChar,Init(as[i], GetMaxX div 2, GetMaxY div 2,
                                                    100, 2*pi*(i-1)/n));
      end;
    end;
 End;
Procedure TString.Move;
Var i:integer;
   Begin
      for i:=n downto 1 do mas[i]^.Move(t);
   End;
Procedure TString.Done;
 Var i:integer;
 Begin
     for i:=1 to n do dispose(mas[i],Done);
 End;
Var s:string; M:array[1..3] of TString;  i,t,dr,md:integer;
Begin
    Write('Введите строку до 10 символов:');
    ReadLn(s);
    InitGraph(dr,md,'d:\bp\bgi');
    for i:=1 to 3 do M[i].Init(s,i);
    t:=0;
    while not KeyPressed and (t<=1000) do
        begin
          for i:=1 to 3 do M[i].Move(t);   {перемещаем строки}
           t:=t+1;
           for i:=1 to 1000 do delay(1000);
        end;
     for i:=1 to 3 do M[i].Done;
      CloseGraph;
End.
```

В нашем случае каждая строка содержит символы – объекты одного класса, но это вовсе не обязательно, просто, если в нашей задаче объекты-символы будут разных типов, то буквы строки будут «разлетаться» в разные стороны.

Задания для самопроверки

Задание 1. Разработайте программу, изображающую на экране снежинки трех типов: меняющие цвет, растущие до заданного значения и падающие вниз. Для реализации указанного эффекта спроектируйте класс, реализующий объект, управляющий изменением n снежинок, тип каждой из которых определяется случайным образом.

Задание 2. Дополните программу предыдущего задания возможностью формирования снежинок, перемещающихся по горизонтали слева направо. Оцените объем и локализацию *изменений*.

12. РАЗРАБОТКА БИБЛИОТЕКИ ИНТЕРФЕЙСНЫХ КОМПОНЕНТОВ

Создание пользовательских интерфейсов – одна из наиболее трудоемких задач программирования, и для ее решения успешно применяют средства ООП. Современный программист, работающий в одной из профессиональных сред программирования, при написании программы с развитым интерфейсом обычно использует большие и достаточно сложные библиотеки классов для реализации интерфейсных компонентов. Прежде чем изучать подобные библиотеки, желательно ознакомиться с общими принципами их построения. С этой целью попытаемся разработать упрощенный вариант такой библиотеки.

12.1. Анализ реальной программы и определение основных интерфейсных компонентов

Создание библиотеки универсальных интерфейсных элементов начнем с того, что на примере конкретной задачи выясним, какие элементы целесообразно в такую библиотеку включать.

Пример 12.1. Разработать программу «Записная книжка», которая должна осуществлять: создание новой книжки (файла), добавление записей (фамилии, имени и телефона), поиск записей по фамилии и/или имени.

По сути дела данная программа должна обеспечивать удобный интерфейс для хранения и поиска информации в некотором файле. Разработку начинаем с уточнения интерфейса.

Главное меню программы в соответствии с условием задачи должно вызывать основные функции для работы с «Записной книжкой» (рис. 12.1). Выбор функции будем осуществлять клавишами горизонтального перемещения курсора, а ее вызов – нажатием клавиши Enter. Для выхода из программы предусмотрим специальный пункт меню, но будем осуществлять завершение программы или возврат из функций и по нажатию клавиши Esc.

Диаграмма переходов состояний интерфейса, отражающая процесс работы пользователя с программой, приведена на рис. 12.2.

Рис. 12.1. Окно главного меню

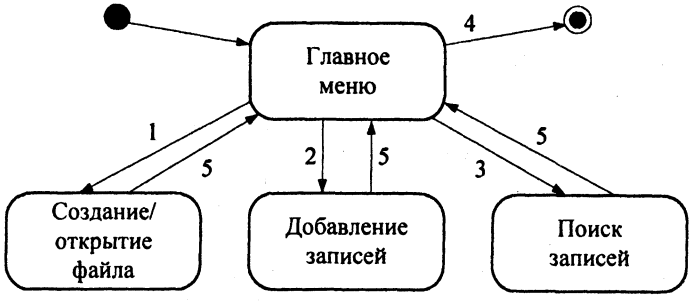


Рис. 12.2. Диаграмма переходов состояний интерфейса:
1 – 4 – выбор пунктов «Открытие файла», «Добавление записей», «Поиск записей» и «Выход» соответственно; 5 – выбор пунктов «Выход» на формах «Открытие файла», «Добавление записей», «Поиск записей»

Каждая функция будет выполняться в своем окне. По необходимости окна функций будут иметь локальные меню, управление которыми также будет осуществляться с помощью клавиатуры.

При выполнении пункта «Создать или открыть книжку» на экране должно появиться окно ввода имени файла (рис. 12.3), причем до ввода имени файла выбор прочих пунктов меню, кроме «Завершения работы», должен быть блокирован. После ввода имени файла, которое должно быть проверено с точки зрения синтаксиса имен файлов, программа будет пытаться открыть файл. Если файл с указанным именем не будет обнаружен, то она создаст файл с таким именем.

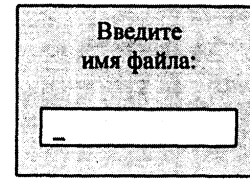


Рис. 12.3. Окно ввода имени файла

При выборе пункта «Добавление записей» на экране должна появляться форма ввода записей (рис. 12.4), которая содержит поля ввода и локаль-

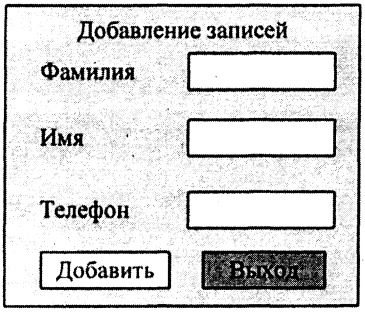


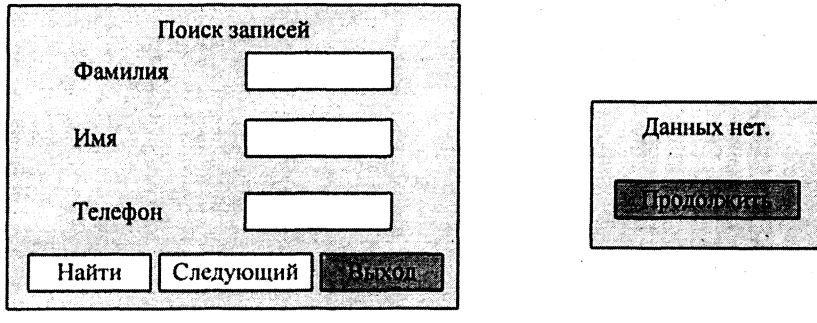
Рис. 12.4. Форма ввода записей

ное меню. Добавление записей происходит при выборе пункта «Добавить» локального меню. При этом все поля очищаются, а курсор вновь устанавливается в поле ввода фамилии.

Добавляемая информация может быть не полной, например, может быть известно только имя или только фамилия абонента, тогда пользователь должен перейти в следующее поле формы, нажав клавишу Enter: соответствующее поле записи файла будет пустым. После внесения всех записей пользователь должен выбрать пункт «Выход» и вновь вернуться в главное меню.

При выборе пункта «Поиск записей» на экране должна появляться форма поиска (рис. 12.5, *а*), которая содержит поле ввода фамилии, имени, поле вывода телефона и локальное меню из трех пунктов. Поиск должен осуществляться по вводу фамилии и/или имени. Начало поиска по выбору пункта «Поиск». Если запись не найдена, то программа должна выдавать сообщение об отсутствии данных (рис. 12.5, *б*). Если необходимо найти несколько записей, то переход к поиску следующей записи осуществляется по выбору пункта «Следующий» (рис. 12.6). После завершения поиска пользователь вновь должен вернуться в главное меню, выбрав пункт «Выход».

Осуществим объектную декомпозицию программы. Будем считать, что каждому элементу интерфейса соответствует интерфейсный объект. Полученная при этом диаграмма интерфейсных объектов – результат объектной



а *б*

Рис. 12.5. Форма поиска записей (*а*) и сообщение об отсутствии данных (*б*)

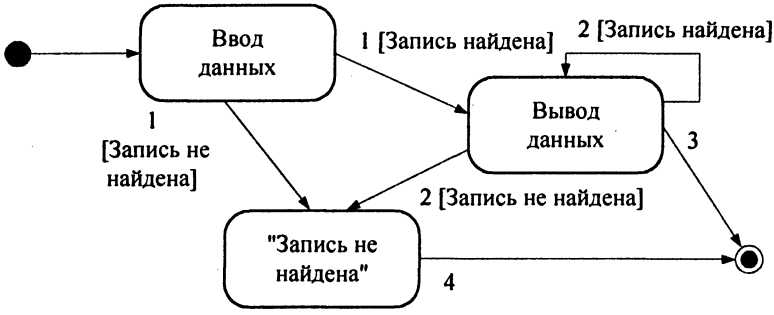


Рис. 12.6. Диаграмма перехода состояний для интерфейса пункта «Поиск записей»:
1 – выбор пункта «Поиск»; 2 – выбор пункта «Следующий»;
3 – выбор пункта «Выход»; 4 – подтверждение получения сообщения об отсутствии данных

декомпозиции интерфейсной части предметной области программы – приведена на рис. 12.7.

Описание объектов начнем с того, что расположим окна в порядке возрастания сложности. При этом просматривается определенная закономерность.

Анализ окон показывает, что их четыре вида:

а) *окно ввода* информации (рис. 12.8, *а*) – пассивное окно, которое должно закрываться при завершении ввода данных (по нажатию Enter);

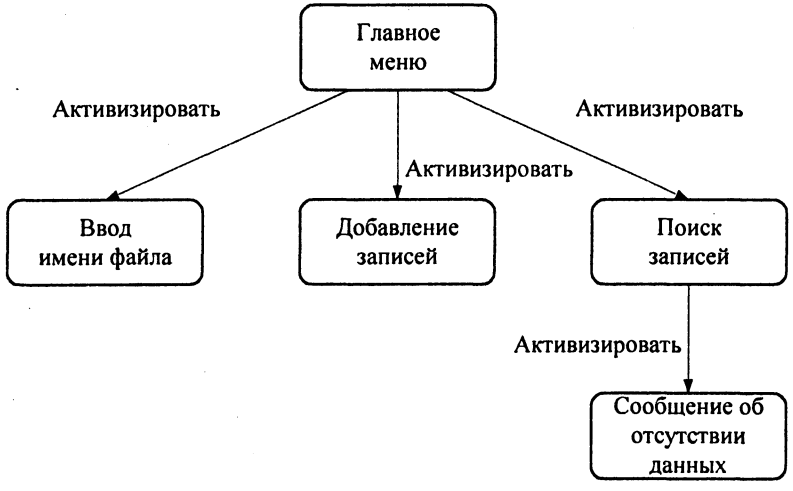


Рис. 12.7. Диаграмма объектов – результат декомпозиции интерфейсной части предметной области программы

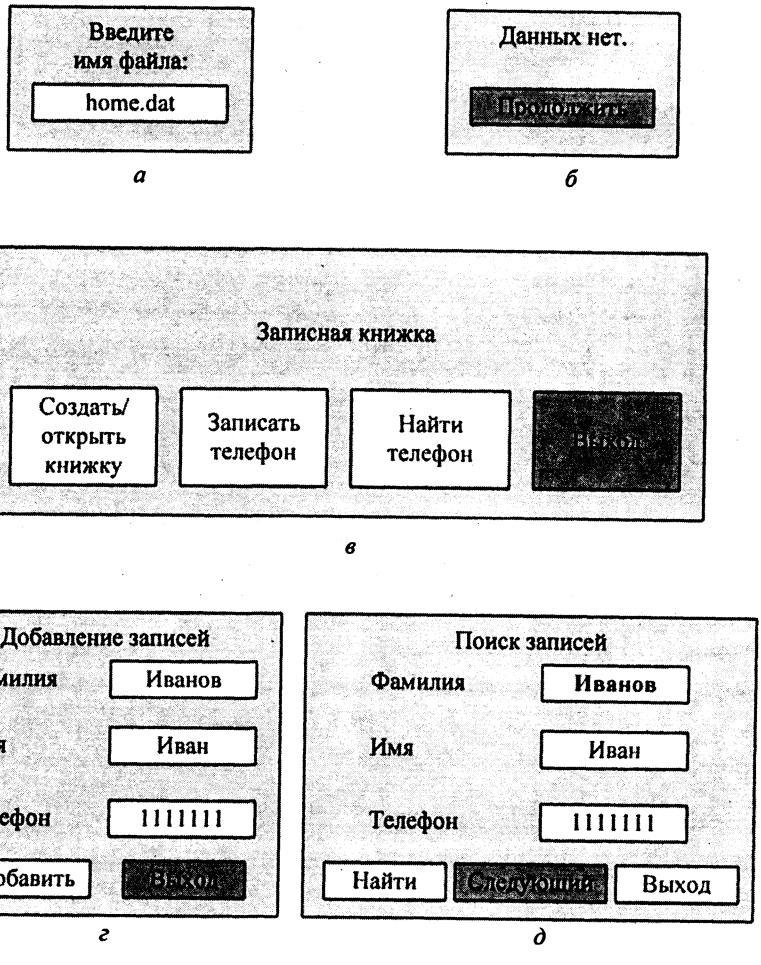


Рис. 12.8. Определение структуры окон программы

б) *окно сообщения* (рис. 12.8, *б*) – активное окно, которое ожидает нажатия любой клавиши – подтверждения получения сообщения;

в) *меню* (рис. 12.8, *в*) – активное окно, которое циклически обрабатывает выбор некоторого пункта пользователем и закрывается при выборе пункта «Выход»;

г) *форма*, включающая окна ввода и меню (рис. 12.8, *г*, *д*) – активное окно, которое помимо циклической обработки выбора пункта обеспечивает возможность осуществления ввода-вывода информации.

Отсюда следует, что достаточно иметь четыре настраиваемых компонента – по одному для реализации объектов каждого вида (рис. 12.9).

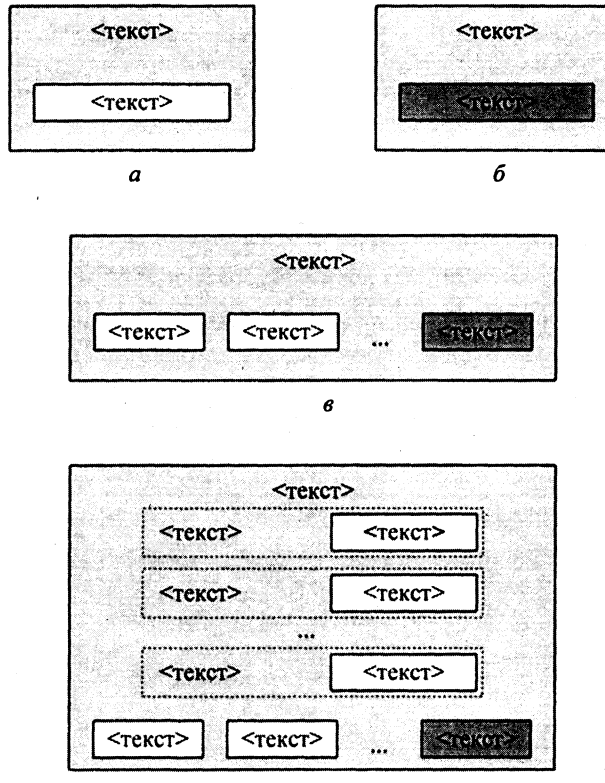


Рис. 12.9. Обобщенное представление компонентов интерфейса (пунктиром выделены компоненты ввода на форме)

Уточнив характеристики интерфейсных компонентов, переходим к их проектированию.

12.2. Проектирование классов

Нетрудно видеть, что любое окно на рис. 12.9 включает строку текста. Следовательно, классы для соответствующих объектов можно наследовать от класса, реализующего окно со строкой текста.

Анализ обобщенных представлений также показывает, что каждая форма включает несколько окон с текстом в качестве пунктов меню, окон ввода и т.д.; соответственно помимо наследования при построении классов будем использовать композицию.

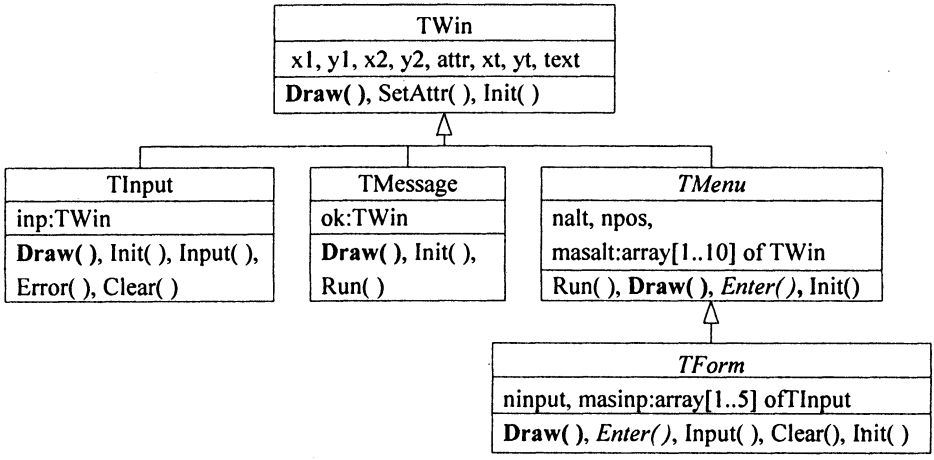


Рис. 12.10. Иерархия классов для реализации простейших интерфейсных элементов

Для реализации каждого вида окон будем строить свой класс.

Вспомогательный базовый класс TWin, реализующий окно с текстом, должен содержать поля для хранения координат окна, цвета фона, цвета символов, координат начала текста и самой строки текста. Цвет фона и цвет символа будем хранить в одном поле attr в виде атрибута – см. параграф 8.1. Для вывода окна класс должен включать метод Draw, для изменения цвета окна – метод SetAttr. Инициализацию полей будет выполнять метод Init (рис. 12.10).

Класс для реализации окна ввода TInput наследуем от TWin и включим в него объектное поле inp типа TWin, которое будет содержать введенное имя файла в качестве текста (на рис. 12.10 композиция не показана, чтобы не затенять общую картину). Класс будет определять свои собственные методы инициализации Init и рисования Draw, так как он должен рисовать два окна с текстом и соответственно хранить информацию о них. Кроме этого класс должен включать метод ввода Input, метод проверки правильности ввода Error, метод очистки поля ввода Clear, которые будут обеспечивать ввод информации.

Класс для реализации сообщения TMessage также наследуем от TWin. Объектное поле ok введено для подтверждения получения сообщения. Класс будет включать методы вывода окна Draw, инициализации полей Init и метод Run, обеспечивающий выход по нажатию любой клавиши.

Класс для реализации меню TMenu будем наследовать от TWin, так как если его наследовать от TMessage, то объекты будут содержать лишнее поле ok. Этот класс будет включать несколько, например десять, объектных полей типа TWin, собранных в массив masalt. Каждое из полей – пункт меню. Количество реально существующих пунктов будем хранить в специальном поле nalt. В процессе работы с меню необходимо также знать номер выделен-

ного пункта меню npos. Метод Run будет обеспечивать работу с меню. Для вывода меню на экран он будет вызывать метод Draw, также переопределенный в данном классе. Каждый раз при выборе пункта меню метод Run будет вызывать метод Enter, который обеспечит выполнение требуемых действий. Для инициализации полей класса TMenu переопределим метод Init.

Класс для реализации форм ввода и поиска TForm наследуем от TMenu, чтобы повторно не описывать работу с меню. Для ввода-вывода информации добавим окна ввода, объединенные в массив masinp, количество используемых окон ввода будем хранить в поле ninput. Метод Run данный класс будет наследовать, а вызываемые из него методы Draw и Enter – переопределять. Следовательно, и в классе TMenu, и в классе TForm методы Draw и Enter должны объявляться виртуальными полиморфными (1-й случай обязательного использования позднего связывания). Кроме реализации меню класс TForm должен обеспечивать выполнение операций ввода-вывода. Следовательно, он будет включать методы ввода Input и очистки полей Clear.

При проектировании классов те методы, реализация которых будет зависеть от конкретной задачи, программируются не содержащими действий (абстрактными). Предполагается, что они будут переопределяться при наследовании класса от библиотечного.

12.3. Реализация универсальных интерфейсных компонентов

Классы будем описывать в отдельных модулях. Это позволит включать в программу описания только тех классов, которые в ней будут использоваться.

Класс TWin описываем в модуле Win: в интерфейсной части – само описание класса, а в части реализации – методы класса.

```
Unit Win;
Interface
  Uses crt;
  Type str80=string[80];
    TWin=Object
      x1, y1, x2, y2:integer;  {координаты окна}
      attr:integer;            {атрибут}
      xt, yt:integer;          {начало текста}
      text:str80;              {строка текста}
      procedure Init(ax1,ay1,ax2,ay2,aattr,axt,ayt:integer;atext:str80);
      procedure SetAttr(aattr:integer); {изменение цвета и фона окна}
      procedure Draw;          {вывод окна на экран}
    End;
```

```
Implementation
 Procedure TWin.Init;   {инициализация полей}
  Begin
        x1:=ax1;
        y1:=ay1;
        x2:=ax2;
        y2:=ay2;
        attr:=aattr;
        xt:=axt;
        yt:=ayt;
        text:=atext;
  End;
 Procedure TWin.Draw;   {вывод окна на экран}
  Begin
      TextBackGround(attr div 16);
      TextColor(attr mod 16);
      Window(x1,y1,x2,y2);
      Clrscr;  {выделение окна}
      Gotoxy(xt,yt);
      Write(text);       {вывод строки текста}
  end;
 Procedure TWin.Setattr; {изменение цвета фона и текста}
  Begin
     attr:=aattr;    {изменение атрибута}
     Draw;           {вывод окна с другим атрибутом}
  end;
End.
```

Класс TInput описываем в модуле Input. Метод Error объявляем виртуальным, так как он вызывается из другого метода класса и будет переопределяться в классах потомках, в то время как метод, из которого он вызывается, скорее всего переопределяться не будет (1-й случай обязательного использования позднего связывания).

```
Unit Input;
Interface
 Uses crt,Win;
 Type TInput=Object(TWin)
        inp:TWin;   {окно ввода}
      Constructor Init(ax1,ay1,ax2,ay2,aattr,axt,ayt:integer;atext:str80;
                       bx1,by1,bx2,by2,battr,bxt,byt:integer;btext:str80);
      procedure Draw;  {вывод окна}
      procedure Clear;  {очистка поля ввода}
```

```
        procedure Input;                    {ввод строки из окна}
        function  Error:boolean; virtual; {проверка введенных данных}
    end;
Implementation
 Constructor TInput.Init;
  Begin
      inherited Init(ax1,ay1,ax2,ay2,aattr,axt,ayt,atext);
      Inp.Init(bx1,by1,bx2,by2,battr,bxt,byt,'');
  end;
 Procedure TInput.Draw;
  Begin
      inherited Draw;
      inp.Draw;
  End;
 Procedure TInput.Clear;
  Begin
       inp.text:='';
       inp.Draw;
  End;
 Procedure TInput.Input;
  Begin
       Window(inp.x1,inp.y1,inp.x2,inp.y2);
       TextBackGround(inp.attr div 16);
       TextColor(inp.attr mod 16);
       repeat
            Gotoxy(inp.xt,inp.yt);
            Clear;
            ReadLn(inp.text);
            Gotoxy(inp.xt,inp.yt);
            Write(inp.text);
       until not Error;
  end;
 Function TInput.Error;   {проверка не выполняется}
  begin
      Error:=false;
  end;
End.
```

Класс TMessage описываем в модуле Message.

```
Unit Message;
Interface
 Uses crt,Win;
```

```
Type TMessage=Object(TWin)
      ok:TWin;              {окно подтверждения}
      procedure Init(ax1,ay1,ax2,ay2,aattr,axt,ayt:integer;atext:str80;
                     bx1,by1,bx2,by2,battr,bxt,byt:integer;btext:str80);
      procedure Run;     {ожидание подтверждения}
      procedure Draw;  {вывести окно}
   end;
Implementation
 Procedure TMessage.Init;
  Begin
      inherited Init(ax1,ay1,ax2,ay2,aattr,axt,ayt,atext);
      ok.Init(bx1,by1,bx2,by2,battr,bxt,byt,btext);
  end;
 Procedure TMessage.Draw;
  Begin
      inherited Draw;    {выводим родительское окно}
      ok.Draw;   {выводим окно запроса на продолжение работы}
  End;
 Procedure TMessage.Run;
  Begin
    Draw;       {выводим окно}
    ReadKey;  {ожидаем подтверждения}
  End;
End.
```

Класс TMenu описываем в модуле Menu. Инициализацию объектных полей-массивов целесообразно выполнять типизированными константами. Несовпадения размерности массивов, если используются не все пункты меню, можно избежать, описав параметры как открытые массивы, не забыв, что индекс элементов открытого массива начинается с нуля.

```
Unit Menu;
Interface
 Uses crt,Win;
 Type TMenu=Object(TWin)
      nalt:integer;                    {количество альтернатив в меню}
      masalt:array[1..10] of TWin; {массив альтернатив меню}
      npos:integer;                    {номер выбранной альтернативы}
      constructor Init(ax1,ay1,ax2,ay2,aattr,
                  axt,ayt:integer;atext:str80;n:integer;
                  const w:array of TWin);  {открытый массив TWin}
      procedure Run;              {реализация работы с меню}
      procedure Draw; virtual;  {вывести окно}
```

```
      procedure Enter; virtual;  {при нажатии на Enter ...}
   end;
Implementation
 Constructor TMenu.Init;
  Var  i:integer;
  Begin
      inherited Init(ax1,ay1,ax2,ay2,aattr,axt,ayt,atext);
      nalt:=n; {количество реально используемых пунктов меню}
      for i:=1 to nalt do
            masalt[i].Init(w[i-1].x1, w[i-1].y1, w[i-1].x2, w[i-1].y2,
                           w[i-1].attr, w[i-1].xt, w[i-1].yt, w[i-1].text);
  End;
 Procedure TMenu.Draw;
  Var i:integer;
  Begin { очищаем экран}
        TextBackGround(0);
        TextColor(1);
        Window(1,1,80,25);     Clrscr;
         inherited Draw;                    {выводим основное окно}
        for i:=1 to nalt do masalt[i].Draw;  {выводим окна пунктов}
  End;
 Procedure TMenu.Run;
  Var ch1,ch2:char;
      temp:integer;
  Begin
      Draw;
      npos:=nalt;
      masalt[npos].Setattr(71);
      repeat
          ch1:=Readkey; {читаем код клавиши}
          if ch1=#0 then ch2:=Readkey;
          case ch1 of
          #0: case ch2 of
              #75:begin       {перемещение курсора влево}
                    temp:=npos-1;
                    if temp=0 then temp:=nalt;    {закольцовываем}
                    masalt[npos].Setattr(113);    {убираем выделение}
                    masalt[temp].Setattr(71);     {выделяем пункт}
                    npos:=temp;
                  end;
             #77:begin       {перемещение курсора вправо}
                    temp:=npos+1;
                    if temp=nalt+1 then temp:=1; {закольцовываем}
```

```
                masalt[npos].Setattr(113);  {убираем выделение}
                masalt[temp].Setattr(71);   {выделяем пункт}
                npos:=temp;
              end;
          end;
        #13: begin
               masalt[npos].Setattr(113); {убираем выделение}
                Enter;    {при нажатии Enter – выполняем пункт}
                Draw;     {выводим главное меню}
                masalt[npos].Setattr(71);  {выделяем пункт меню}
             end;
        end
    until((npos=nalt)and(ch1=#13))or(ch1=#27);{до завершения работы}
  End;
 Procedure TMenu.Enter;  {абстрактный метод выполнения пунктов}
  Begin  end;
End.
```

Класс TForm описываем в модуле Form. Инициализацию объектных полей – окон ввода также будем выполнять с использованием типизированных констант.

```
Unit Form;
Interface
 Uses crt,Win,Input,Menu;
 Type TForm=Object(TMenu)
       ninput:integer; {количество полей ввода}
       masinp:array[1..5] of TInput; {массив полей ввода}
       constructor Init(ax1,ay1,ax2,ay2,aattr,axt,ayt:integer;atext:str80;
                        n:integer;   const w1:array of TWin;k:integer;
                        const w2:array of TInput);
       procedure Draw; virtual; {вывод окна}
       procedure Clear; {очистка окон ввода}
       procedure Input; {ввод информации из окон ввода}
    end;
Implementation
 Constructor TForm.Init;
  Var  i:integer;
  Begin
       inherited Init(ax1,ay1,ax2,ay2,aattr,axt,ayt,atext,n,w1);
       ninput:=k;    {количество задействованных окон ввода}
```

```
        for i:=1 to ninput do
        masinp[i].Init(w2[i-1].x1, w2[i-1].y1,
                       w2[i-1].x2, w2[i-1].y2, w2[i-1].attr,
                       w2[i-1].xt, w2[i-1].yt,w2[i-1].text,
                       w2[i-1].inp.x1, w2[i-1].inp.y1,
                       w2[i-1].inp.x2, w2[i-1].inp.y2,
                       w2[i-1].inp.attr, w2[i-1].inp.xt, w2[i-1].inp.yt,
                       w2[i-1].inp.text);
  End;
 Procedure TForm.Draw;
  Var i:integer;
  Begin
        inherited Draw;                          {выводим основное окно}
        for i:=1 to nalt do masalt[i].Draw;      {выводим окна пунктов}
        for i:=1 to ninput do masinp[i].Draw; {выводим окна ввода}
  End;
 Procedure TForm.Clear;
  Var i:integer;
  Begin
     for i:=1 to ninput do masinp[i].Clear;  {чистим окна ввода}
  End;
 Procedure TForm.Input;
  Var i:integer;
  Begin
       Clear;
       for i:=1 to ninput do masinp[i].Input;     {вводим данные}
  End;
End.
```

12.4. Создание программы с использованием библиотеки интерфейсных компонентов

В параграфе 12.1 для программы «Записная книжка» уже была выполнена декомпозиция интерфейсной части предметной области (рис. 12.7). Осталось учесть взаимодействие программы с файлом: открытие, запись информации и ее поиск. Будем считать, что все эти действия выполняет объект Файл. Этот объект должен обрабатывать сообщения: Открыть/создать, Добавить и Найти. На рис. 12.11 показан результат объектной декомпозиции всей предметной области программы.

Проектирование и реализация объекта Файл. Для реализации объекта Файл разработаем класс TBase (рис. 12.12).

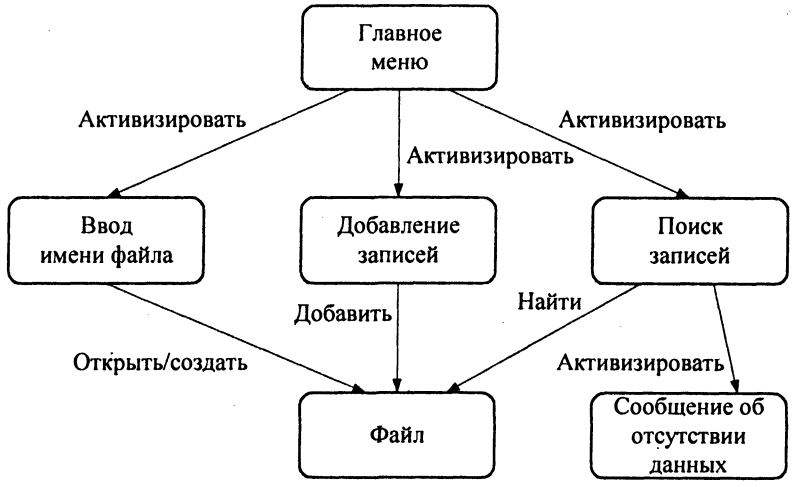


Рис. 12.11. Диаграмма объектов предметной области программы

Этот класс должен хранить файловую переменную f, через которую будет осуществляться доступ к файлу.

Для поиска данных класс будет включать поля исходных данных поиска: p_family – фамилия и p_name – имя, ключи k1, k2, которые устанавливаются в зависимости от того, заданы или нет фамилия или имя и поля результатов поиска family, name, telefon. Кроме того, класс должен включать метод открытия существующего или создания нового файла Open, метод добавления записей Add, метод поиска первой подходящей записи Find и метод поиска следующих подходящих записей FindNext.

Наибольший интерес представляют функции поиска информации, которые в соответствии с заданием должны обеспечивать поиск по неполным данным. При этом поля Фамилия и Имя могут быть заданы, а могут быть пропущены, могут совпадать, а могут не совпадать. Всего возможно $2^4 = 16$ вариантов. В пяти случаях результат поиска положителен (табл. 12.1), т.е. записи считаются удовлетворяющими заданным условиям. Эти случаи можно объединить в одно логическое выражение:

$$ff = (k1 \wedge k2 \wedge k3 \wedge k4) \vee (k1 \wedge \overline{k2} \wedge k3) \vee (\overline{k1} \wedge k2 \wedge k4),$$

TBase
f, family, name, telefon, p_family, p_name, k1, k2
Init( ), Open( ), Add( ), Find( ), FindNext( ), Closef( )

Рис. 12.12. Класс TBase

где $\wedge$ – соответствует логическому «и», а $\vee$ – логическому «или».

Именно это выражение и заложено в метод поиска для определения «подходящих» записей.

Таблица 12.1

Фамилия указана k1	Имя указано k2	Фамилия совпадает k3	Имя совпадает k4	Результат поиска ff
Да	**Да**	**Да**	**Да**	**Да**
Да	Да	Да	Нет	Нет
Да	Да	Нет	Да	Нет
Да	Да	Нет	Нет	Нет
Да	**Нет**	**Да**	Да	**Да**
Да	**Нет**	**Да**	Нет	**Да**
Да	Нет	Нет	Да	Нет
Да	Нет	Нет	Нет	Нет
Нет	**Да**	Да	**Да**	**Да**
Нет	Да	Да	Нет	Нет
Нет	**Да**	Нет	**Да**	**Да**
Нет	Да	Нет	Нет	Нет
Нет	Нет	Да	Да	Нет
Нет	Нет	Да	Нет	Нет
Нет	Нет	Нет	Да	Нет
Нет	Нет	Нет	Нет	Нет

Ниже приведен текст модуля Base, содержащего описание данного класса.

```
Unit Base;
Interface
  Type str30=string[30];
     rec=record
            rfamily, rname, rtelefon:str30;
         end;
  Type TBase=Object
       f:file of rec;
       family, name, telefon:str30;  {результаты поиска}
       p_family,p_name:str30;        {данные поиска}
       k1, k2:boolean;               {ключи поиска}
       procedure Open(fname:str30);        {открытие/создание файла}
       procedure Add(afamily,aname,atelefon:str30);{добавление записей}
       function Find(afamily,aname:str30):boolean; {поиск первого}
```

```
      function FindNext:boolean;          {поиск следующего}
       procedure Closef;                  {закрытие файла}
    End;
Implementation
 Procedure TBase.Open;
  Begin
     Assign(f,fname);  {инициализация файловой переменной}
     {$I-}
       Reset(f);
     {$I+}  {открытие с проверкой существования}
     if IOResult<>0 then ReWrite(f);  {создание файла}
  End;
Procedure TBase.Add;
 Var r:rec;
  Begin
       Seek(f,FileSize(f));{устанавливаем файловый указатель на
                                                    конец файла}
       r.rfamily:=afamily;  {создаем запись}
       r.rname:=aname;
       r.rtelefon:=atelefon;
       Write(f,r);              {выводим запись в файл}
  End;
Function TBase.Find;
  Begin
       Close(f);                {закрываем файл}
       ReSet(f);                {открываем файл для чтения}
       p_family:=afamily;   {сохраняем данные поиска}
       p_name:=aname;
       k1:=p_family<>'';    {устанавливаем два ключа поиска}
       k2:=p_name<>'';
       Find:=FindNext;      {ищем запись по ключам}
  End;
Function TBase.FindNext;
 Var r:rec;
     k3, k4, ff:boolean;  {ключи поиска и его результат}
 Begin
   ff:=false;  {ключ поиска «запись не найдена»}
   while not Eof(f) and not ff do
     begin
       Read(f,r);
       k3:=p_family=r.rfamily;  {строим еще два ключа поиска}
       k4:=p_name=r.rname;
```

```
      if (k1 and k2 and k3 and k4)   {выбираем записи}
         or (not k1 and k2 and k4) or (k1 and not k2 and k3) then
           begin
             ff:=true;              {ключ поиска «запись найдена»}
             family:=r.rfamily;  {копируем результаты поиска}
             name:=r.rname;
             telefon:=r.rtelefon;
           end
    end;
    FindNext:=ff;    {возвращаем ключ поиска}
   end;
Procedure TBase.Closef;
 begin
  Close(f);   {закрываем файл}
 end;
End.
```

Проектирование интерфейса с использованием библиотеки интерфейсных компонентов. Библиотека интерфейсных компонентов содержит классы, реализующие обобщенные объекты, вид и поведение которых, связанные с решением конкретной задачи, не определены. Для того чтобы уточнить вид компонентов интерфейса, обычно достаточно их соответствующим образом инициализировать. Конкретизацию поведения компонентов в реальной задаче выполняют за счет наследования классов реальной задачи от библиотечных.

На рис. 12.13 представлена полная диаграмма классов программы, на которой показано наследование интерфейсных классов от описанных в библиотеке и уточнены их ассоциации с классом TBase.

Класс TIName будет использован для реализации объекта Ввод имени файла. Он переопределяет метод Error класса TInput, задавая свой вариант проверки правильности введенной информации.

Класс TMain строится для реализации объекта Главное меню. Он переопределяет метод Enter, конкретизируя поведение программы при выборе пунктов главного меню. Количество пунктов меню регулируется соответствующей инициализацией поля nalt. Добавляемое поле may логического типа будет использовано для контроля открытия файла.

Классы TAdd и TFind будут использованы для реализации форм Добавление и Поиск. Очевидно, что они должны переопределять абстрактный метод Enter прародителя – класса TMenu. Кроме этого в классе TFind необходимо переопределить метод ввода, который должен обеспечить ввод информации не из всех полей, а только из поля фамилии и поля имени, и добавить метод Show для вывода найденных данных (телефона и, при поиске по неполным данным, имени или фамилии) в окно поиска.

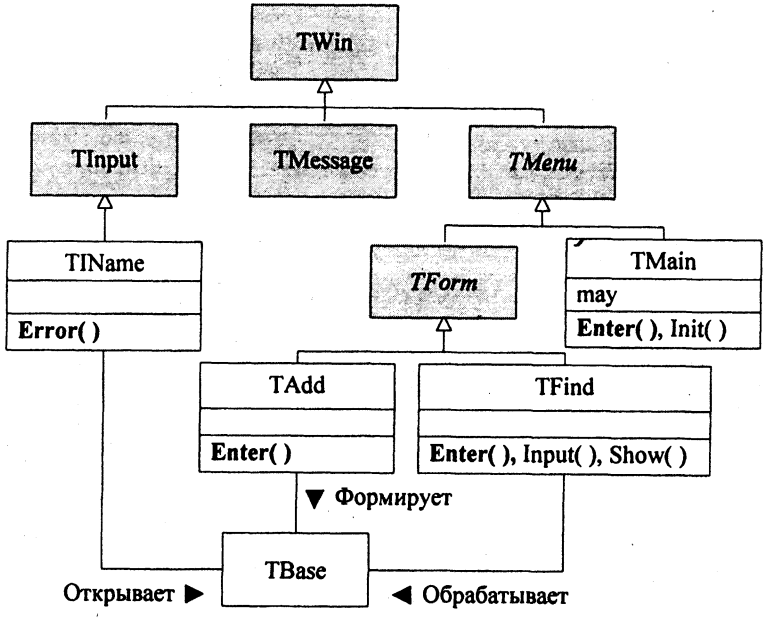


Рис. 12.13. Диаграмма классов программы «Записная книжка»

Для реализации объекта Сообщение об отсутствии данных нового класса создавать не нужно, используем непосредственно класс TMessage.

После объявления наследуемых классов перед описанием переопределенных методов необходимо объявить объекты-переменные, так как методы будут обращаться к этим переменным для программирования требуемых действий.

Ниже представлен полный текст программы.

```
Program Memory;
Uses  crt,Win,Input,Message,Menu,Form,Base;
 {объявление классов – потомков библиотечных классов}
Type TMain=Object(TMenu)  {главное меню}
       may:boolean;          {признак открытия файла}
       Procedure Enter; virtual;
     end;
Type TIName=Object(TInput)     {ввод имени файла}
       Function Error:boolean; virtual;  {проверка имени файла}
     end;
Type TAdd=Object(TForm)      {форма для добавления записей}
       Procedure Enter; virtual;  {завершение ввода одной записи}
     end;
```

```
Type TFind=Object(TForm)      {форма для поиска телефонов}
       Procedure Input; virtual;  {ввод данных поиска}
       Procedure Enter; virtual;  {поиск одной записи}
       Procedure Show;            {вывод результата поиска в окна}
     end;
 {объявление объектов-переменных}
Var M:TMain;                {объект Главное меню}
    N:TIName;               {объект Ввод имени файла}
    A:TAdd;                 {объект Добавление записей}
    F:TFind;                {объект Поиск записей}
    ND:TMessage;            {объект Сообщение об отсутствии данных}
    B:TBase;                {объект Файл}
 {описание дополнительных методов}
Procedure TMain.Enter; {обработка выбора пунктов главного меню}
Begin
 case npos of
  1:begin
      N.Draw;    {выводим окно ввода}
      N.Input;   {вводим имя файла, проверяя его допустимость}
      B.Open(N.inp.text); {если файл существует, то открываем,
                                                иначе – создаем}
      may:=true; {устанавливаем признак открытия файла }
    end;
  2:if may then   {если определен файл данных }
      A.Run;      {осуществляем добавление записей}
  3:if may then   {если определен файл данных }
      F.Run;      {осуществляем поиск записей}
 end;
End;
Function TIName.Error;  {проверка имени файла}
 Var l:integer;
 Begin l:=Pos('.', inp.Text);
    if l=0 then  l:=length(inp.Text);
    if (l>0) and (l<=8) then Error:=false
                        else  Error:=true;
 End;
Procedure TAdd.Enter; {обработка пунктов меню добавления}
 Begin
   case npos of
    1:begin
        Input; {вводим фамилию, имя и телефон}
        B.Add(masinp[1].inp.text, masinp[2].inp.text,
              masinp[3].inp.text);   {записываем в файл}
```

```
        end;
    end; {case}
End;
Procedure TFind.Enter; {обработка пунктов меню поиска}
Begin case npos of
     1:begin
         Input; {вводим фамилию и имя }
         if B.Find(masinp[1].inp.text,masinp[2].inp.text) then Show
         else ND.Run; {выводим сообщение об отсутствии данных }
       end;
     2:begin
         if B.FindNext then Show
         else ND.Run; {выводим сообщение об отсутствии данных }
       end;
   end;
End;
Procedure TFind.Input; {ввод данных для поиска информации}
Begin
        Clear;              {очищаем поля ввода}
        masinp[1].Input; {вводим фамилию}
        masinp[2].Input; {вводим имя}
End;
Procedure TFind.Show; {вывод найденной информации в окно}
Begin
   Clear;
   masinp[1].inp.text:=B.family; masinp[1].Draw; {выводим фамилию}
   masinp[2].inp.text:=B.name;   masinp[2].Draw;{выводим имя}
   masinp[3].inp.text:=B.telefon; masinp[3].Draw; {выводим телефон}
End;
{описание констант для инициализации полей-массивов}
Const menu1:array[1..4] of TWin=
        ((x1:10;y1:14;x2:23;y2:18;attr:113;xt:3;yt:2;
                    text:'Создать / открыть книжку'),
         (x1:26;y1:14;x2:39;y2:18;attr:113;xt:4;yt:2;
                    text:'Записать телефон'),
        (x1:42;y1:14;x2:55;y2:18;attr:113;xt:5;yt:2;
                    text:'Найти телефон'),
        (x1:58;y1:14;x2:71;y2:18;attr:113;xt:4;yt:2;
                    text:'Завершить работу'));
   menu2:array[1..2] of TWin=
        ((x1:28;y1:18;x2:38;y2:21;attr:113;xt:2;yt:2;text:'Добавить'),
         (x1:42;y1:18;x2:52;y2:21;attr:113;xt:2;yt:2;text:'Выход'));
```

```
    menu3:array[1..3] of TWin=
          ((x1:23;y1:18;x2:33;y2:21;attr:113;xt:2;yt:2;text:'Найти'),
          (x1:35;y1:18;x2:45;y2:21;attr:113;xt:2;yt:2;text:'Следующий'),
           (x1:47;y1:18;x2:57;y2:21;attr:113;xt:2;yt:2;text:'Выход'));
    inpp:array[1..3] of TInput=
          ((x1:22;y1:8;x2:32;y2:8;attr:94;xt:1;yt:1;text:'Фамилия';
                Inp:(x1:34;y1:8;x2:54;y2:8;attr:112;xt:1;yt:1;text:'')),
           (x1:22;y1:10;x2:32;y2:10;attr:94;xt:1;yt:1;text:'Имя';
                Inp:(x1:34;y1:10;x2:54;y2:10;attr:112;xt:1;yt:1;text:'')),
           (x1:22;y1:12;x2:32;y2:12;attr:94;xt:1;yt:1;text:'Телефон';
                Inp:(x1:34;y1:12;x2:54;y2:12;attr:112;xt:1;yt:1;text:'')));
{основная программа}
Begin
  {инициализируем объекты}
  M.Init(5,5,76,20,30,5,3,'Записная книжка',4,menu1);
  A.Init(20,2,60,22,94,5,3,'Добавление записей',2,menu2,3,inpp);
  N.Init(30,8,50,19,94,3,3,'Введите имя файла:',
       35,12,45,12,112,1,1,' ');
  F.Init(20,2,60,22,94,5,3,'Поиск записей',3,menu3,3,inpp);
  ND.Init(30,6,50,14,30,6,2,'Нет данных',
      34,11,46,12,71,2,1,'Продолжить');
  {начинаем работу}
  M.may:=false;  {устанавливаем признак «файл не открыт»}
  M.Run;         {передаем управление Главному меню}
  if may then B.Closef;    {закрываем файл}
  {очищаем экран}
  TextBackGround(0);
  TextColor(1);
  Window(1,1,80,25);
  Clrscr;
End.
```

К основным достоинствам объектно-ориентированного подхода следует отнести:

- уменьшение количества параметров подпрограмм;
- увеличение объема повторно используемых кодов;
- возможность унификации программных элементов (в том числе интерфейсных);
- относительную простоту распределения разработки сложных программных продуктов между несколькими программистами.

Рассмотрим эти достоинства подробнее.

Уменьшение количества параметров подпрограмм связано с тем, что при вызове методов мы не должны указывать поля объекта. В свою очередь, чем

меньше параметров, тем меньше вероятность ошибки при вызове подпрограммы, а ведь именно такие ошибки обычно выявляются при сборке программ из модулей, т.е. на последнем самом сложном этапе отладки программы.

Увеличение объема повторно используемых кодов вызвано появлением возможности использования уже существующих классов, приспособив их к своим целям без изменения текстов уже написанных программ. Методы, которые нас не устраивают, мы просто заменяем при наследовании, сохраняя возможность вызвать их из заменяющего метода и даже из программы.

Возможность унификации программных элементов также является следствием появления средств построения новых классов на базе существующих. Работать с такими элементами удобно, так как деталей реализации их, как правило, можно не знать.

Поскольку независимость объектов существенно выше, чем подзадач, уменьшается количество вопросов, которые должны быть учтены при ведении разработки несколькими программистами.

Однако у объектно-ориентированного подхода есть и недостатки. И к ним в первую очередь относят некоторое увеличение размера программы за счет большого числа небольших подпрограмм, которые вряд ли появились бы при использовании структурного подхода. Соответственно объектно-ориентированные программы медленнее работают, так как каждый вызов подпрограммы требует времени.

Существует и еще один недостаток – сложность обработки нештатных ситуаций, таких как деление на нуль и т. п. Дело в том, что ситуации такого рода обнаруживаются обычно не в том месте и часто даже не во время работы того объекта, где определяются некорректные данные и соответственно возможна их корректировка. Этот недостаток исправлен в более развитых объектных моделях, например в Delphi Pascal, за счет реализации механизма исключений.

Задания для самопроверки

Задание 1. Разработайте программу, которая организует хранение в файле информации о печатных изданиях. Для каждого издания определены: наименование, периодичность, тираж, типография. Программа должна в диалоговом режиме корректировать информацию в файле и давать ответы на каждый из перечисленных вопросов:

1) вывести наименования всех ежедневных изданий, печатаемых указанной типографией;

2) определить наименование издания данной периодичности с наибольшим тиражом;

3) определить наименования всех изданий, печатаемых типографией, на которую приходится максимальный суммарный тираж.

При проектировании интерфейса используйте разработанные в настоящей главе классы интерфейсных элементов.

Задание 2. Разработайте программу, которая организует хранение в файле информации о товарах на складе. По каждому товару необходимо хранить: наименование, дату изготовления, сведения об изготовителе, дату поступления, количество единиц хранения. Программа должна в диалоговом режиме корректировать информацию в файле при его отгрузке или поступлении и давать ответы на каждый из перечисленных вопросов:

1) вывести список товаров на складе на текущий момент времени;

2) выполнить поиск товаров по наименованию;

3) вывести список товаров, отсортированный по дате поступления.

При проектировании интерфейса используйте разработанные в настоящей главе классы интерфейсных элементов.

Задание 3. Предложите собственную библиотеку интерфейсных элементов для реализации интерфейса программы заданий 1и 2.

Приложение

П1. Основные стандартные процедуры и функции

Объявление	Описание
abs(x:integer):integer ***abs(x:real):real***	Возвращает модуль (абсолютное значение) аргумента
arctan(x:real):real	Возвращает арктангенс аргумента
chr(i:integer):char	Возвращает i-й символ таблицы символов
cos(x:real):real	Возвращает косинус аргумента
dec(var i [; di:longint])	Уменьшает значение i на di единиц
exp(x:real):real	Возвращает значение e^x
frac(x:real):real	Возвращает дробную часть аргумента
inc(var i [; di:longint])	Увеличивает значение i на di единиц
int(x:real):real	Возвращает целую часть аргумента
hi(w:word):byte	Возвращает старший байт слова аргумента
lo(w:word):byte	Возвращает младший байт слова аргумента
ln(x:real):real	Возвращает ln x (натуральный)
ord(x):longint	Возвращает номер значения порядкового типа
pi:real	Возвращает значение p
pred(x):<тип x>	Возвращает значение, предшествующее аргументу
random:real	Возвращает случайное число $0 \le r < 1$
random(i:integer):integer	Возвращает случайное число $0 \le n < i$
randomise	Инициализирует датчик случайных чисел
round(x:real):integer	Округляет аргумент до ближайшего целого числа
sin(x:real):real	Возвращает sin x
sqr(x:real):real ***sqr(i:integer):integer***	Возвращают x^2
sqrt(x:real):real	Возвращает корень квадратный из аргумента
succ(x): <mun x>	Возвращает значение, следующее за аргументом
trunc(x:real):integer	Возвращает целое, полученное отбрасыванием дробной части

П2. Русская кодовая таблица для MS DOS (страница 866)

x\y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫
16	►	◄	↕	!!	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
32	пробел	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
128	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
144	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
160	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
176	░	▒	▓	|	┤	╡	╢	╖	╕	╣	‖	╗	╝	╜	╛	┐
192	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
208	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	▉	▄	▌	▐	▀
224	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
240	Ё	ё	Є	є	Ъ	ї	Ы	ў	°	•	·	√	№	¤	■	

Примечания:

1. Код символа = x + y .
2. Пробел имеет код 32. Символы с кодами 0 и 255 называют пустыми.
3. При выводе символа с кодом 8 предусмотрен звуковой сигнал.
4. При выводе символа с кодом 10 происходит переход на следующую строку.
5. При выводе символа с кодом 13 происходит установка курсора на начало строки.

П3. Расширенные scan-коды

Комбинации клавиш	Расширенный scan-код
Cntl + 2	3
Shift +Tab	15
Alt + Q..Alt + P (верхний ряд букв)	16..25
Alt + A..Alt + L (средний ряд букв)	30..38
Alt + Z..Alt + M (нижний ряд)	44..50
F1..F10	59..68
Home	71
↑	72
Page up	73
←	75
→	77
End	79
↓	80
Page down	81
Ins	82
Del	83
Shift + F1..Shift + F10	84..93
Ctrl + F1..Ctrl + F10	94..103
Alt + F1..Alt + F10	104..113
Ctrl + Print screen	114
Ctrl + ←	115
Ctrl + →	116
Ctrl + End	117
Ctrl + Page down	118
Ctrl + Home	119
Alt +1..Alt + = (верхний ряд)	120..131
Ctrl + Page up	132

2. Строковые типы. Добавлены новые строковые типы, изменено значение String. В результате определены следующие типы строк:

- ShortString = String Borland Pascal;
- AnsiString – переменная, содержащая указатель на строку из символов AnsiChar размером до 2147483547 байт, завершающуюся нулем (#0), первый символ расположен в элементе с номером 1;
- WideString – переменная, содержащая указатель на строку из символов WideChar размером до 2147483547 байт, завершающуюся нулем (#0), первый символ расположен в элементе с номером 1;
- PChar – переменная, содержащая указатель на массив array[0..n] of Char, которая завершается нулем (#0), первый символ расположен в элементе с номером 0;
- String соответствует ShortString при {$H-} и AnsiString при {$H+}.

Строки первых трех типов совместимы. При присваивании строк любого типа строке PChar используют явное преобразование типа в тип PChar вида: PChar(<строковая переменная>).

3. Файлы. Изменены имена нескольких стандартных процедур в связи с тем, что эти имена используются как имена методов визуальных компонентов: Assign → AssignFile, Close → CloseFile.

4. Инициализация переменных. В качестве инициализированных переменных при {$J+} используют типизированные константы, как в Borland Pascal, а при {$J-} – инициализированные переменные:

Var <имя>:<тип>=<значение>; ...

5. Функции. Функция может возвращать параметр любого типа, кроме файла. Внутри функции определена специальная переменная Result, тип которой совпадает с типом возвращаемого значения. Этой переменной и должно присваиваться значение результата. Допускается использовать старый вариант, при котором значение результата присваивается переменной, имя которой совпадает с именем функции.

6. Модули. При описании модуля можно указывать новые секции:

Unit <имя>;
  Interface <интерфейсная секция>;
  Implementation <секция реализации>;
  Initialization <секция инициализации>;
  Finalization <секция завершения>
end.

Операции, определенные в секции инициализации, выполняются при подключении модуля, а операции, определенные в секции завершения – при завершении программы.

П4. Основные отличия Delphi Pascal от Borland Pascal 7.0

Языком программирования для среды Delphi является Object Pascal – более развитая версия Borland Pascal 7.0. Все отличия Object Pascal от базовой версии языка разобьем на две группы: отличия средств процедурного программирования и отличие объектной модели.

Средства процедурного программирования Delphi Pascal.

1. Классификация типов данных. В Delphi Pascal изменились диапазоны значений существовавших типов данных и появились новые типы. Обновленная классификация типов выглядит следующим образом.

Целые:

Тип	Диапазон	Формат	Размер, байт
ShortInt	-128...127	знаковый	1
Byte	0...255	беззнаковый	1
SmallInt	-32768...32767	знаковый	2
Word	0...65535	беззнаковый	2
LongInt	-2147483648...2147483647	знаковый	4
Integer	То же	знаковый	4
Cardinal	0...2147483547	беззнаковый	4

Логические:

Тип	Размер, байт	Примечание
Boolean	1	ord(false)=0, ord(true)=1
ByteBool	1	ord(false)=0, ord(true)≠0
WordBool	2	То же
LongBool	4	»

Символьные:

Тип	Размер, байт	Кодировка
AnsiChar = Char	1	Кодировка ANSI
WideChar	2	Кодировка Unicode

Вещественные:

Тип	Диапазон	Значащих цифр	Размер, байт	Примечание
Real	$\pm 2{,}9\cdot 10^{-39}...1{,}7\cdot 10^{38}$	11...12	6	Не эффективен
Single	$\pm 1{,}5\cdot 10^{-45}...3{,}4\cdot 10^{38}$	7...8	4	
Double	$\pm 5\cdot 10^{-324}...1{,}7\cdot 10^{308}$	15...16	8	
Extended	$\pm 3{,}4\cdot 10^{-4932}...1{,}1\cdot 10^{4932}$	19...20	10	
Comp	$-2^{63}...2^{63}-1$	19...20	8	
Currency	±922337203685477.5808	19...20	8	Денежный

Объектная модель Delphi Pascal. Основное отличие объектной модели заключается в том, что она предполагает использование только динамических объектов-переменных. При конструировании таких объектов им необходимо выделять память, а при уничтожении – освобождать ее. Эти функции выполняют конструктор Create и деструктор Destroy класса TObject, от которого наследуются все классы Delphi. Следовательно, каждый объект при конструировании и уничтожении должен вызывать эти методы. Если класс определяет собственные конструкторы и деструкторы, то они должны включать вызов конструкторов и деструкторов родительского класса:

```
Constructor <имя класса>.Create;
  begin
     inherited Create;
       ...
  end;
Destructor <имя класса>.Destroy;
  begin
       ...
     inherited Destroy;
  end;
```

Обращение к объектам осуществляется по имени указателя *без использования операции разыменования*, что, к сожалению, заставляет забывать об их динамической природе:

Borland Pascal 7.0:

```
Type pTNum = ^TNum;
      TNum = Object
        n: integer;
        constructor Init (an:integer);
      end;
Constructor TNum.Init;
 begin
    n:=an;
 end;
  . . .
  Var p:pTNum;
  Begin
    New(p, Init(5));
    WriteLn(p^.n);
    Dispose(p);
  End.
```

Delphi Pascal:

```
Type
  TNum = class
      public  n:integer;
      constructor Create (an:integer);
    end;
Constructor TNum.Create;
  begin     inherited Create;
            n:=an;
  end;
 . . .
 Var A:TNum;
Begin
    A:=TNum.Create(5);
    WriteLn(A.n);
    A.Destroy;
End;...
```

Изменился синтаксис описания класса, в Delphi Pascal он выглядит так:

```
Type <имя объявляемого класса> = class (<имя родителя>)
        private      <скрытые элементы класса>
        protected    <защищенные элементы класса>
        public       <общедоступные элементы класса>
        published    <опубликованные элементы класса>
        automated    <элементы, реализующие OLE-механизм>
    end;
```

Секция *protected* содержит объявление полей и методов, доступных в пределах модуля и методам классов потомков. Секция *published* используется для объявлений компонент, доступных через Инспектор объектов (см. приложение 6).

Изменилось описание виртуальных методов: только самый первый виртуальный метод в иерархии описывается *virtual*, все методы, перекрывающие его, описываются с директивой *override*. Если для некоторого класса объявляется деструктор, то он описывается с директивой override, так как деструктор класса TObject является виртуальным.

Для описания абстрактных методов класса в объектной модели Delphi используют специальную директиву *abstruct*, например:

```
Type TNumber=class(TObject)
      public
        Procedure Print; virtual; abstruct;
    end;
   TIntNumber=class(TNumber)
     private i:integer;
     public
       Constructor Create(ai:integer);
       Procedure Print;override;
    end;
```

Кроме этого в объектной модели Delphi Pascal появился целый ряд новых возможностей и средств:

- динамические и абстрактные полиморфные методы;
- свойства (простые, массивы, индексные);
- средства реализации делегирования методов;
- метаклассы;
- перегрузка методов и т.п.

Особенности их использования подробно рассмотрены в [5].

П5. Создание приложений Windows с использованием среды программирования Delphi

Интегрированная среда программирования Delphi (рис. П5.1) предназначена для создания 32-разрядных приложений Windows. В отличие от программ, исполняемых в MS DOS, приложения Windows используют принцип *событийного программирования*, согласно которому программа представляет собой совокупность подпрограмм – обработчиков событий, таких, как получение сигналов от клавиатуры, мыши, таймера и т.п. Она не имеет алгоритма в традиционном смысле, так как связь между отдельными частями не задана жестко, а зависит от последовательности наступления событий.

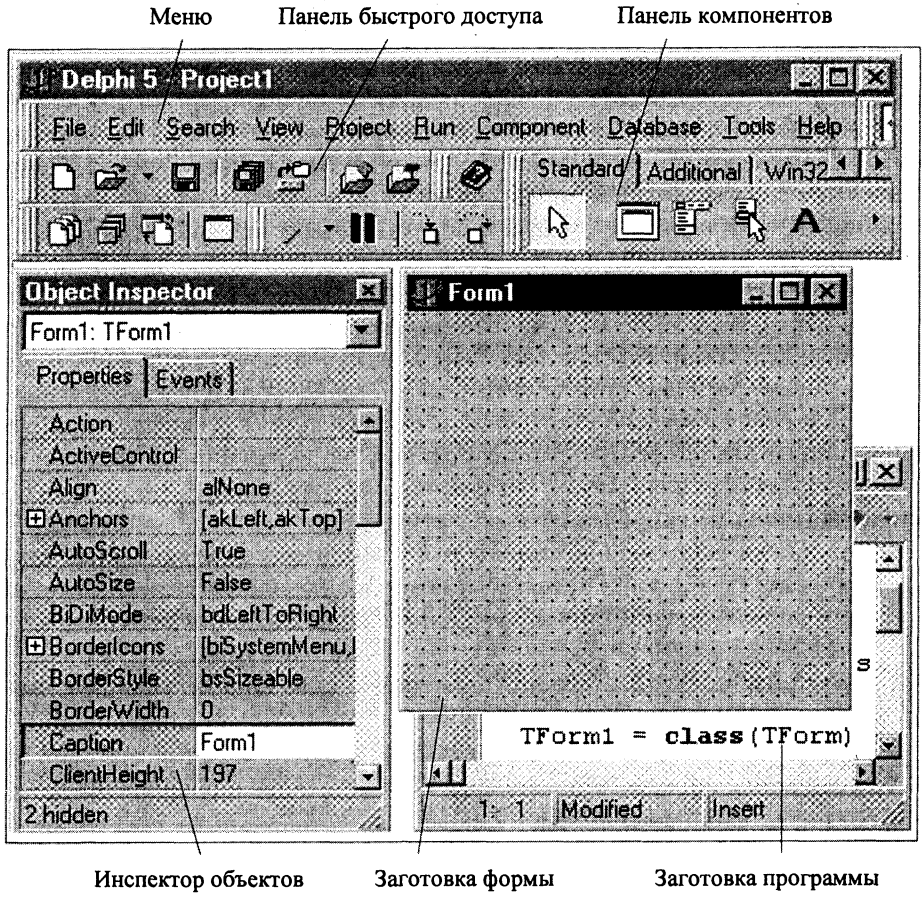


Рис. П5.1. Вид экрана при входе в Delphi

Delphi также поддерживает технологию визуального проектирования пользовательского интерфейса, согласно которой разработчику предоставляется возможность прямо на экране формировать интерфейс приложения из стандартных элементов, расположенных на панели компонентов.

Создание приложения начинают с заготовки, предоставляемой Delphi. (Она представляет собой минимальное приложение и может быть запущена на выполнение, правда при отсутствии полезного эффекта.) Мышью выбирают на панели необходимый интерфейсный компонент и переносят его на заготовку формы. При этом на первой закладке Properties Инспектора Объектов высвечиваются параметры компонента, заданные по умолчанию, которые можно переопределить, а на второй Events – перечень событий, которые он может обрабатывать. То же происходит при выделении мышью уже установленного компонента.

Визуально разработав интерфейс, программист определяет множество событий, необходимых для выполнения требуемых функций, распределяет эти события между компонентами и программирует соответствующие обработчики.

Среда Delphi предназначена для создания больших программ, элементы которых размещаются в разных файлах, но образуют единый проект. Основная программа, содержащая описание проекта, размещается в файле с расширением .dpr. Как правило, она формируется самой средой Delphi, но при необходимости программист может ее изменить.

Кроме этого приложение включает одну или несколько интерфейсных объектов-форм, каждой из которых соответствует файл описания с расширением .dfm и модуль исходного текста на Delphi Pascal с расширением .pas. Файл описания формы и описание класса в модуле формируются автоматически в процессе визуального создания интерфейса, а тела методов проектируются и реализуются программистом. Возможно включение в проект и модулей, не связанных с формами, например, для хранения описаний классов, реализующих объекты предметной области.

В результате успешной компиляции и компоновки программы создаются: исполняемый файл типа «exe», имя которого совпадает с именем проекта, объектные модули с расширением .dcu и файлы ресурсов с расширением .res.

Рассмотрим последовательность создания простейших приложений в Delphi на конкретных примерах.

Пример П5.1. Разработать калькулятор – приложение Windows, выполняющее основные арифметические операции.

Разработку будем выполнять по шагам.

1. *Определение имени проекта и первого модуля.* Для определения имени проекта используют пункт меню File\Save Project As... В появившемся окне создайте новую папку и сохраните в ней файл с именем MyProject. Для определения имени модуля необходимо щелкнуть либо по заготовке формы

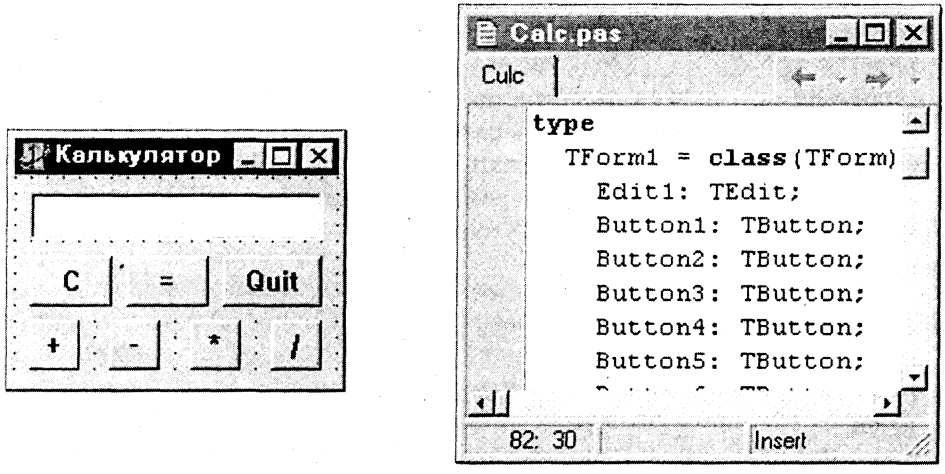
(Form1), либо по окну заготовки программа (Unit1). Затем, используя пункт меню File\Save As..., сохраняем в той же папке, что и проект, модуль формы и саму форму под именем Calc.

2. *Определение заголовка окна программы.* Заголовок окна программы задается в свойствах окна формы. Для определения необходимо, предварительно выделив щелчком форму, на странице Propeties Инспектора объектов щелчком выделить свойство Caption (Заголовок) и ввести имя «Калькулятор».

3. *Размещение на форме окна ввода/отображения чисел* – однострочного текстового редактора. Для выполнения этой операции на странице Standart палитры компонентов необходимо найти кнопку Edit, мышью перетащить этот компонент в нужное место формы (рис. П5.2 , *а*) и мышью же скорректировать его размер. Чтобы удалить текст из окна компонента на странице Properties Инспектора объектов необходимо найти свойство Text и очистить его поле.

Одновременно с размещением компонент Delphi добавляет объектные поля в класс с именем TForm1, наследуемый от стандартного класса TForm (рис. П5.2, *б*).

4. *Размещение кнопок операций на форме.* Для выполнения этой операции на странице Standart палитры компонентов необходимо найти кнопку Button. Для того чтобы не перетаскивать каждый компонент отдельно, перед выбором мышью компонента следует нажать клавишу Shift. Теперь, щелкая мышью в нужных местах, можно установить сразу все семь кнопок. Для отмены работы с кнопкой необходимо щелкнуть мышью по стрелке под словом Standart палитры компонентов. Затем, последовательно щелкая мышью



а *б*

Рис. П5.2. Вид формы (*а*) и окна программы (*б*) во время проектирования

по установленным кнопкам, измените их размеры. После этого, используя Инспектор объектов, установите требуемые названия кнопок (свойство Caption на странице Properties инспектора объектов) соответственно на «C», «=», «+», «-», «*», «/» и «Quit». Теперь программу необходимо «научить» выполнять требуемые функции.

5. *Добавление обработчиков событий.* Для добавления обработчиков событий обычно используют страницу Events Инспектора объектов. На этой странице указаны все события, на которые может реагировать выделенный компонент. Выделите компонент Button1 и, щелкнув по соответствующей строке, выберете событие OnClick («щелчок мыши по компоненту»). Двойным щелчком по полю рядом с ним вызовите заготовку обработчика данного события и введите его текст:

```
Procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Clear;       {очистить окно компонента Edit1}
  operation:='@';    {установить состояние «первая операция»}
  Edit1.setfocus;    {установить активным окно компонента Edit1}
end;
```

Если обработчик некоторого события уже определен, то по двойному щелчку мыши по его имени осуществляется переход на его текст.

Аналогично добавляем тексты обработчиков других событий:

• Button2Click – щелчок по кнопке «=»:

```
Procedure TForm1.Button2Click(Sender: TObject);
Var s:string;
begin
  operate;           {выполнить предыдущую операцию}
  operation:='=';    {установить состояние «операция =»}
  Str(sum:6:3,s);    {преобразовать результат в строку}
  Edit1.text:=s;     {вывести строку в окно компонента Edit1}
  Button1.setfocus;  {установить курсор на кнопку Button1}
end;
```

• Button3Click – щелчок по кнопке «+»:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  operate;           {выполнить предыдущую операцию}
  operation:='+';    {установить состояние «операция +»}
  Edit1.setfocus;    {установить активным окно компонента Edit1}
end;
```

• Button4Click – щелчок по кнопке «-»:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
   operate;          {выполнить предыдущую операцию}
   operation:='-';   {установить состояние «операция -»}
  Edit1.setfocus;    {установить активным окно компонента Edit1}
end;
```

• Button5Click – щелчок по кнопке «*»:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
   operate;          {выполнить предыдущую операцию}
   operation:='*';   {установить состояние «операция *»}
   Edit1.setfocus;   {установить активным окно компонента Edit1}
end;
```

• Button6Click – щелчок по кнопке «/»:

```
procedure TForm1.Button6Click(Sender: TObject);
begin
   operate;          {выполнить предыдущую операцию}
   operation:='/';   {установить состояние «операция /»}
   Edit1.setfocus;   {установить активным окно компонента Edit1}
end;
```

• Button7Click – щелчок по кнопке «Quit»:

```
procedure TForm1.Button7Click(Sender: TObject);
begin
  Close;             {завершить работу приложения}
end;
```

6. *Добавление процедуры.* Для выполнения вычислений необходимо добавить процедуру Operate. Поскольку она вызывается из обработчиков событий, ее необходимо вставить *перед* ними в секции реализации модуля:

```
Procedure Operate;
 Var  s:string;  code:integer;  n:double;
 Begin
      s:=Form1.Edit1.text;  {читаем сроку из Edit1.text }
      Form1.Edit1.clear;    {очищаем Edit1}
```

```
        Val(s,n,code);          {преобразуем строку в число}
        case operation of       {выполняем операцию}
          '@': sum:=n;
          '+': sum:=sum+n;
          '-': sum:=sum-n;
          '*': sum:=sum*n;
          '/': sum:=sum/n;
        end;
end;
```

7. *Объявление переменных.* Поскольку переменные, необходимые для вычислений, являются внутренними, объявляем их в секции реализации модуля (после служебного слова implementation):

```
Var   Sum:double;
      operation:char='@';
```

8. *Сохранение проекта.* Для сохранения проекта используют пункт меню File\SaveAll.

9. *Компиляция проекта.* Для выполнения компиляции используют комбинацию клавиш Ctrl-F9 или пункт меню Project\Compile. Если при компиляции обнаружены ошибки, то внимательно проверьте текст программы.

10. *Запуск программы на выполнение.* Для запуска программы используют клавишу F9 или пункт меню Run/Run или кнопку Run на панели быстрого доступа.

Второй пример возьмем немного сложнее и выполним некоторые проектные операции в процессе его разработки. Причем, несмотря на очевидную простоту приложения, будем использовать объектную технологию как для представления пользовательского интерфейса, так и для предметной области задачи.

Пример П5.2. Разработать приложение Windows для возведения вводимых чисел в квадрат.

А н а л и з. Начинаем с объектной декомпозиции интерфейсной и предметной частей приложения. Интерфейсная часть включает только Окно приложения. Предметная – объект Число, который отвечает за инициализацию числа и возведение его в квадрат (рис. П5.3).

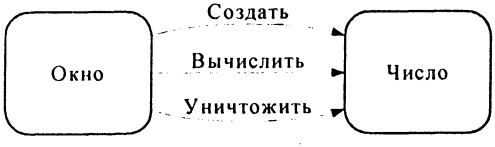


Рис. П5.3. Объектная декомпозиция

При анализе реализуемых функций учтем, что при вводе чисел возможны ошибки. Следовательно, необходим интерфейс с тремя основными состояниями: Ввод числа, Демонстрация результата и Демонстрация сообще-

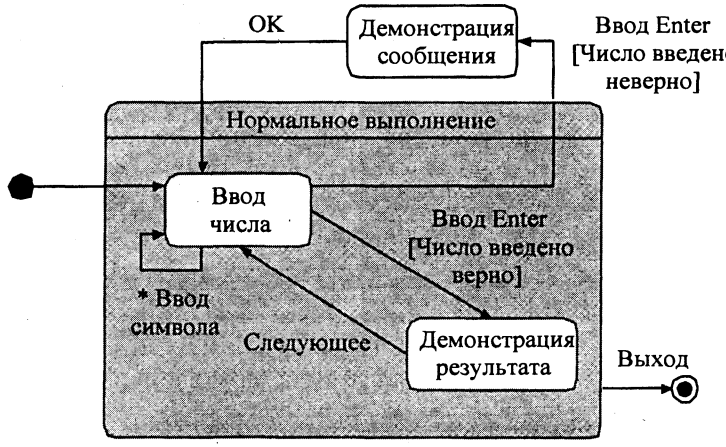


Рис. П5.4. Диаграмма переходов состояний интерфейса

ния об ошибке (рис. П5.4). Из двух первых состояний необходимо обеспечить возможность выхода, поэтому объединим их в дополнительное состояние – Нормальное выполнение.

Проектирование интерфейса. Основное окно приложения проектируем, используя средства визуальной среды Delphi (рис. П5.5), аналогично тому, как это было сделано в предыдущем примере. Затем уточняем внешний вид приложения в каждом из состояний интерфейса (рис. П5.6): определение жестких ограничений на действия пользователя позволяет избежать некорректной работы программы в условиях событийного программирования.

Используя Инспектор объектов, определяем имена объектов (по умолчанию Delphi, как было показано в предыдущем примере, называет объекты, используя имя компонента и номер объекта, что неудобно, если разных компонентов включается в форму много) и значения некоторых свойств:

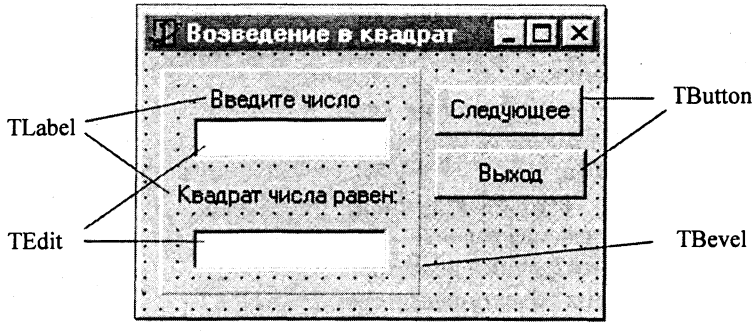
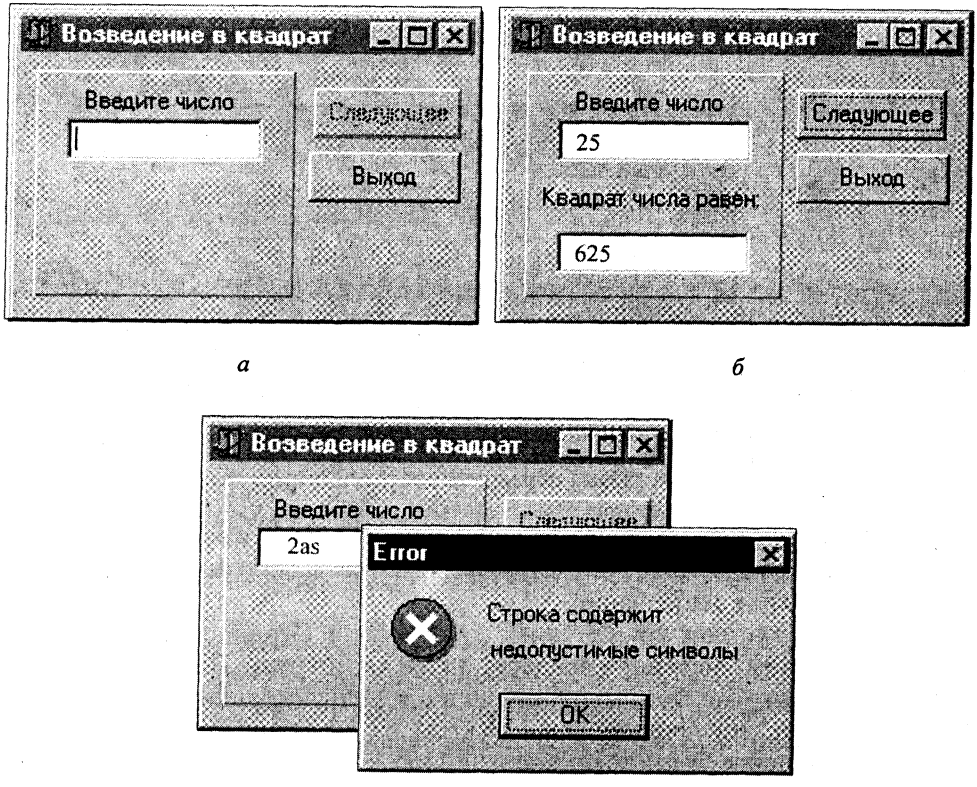


Рис. П5.5. Проектирование формы основного окна приложения



а *б* *в*

Рис. П5.6. Интерфейс приложения «Возведение числа в квадрат»:
а – ввод числа (исходное состояние); *б* – демонстрация результата;
в – демонстрации сообщения об ошибке

Form1 – главная форма приложения:
Name: MainForm – имя объекта-формы;
Caption: 'Возведение числа в квадрат' – заголовок окна;
Label1 – первая метка:
Name: InputLabel – имя объекта-метки;
Caption: 'Введите значение' – текст метки;
Label2 – вторая метка:
Name: OutPutLabel – имя объекта-метки;
Caption: 'Квадрат значения равен: ' – текст метки;
Edit1 – первый однострочный редактор:
Name: InputEdit – имя объекта-редактора;
Edit2 – второй однострочный редактор:
Name: OutPutEdit – имя объекта-редактора;

Enable: false – запрет активизации редактора пользователем;
ReadOnly: true – запрет ввода текста в редактор;
Button1 – первая кнопка:
Name: NextButton – имя объекта-кнопки;
Caption: 'Следующее' – название кнопки;
Button2 – вторая кнопка:
Name: ExitButton – имя объекта-кнопки;
Caption: 'Выход' – название кнопки.

Проектирование объекта предметной области. Анализ взаимодействия объектов, полученных в процессе объектной декомпозиции показывает, что класс TNumber для реализации объекта Число должен включать поле Num для хранения введенного значения, конструктор и метод определения квадрата числа. Деструктор он может наследовать от класса TObject. Полная диаграмма классов приложения показана на рис. П5.7 (серым выделены стандартные классы Delphi). Классы TMainForm и TNumber связаны отношением ассоциации, так как класс формы посылает классу объекта сообщения.

Реализация. Описание класса TNumber и его методов целесообразно поместить в специальном модуле:

```
Unit UnitNumber;
Interface
 Type
     TNumber=class(TObject)
        private Num:single;
        public  Constructor Create(aNum:single);
                Function SqrNumber:single;
     end;
```

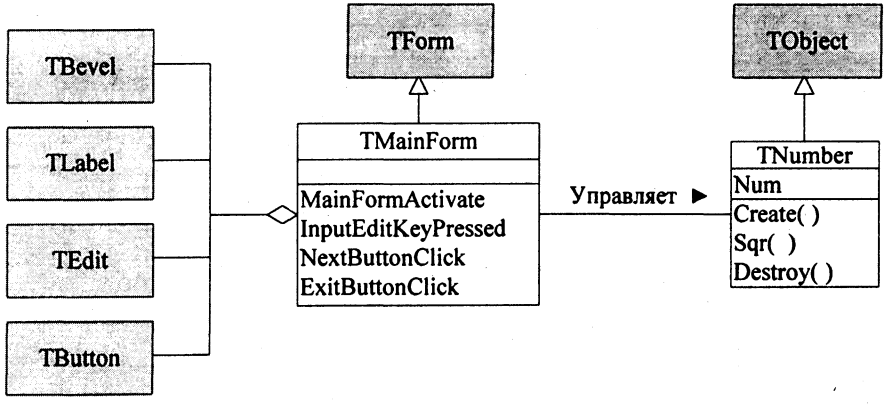


Рис. П5.7. Диаграмма классов приложения

```
 Var N:TNumber;
Implementation
   Constructor TNumber.Create(aNum:single);
     begin
        inherited Create;
        Num:=aNum;
     end;
   Function TNumber.SqrNumber:single;
     begin    Result:=Sqr(Num);
     end;
end.
```

В секции реализации модуля MainForm указываем использование модуля UnitNumber.

Обработчики событий. Привязываем проектируемые события, указанные на диаграмме переходов состояний интерфейса, к событиям, обрабатываемым визуальными компонентами, и описываем необходимые действия приложения (табл. П5.1).

Таблица П5.1

Реальные события	События компонентов	Действия
Запуск приложения	MainForm: OnActivate – получение окном управления	Настройка интерфейса на ввод числа (рис. П5.6, *а*)
Ввод цифр	–	Обрабатывается компонентом TEdit автоматически
Ввод Enter	InputEdit: OnKeyPressed – ввод алфавитно-цифровой информации в окно компонента редактирования	Если введено не число, то вывод сообщения об ошибке (рис. П5.6, *б*), иначе – создание объекта Число, генерация запроса о квадрате числа, вывод результата (рис. П5.6, *в*) и уничтожение объекта
Щелчок по кнопке OK	–	Обрабатывается окном сообщения автоматически
Щелчок по кнопке Следующее	NextButton: OnClick – щелчок мышью по кнопке	Настройка интерфейса на ввод числа (рис. П5.6, *а*)
Щелчок по кнопке Выход	ExitButton: OnClick – щелчок мышью по кнопке	Завершение приложения

Последовательно выделяем компоненты, выбираем в списке событий Events Инспектора объектов нужные события и дважды щелкаем мышью на чистом поле рядом с ними. В результате создаются заготовки обработчиков событий, в которые необходимо вписать соответствующие фрагменты программы. В конечном итоге получаем полный текст модуля MainForm, приведенный ниже (операторы, вводимые программистом, выделены полужирным шрифтом).

```
Unit MainForm;
Interface
Uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
     Dialogs, StdCtrls, ExtCtrls;  {модули Delphi}
Type
 TMainForm= class(TForm){описание формы – создано автоматически}
   InputLabel: TLabel;    OutputLabel: TLabel; {метки}
   InputEdit: TEdit;      OutputEdit: TEdit;   {редакторы}
   NextButton: TButton;  ExitButton: TButton;  {кнопки}
   Bevel1: TBevel;                             {рамка}
   procedure FormActivate(Sender: TObject);
   procedure InputEditKeyPress(Sender: TObject; var Key: Char);
   procedure NextButtonClick(Sender: TObject);
   procedure ExitButtonClick(Sender: TObject);
 end;
   Var  MainForm: TMainForm;  {объект – создано автоматически }
Implementation
uses UnitNumber;   {описание класса TNumber}
{$R *.DFM}
Procedure TMainForm.FormActivate(Sender: TObject);
begin
   OutputEdit.Visible:=false;   {сделать редактор вывода невидимым}
   OutputLabel.Visible:=false; {сделать метку вывода невидимой}
   NextButton.Enabled:=false;{сделать кнопку Следующий
                                                  недоступной}
   InputEdit.Clear;             {очистить редактор ввода}
   InputEdit.ReadOnly:=false;  {разрешить ввод}
   InputEdit.SetFocus;       {установить фокус ввода на редактор ввода}
end;
Procedure TMainForm.InputEditKeyPress(Sender: TObject; var Key: Char);
  Var k:single; Code:integer;
begin
   if Key=#13 then
     begin
       Key:=#0; {чтобы не выдавался звуковой сигнал}
```

```
        Val(InputEdit.Text,k,Code);
        if Code=0 then
          begin
           N:=TNumber.Create(strtofloat(InputEdit.Text)); {создать объект}
           OutPutEdit.Text:=floattostr(N.SqrNumber); {вывести результат}
           N.Destroy; {уничтожить объект – деструктор TObject}
           OutPutEdit.Visible:=true; {сделать редактор вывода видимым}
           OutputLabel.Visible:=true; {сделать метку вывода видимой}
           InputEdit.ReadOnly:=true; {запретить ввод}
           NextButton.Enabled:=true; {сделать кнопку Следующий
                                                    доступной}
           NextButton.SetFocus; {установить фокус ввода на кнопку
                                                    Следующий}
          end
        else {вывести сообщение об ошибке}
            MessageDlg('Строка содержит недопустимые символы',
                                                    mtError,[mbOk],0)
    end;
end;
Procedure TMainForm.NextButtonClick(Sender: TObject);
  begin    FormActivate(NextButton); end; {вызываем метод}
Procedure TMainForm.ExitButtonClick(Sender: TObject);
  begin    Close;  end;  {закрываем окно и завершаем приложение}
End.
```

Основную программу приложения – проект Delphi формирует автоматически:

```
Program ProectXQ;
Uses
  Forms,
  MainFormUnit in 'MainFormUnit.pas', {Mainform}
  UnitNumber in 'UnitNumber.pas';
{$R *.RES}
Begin
  Application.Initialize;   {инициализация объекта Приложение}
  Application.CreateForm(TMainForm, MainForm);{создание формы}
  Application.Run;     {запуск цикла обработки сообщений}
End.
```

Чтобы лучше показать возможности среды, в качестве третьего примера рассмотрим, как можно реализовать в Delphi уже знакомый нам пример с записной книжкой (см. пример 12.1).

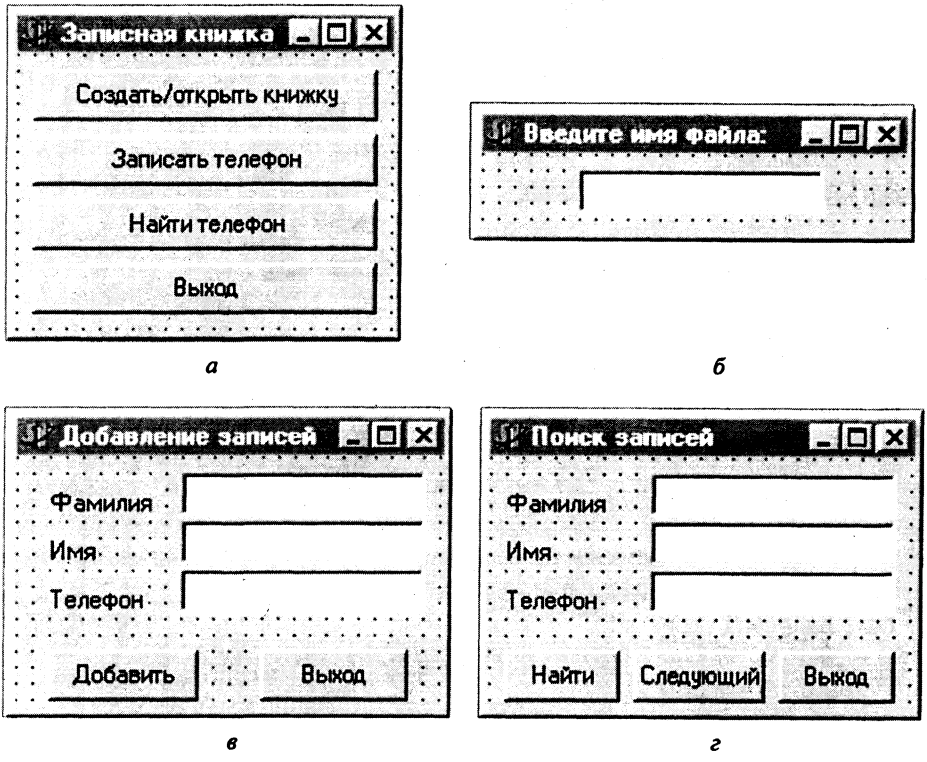


Рис. П5.8. Формы приложения «Записная книжка»:
а – основная форма; *б* – форма создания/открытия файла;
в – форма добавления записей; *г* – форма поиска записей

Пример П5.3. Разработать приложение «Записная книжка». На рис. П5.8 представлены формы разрабатываемого приложения. Их проектируют с использованием визуальной технологии.

На рис. П5.9 представлена диаграмма классов приложения. Основное ее отличие от диаграммы, приведенной на рис. 12.13, заключается в том, что в ней использованы интерфейсные компоненты Delphi, и класс TBase наследуется от класса TObject. Кроме того, не предусмотрен специальный класс для выдачи сообщения пользователю, так как в Delphi с этой целью обычно используют специальную процедуру MessageDlg, которая выводит нужное сообщение.

Описание класса TBase практически полностью совпадает с выполненным на Borland Pascal 7.0. Внесены всего три изменения: заменены имена процедур Assign и Close соответственно на AssignFile и CloseFile и объявление объекта Base класса TBase перенесено в модуль, содержащий описание класса.

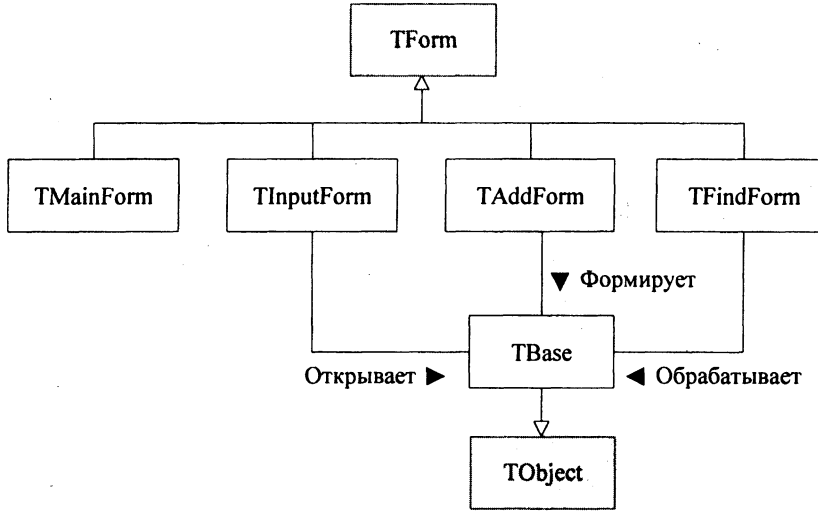


Рис. П5.9. Диаграмма классов приложения Записная книжка

```
Unit BaseUnit;
interface
 Type str30=string[30];
    rec=record
        rfamily, rname, rtelefon:str30;
       end;
 Type TBase=Object
      public
       f:file of rec;
       family, name, telefon:str30;  {результаты поиска}
       p_family,p_name:str30;   {данные поиска}
       k1, k2:boolean;              {ключи поиска}
       procedure Open(fname:str30); {открытие/создание файла}
       procedure Add(afamily,aname,atelefon:str30);{добавление записей}
       function Find(afamily,aname:str30):boolean; {поиск первого}
       function FindNext:boolean;                  {поиск следующего}
       procedure Closef;                           {закрытие файла}
    End;
 Var
    Base:TBase;
implementation
Procedure TBase.Open;
 Begin
      AssignFile(f,fname);  {инициализация файловой переменной}
```

```
        {$I-}
         Reset(f); {открытие с проверкой существования}
        {$I+}
       if IOResult<>0 then ReWrite(f);   {создание файла}
  End;
Procedure TBase.Add;
 Var r:rec;
  Begin
    Seek(f,FileSize(f));    {устанавливаем указатель на конец файла}
    r.rfamily:=afamily;     {создаем запись}
    r.rname:=aname;
    r.rtelefon:=atelefon;
    Write(f,r);             {выводим запись в файл}
  End;
Function TBase.Find;
  Begin
    CloseFile(f);           {закрываем файл}
    ReSet(f);               {открываем файл для чтения}
    p_family:=afamily;      {сохраняем данные поиска}
    p_name:=aname;
    k1:=p_family<>'';       {строим ключи поиска}
    k2:=p_name<>'';
    Find:=FindNext;         {ищем запись по ключам}
  End;
Function TBase.FindNext;
 Var r:rec; k3, k4, ff:boolean;  {ключи поиска и его результат}
 Begin
   ff:=false;   {ключ поиска "запись не найдена"}
   while not Eof(f) and not ff do
     begin
        Read(f,r);
        k3:=p_family=r.rfamily;   {строим еще два ключа поиска}
        k4:=p_name=r.rname;
        if (k1 and k2 and k3 and k4)   {выбираем записи}
          or (not k1 and k2 and k4)
          or (k1 and not k2 and k3) then
          begin
            ff:=true;          {ключ поиска "запись найдена"}
            family:=r.rfamily;  {копируем результаты поиска}
            name:=r.rname;
            telefon:=r.rtelefon;
          end
     end;
```

```
    FindNext:=ff;    {возвращаем ключ поиска}
   end;
Procedure TBase.Closef;
 begin
  CloseFile(f);    {закрываем файл}
 end;
end.
```

Объект MainForm класса TMainForm для всех кнопок, кроме кнопки Выход, должен вызывать соответствующий объект для выполнения основных операций с файлом. Имя файла должен вводить объект InputForm, он же должен создать/открыть файл. До открытия файла нельзя разрешать добавление и поиск записей, поэтому для компонентов MainForm AddButton и FindButton в Инспекторе объектов устанавливают свойство Enable в состояние false. Тогда при запуске программы эти кнопки будут не активны. После нормального выполнения InputForm использование этих кнопок будет разрешено: указанное свойство будет установлено в состояние true программно.

```
Unit MainUnit;
interface
uses
      Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
      Dialogs, StdCtrls;
type
 TMainForm = class(TForm)
  OpenButton: TButton;
  AddButton: TButton;
  FindButton: TButton;
  ExitButton: TButton;
  procedure OpenButtonClick(Sender: TObject);
  procedure AddButtonClick(Sender: TObject);
  procedure FindButtonClick(Sender: TObject);
  procedure ExitButtonClick(Sender: TObject);
 end;
var  MainForm: TMainForm;
implementation
 uses AddUnit, FindUnit, InputUnit;
 {$R *.DFM}
procedure TMainForm.OpenButtonClick(Sender: TObject);
begin
   if InputForm.ShowModal=mrOk then  {если выполнение InputForm}
     begin                           {завершилось благополучно, то}
      AddButton.Enabled:=true;       {разрешить добавление}
```

```
        FindButton.Enabled:=true; {разрешить поиск}
      end;
end;
procedure TMainForm.AddButtonClick(Sender: TObject);
begin
   AddForm.ShowModal;      {активизировать AddForm}
end;
procedure TMainForm.FindButtonClick(Sender: TObject);
begin
   FindForm.ShowModal;     {активизировать FindForm}
end;
procedure TMainForm.ExitButtonClick(Sender: TObject);
begin
  Close;    {завершить приложение}
end;
end.
```

При активации формы InputForm фокус должен быть установлен на редактор ввода. Ввод обрабатывается автоматически. Завершение ввода осуществляется нажатием клавиши Enter.

```
Unit InputUnit;
interface
uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
 StdCtrls;
type
 TInputForm = class(TForm)
   InputEdit: TEdit;
   procedure InputEditKeyPress(Sender: TObject; var Key: Char);
   procedure FormActivate(Sender: TObject);
 end;
var  InputForm: TInputForm;
implementation
 uses BaseUnit;
{$R *.DFM}
procedure TInputForm.FormActivate(Sender: TObject);
begin
   InputEdit.SetFocus;
end;
procedure TInputForm.InputEditKeyPress(Sender: TObject; var Key: Char);
begin
   if Key=#13 then
```

```
        begin
          Key:=#0;
          Base.Open(InputEdit.Text);
          ModalResult:= mrOK;      {завершить благополучно}
        end;
end;
end.
```

Объект AddForm отвечает за добавление записей. На рис. П5.10 представлена диаграмма переходов состояний интерфейса для объекта AddForm, которая уточняет его поведение при вводе записи по полям. Из диаграммы следует, что ввод данных начинается с нажатия кнопки Добавить, переход от поля к полю осуществляется нажитием клавиши Enter, запись в файл происходит после ввода телефона, а выход возможен из любого состояния. Внутренние переходы по вводу отдельных символов не показаны: они обрабатываются компонентами автоматически.

```
Unit AddUnit;
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls;
```

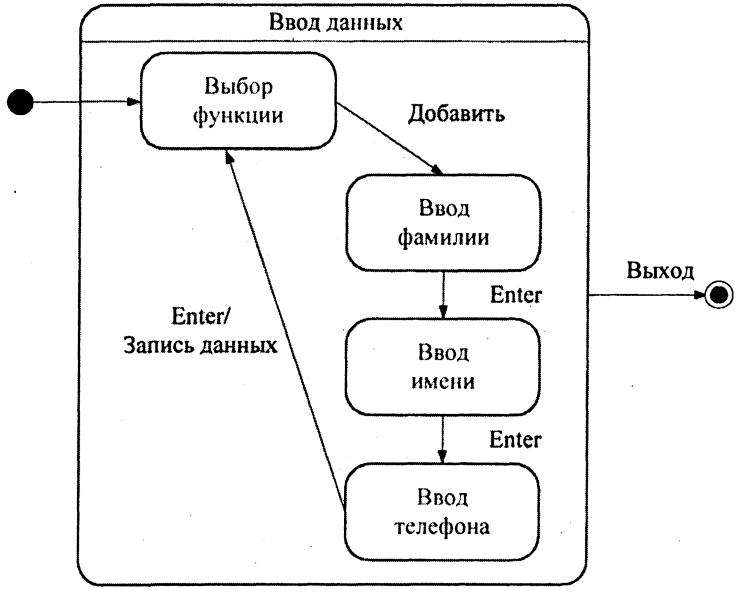


Рис. П5.10. Диаграмма переходов состояний интерфейса для объекта AddForm

```
type
  TAddForm = class(TForm)
    FamLabel: TLabel;
    NameLabel: TLabel;
    FonLabel: TLabel;
    AddButton: TButton;
    ExitButton: TButton;
    FamEdit: TEdit;
    NameEdit: TEdit;
    FonEdit: TEdit;
    procedure FamEditKeyPress(Sender: TObject; var Key: Char);
    procedure NameEditKeyPress(Sender: TObject; var Key: Char);
    procedure FonEditKeyPress(Sender: TObject; var Key: Char);
    procedure FormActivate(Sender: TObject);
    procedure AddButtonClick(Sender: TObject);
    procedure ExitButtonClick(Sender: TObject);
  end;
var
    AddForm: TAddForm;
implementation
  uses BaseUnit;
{$R *.DFM}
procedure TAddForm.FormActivate(Sender: TObject);
begin
   ExitButton.SetFocus;
end;
procedure TAddForm.AddButtonClick(Sender: TObject);
begin
   FamEdit.SetFocus;
end;
procedure TAddForm.FamEditKeyPress(Sender: TObject; var Key: Char);
begin
   if Key=#13 then
          begin
            Key:=#0;
            NameEdit.SetFocus;
          end;
end;
procedure TAddForm.NameEditKeyPress(Sender: TObject; var Key: Char);
begin
   if Key=#13 then
          begin
            Key:=#0;
```

```
            FonEdit.SetFocus;
          end;
end;
procedure TAddForm.FonEditKeyPress(Sender: TObject; var Key: Char);
begin
  if Key=#13 then
          begin
            Key:=#0;
            Base.Add(FamEdit.Text,NameEdit.Text,FonEdit.Text);
            ExitButton.SetFocus;
          end;
end;
procedure TAddForm.ExitButtonClick(Sender: TObject);
begin
  ModalResult:= mrOK;      {завершить благополучно}
end;
end.
```

Объект FindForm отвечает за поиск записей. Переход от поля к полю при вводе осуществляется нажитием клавиши Enter.

```
unit FindUnit;
interface
uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
 StdCtrls;
type
 TFindForm = class(TForm)
   FamLabel: TLabel;
   NameLabel: TLabel;
   FonLabel: TLabel;
   FindButton: TButton;
   ExitButton: TButton;
   NextButton: TButton;
   FamEdit: TEdit;
   NameEdit: TEdit;
   FonEdit: TEdit;
   procedure FamEditKeyPress(Sender: TObject; var Key: Char);
   procedure NameEditKeyPress(Sender: TObject; var Key: Char);
   procedure FormActivate(Sender: TObject);
   procedure FindButtonClick(Sender: TObject);
   procedure NextButtonClick(Sender: TObject);
   procedure ExitButtonClick(Sender: TObject);
```

```
  private
    Procedure Show;
  end;
var
    FindForm: TFindForm;
implementation
  uses BaseUnit;
{$R *.DFM}
Procedure TFindForm.Show;
begin
  FamEdit.Text:=Base.family;
  NameEdit.Text:=Base.name;
  FonEdit.Text:=Base.telefon;
end;
procedure TFindForm.FormActivate(Sender: TObject);
begin
    ExitButton.SetFocus;
end;
procedure TFindForm.FindButtonClick(Sender: TObject);
begin
    FamEdit.SetFocus;
end;
procedure TFindForm.FamEditKeyPress(Sender: TObject; var Key: Char);
begin
    if Key=#13 then
            begin
              Key:=#0;
              NameEdit.SetFocus;
            end;
end;
procedure TFindForm.NameEditKeyPress(Sender: TObject; var Key: Char);
begin
   if Key=#13 then
     begin
       Key:=#0;
       if Base.Find(FamEdit.Text,NameEdit.Text) then
          begin
            Show;
            NextButton.SetFocus;
          end
       else
          begin
            MessageDlg( 'Нет данных',mtInformation,[mbOk],0);
```

```
            ExitButton.SetFocus;
          end
      end;
  end;
  procedure TFindForm.NextButtonClick(Sender: TObject);
  begin
     if Base.FindNext then
        begin
          Show;
          NextButton.SetFocus;
        end
     else
       begin
         MessageDlg( 'Нет данных',mtInformation,[mbOk],0);
         ExitButton.SetFocus;
       end
  end;
  procedure TFindForm.ExitButtonClick(Sender: TObject);
  begin
    ModalResult:= mrOK;      {завершить благополучно};
  end;
  end.
```

В процессе проектирования приложения Delphi автоматически строит файл проекта.

```
program BookProject;
uses  Forms,
  MainUnit in 'MainUnit.pas' {MainForm},
  InputUnit in 'InputUnit.pas' {InputForm},
  AddUnit in 'AddUnit.pas' {AddForm},
  FindUnit in 'FindUnit.pas' {FindForm},
  BaseUnit in 'BaseUnit.pas';
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.CreateForm(TInputForm, InputForm);
  Application.CreateForm(TAddForm, AddForm);
  Application.CreateForm(TFindForm, FindForm);
  Application.Run;
end.
```

СПИСОК ЛИТЕРАТУРЫ

1. *Бадд Т.* Объектно-ориентированное программирование в действии. СПб.: Питер, 1997.

2. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином; СПб.: Невский диалект, 1998.

3. *Вирт Н.* Алгоритмы и структуры данных. М.: Мир, 1989.

4. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование. М.: Мир, 1975.

5. *Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К.* Объектно-ориентированное программирование. М.: Изд-во МГТУ им. Н.Э. Баумана, 2001.

6. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.

7. *Майерс Г.* Искусство тестирования программ. М.: Мир, 1982.

8. *Фаронов В.В.* Турбо-Паскаль. Основы Турбо-Паскаля. М.: «МВТУ – ФЕСТО ДИДАКТИК», 1992.

9. *Хьюз Дж., Мичтом Дж.* Структурный подход к программированию. М.: Мир, 1980.

Предметный указатель

Учебное издание

ИНФОРМАТИКА В ТЕХНИЧЕСКОМ УНИВЕРСИТЕТЕ

Иванова Галина Сергеевна

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Редактор *Н.Е. Овчеренко*
Художники *С.С. Водчиц, Н.Г. Столярова*
Корректоры *Л.И. Малютина, О.Ю. Соколова*
Компьютерная верстка *Б.А. Иванов*

Подписано в печать 16.05.2002. Формат 70x100/16. Печать офсетная. Бумага газетная
Гарнитура «Таймс». Усл. печ. л. 33,8. Уч.-изд. л. 33,25. Тираж 5000 экз.
Заказ 1433

Издательство МГТУ им. Н.Э. Баумана
105005, Москва, 2-я Бауманская, 5

Отпечатано с оригинал-макета во ФГУП ИПК «Ульяновский
Дом печати». 432980, г. Ульяновск, ул. Гончарова, 14