

ВМК МГУ – ШКОЛЕ



Т. Ю. Грацианова

# ПРОГРАММИРОВАНИЕ в примерах и задачах



# ИНФОРМАТИКА



ЛАБОРАТОРИЯ

**ПИЛОТ**

ВМК МГУ – ШКОЛЕ



Т. Ю. Грацианова

# ИНФОРМАТИКА

## ПРОГРАММИРОВАНИЕ

в примерах и задачах

3-е издание,  
исправленное и дополненное  
(электронное)



Москва  
Лаборатория знаний  
2016

УДК 004.9  
ББК 32.97  
Г78

**Грацианова Т. Ю.**

**Г78** Программирование в примерах и задачах [Электронный ресурс] / Т. Ю. Грацианова. — 3-е изд., испр. и доп. (эл.). — Электрон. текстовые дан. (1 файл pdf : 373 с.). — М. : Лаборатория знаний, 2016. — (БМК МГУ — школе). — Систем. требования: Adobe Reader XI ; экран 10".

ISBN 978-5-00101-436-2

Пособие поможет подготовиться к экзамену по информатике, научиться решать задачи по программированию на языке Паскаль. Рассмотрено большое количество программ; листинги приведены в расчете на использование среды Турбо Паскаль 7.0, однако в большинстве своем будут работать без всяких изменений и в других версиях Паскаля. Некоторые задачи имеют несколько вариантов решений, и в пособии подробно разобрано, какое из них является наилучшим.

Для школьников 8–11 классов, учителей информатики и методистов, а также студентов первых курсов технических вузов.

**УДК 004.9  
ББК 32.97**

**Деривативное электронное издание на основе печатного аналога:** Программирование в примерах и задачах / Т. Ю. Грацианова. — 3-е изд., испр. и доп. — М. : Лаборатория знаний, 2016. — 368 с. : ил. — (БМК МГУ — школе). — ISBN 978-5-00101-000-5.

**В соответствии со ст. 1299 и 1301 ГК РФ при устранении ограничений, установленных техническими средствами защиты авторских прав, правообладатель вправе требовать от нарушителя возмещения убытков или выплаты компенсации**

ISBN 978-5-00101-436-2

© Лаборатория знаний, 2015

# Введение

---

Вы держите перед собой учебное пособие, которое поможет вам подготовиться к экзамену по информатике, научиться программировать, решать задачи на компьютере. В качестве языка программирования мы выбрали язык Паскаль. Этот язык создан специально для учебных целей, удобен в качестве первого изучаемого языка программирования, так как, с одной стороны, он не очень сложен, а с другой — на нем можно решать достаточно серьезные задачи.

Никаких специальных знаний для того, чтобы начать заниматься программированием, не требуется. Чтобы усвоить материал нашего пособия, достаточно уметь читать и немного — считать, поэтому можно начать занятия и в 11-м классе, и в 10-м, и даже раньше.

Главное место в учебном пособии занимают задачи с решениями. Даже необходимые теоретические сведения и определения подаются с помощью задач. Решая задачи, мы объясняем, зачем нужны новые конструкции языка, как их использовать, показываем способы разработки алгоритмов, приемы программирования. В тексте разобрано порядка полутора сотен задач, причем почти все решения снабжены подробными пояснениями и доведены до полной программы (ее можно «набить» на компьютере и выполнить — будет работать!).

Как «набить» программу? Для этого вам понадобится компьютер, умеющий работать с одной из версий языка Паскаль. В пособии подробно рассматривается, как работать в среде Borland Pascal 7.0 (Борланд Паскаль, Турбо Паскаль), — все приведенные программы писались и отлаживались именно в ней. Также вы найдете пояснения по работе со средами Free Паскаль и ABC-Паскаль. Приведенные в пособии программы, за некоторым исключением, будут без изменений правильно работать в любой среде, так как писались на стандартном Паскале. Все случаи, где применены какие-то особенности Турбо Паскаля, в тексте отмечены, и вы сможете работать с такой программой в своей версии языка (например, ABC-Паскаль), может быть, слегка изменив ее.

Конечно, подбирая задачи, мы учитывали, что большинству из вас наше пособие понадобится для подготовки к ЕГЭ. Однако здесь вы не найдете решений вариантов ЕГЭ разных лет (хотя, конечно же, все типовые задачи рассматриваются). Наша цель — не показать вам, как решаются задачи,

которые когда-то были на экзаменах, а помочь научиться думать, составлять новые алгоритмы, писать программы и выполнять их на компьютере.

Немного о содержании учебника. Глава 1 знакомит с основными, базовыми понятиями программирования. Уже в ней мы разбираем достаточно сложные задачи, правда, пока на уровне блок-схем. В главах 2–12 мы изучаем основные конструкции языка Паскаль, учимся решать задачи. В принципе, изучив этот материал, можно решить любую задачу по программированию из предлагавшихся на ЕГЭ. В главах 13–14 содержится дополнительный материал, он поможет решать «обычные» задачи лучше и быстрее, а также пригодится тем, кто собирается участвовать (и побеждать!) в олимпиадах. А глава 15, можно сказать, развлекательная. Материал, который в ней рассматривается, обычно не входит ни в ЕГЭ, ни в школьную программу. Прочитав эту главу, вы сможете научить свой компьютер рисовать, сочинять музыку, сможете сами писать простенькие компьютерные игры, а заодно повторите все, чему научились ранее. В главе 16 приводятся примеры достаточно сложных задач, для решения которых понадобятся знания из разных глав учебника.

В учебнике много задач (около 600). Часть из них вы найдете после объяснения очередной темы. Многие из таких задач очень похожи на примеры, разобранные в тексте главы. Так что, если непонятно, как решать задачу, попробуйте перечитать материал еще раз, поищите аналогичную задачу. Конечно, есть и более сложные задачи, над которыми придется подумать. Они помечены звездочкой. В конце каждой главы вы найдете задачи по всему материалу, рассмотренному в главе; в этом случае не всегда удастся найти в главе аналог задачи, надо самостоятельно продумать метод решения.

Правильность решения практически всех задач очень легко проверить — в этом поможет компьютер. Введите текст программы, входные данные, запустите ее и сравните полученный ответ с правильным. Откуда взять правильный ответ? Подсчитать вручную! Не беспокойтесь, это просто, никаких сложных вычислений нигде делать не надо. Только не ленитесь проверять программы для разных наборов входных данных. К сожалению, программа может быть правильной, но неэффективной (например, слишком громоздкой), — здесь уже компьютер не поможет. Мы постараемся научить вас писать хорошие программы: некоторые задачи имеют несколько вариантов решений, и мы подробно разбираем, какой из них лучше.

Автору в работе над этой книгой большую помощь оказали преподаватели подготовительных курсов факультета Вычислительной математики и кибернетики (ВМК) МГУ имени М. В. Ломоносова Родин В. И., Вовк Е. Т., Линев Н. Б., Дорофеева И. Д., Лапониная О. Р.

# Глава 1

## Основные понятия и определения

---

### Программирование

Решение многих задач, возникающих в самых различных сферах человеческой деятельности, было бы невозможно без применения компьютеров. Причем компьютеры — это не только вычислительные устройства с дисплеем, которые стоят в классе информатики или дома на столе. Они окружают нас повсюду: «притаились» в плеере, мобильном телефоне, фотоаппарате, в турникетах в метро и в школе, и даже в домашних бытовых приборах.

Для каких бы целей ни было предназначено устройство: полет ракеты на Марс, расчет зарплаты, выдача денег в банкомате, пропуск учеников в школу — надо «научить» его решать поставленную задачу, т. е. написать инструкции, детально разъясняющие, как действовать во всех возможных ситуациях. Деятельность по написанию таких инструкций (команд) для вычислительных машин или управляющих чем-либо устройств называется программированием, а сами инструкции — программами.

### Этапы решения задачи

Предположим, что перед нами стоит задача, для решения которой необходимо написать программу. Решение задачи разбивается на этапы.

1. **Постановка задачи.** Необходимо определить, каковы будут исходные данные (что подается «на вход») и каких результатов надо достичь (что получить «на выходе»). При решении учебных задач этот этап иногда может отсутствовать, так как исходные данные и конечные результаты определены в формулировке задания.
2. **Выбор метода решения и разработка алгоритма.** Это один из основных, главных этапов решения задачи. От правильности выбора метода и эффективности алгоритма зависят размер программы и ее быстродействие.
3. **Составление программы и ввод ее в память компьютера.** Часто именно этот этап неправильно называют программированием. На самом деле, составление программы — лишь некоторая часть решения задачи с помощью компьютера. Сейчас этот процесс принято называть **кодированием**.

4. **Отладка программы.** Созданная программа может содержать ошибки, допущенные как в процессе кодирования, так и на любом из предыдущих этапов (может быть, неправильно разработан алгоритм, неверно выбраны исходные данные и т. д.). Ошибки выявляются в процессе отладки и тестирования — выполнения программы с заранее подготовленными наборами исходных данных, для которых известен результат.
5. **Вычисление и обработка результатов.**

Остановимся подробнее на главном этапе — разработке алгоритма.

## Что такое алгоритм?

**Алгоритм** — это система правил, набор инструкций, позволяющий решить некоторую задачу, детально разработанное описание методов ее решения.

С алгоритмами мы сталкиваемся в математике (алгоритм сложения многозначных чисел в столбик), физике, химии, при изучении языков (правила правописания); они подстерегают нас и в повседневной жизни: алгоритм перехода улицы, правила пользования лифтом и др. Поваренная книга также является сборником алгоритмов для приготовления блюд.

***Задание.** Приведите еще примеры алгоритмов из математики, физики, химии, повседневной жизни.*

Любой алгоритм рассчитан на то, что его инструкции будет кто-то выполнять. Назовем этого кого-то «исполнителем» и будем иметь в виду, что наши инструкции должны быть понятны ему, он должен уметь выполнять действия, записанные в алгоритме.

Существуют разные способы представления алгоритмов: с помощью формул, схем, рисунков, также можно записать алгоритм в виде программы и, конечно же, просто словами на обычном (естественном) языке.

## Словесная формулировка алгоритма

Часто алгоритмы записываются в виде списка инструкций. Сформулируем таким образом алгоритм пользования лифтом.

1. Нажмите кнопку вызова.
2. Когда автоматические двери откроются, войдите в лифт.
3. Встаньте так, чтобы не мешать закрытию дверей.
4. Нажмите кнопку нужного этажа.
5. Когда лифт приедет на нужный этаж, дождитесь открывания дверей.
6. Выйдите из лифта.

В получившемся алгоритме все инструкции выполняются ровно один раз, последовательно друг за другом. Такие алгоритмы называются **линейными**.

Все ли учтено в нашем списке инструкций? Вдруг лифт сломался и никогда не придет к тому, кто его вызвал, — не указано, что делать в этом случае. А если лифт остановился между этажами? Все знают, что в этом случае нужно вызывать диспетчера. Включим эти инструкции в алгоритм.

Пункт 1 теперь будет выглядеть так.

1. Нажмите кнопку вызова. Если никакой реакции нет, значит, лифт сломан и вам придется идти пешком, если же все в порядке — выполняйте пункт 2.

А пункт 5 так:

5. Дождитесь остановки лифта. Если двери открылись, то выйдите из лифта, иначе нажмите кнопку вызова диспетчера и выполняйте его указания.

Отметим, что здесь мы использовали прием разработки алгоритмов, называемый «сверху вниз». При этом способе задача сначала разбивается на несколько подзадач, а потом некоторые из них, в свою очередь, могут быть разбиты на отдельные части.

Получившийся алгоритм уже не является линейным, в нем есть **разветвления**. В случае поломки лифта пункты 2–6 выполняться не будут, пункт 5 на самом деле состоит из двух частей, и в каждом случае выполняется одна из них.

## Блок-схема. Основные конструкции

Для алгоритма с разветвлениями словесная запись в виде перечня инструкций не всегда удобна, в этом случае часто применяется графическая запись алгоритма в виде блок-схемы: каждая инструкция помещается в блок (на чертеже — геометрическая фигура), блоки соединяются стрелочками, определяющими порядок выполнения инструкций. Этот способ представления алгоритмов достаточно популярен и общепринят, существует даже Государственный стандарт на изображение блок-схем. Мы не будем детально его изучать и слишком буквально ему следовать, используем некоторые конструкции, с помощью которых удобно графически представлять изучаемые нами алгоритмы.

Каждый алгоритм должен иметь начало (точка входа), для его обозначения будем использовать значок овал, а для обозначения конца (выхода) — значок овал с крестиком внутри. Каждый алгоритм имеет ровно один вход, а вот выходов может быть несколько (но никак не менее одного!), так как процесс решения задачи должен когда-то заканчиваться с некоторым результатом, причем в зависимости от условий результаты могут получаться разные.

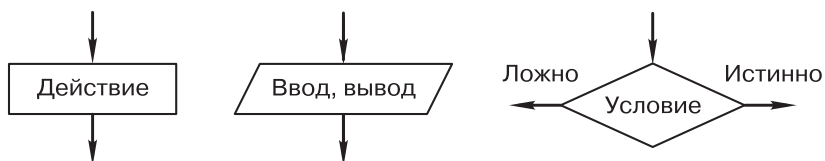
Рассмотрим наиболее часто используемые типы блоков (рис. 1.1). Сразу заметим, что в любой из этих блоков должен быть хотя бы один вход.

**Прямоугольник.** В нем приводится описание некоторых действий, которые необходимо выполнить. Из прямоугольника всегда выходит только одна стрелочка (т. е. следующая инструкция всегда четко определена).

**Параллелограмм.** В нем записываются данные, которые необходимо ввести или вывести (исходные или выходные данные). Из параллелограмма выходит тоже только одна стрелочка.

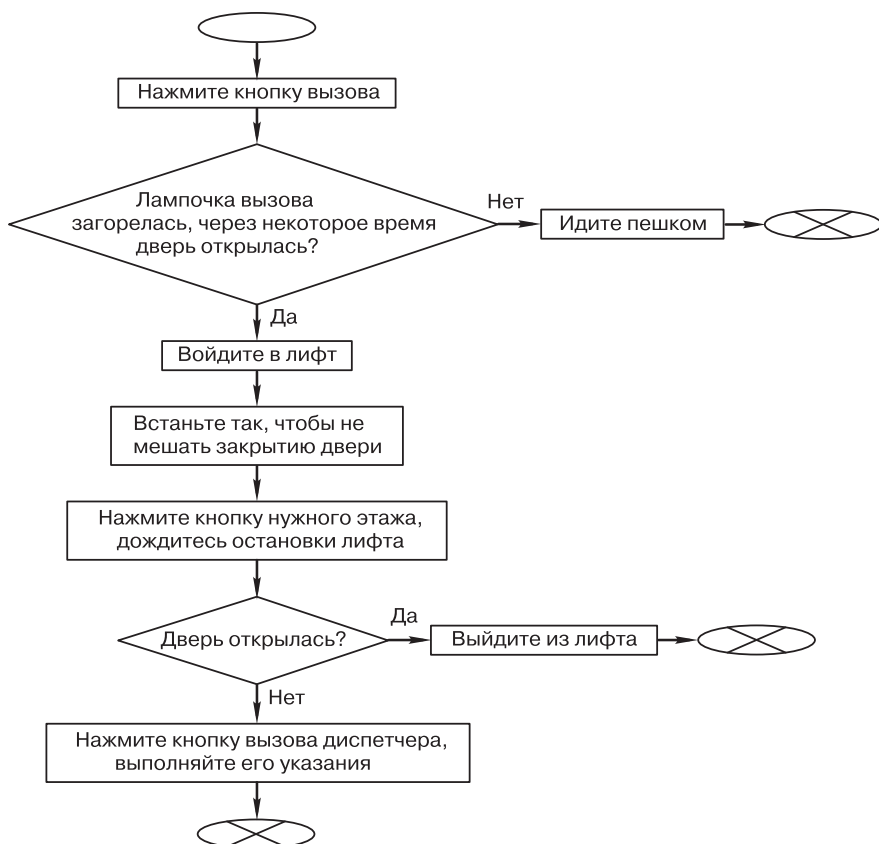


**Ромб.** Используется для обозначения ветвления. В ромбе размещается вопрос (условие), на который возможен ответ «да» или «нет» (забегая вперед, скажем, что такая конструкция называется «логическое выражение»). Из ромба должны выходить две стрелочки. Одна помечается словом «да» (или «верно», «истинно»), другая — словом «нет» (или «неверно», «ложно»). В зависимости от значения истинности помещенного в ромбик условия далее выполняется переход по одной из стрелочек.

**Рис. 1.1**

Есть и некоторые другие блоки, о которых мы расскажем позже.

Изобразим алгоритм пользования лифтом в виде блок-схемы (рис. 1.2).

**Рис. 1.2**

Обратите внимание, в этой схеме мы не использовали блоки ввода и вывода. Так получилось потому, что это алгоритм не для компьютера, а «бытовой». В программах, которые мы будем писать для компьютера, обычно (но не обязательно) будет присутствовать ввод данных и обязательно — вывод результатов (если в результате выполнения программы ничего не будет выведено — как же понять, что она сделала).

**Задание.** Понятно, что наш алгоритм неполон. Например, лифт может остановиться, не доехав до нужного этажа, если его кто-то вызвал. Продолжите разработку данного алгоритма. Подумайте, какие инструкции нуждаются в уточнении.

## Переменная. Присваивание

Составим алгоритм для решения следующей задачи. *Человек приобрел огород треугольной формы. Какой длины забор ему понадобится, чтобы огородить свой участок?*

Понятно, что надо измерить все стороны треугольника и найти его периметр, сложив полученные величины. Для записи формул в математике, физике обычно используют буквенные обозначения. Мы тоже ими воспользуемся, обозначив длины сторон треугольника  $a$ ,  $b$  и  $c$ , а периметр —  $P$ . Блок-схема алгоритма изображена на рис. 1.3.

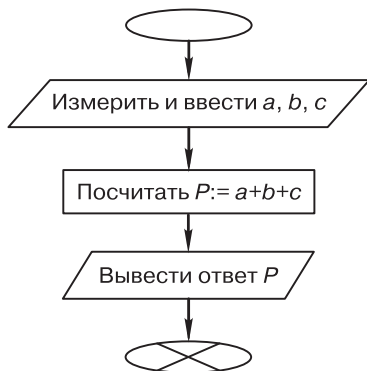


Рис. 1.3

Обозначенные буквами величины мы будем называть **переменными**. При разных начальных условиях (разных огородах) они будут принимать разные числовые значения, меняться.

Обратите внимание на запись формулы  $P := a + b + c$ . Здесь вместо принятого в математике знака « $=$ » использован знак, состоящий из двух символов « $:=$ ». Будем называть его **присваиванием**.

В математике знак равенства используется в двух случаях: когда надо проверить, равна ли правая часть левой (например, «если  $x = 0$ , то...»),

и в формулах, когда надо подсчитать значение правой части и положить переменную в левой части равной этому значению (т. е. присвоить ей это значение). Чтобы избежать двойного толкования этого знака, будем в первом случае использовать знак равенства « $=$ » и называть это действие сравнением, а во втором — знак присваивания « $:=$ », а действие — присваиванием переменной нового значения.

Этот значок произошел от такой стрелочки:  $\Leftarrow$ . В записи программ для первых компьютеров эта стрелочка показывала, что некоторое число пересылается в определенную ячейку, т. е. запись  $X \Leftarrow 5$  обозначала инструкцию: «число 5 переслать (положить) в ячейку, предназначенную для  $X$ ». Таким образом, в левой части операции присваивания всегда пишется переменная, которая должна получить новое значение, а в правой — само значение (оно может вычисляться, т. е. записываться в виде выражения).

С точки зрения компьютера **переменная** — это фрагмент оперативной памяти, в котором хранится значение. Компьютер обращается к этому участку памяти и записывает туда новое значение или считывает то, что там находится. Для наглядности память компьютера можно представить себе как большой шкаф с ящиками. Каждый ящик — это место для переменной. Для того чтобы можно было найти нужный ящик, наклеиваем на него табличку с именем. Теперь при необходимости туда можно положить значение или посмотреть, что лежит в ящике. И еще одна важная особенность: при обращении к переменной мы забираем из ящика не само значение, а его копию, т. е. само значение остается в ящике без изменения.

При выполнении действия  $P := a + b + c$  сначала вычисляется правая часть, затем результат вычисления помещается в ящик с именем (табличкой)  $P$ . Это и есть операция присваивания переменной значения. Она всегда работает справа налево.

Получившийся алгоритм — линейный. Займемся теперь нелинейными алгоритмами.

## Условие. Виды разветвлений

Простейшая задача с разветвлением, пожалуй, такая: из двух чисел выбрать наибольшее (договоримся, что, если числа равны, «наибольшим» будет любое из них). Блок-схема этого алгоритма представлена на рис. 1.4.

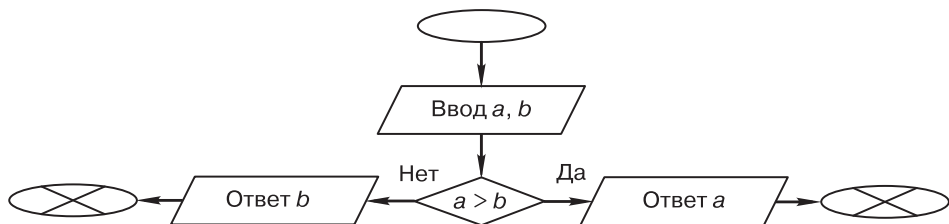


Рис. 1.4

**Задание.** Посмотрите, как усложняется алгоритм для трех чисел (составьте соответствующую блок-схему).

На рис. 1.5 вы видите, как выглядит блок проверки условия (разветвление) в общем случае. Официально он называется **полной условной конструкцией**.

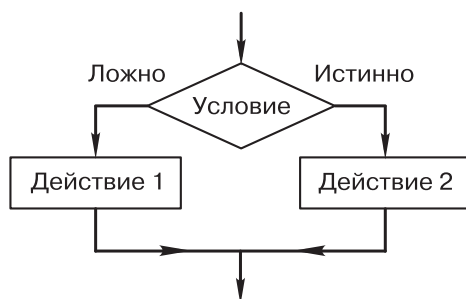


Рис. 1.5

В таком алгоритме выполняется одно из двух действий в зависимости от истинности условия. Однако часто бывают случаи, когда в подобной схеме присутствует только одно действие, которое надо реализовать при выполнении некоторого условия, а в противном случае делать ничего не надо. (Из жизни можно привести такой пример: «Если идет дождь, возьми зонт».)

На рис. 1.6 изображен частный случай разветвления — **неполная условная конструкция**, которую удобно называть **обходом**.

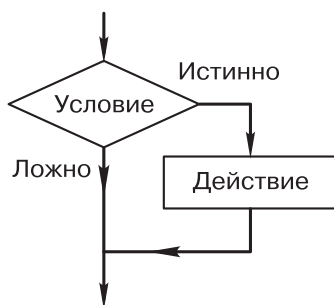


Рис. 1.6

Чтобы реализовать присваивание  $X := |X|$ , понадобится именно обход, так как значение  $X$  надо изменять только в том случае, когда значение  $X$  отрицательно (рис. 1.7):

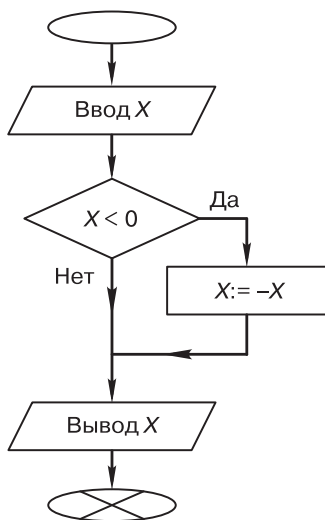


Рис. 1.7

**Пример 1.1.** Компьютеру сообщается значение  $X$ , а он печатает значение  $Y$ , которое находит по формуле:

$$Y = \begin{cases} X^2 & \text{при } X \leq 0; \\ \sqrt{X} & \text{при } X > 0. \end{cases}$$

Блок-схема решения представлена на рис. 1.8.

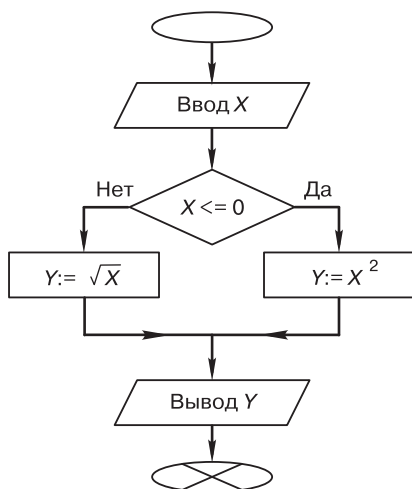


Рис. 1.8

**Пример 1.2.** Заданы 3 числа. Определить, можно ли построить треугольник с такими сторонами.

Чтобы определить, можно ли построить треугольник с заданными длинами сторон, существует несколько способов:

1. Наибольшая сторона должна быть меньше суммы двух других сторон.
2. Наименьшая сторона должна быть больше разности двух других сторон.
3. Каждая сторона должна быть меньше суммы двух других сторон.
4. Каждая сторона должна быть больше модуля разности двух других сторон.

Выбираем третий способ. Обратите внимание: если окажется, что какое-то неравенство не выполняется, то другие проверять уже не надо. Блок-схема решения задачи представлена на рис. 1.9.

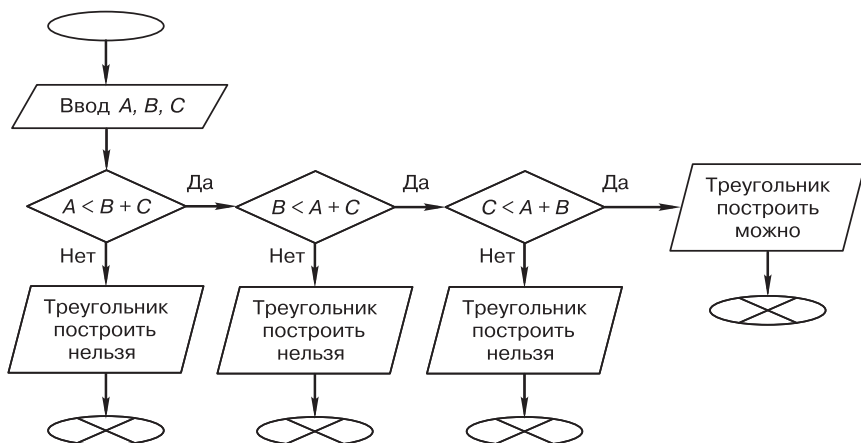


Рис. 1.9

В этой схеме у нас один и тот же блок — параллелограмм, в котором записано, что построение невозможно, — повторяется трижды. Можно ли сократить схему, нарисовав только один такой блок и подведя к нему три стрелочки? Этого делать не стоит: получившаяся схема уже не разделится на блоки, описанные выше (условные конструкции в ней нельзя будет считать ни полными, ни конструкциями обхода). Мы будем в наших схемах использовать только описанные выше блоки, так как потом научимся их «переводить» в программы на языке Паскаль.

### Задание

1. Решите задачу из примера 1.2 другими способами.
2. Дополните алгоритм. Пусть в случае, если треугольник построить можно, определяется его вид: является ли он равносторонним, равнобедренным, прямоугольным, тупоугольным, остроугольным.

## Цикл

Ранее мы успешно справились с алгоритмом определения периметра треугольника (в задаче про огород). Если бы вместо него в задаче был четырехугольник, наверное, мы ее решали бы так же. А если 10-угольник? К тому же, если сначала измерить все стороны и только после этого их суммировать, понадобится все измерения хранить в памяти. Удобнее измерять очередную сторону и прибавлять ее значение к уже накопленной сумме.

**Пример 1.3.** Найдём предложенным на рис. 1.10 способом периметр четырехугольника.

При таком решении задачи мы будем как бы накапливать значение периметра, постепенно добавляя к нему значения сторон. Значит, когда мы еще ничего не измерили (и не прибавили), периметр равен нулю.

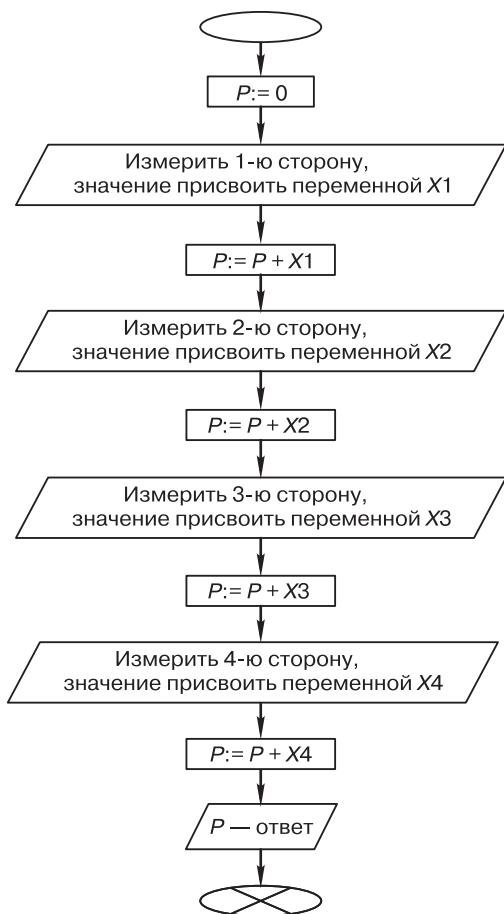


Рис. 1.10

Обратим внимание на инструкции вида  $P := P + X1$ . Если бы мы не договорились использовать знак «:=», а записывали эти инструкции со знаком «=», получилось бы правильное с точки зрения математики выражение только при  $X1 = 0$ . В этом и состоит отличие операции присваивания от операции сравнения. Мы сначала вычисляем значение выражения в правой части (используя уже известное значение  $P$ ), а потом значение  $P$  изменяем (она ведь у нас и называется «переменная»).

Задача решена совершенно правильно, но давайте посмотрим, нужно ли нам длину каждой стороны хранить в особой переменной — ведь она нам нужна только один раз в следующем операторе. Значит, можно длины всех сторон по очереди присваивать одной и той же переменной  $X$ . Перерисуйте блок-схему в соответствии с этой договоренностью и хорошенько посмотрите на нее. Все правильно, но как-то нехорошо, что практически одни и те же действия записаны 4 раза. А если бы пришлось искать периметр многоугольника, имеющего 100 сторон? Не хочется рисовать такой длинный алгоритм? В нашем алгоритме есть 4 повторяющихся фрагмента, которые различаются только номером измеряемой стороны. Давайте обозначим этот номер переменной  $K$ , а результат измерения длины стороны будем обозначать все время одинаково  $X$  (это можно сделать, так как значение длины стороны запоминать не надо, оно нужно только один раз, чтобы прибавить его к периметру). Получим фрагмент, представленный на рис. 1.11.



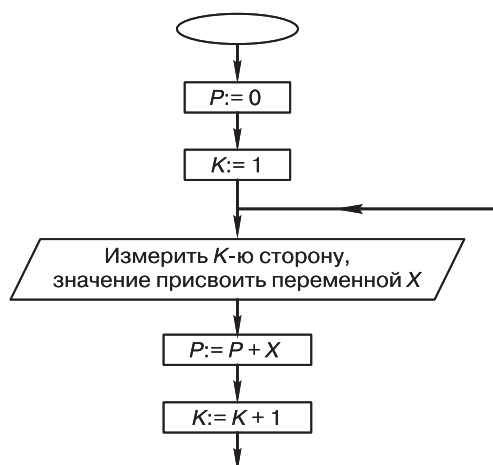
Рис. 1.11

Организуем работу так, чтобы этот фрагмент повторялся. При этом будем изменять значение  $K$  (рис. 1.12). Так как вначале нам нужна первая сторона, присвоим  $K$  единицу, а потом эту переменную будем увеличивать на 1.

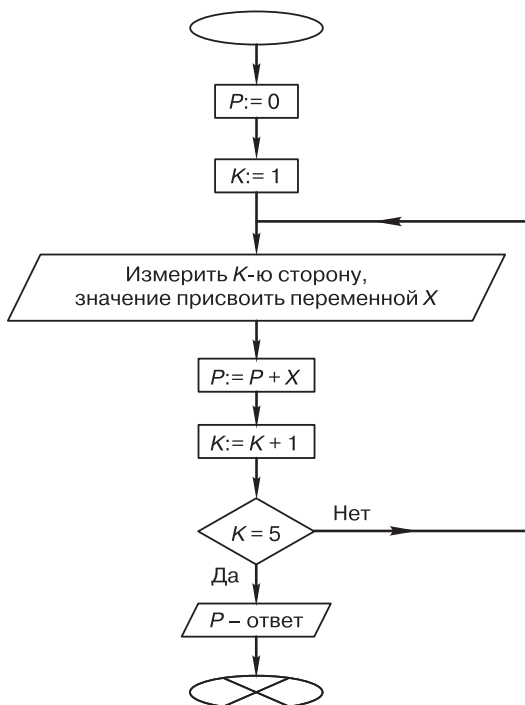
После изменения значения  $K$ , если мы направим стрелочку к нашему повторяющемуся фрагменту, будет измеряться уже вторая сторона многоугольника. Так можно измерить и сложить все стороны. Но где же написать ответ? Где конец нашего алгоритма? Из прямоугольника может выходить только одна стрелочка, и мы направили ее вверх. Откуда взять вторую стрелочку — к ответу?

Дело в том, что после добавления длины очередной стороны к периметру надо не всегда переходить к измерению следующей стороны многоугольника (иначе если мы будем делать это всегда, то «зациклимся», т. е. будем вечно ходить по многоугольнику и измерять его стороны бесконечное



**Рис. 1.12**

количество раз). Как только все стороны многоугольника измерены, вычисления надо прекратить. Как узнать, что четырехугольник «закончился»? Очень просто — мы ведь знаем, что у него всего 4 стороны, поэтому, если  $K$  стало равно 5, вычисления следует прекратить, так как пятой стороны у четырехугольника нет. Правильный алгоритм представлен на рис. 1.13.

**Рис. 1.13**

Заметим, что записанный таким образом алгоритм годится для вычисления периметра не только 4-угольника, а любого многоугольника. Для  $N$ -угольника надо просто число «5» в блоке  $K = 5$  заменить на  $N + 1$ .

**Вопрос.** Как изменить блок-схему, чтобы в этом блоке фигурировало не  $N + 1$ , а  $N$  (естественно, алгоритм должен оставаться правильным и эффективным)?

В этой задаче мы познакомились с понятием **цикл**. Слово это происходит от греческого κύκλος — круг, и если на нашей блок-схеме стрелочки рисовать изогнутыми, как раз круг в этом месте и получится.

Во многих задачах часто приходится одни и те же действия выполнять несколько раз подряд. Часть алгоритма, в которой группа действий записывается только один раз, а выполняется многократно, и называется **циклом**. В алгоритмах с циклами надо внимательно следить, чтобы цикл правильно и своевременно заканчивался. Обязательно должно присутствовать условие завершения цикла, причем оно должно быть достижимым (должны существовать такие значения переменных, при которых работа пойдет по стрелочке, выводящей из цикла).

В любом цикле присутствует действие, которое нужно повторять, и проверяется условие: продолжать повтор или закончить цикл. В зависимости от порядка выполнения этих этапов циклы можно подразделить на два типа: **с предусловием** и **с постусловием**. В цикле с предусловием сначала проверяется некоторое условие, потом, если надо, выполняются действия, после которых опять проверяется условие. В цикле с постусловием — наоборот, сначала выполняются действия, потом проверяется, надо ли их повторять (рис. 1.14).

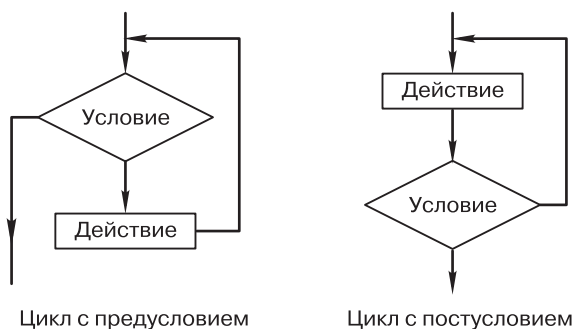


Рис. 1.14

Обратим внимание на существенное различие в этих циклах. Цикл с постусловием всегда выполняется хотя бы один раз, каким бы ни было условие, так как оно проверяется после действия. Цикл с предусловием может не выполниться ни разу, если таково условие.

Поскольку очень часто бывают задачи, в которых число повторений четко известно, выделяется еще один вид цикла: **с известным числом повторений** — «выполнить  $N$  раз». В блок-схемах договоримся изображать его так, как представлено на рис. 1.15.

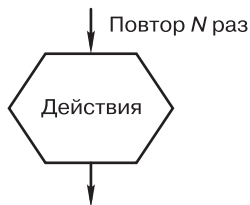


Рис. 1.15

Изображая так цикл, мы отмечаем, что в данном случае не важно, когда будет проверяться условие и как будет организована проверка, важно, чтобы количество повторений было именно такое, какое требуется. Цикл этого вида можно реализовать и как цикл с предусловием, и как цикл с постусловием. При этом надо завести какой-то счетчик (переменную, в которой будет храниться количество выполненных действий), присвоить ему начальное значение (например,  $K := 1$ ), а при каждом действии увеличивать  $K$  на 1 и сравнивать  $K$  и  $N$  — не пора ли закончить. Порядок этих операций при реализациях в циклах с предусловием и с постусловием будет разным.

**Задание.** Начертите блок-схему «повторить действие  $N$  раз», используя цикл с предусловием и с постусловием. Проверьте, правильно ли работает цикл при  $N = 0$ ,  $N = 1$ .

Примеры циклов:

- с постусловием: читай стихотворение, пока не выучишь его наизусть (хотя бы один раз придется стихотворение прочитать; если уже удалось его выучить — прекращаем это занятие, иначе читаем опять);
- с предусловием: пока у тебя хватает денег, покупай порцию мороженого (если с самого начала денег мало, можно остаться без мороженого вообще);
- с известным числом повторений: проехать на автобусе 5 остановок, выпить 3 стакана сока.

В рассмотренной выше задаче про многоугольник мы использовали цикл с постусловием, но ее можно решить, используя и алгоритм с предусловием — блок-схема почти не изменится, можно в этой задаче использовать и цикл с известным числом повторений (ведь число сторон многоугольника определено). *Сделайте это самостоятельно.*

С задачами такого типа вам придется часто сталкиваться. Строгая математическая формулировка этой задачи такова: найти сумму  $N$  чисел (ведь на самом деле все равно, что складывать: длины сторон, количество конфет, стоимость товаров).

## Вложенные циклы

*Пример 1.4. Робот умеет выполнять следующие команды:*

- *вперед* — пройти 1 шаг вперед (команда может быть выполнена, если путь свободен);
- *налево, направо* — поворот в соответствующую сторону на 90 градусов;
- *стена?* — отвечает «да», если перед ним стена, и «нет», если путь свободен.

Пусть робот находится в левом нижнем углу прямоугольной комнаты (угол будет так расположен, если ее нарисовать на бумаге — вид сверху), спиной к нижней (на плане) стене. Написать инструкцию, действуя по которой, он обойдет всю комнату по периметру.

Решение представлено на рис. 1.16.

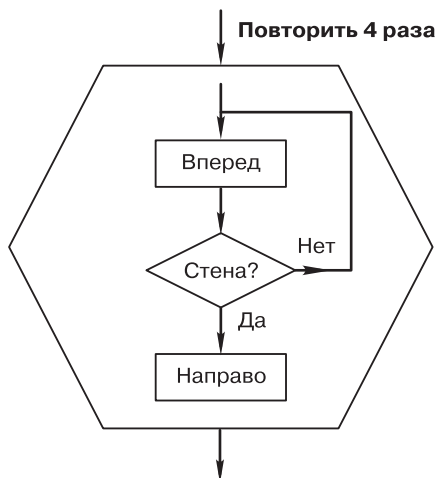


Рис. 1.16

Робот должен пройти вдоль четырех стен. Каждый прямолинейный отрезок — это путь вперед, пока он не достигнет стены. Используем цикл с известным числом повторений (в нашем случае оно равно 4).

Проход робота по прямой от стены до стены организован в виде цикла с постусловием. После очередного шага проверяем, достигли мы стены или нет.

Пусть робот идет по прямой от стены до стены 5 шагов. Тогда команда «направо» выполнится 4 раза (она находится только во внешнем цикле), а вот команда «вперед» — целых 20 раз, так как  $5 * 4 = 20$  (внутренний цикл выполняется 4 раза, и в каждом таком цикле команда выполняется 5 раз).

Такие алгоритмы называются алгоритмами с вложенными циклами. Вложенность бывает и больше, ее глубина ничем не ограничена. При программировании алгоритмов с вложенными циклами надо внимательно следить, какие действия и сколько раз должны выполняться и, соответственно, на каком уровне вложенности они находятся (чем больше уровень вложенности, тем большее количество раз выполняется действие).

Вложенными часто бывают и циклы с известным числом повторений.

## Массив

Попробуем описать в виде блок-схемы один из алгоритмов, известных нам еще в начальной школе.

***Пример 1.5.** Напишите блок-схему для сложения чисел в столбик.*

На первый взгляд может показаться, что такой алгоритм нужен только для человека, а компьютер умеет «сам» складывать числа быстро и правильно. На самом деле это не так. Во-первых, компьютер, складывая числа, пользуется примерно таким же алгоритмом (давайте посмотрим, как он это делает, — ведь интересно!), а во-вторых, подобные алгоритмы нужны для работы с очень большими числами.

Если необходимо сложить огромные целые числа (ну очень большие — из 30, из 50 цифр), компьютер с такими числами уже не может оперировать по обычным правилам. Такое число просто не удастся хранить в памяти «целиком» — не помещается. Приходится число разбивать на части — отдельные цифры и хранить в памяти в виде массива его цифр. А арифметические операции производить именно по этому алгоритму.

Только что было использовано слово «массив». Оно, наверное, знакомо вам из обычной жизни. Так говорят, когда много чего-то одинакового: «лесной массив» — много деревьев, «горный массив» — много гор.

В информатике этот термин используется в похожем смысле. Так называют некоторый упорядоченный набор однотипных данных. Их количество должно быть вполне известным, определенным, а элементы стоят в определенном порядке, их можно пронумеровать (на лесной массив уже не похоже — деревья не всегда пронумерованы, да и точное количество их известно только в ботанических садах).

Все данные, весь массив, имеют только одно, общее название, а чтобы как-то назвать один из элементов, надо использовать общее имя (имя массива) и номер элемента в массиве.

Таким образом, список 9а класса, где под номерами перечислены фамилии учеников, с точки зрения информатики, можно считать массивом, если обращаться к каждому ученику вот таким хитрым образом: «ученик из класса 9а номер такой-то».

Для однотипных данных заводят массив, если для решения задачи требуется хранение всех данных в памяти (например, надо использовать данные несколько раз).

Итак, имеется 2 числа. Цифры каждого из них поместим в массивы  $A$  и  $B$ . Так как при сложении в столбик мы работаем справа налево, пронумеруем цифры в числах именно таким образом. У последней цифры будет номер 1, у предпоследней — 2 и т. д.

Предположим, что в каждом числе  $N - 1$  цифр. Если это не так, можно всегда дополнить число слева незначащими нулями. Цифры первого числа будут обозначаться справа налево:  $a_1, a_2, \dots, a_{n-1}$ ; соответственно, цифры второго числа  $b_1, b_2, b_3, \dots, b_{n-1}$ . А результат будет  $c_1, c_2, \dots, c_n$ . Почему для результата мы отвели больше места? В сумме может быть на одну цифру больше, чем в слагаемых.

Чтобы все массивы были одной длины, добавим цифры  $a_n$  и  $b_n$ , сделаем их равными нулю (при работе на компьютере можно числа вводить по цифрам, начиная с конца, дополняя нулями).

Рассмотрим блок-схему этого алгоритма (рис. 1.17). Работу начинаем с самой правой цифры, она у нас имеет номер 1. Цифры надо сложить. А вот можно ли сразу считать, что получившаяся сумма и есть цифра результата  $c_1$ ? Нет, ведь может быть, что в сумме получилось число, большее 9. Тогда надо от суммы отнять число 10 (результатом и будет нужная цифра), и единицу «запомнить в уме» для того, чтобы прибавить к сумме следующих цифр, сделать перенос в следующий разряд. Таким образом, у нас появляется новая переменная, в которой будем хранить значение этого переноса (0 или 1). Назовем ее  $U$ . И складывать будем теперь не только 2 очередные цифры, но еще и прибавлять к ним  $U$  (пусть оно в некоторых случаях будет равно 0). Кстати, чтобы со всеми цифрами работать одинаково, складывая первые цифры, тоже прибавим к ним  $U$ , равное 0.

Эти операции надо проделать со всеми цифрами, от 1-й до  $n$ -й. Можно использовать цикл с известным числом повторений. В цикле мы опишем, как будем действовать с числами под номером  $I$ , а этот счетчик  $I$  будет меняться от 1 до  $N$ .

Последние (самые левые) цифры наших чисел-слагаемых равны нулю, но, складывая их и прибавляя то, что «в уме», мы можем получить самую левую цифру результата равной 1.

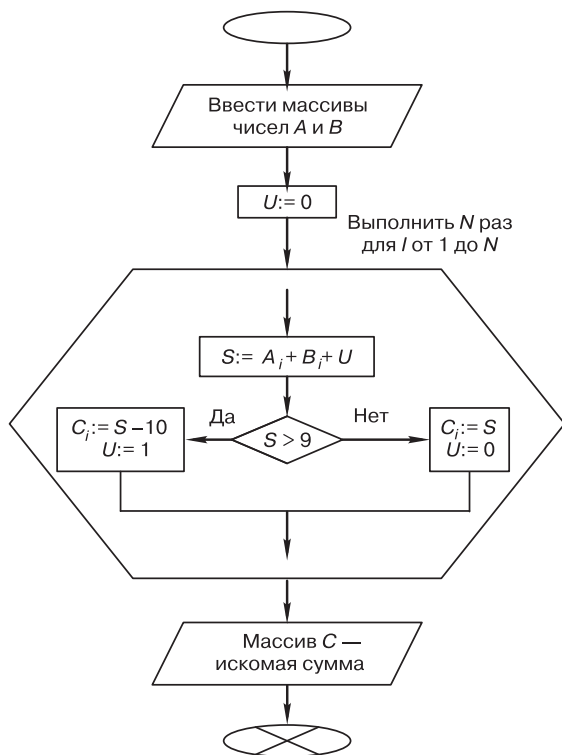


Рис. 1.17

А теперь подумаем, обязательно ли в нашей задаче использование массивов, нельзя ли обойтись без них? Можно, но тогда пришлось бы вводить и выводить числа не естественным образом (сначала ввести первое число, потом второе, затем посчитать и вывести результат), а довольно странно: сначала ввести правые цифры обоих чисел, подсчитать и вывести самую правую цифру результата, потом ввести две следующие цифры и т. д. Это не слишком удобно. Вот мы и воспользовались массивом. К тому же, если результат не записывать в массив  $C$ , а сразу печатать, напечатается число в «зеркальном» виде, ведь мы получаем цифры справа налево.

## Подпрограммы

**Пример 1.6.** Пусть наш робот (из примера 1.4) выполняет работу пылесоса. Ему надо аккуратно обойти всю комнату, не пропустив ни единого сантиметра, чтобы вычистить ее. Начальная позиция — та же, что и в примере 1.4.

Как обойти комнату? Можно, конечно, хаотично двигаться в разных направлениях, но тогда может оказаться, что какие-то участки робот почистит несколько раз, а в каких-то не побывает. Надо выработать некую систему обхода. Их существует множество. Можно идти по спирали, постепенно approaching к середине комнаты, а можно двигаться так называемым «ходом быка» (рис. 1.18), обходя комнату полосками (именно так двигается по полю бык или — кому как привычнее — трактор, когда вспахивает поле).

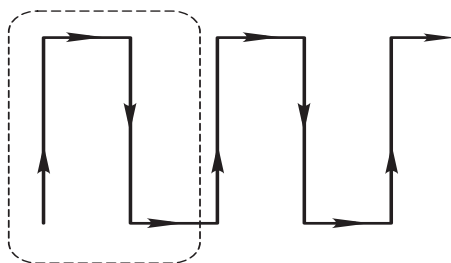


Рис. 1.18

Траектория состоит из следующих частей:

- путь «вверх» до стены — это цикл, похожий на внутренний в задаче 1.4;
- поворот направо, 1 шаг, еще раз направо;
- путь вниз до стены (аналогично первому циклу);
- поворот налево, 1 шаг, еще раз налево.

А далее опять повторяется все сначала, так как положение робота оказывается аналогичным положению в самом начале движения — около нижней стены спиной к ней. На рисунке часть пути, которая повторяется несколько раз, обведена. Вот только когда надо будет остановиться? Когда дойдем до правой стены. Обнаружить это мы сможем после поворота направо или налево, когда некуда будет сделать 1 шаг.

Можно ли в каком-нибудь случае здесь использовать цикл с известным числом повторений? Нет, мы не знаем размеров комнаты.

Получаем блок-схему, изображенную на рис. 1.19. Рассмотрим ее внимательно. В ней есть два совершенно одинаковых фрагмента, описывающих движение робота по прямой (вниз или вверх на рисунке) до стены. «Свернуть» эти два куса в один с помощью оператора цикла не получается, потому что между ними есть неповторяющийся кусок. В таких случаях бывает удобно этой повторяющейся части блок-схемы дать название (в нашем случае, например, «двигаться до стены») и заменить ее одним блоком



(с этим названием). Содержимое самого блока можно нарисовать отдельно (фрагмент блок-схемы «двигаться до стены» — рис. 1.20). Этим действием мы увеличиваем возможности, «умения» нашего робота. Теперь кроме элементарных команд он может выполнить и вот такую сложную.

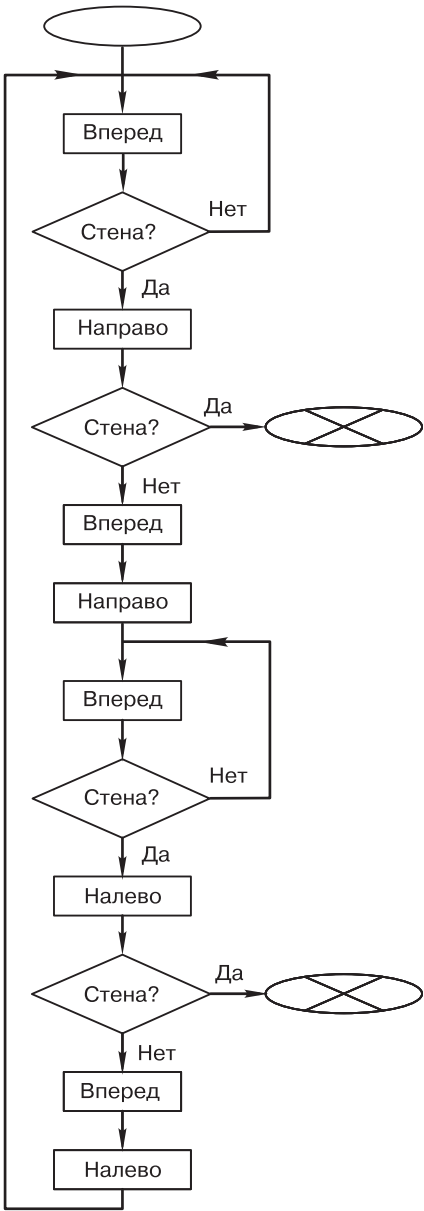


Рис. 1.19

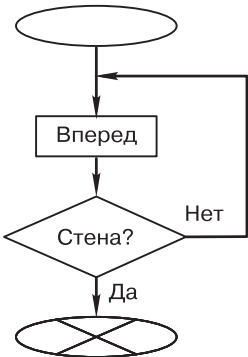


Рис. 1.20

Получившаяся после замены блок-схема (рис. 1.21) короче и понятнее, правда, в ней появился новый блок.

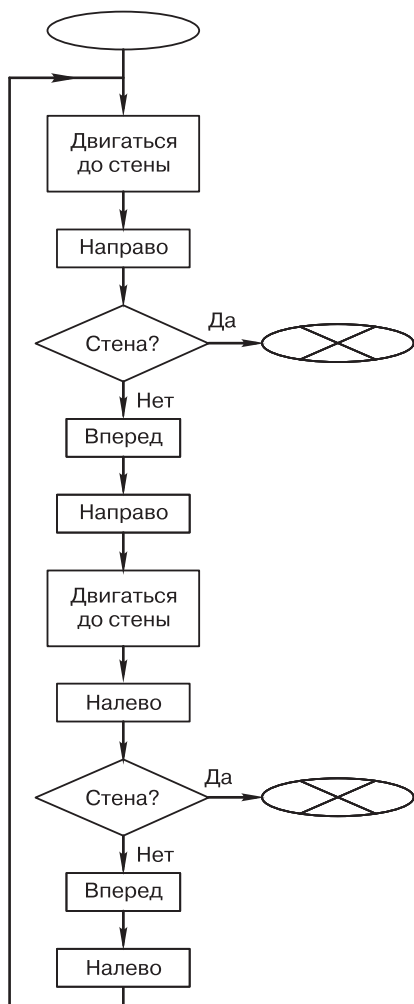


Рис. 1.21

Обучение исполнителя новым действиям, а потом использование таких более сложных действий при составлении алгоритмов часто применяется в программировании. При написании программ на Паскале также можно использовать этот прием: новые умения (команды, операторы) можно описывать в виде самостоятельных частей программы — **подпрограмм**, так называемых **процедур**.

Это слово знакомо вам из повседневной жизни. Наверняка вы слышали фразы «Больному назначены водные процедуры», «Законодательством предусмотрен ряд юридических процедур для обеспечения ...». **Процедурой** называют некоторый набор последовательных действий, этим действиям дают общее название.

Часто подпрограммы, написанные для одних задач, используют в других задачах. Например, в нашем случае очень легко будет написать блок-схему для программирования робота, который должен 10 раз пройти туда-обратно вдоль одной из стен (робот-часовой). А если еще написать процедуру «поворот на 180°», эта задача станет совершенно элементарной.

## Тестирование

*Пример 1.7. Обратная задача. Задана блок-схема (рис. 1.22). Какую задачу она решает?*

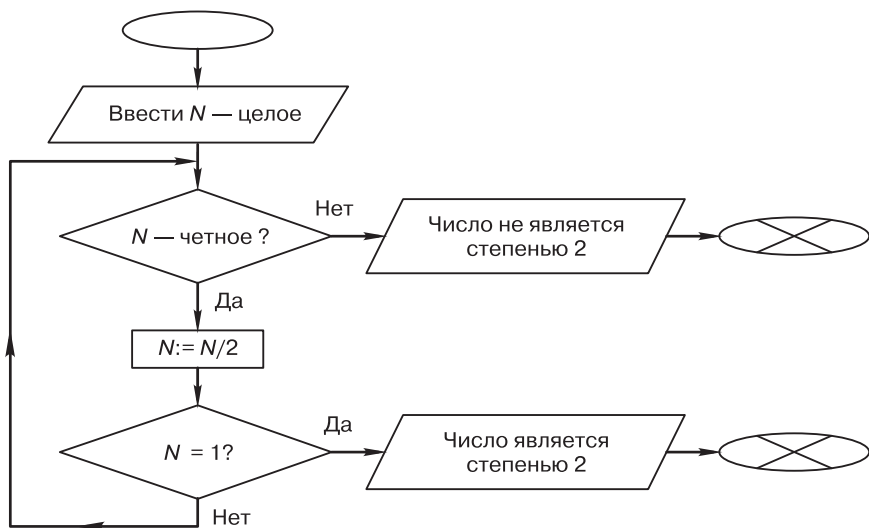


Рис. 1.22

Из ответов, которые предполагаются в этой блок-схеме, ясно, что алгоритм пытается выяснить, является ли заданное число степенью двойки. Как он это делает? Если число нечетное, ответ понятен сразу — нечетное число степенью двойки быть не может. Четное число мы будем делить на 2 до тех пор, либо пока не получим единицу (это означает, что число является степенью двойки), либо пока не получится другое нечетное число (а этот случай мы уже рассмотрели).

Проверим, правильно ли работает алгоритм. Как уже говорилось выше, этот процесс называется тестированием. Надо взять входные данные, для которых легко посчитать ответ, и выполнить алгоритм «вручную». Если ответы получаются правильные, есть надежда, что в алгоритме ошибок нет. Почему только «есть надежда»? Да потому, что работу алгоритма для всех наборов входных данных проверить обычно невозможно, а подобрать данные, с помощью которых можно протестировать алгоритм «со всех сторон», — задача достаточно сложная.

Проверим работу нашего алгоритма для  $N = 5$ . Работа завершается на первом же блоке, ответ правильный.

Теперь возьмем  $N = 4$ . Будет сделано действие  $N := 4 / 2$ , потом  $N := 2 / 1$ , работа завершится, ответ правильный. Понятно, что для всех остальных четных чисел, являющихся степенью двойки, алгоритм будет работать аналогичным образом, только шагов будет больше. Осталось проверить для четного числа, которое степенью 2 не является. Возьмем  $N = 6$ . Сначала будет выполнено действие  $N := 6 / 2$ , а потом, так как число получится нечетное, работа завершится.

Все ли случаи мы рассмотрели? Давайте подумаем, какие входные данные допустимы для нашего алгоритма, какова его область применимости? Проверять на четность/нечетность можно только целые числа. Имеет смысл обсуждать, является ли число степенью 2, только если оно положительное. Таким образом, получаем целые  $N \geq 1$ . При тестировании алгоритмов всегда надо проверять «граничные» значения. Верхней границы у нас не существует (да и понятно, что работа будет аналогичная), а вот правильно ли будет работать алгоритм при  $N = 1$ ? Это нечетное число, поэтому сразу же получится ответ, что 1 степенью 2 не является, что неверно,  $2^0 = 1$ .

Как исправить алгоритм? Можно «в лоб»: добавить в самое начало проверку  $N = 1$ . Если равно — заканчивать с ответом «является», а если нет, действовать, как и раньше. Получается не очень красиво: в блок-схеме две одинаковых проверки  $N = 1$ . *Подумайте, как исправить блок-схему, чтобы эта проверка выполнялась только один раз.*

**Задание.** Решите ту же задачу, но когда число является степенью двойки, надо найти и показатель этой степени.

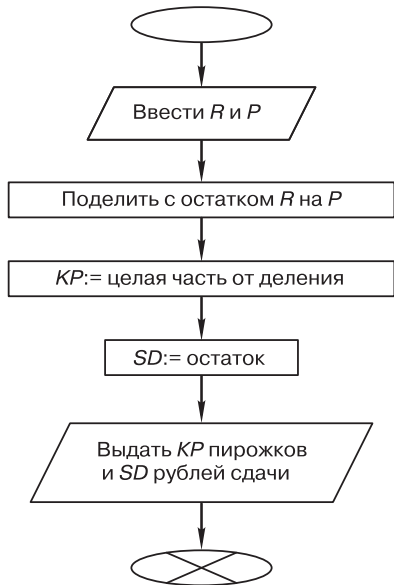
## Исполнитель алгоритма

Одно из свойств алгоритма — понятность для исполнителя. Из-за этого алгоритмы решения одной и той же задачи для разных исполнителей могут различаться. Например, когда мы объясняем, как куда-то доехать, алгоритмы будут разными для человека, который поедет на машине, и для того, кто воспользуется общественным транспортом.

Рассмотрим это свойство на примере следующей задачи.

**Пример 1.8.** Автомат продает пирожки. Цена одного пирожка  $P$  рублей, человек опускает в автомат  $R$  рублей (оба числа целые). Написать алгоритм, который выдает человеку максимальное количество пирожков, возможное за эти деньги, и полагающуюся сдачу.

Если в автомате имеется калькулятор, умеющий выполнять операцию «деление с остатком», алгоритм получается линейным и выглядит очень просто (обозначим  $KP$  — количество пирожков, которое должен выдать автомат покупателю,  $SD$  — сдача) — рис. 1.23.



**Рис. 1.23**

Можно ли создать автомат подешевле, без такого сложного калькулятора, который сможет справиться с этой задачей? Можно, но алгоритм будет совсем другой.

Автомат должен оценить, хватит ли денег хотя бы на один пирожок. Если нет — вернуть все деньги (сдача без пирожков), если да — выдать один пирожок, из поданных денег вычесть его стоимость, и повторить те же действия. При этом автомату для правильной работы считать количество пирожков  $KP$  совершенно не обязательно, мы же посчитаем, не забыв придать ему начальное значение 0 (рис. 1.24).

Что же у нас получилось? Количество пирожков ( $KP$ ) — это целая часть от деления  $R$  (столько дали денег) на  $P$  (стоимость пирожка), а сдача (полученное значение  $R$ ) — остаток от деления. Таким образом, мы реализовали операции деления нацело и нахождения остатка с помощью вычитания, «научили» нашего исполнителя выполнять действия, которые в него не встроены. Для другого исполнителя у нас получился совсем другой алгоритм.

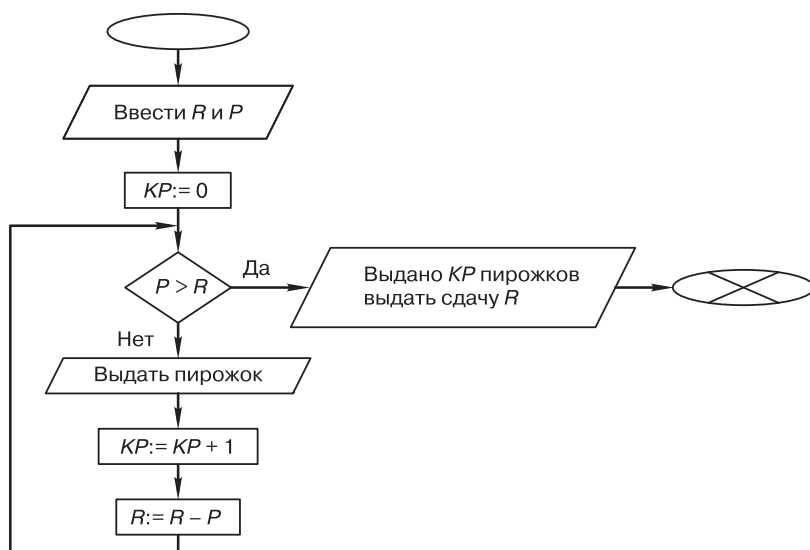


Рис. 1.24

Для создания грамотного алгоритма мы должны ориентироваться на его будущего исполнителя, знать, что он «умеет», какие действия ему доступны. Так как мы собираемся решать задачи на языке Паскаль, для построения правильных алгоритмов надо знать его возможности.

## Оптимальный алгоритм. Сложность алгоритма

Для решения одной и той же задачи можно придумать разные алгоритмы не только из-за того, что могут быть разные исполнители. Большинство задач можно решить разными способами даже в расчете на одного исполнителя.

**Пример 1.9.** Известен первый член арифметической прогрессии ( $A_1$ ) и ее разность ( $d$ ). Найти сумму первых 10 членов.

По определению арифметической прогрессии, каждый следующий член отличается от предыдущего на разность, т. е. второй член можно найти по формуле  $A_2 = A_1 + d$ , третий —  $A_3 = A_2 + d$ . Действуя таким образом, можно последовательно найти все члены прогрессии вплоть до 10, а потом сложить получившиеся 10 чисел. Задача решена.

Можно ли ее решить другим способом? Можно, если вспомнить, что существует формула суммы  $N$  первых членов геометрической прогрессии:  $S = (2 * A_1 + d * (N - 1)) * N/2$ , по ней, подставив нужные значения, можно сразу посчитать ответ.

Как определить, какой алгоритм лучше? Для человека, возможно, первый (особенно если он забыл формулу), а вот для компьютера, может быть, второй, так как вычислений здесь меньше, а то, что они чуть «труднее», компьютеру не страшно.

А как в общем случае из нескольких алгоритмов выбрать наилучший, наиболее оптимальный? Тот, который короче, быстрее, красивее, понятнее? Увы, это бывает достаточно сложно: «хороший» по одним параметрам алгоритм может оказаться «плохим» по другим. Скажем сразу, не всегда алгоритм, плохо понятный человеку, сложный для него, является таковым для компьютера. Компьютер не «запутается» в сложной логической структуре программы, не «испугается» громоздких вычислений. Бывает, что алгоритм, кажущийся «коротким», на самом деле выполняется дольше, чем «длинный». Это может произойти из-за того, что цикл только записывается компактно, а выполняется долго. Существует понятие **вычислительной сложности алгоритма**. Это количество элементарных действий, которое выполняется в процессе решения задачи. Так, в нашем случае (задача про сумму первых членов прогрессии), если за элементарное действие принять выполнение одной арифметической операции, вычислительная сложность нахождения одного члена прогрессии — 1, всех десяти — 9 (первый нам известен), а для определения вычислительной сложности всего алгоритма (первый способ) надо прибавить еще 9 (нахождение суммы). Итого 18. При втором способе составления алгоритма вычислительная сложность равна 6.

Также важным критерием оценки качества алгоритма является объем необходимой ему памяти. Даже в нашем примере количество переменных, которое потребуется использовать для решения задачи разными способами, — разное. В первом случае нужно хранить в памяти все 10 первых членов прогрессии, во втором — только первый и разность.

Понятно, если наиболее «быстрый» алгоритм является еще и самым «красивым», компактным и т. п., выбирают именно его, однако чаще параметры оптимизации задаются, исходя из условия задачи. Например, при написании каких-то программ надо стараться, чтобы использовалось как можно меньше памяти, при составлении других — обращать внимание на время работы программы.

Конечно, мы пока будем писать только небольшие, учебные программы, но это не значит, что не следует обращать внимание на эффективность алгоритма. Если существует несколько способов решения задачи, нужно выбирать наиболее оптимальный алгоритм.

## Задачи 1.1—1.26. Составление алгоритмов

Для решения задач надо составить алгоритмы на русском языке в виде пошаговой инструкции, либо в виде блок-схемы.

- 1.1. Петя живет в 16-этажном доме, в котором 10 подъездов, в квартире № 361. На каждом этаже по 4 квартиры. Коля идет в гости к Пете, но забыл, в каком подъезде и на каком этаже живет Петя. Помогите Коле.

- 1.2. С начала суток прошло  $n$  секунд. Определить, сколько полных часов прошло с начала суток, сколько полных минут прошло с начала очередного часа, сколько секунд прошло с начала очередной минуты.
- 1.3. Даны два числа (например,  $a = 3$ ,  $b = 9$ ). Поменять значения переменных ( $a = 9$ ,  $b = 3$ ):
- а) используя дополнительную переменную;
  - б) без использования дополнительной переменной.
- 1.4. Записать в порядке возрастания три числа. Постарайтесь сделать алгоритм наиболее эффективным (чтобы в нем было как можно меньше сравнений). В каждом блоке записывайте только одно сравнение.
- 1.5. Вычислить значение функции:

$$Y = \begin{cases} -1, & \text{если } X < 0; \\ 0, & \text{если } X = 0; \\ 1, & \text{если } X > 0. \end{cases}$$

- 1.6. Задана функция  $F(x) = \frac{\sqrt{x+2}}{x*(x-1)}$ . Написать блок-схему: вводится значение  $X$ , на выходе — входит ли оно в область определения функции (в каждом блоке можно проверять только одно условие).
- 1.7. Перед вами — треугольник. Можно измерять любые стороны и углы. Составить инструкции для определения его вида: является ли он прямоугольным, остроугольным, тупоугольным, равносторонним, равнобедренным.
- 1.8. Перед вами — четырехугольник. Можно измерять любые стороны и углы. Составить инструкции для определения его вида: является ли он прямоугольником, параллелограммом, ромбом, квадратом (написать надо что-то одно: для квадрата, например, не надо сообщать, что это еще и ромб, и прямоугольник).
- 1.9. Написать блок-схему для вычисления значения функции, изображенной на графике (рис. 1.25).

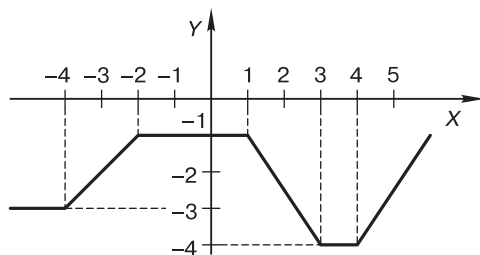


Рис. 1.25



- 1.10. В начале рабочего дня в помещении включают автомат-кондиционер. Составить программу, по которой он будет через некоторые промежутки времени измерять температуру в помещении и включать-выключать обогрев или охлаждение. При наступлении времени «окончание рабочего дня» он автоматически выключается. Температура должна находиться в пределах  $[T_1, T_2]$ .
- 1.11. Придумать и составить инструкцию охраннику школы: в какой последовательности и что он должен проверять (наличие пропуска, соответствие фотографии, не просрочен ли пропуск, есть ли сменная обувь и т. п.) и как реагировать (вызвать милицию, отправить домой, поругать, но пропустить и т. п.).
- 1.12. Водитель автобуса, в котором  $K$  мест, продает билеты и по одному пропускает пассажиров в автобус. Он должен уехать либо когда все желающие войдут в автобус, либо когда все места будут заняты. Составить программу его действий.
- 1.13. В рабочем помещении установлен автомат, включающий и выключающий освещение. Он должен обеспечить следующее: в нерабочее время свет должен быть выключен; в рабочее время свет должен включаться, если без него становится темно, и выключаться, если становится светло (освещенность на улице и в комнате автомат измерять умеет).
- 1.14. Задано  $N \geq 0$ . Напечатать степени 2 от 0 до  $N$  включительно. (Можно складывать, сравнивать, умножать; операции «возведение в степень» нет.)
- 1.15. Задаются два числа. Выяснить, делится ли первое на второе нацело, не используя операции деления.
- 1.16. Промышленный манипулятор находится около непрерывно движущейся ленты с деталями. Он умеет отличать бракованные детали от качественных. Манипулятор берет хорошие детали и складывает их в ящик, в который помещается  $K$  деталей. Когда ящик заполнится, манипулятор должен остановить конвейер и прекратить работу. Прекратить работу он должен также, если ему попалось  $N$  бракованных деталей подряд.
- 1.17. Автомат умеет считать, сравнивать числа и определять, какого цвета кнопка нажата (имеются кнопки четырех цветов). Он зажигает красную лампочку, если 5 раз нажата красная кнопка, синюю лампочку, если 3 раза нажата синяя, и включает сирену, если было 30 нажатий кнопок, но ни красную, ни синюю так ни разу и не нажали.
- 1.18. Некоторая фирма решила провести статистические исследования покупателей своих товаров. Она наняла человека, который у каждого купившего товар будет узнавать его возраст. Составьте алгоритм, следуя которому (и делая как можно меньше записей) можно определить:
  - 1.18.1. Средний возраст покупателей.

- 1.18.2. Средний возраст покупателей-женщин.
- 1.18.3. самого старшего покупателя.
- 1.19. Спортсмен-стрелок готовится к соревнованиям, стреляет по мишени. Ему в этом помогает специальное устройство. Оно командует «выстрел», после каждого выстрела сообщает, сколько он очков за этот выстрел получает. Составить следующие программы:
  - 1.19.1. Устройство предлагает спортсмену продолжать тренировки до тех пор, пока он не наберет  $N$  очков. По окончании тренировок оно сообщает ему, сколько в среднем очков «стоит» каждый его выстрел (набранная сумма делится на количество выстрелов).
  - 1.19.2. Устройство предлагает продолжать тренировку, пока стрелок не сделает 10 хороших (более  $M$  очков) выстрелов подряд.
  - 1.19.3. Устройство считает, что можно прекратить тренировку, только если в очередной серии из 10 выстрелов спортсмен ни разу не промахнулся (за каждый выстрел получал больше 0) и набрал за все выстрелы вместе не менее  $K$  очков. Если эта цель не достигнута, предлагается сделать еще серию из 10 выстрелов.
  - 1.19.4. На все предложенные стрелку выше программы накладывается ограничение — нельзя сделать более 9999 выстрелов. Если за время такой долгой тренировки стрелок так и не достиг нужных результатов, устройство предлагает ему переквалифицироваться в штангиста.
- 1.20. Устройство умеет обрабатывать некоторый вид социальных карт. Оно умеет считывать с карты фамилию, пол и возраст. На ввод подается стопка карточек.
  - 1.20.1. Подсчитать количество мужчин и женщин.
  - 1.20.2. Определить, есть ли в стопке карточки детей (до 16 лет).
  - 1.20.3. Подсчитать количество и напечатать фамилии мужчин призывного возраста.
  - 1.20.4. Найти самого старого мужчину (одного такого, если их несколько). Учесть возможность того, что в стопке может вообще не быть карточек мужчин! Напечатать фамилию и возраст (или сообщение, что такого нет).
  - 1.20.5. Такая же задача, но, если самых старых мужчин оказалось несколько, надо напечатать фамилии всех.
- 1.21. Даны 3 целых положительных числа  $A < B < C$ . Можно ли получить  $C$ , прибавив к  $A$  сколько-нибудь раз  $B$ ?
- 1.22. Задано некоторое целое число  $N > 2$ . Определить, является ли оно простым (т. е. делится нацело только на 1 и само на себя).

- 1.23. Имеются гири 1, 2, 4, 8, 16 кг — по одной штуке. Определить с точностью до 1 кг вес положенного на весы груза (он весит не более 32 кг) за минимальное количество взвешиваний. (По внешнему виду делать предположения о весе груза нельзя.)
- 1.24. Робот из примеров 1.4, 1.6.
- 1.24.1. Определить, есть ли рядом с роботом стена, т. е. может ли он увидеть стену, лишь поворачиваясь в какую-либо сторону, но не делая ни одного шага.
  - 1.24.2. Робот находится где-то в комнате (точное положение и куда смотрит, неизвестно). Робот должен дойти до любой стены и встать спиной к ней. Если он уже стоит рядом со стеной, идти никуда не надо, надо только повернуться. В задаче можно использовать уже предварительно приобретенные роботом умения (определить, есть ли рядом стена, пройти всю комнату до стены).
  - 1.24.3. Положение робота неизвестно, смотрит «вверх» (на плане). Робот должен дойти до левого нижнего угла. (Опять же полезно использовать уже приобретенные роботом умения.)
  - 1.24.4. Робот находится где-то в комнате. Определить ее площадь (считать, что 1 шаг робота равен 1 м). Составить наиболее эффективный алгоритм.
- 1.25. Заданы 2 числа в виде массива — смотри пример 1.5 в тексте, первое больше второго. Найти их разность.
- 1.26. Заданы 2 числа в виде массива — смотри пример 1.5 в тексте. Сравнить их.

## Глава 2

# Первая программа на языке Паскаль

---

Пусть наша первая программа будет такой (это достаточно известная стандартная программа, используемая при изучении языков программирования):

```
Program FIRST;  
{Первая программа}  
Begin  
    Writeln('Здравствуй, мир!')  
End.
```

Заметим, что, печатая текст этой программы на компьютере, вы должны в точности повторить все, как это написано здесь, соблюдая разбиение на строки, ставя все значки: пробелы, скобочки и т. п. Впоследствии мы узнаем, что некоторые из них обязательны, другие же нужны «для красоты». Не забудьте точку в конце текста, обратите внимание, что русский текст в предпоследней строчке — в апострофах, на клавиатуре этот значок справа, на той же клавише, что и русская буква «э», в нижнем регистре.

## Понятие об алфавите языка

Первое, что мы узнаем — какие же буквы, символы, значки можно использовать в программе на Паскале. Ведь алфавит — конечный набор знаков, из которых строятся все конструкции, — является основой любого языка. Далее перечислены составляющие алфавита языка Паскаль:

- цифры (обычные десятичные арабские);
- латинские буквы, прописные и заглавные (будем их в дальнейшем называть маленькими и большими). К буквам еще относится символ «\_» (подстрочник или нижнее подчеркивание), так как он используется именно в качестве буквы. Во многих конструкциях языка маленькие и большие буквы не различаются;
- специальные символы.

К специальным символам относятся разделители (пробел, разные скобки, знаки препинания, апострофы), знаки арифметических операций, операций сравнения, например:

+   -   \*   /   =   '   .   :   ;   <   >   [   ]   (   )   {   }

Есть среди специальных символов и такие, которые при печати на клавиатуре обозначаются двумя значками (парные символы):

<>   <=   >=   :=   (\*   \*)   (.   .)

В языке они интерпретируются как один символ (писать символы, входящие в пару, надо обязательно подряд, в одной строке без пробела).

Особое место среди специальных символов занимают служебные (зарезервированные) слова. В языке имеется фиксированный небольшой набор (несколько десятков) слов, зарезервированных для определенных целей (для любых других целей их использовать запрещается). Для того чтобы отличить служебные слова от других наборов символов, в учебниках их принято выделять жирным шрифтом или курсивом, писать с большой буквы, на письме — подчеркивать. Служебные слова состоят из букв — символов алфавита, однако мы их выделяем как особые неделимые единицы алфавита языка.

Мы здесь не задаемся целью привести полностью весь набор символов, допустимых в языке, не предъявляем список служебных слов для заучивания — со всеми этими объектами мы познакомимся в процессе изучения материала.

Перечисленные символы входят в стандарт языка и используются для написания его конструкций. Однако для написания подсказок пользователю или некоторых конструкций в процессе работы программы этих символов бывает недостаточно. Для адаптации программы под пользователей разных стран в язык ввели четвертую группу символов, не входящую в стандарт языка. Это символы национального алфавита, они имеют особый статус, их использование ограничено.

В естественном языке из символов алфавита составляются слова, фразы. Так и в Паскале из символов алфавита строятся сложные конструкции: имена, константы, операторы и, в конце концов, целая программа.

Вернемся к нашей программе, посмотрим, какие элементы алфавита мы в ней использовали.

Слова **Program**, **Begin**, **End** относятся к служебным. Как видим, служебные слова «заимствованы» из английского языка, так что выучить их будет несложно. Большие и маленькие буквы в служебных словах Паскаль не различает.

В нашей программе использовано много разделителей, а вот никаких знаков операций нет. Во второй строке мы видим текст на русском языке. Это — комментарий, пояснение к программе. Комментарий представляет собой один из особых случаев, где можно использовать символы национального алфавита.

Также в нашей программе использованы имена и константы.

**Константы.** С точки зрения языка константы определяют изображения значений, запись которых подчиняется определенным правилам. Константы могут быть числовые, логические, символьные и строковые (об этом мы, естественно, поговорим подробнее позже).

В нашей программе присутствует одна константа-строка:

```
'Здравствуй, мир!'
```

Текст, составляющий содержимое константы-строки, заключен в символы-апострофы. В строковой константе (внутри апострофов) также можно использовать символы национального алфавита (и вообще любые символы, которые есть на клавиатуре). При этом большие и маленькие буквы различаются. Так, константы **'TIGR'**, **'tigr'** и **'Tigr'** — различные.

**Имена.** Объекты программы (переменные, функции, процедуры) должны иметь имена (идентификаторы). Иногда эти имена определены заранее, они называются стандартными (например, существует функция **sin**). В отличие от служебных слов назначение стандартного идентификатора может быть переопределено программистом (но делать это без особых причин во избежание путаницы не следует). Большинство же имен в своей программе придумывает программист.

В нашей программе именем является слово **FIRST**, мы его придумали, оно не определено языком. **Writeln** — тоже имя, но оно стандартное.

Имена (часто их называют идентификаторы) могут состоять из букв и цифр, начинаться должны с буквы, причем под буквами мы здесь подразумеваем только латинские буквы и значок «\_». Большие и маленькие буквы в именах язык не различает. Длина имени, вообще говоря, не ограничивается (хотя понятно, что превышать длину строки она никак не может), но различаются идентификаторы по некоторому количеству первых символов (в большинстве версий — 63). Например, имена

A, A234, X45G, Dlina, Summa\_Otr, Cos3 — правильные;

234A, СТОЛ, Summa-otr, cos(3) — неправильные;

KROKODIL, krokodil, Krokodil, KrokoDil — одинаковые.

## Принципы записи и «внешний вид» программы

По правилам языка программа на Паскале может записываться вся в одну строчку, может в столбик (в любом месте, где стоит разделитель, может быть и конец строки); буквы, как мы уже говорили, можно использовать большие и маленькие вперемешку. До и после некоторых разделителей (например, пробел, знаки арифметических операций, скобки) разрешается добавлять любое количество пробелов. Таким образом, одна и та же программа может выглядеть совершенно по-разному. Какой способ записи предпочесть? Такой, при котором программа будет наиболее понятна человеку (для компьютера способ записи не важен), удобна для чтения и редактирования. Поэтому в процессе знакомства с конструкциями языка мы выработаем правила их записи, от которых постараемся не отступать.

О больших и маленьких буквах можем договориться уже сейчас. Не рекомендуется их использование «вперемешку»: компьютер прекрасно поймет и слово **BeGiN**, и **bEGin**, человеку же придется напрягаться, чтобы уловить смысл такой программы.

Допустимы три способа записи программы:

- вся программа пишется либо маленькими, либо большими буквами;
- слова, используемые в программе, начинаются с большой буквы (все остальные буквы в слове — маленькие); этот способ использован в нашей программе;
- часть программы целиком пишется маленькими буквами, часть — большими; этот способ применяется, когда кусок программы хочется особо выделить (например, чтобы были видны последние добавления).

Об именах было сказано, что они представляют из себя любую последовательность букв и цифр, начинающуюся с буквы. Слово «любую» не надо понимать буквально и давать имена объектам программы «бесконтрольно». Не стоит также и впадать в обратную крайность и давать всем объектам почти одинаковые имена типа *A1*, *A2*, *A21* и т. п.

Имена должны быть не слишком длинными, легко читаться и отражать суть объекта. Например: *Summa*, *Kvo\_Otr*, *Factorial*. В математике многие величины часто по традиции обозначаются некоторыми «стандартными» именами. От этой традиции без особых причин нет оснований отступать и в Паскале. Например, стороны прямоугольного треугольника можно называть *A*, *B*, *C*, угол — *Alfa*, радиус — *R*. В программировании тоже есть свои «древние» традиции именования переменных. Например, счетчики, то есть переменные, служащие для пересчета чего-то, обычно называются буквами *I*, *J*, *K*, *L*, *M*, *N* или словами, начинающимися с этих букв.

## Глава 3

# Этапы подготовки программы. Паскаль-среда

---

Как записывать программу, мы выяснили, а вот как ее ввести в компьютер и, главное, что делать с ней дальше, как заставить компьютер ее выполнять? В этом нам поможет специальная программа, точнее, комплекс программ, называемый Паскаль-среда.

## Этапы подготовки программы

Ввести программу в память компьютера можно с помощью какого-либо текстового редактора (можно использовать, например, Блокнот, а вот Word — нельзя, он вставляет в текст дополнительные символы). Таким образом, на первом этапе программа представляет собой обычный текст, ее можно хранить в текстовом файле.

Назвать этот файл можно как угодно, однако для дальнейшей работы будет удобно, если при именовании файлов придерживаться некоторых принципов. Во-первых, будем давать файлам с программой на Паскале расширение PAS. Во-вторых, имена будем составлять только из латинских букв и цифр, и длина их не будет превышать 8 символов.

Но вся беда в том, что компьютер не понимает обычного текста, в том числе и текста на языке Паскаль, не понимает буквы, десятичные числа, а понимает только нули и единицы. Программа, которую может выполнять компьютер, должна быть написана на специальном машинном языке (его алфавит состоит только из нулей и единиц и больше ни единого символа, даже пробела, не содержит!). Кто же выполнит эту нудную работу, переведет программу с языка Паскаль на машинный язык? Не волнуйтесь, самим это делать не придется. Для этого существуют специальные программы-переводчики — трансляторы (от *translate* — переводить).

Есть разные методы трансляции. Транслятор с Паскаля кроме непосредственно перевода может выполнять еще некоторые действия, например, добавлять в программу нужные фрагменты, собирать ее из разных частей, поэтому он называется «компилятор» (от *compile* — собирать, составлять). Результатом работы компилятора является создание так называемого исполняемого файла (он имеет расширение EXE, от слова *execute* — выполнить), этот файл написан на машинном языке и может быть выполнен компьютером.



Таким образом, решение задачи на компьютере разбивается на следующие этапы:

1. Ввод текста программы в память компьютера. Создание соответствующего текстового файла.
2. Перевод программы на машинный язык, компиляция, создание EXE-файла.
3. Выполнение EXE-файла, получение результатов.

В разделе 1 мы говорили, что важной частью программирования является отладка программы, в процессе ее выполнения все эти этапы приходится выполнять несколько раз, только ввод текста заменяется его редактированием.

## Основные функции Паскаль-среды

Чтобы нам не думать, с помощью какой программы какой файл обрабатывать, быстро переходить от одного вида работы к другому, и создан набор программ, который мы будем называть Паскаль-среда. Основные виды ее деятельности таковы.

1. Ввод и редактирование текста программы. Встроенный в среду Паскаль-редактор отличается от обычных текстовых редакторов тем, что ориентирован непосредственно на Паскаль, например «знает» служебные слова и выделяет их специальным шрифтом или цветом, автоматически устанавливает курсор на нужное место при переходе на следующую строку (делает отступы слева), автоматически дает имени файла расширение PAS, дает возможность пользоваться справочной системой и т. п.
2. Трансляция (компиляция) программы, создание выполняемого файла.
3. Выполнение полученной программы, вывод результатов.
4. Работа с файлами: загрузка, сохранение, переименование, копирование и т. п.
5. Отладка программы (можно проследить ход выполнения программы «по шагам», узнать значения переменных в нужных точках).

Существует несколько сред для работы с Паскалем с разными названиями (к слову Паскаль добавляется приставка BORLAND, FREE, GNU, ABC и др.), некоторые из них имеют разные версии, могут запускаться из-под Windows или Linux; в средах могут использоваться разные версии языка Паскаль, разные компиляторы. Однако различия между ними невелики и проявляются лишь в весьма несущественных деталях, как правило, при работе с достаточно сложными и емкими программами, поэтому здесь углубляться в эти подробности не будем.

Дело в том, что язык Паскаль был разработан в 1970 году (автор — швейцарский ученый Никлаус Вирт (Niklaus Wirth)). Язык создавался как сред-

ство для удобной, наглядной записи алгоритмов и обучения программированию, но получил широкое распространение не только для этих целей и нашел практическое применение. При переносе этого «академического», разработанного «на бумаге» языка на реальные компьютеры разработчики программного обеспечения допускали различные отклонения от стандарта, предложенного автором, вносили добавления и изменения. Так появлялись различные «диалекты» языка. Понятно, что в 1970-х гг. не было графики, разных шрифтов, дисплеи были черно-белые. Новые возможности постепенно добавлялись в разные версии языка, иногда с некоторыми различиями. Мы будем здесь использовать версию Турбо Паскаль 7.0. Подавляющее большинство программ будет работать и во всех прочих версиях (например, на Free Паскале), другие вы легко сможете адаптировать, так как, повторим, это должны быть весьма сложные программы, и вы к моменту их написания уже будете обладать достаточными знаниями (некоторые различия мы постараемся отметить).

Рассмотрим основные правила работы в Паскаль-среде (поговорим отдельно о средах Турбо Паскаль или Free Паскаль — они очень похожи даже внешне — и о среде ABC-Паскаль).

## Ввод текста программы и ее редактирование

Редактирование — не самая сложная, но наиболее долгая операция при работе с программой (компьютер считает быстро, а вот человек печатает медленно). Возможно, именно поэтому окно редактора является основным в Паскаль-среде, в него мы попадаем при входе в среду. Эту область экрана можно перемещать, изменять в размерах, перекрывать, закрывать и открывать.

### Среда Турбо Паскаль

Поле для редактирования текста синего цвета, сверху — меню среды, снизу — «горячие клавиши». Слева внизу через двоеточие два числа — текущая позиция курсора (номер строки и номер позиции в строке). Когда вы нажимаете клавиши, курсор передвигается, числа меняются. На экране обычно помещается 20 строк текста по 80 символов в строке. Понятно, что программа может быть много больше как в «длину», так и в «ширину». В таком случае мы на экране видим лишь ее часть, помещающуюся в окно; чтобы увидеть другие части, надо перемещать курсор.

**Задание.** *Наберите длинную строку (более 80 любых символов), посмотрите, что теперь на экране, — программа как бы «убежала», виден только «хвост» длинной строки. Договоримся, что мы без особой нужды в наших программах не будем использовать строки длиной более 80 символов, чтобы текст было удобно читать с экрана.*

Ввод текста производится обычным образом — нажатием буквенных и цифровых клавиш. Для ввода больших букв, символов используется клавиша **Shift**. Вот только для перехода от латинских букв к русским (и наоборот) могут использоваться сочетания разных клавиш (а иногда и вообще может не быть возможности пользоваться русскими буквами). Для стирания неправильно набранных символов используются привычные клавиши **Delete** и **Backspace** (на многих клавиатурах она так не называется, а помечена стрелочкой влево, но находится не вместе с другими стрелочками, а справа вверху основной клавиатуры).

Редактор работает в двух режимах: вставки и замены. Переключение режимов производится клавишей **Insert**. При этом изменяется вид курсора. В режиме замены он изображается большим прямоугольником, в режиме вставки — маленьким (как символ подчеркивания). Назначение этих режимов понятно без особых пояснений: попробуйте сами. В пояснениях нуждается работа клавиши **Enter** (той, что называется «большая клавиша справа»). В режиме замены при ее нажатии происходит переход на следующую строку (причем не на первую позицию, а в ту позицию, где начинается текст предыдущей строки). В режиме вставки, если курсор внутри строки, при нажатии **Enter** строка разделяется на две (соединить части можно, подведя курсор в первую позицию второй строки и нажав клавишу **Backspace**, или подведя курсор в конец первой строки и нажав **Delete**). Если же курсор стоит в конце строки, вставляется новая строка. Ввод программы удобно производить именно в режиме вставки, так как никаким другим образом создать новые строчки нельзя.

К сожалению, так как Паскаль-редактор появился задолго до создания системы Windows, совместимость его с другими программами этой системы неполная. В частности, нельзя текст из какого-либо текстового файла перенести в файл, открытый Паскаль-редактором, с помощью привычных нам в Windows «копировать», «вырезать», «вставить», не действуют и принятые в Windows «горячие клавиши», они в Паскаль-редакторе свои.

Нужный кусок текста в Паскаль-редакторе можно выделить мышкой так же, как и в других привычных нам редакторах. Скажем сразу, что в Паскаль-среде вполне можно обойтись вообще без мышки — часто это даже удобнее. Например, для выделения фрагмента текста можно использовать **Shift** + стрелки дополнительной клавиатуры (есть и другой способ выделения — клавишами **Ctrl + K + B** — начало фрагмента, **Ctrl + K + K** — конец, но он менее удобен).

Копирование текста возможно только внутри программ, открытых Паскаль-редактором (а их одновременно можно открыть несколько) с помощью пункта меню **Edit** (вторая слева «кнопка»). Попасть в меню можно либо клавишей **F10** (а нужный пункт выбрать стрелочками), либо нажав

одновременно две клавиши **Alt** + первая выделенная цветом буква нужного слова меню, либо выбрав нужный пункт мышкой. Для выхода из меню нужно мышкой «кликнуть» по полю редактора или нажать клавишу **Esc**.

Открыв меню **Edit**, мы видим, что здесь есть кнопки:

- Clear** — удалить выделенный фрагмент, не запоминая его;
- Cut** — вырезать выделенный фрагмент и запомнить его (используется для переноса фрагмента текста на другое место или в другой файл);
- Copy** — скопировать выделенный фрагмент, не удаляя его из текста;
- Paste** — вставить выделенный фрагмент на место курсора;
- Undo** — отменить последнюю операцию;
- Redo** — вернуть отмененную операцию.

Рядом с этими словами выписаны так называемые «горячие клавиши», с помощью которых, не вызывая меню, можно выполнить нужные действия. Часто это две клавиши, одновременно их нажать трудно; сначала нажимается первая клавиша, удерживается, а затем та, которая стоит после знака «+».

Иногда при редактировании длинных файлов бывает полезна кнопка **Search**, которая содержит опции для поиска в файле фрагментов текста и их автоматической замены.

## Среда ABC-Паскаль

Поле белого цвета, работа в редакторе среды похожа на работу с Блокнотом. Редактор совместим с другими продуктами Windows: сюда с помощью стандартных операций копирования можно перенести текст, например, из Блокнота. Названия кнопок на русском языке. Средства для работы с редактором вы найдете во вкладке «Правка»: можно выделять, копировать и вставлять фрагменты текста, осуществлять поиск нужного текста — кнопки стандартные, привычные, этим среда более удобна, чем Турбо Паскаль.

## Работа с файлами

Каждая Паскаль-программа должна содержаться в отдельном файле. (Вообще говоря, вы можете поместить в один файл несколько программ, но выполняться будет только первая.) Чтобы Паскаль-среда могла успешно работать с ними, файлы должны иметь расширение PAS. Это расширение среда приписывает вновь создаваемым файлам сама, по умолчанию. Паскаль-редактором можно создавать файлы и с другим расширением (например, TXT), но тогда его надо вводить самостоятельно при сохранении файла.

Мы уже говорили, что имя файла для удобной работы в среде должно состоять не более чем из 8 символов: латинских букв и цифр. Имена, сформированные по-другому, могут неправильно читаться внутри среды.

Меню для работы с файлами в разных средах практически одинаковое (только в среде ABC названия кнопок на русском языке). Меню открывается кнопкой **File (Файл)**. Нажав ее, можно увидеть, что здесь есть пункты:

- |                                     |   |
|-------------------------------------|---|
| <b>New (Новый)</b>                  | — создать новый файл;   |
| <b>Open (Открыть)</b>               | — открыть файл, уже имеющийся на диске;   |
| <b>Save (Сохранить)</b>             | — сохранить файл;   |
| <b>Save as<br/>(Сохранить как)</b>  | — сохранить текущий файл под другим именем; при этом файл со старым именем также останется в памяти. Эта опция полезна для копирования файлов внутри среды, с ее помощью можно создавать новые программы, внося изменения в старые. |
| <b>Save all<br/>(Сохранить все)</b> | — сохранить все файлы, имеющиеся на экране.   |

Поговорим сначала о сохранении файлов, так как именно это действие нам надо срочно произвести с нашей программой «Здравствуй, мир!». Почему срочно? Дело в том, что в процессе ввода и редактирования программу надо сохранять в памяти как можно чаще, а не только тогда, когда она полностью готова. Совсем не трудно периодически нажимать клавишу **F2** или картинку с дискеткой (это горячая клавиша опции **Save**), и в таком случае вы будете застрахованы от потери результатов своего труда из-за каких-то непредвиденных ситуаций: отключили электричество, кошка села на клавиатуру так, что все стерлось, и т. п.

Если вы попытаете записать в память подготовленную вами программу, среда потребует, чтобы вы ввели имя файла, в котором она будет храниться (по умолчанию файл в Турбо называется *NONAME* — безымянный, а в ABC — Program1); имя файла вы можете видеть в верхней строке редактора.

Для ввода имени поверх окна с текстом программы появляется окно меньшего размера, состоящее из нескольких полей. В Турбо-среде эти окна разноцветные, в ABC — все белого цвета, действия в обеих средах надо производить одинаковые, практически такие же, как и при записи файла в любой программе. Верхнее поле — полоска синего цвета, называется **Save file as**. Надо напечатать имя, которое вы хотите дать своему файлу (в Турбо-среде — в окне синего цвета Save file as, в среде ABC — в поле «Имя файла»). Напомним, что расширение PAS набирать не надо (ставить

точку тоже). В большом зеленом поле написаны имена файлов, которые уже есть в рабочей директории. Что такое директория? Это не новое понятие языка Паскаль — этим словом раньше, во времена создания Паскаль-среды, называли то, что мы сейчас называем «папка». Находясь в Паскаль-среде, будем говорить не «папка», а «директория», так как есть команды, в названиях которых отражено именно это слово. Сама рабочая, доступная на данный момент директория указана в самом нижнем, синем поле (там же помещается еще некоторая информация о текущем состоянии среды). Перед записью программы вы должны внимательно посмотреть, какая директория у вас открыта. Та ли, в которую вы обычно записываете свои программы (рекомендуется завести специальную папку для хранения программ)? Если вы запишете программу «не глядя», не туда, куда надо вам, а куда предлагает среда, потом придется ее долго искать.

В нижнем поле выписывается полное имя (путь к файлу). При этом, к сожалению, длинные имена папок, имена с русскими буквами отображаются некорректно. Вывод: папку для Паскаль-программ не надо располагать слишком «глубоко», папкам не надо давать длинные имена и имена, содержащие русские буквы; папку не надо помещать на Рабочий стол.

Может возникнуть необходимость смены директории, если рабочая директория при входе в среду установлена не та, в которой находятся (или должны находиться) программы. Для этого в Турбо-среде в меню **File** есть специальная кнопка **Change dir** (сменить директорию), в ABC-среде директория (папка) выбирается в окошке слева.

При открытии уже имеющегося файла вы волей-неволей должны обратить внимание, какая директория открыта, так как в большом окошечке высвечиваются имена файлов, находящихся именно в открытой рабочей директории (да еще и только с расширением PAS, если не задано иное), других вы просто не увидите.

Файл, который открывается с помощью **Open** или создается с помощью **New**, отображается в новом окне. Да, Паскаль — многооконная система.

В среде ABC можно видеть на экране только одно окно, от других открытых видны только названия в верхней строке, щелкая по этим названиям, можно переходить от одного окна к другому. В Турбо-среде работа с окнами более «продвинутая», есть даже специальное меню **Window**. С его помощью можно менять расположение окон, их размер, переходить от одного окна к другому и т. п. Нам в первую очередь понадобится команда «закрыть окно», горячие клавиши которой **Alt + F3** (запомнить нетрудно — открывается окно с файлом горячей клавишей **F3**).

Окна с файлами, которые уже не нужны, надо убирать с экрана, закрывать. При этом, если файл (последняя версия) не записан в память, среда напоминает об этом. Скажем здесь еще об одной особенности Паскаль-редактора Турбо, которой нет во многих других редакторах. При записи

в память очередного варианта изменений файла старая версия тоже остается в памяти. Она помещается в ту же директорию, под тем же именем, с расширением ВАК. Хранится только одна версия, при очередном редактировании программы содержимое этого файла изменяется.

Мы не советуем вам открывать одновременно много окон. В большинстве случаев для работы нужно одно — с программой, которую вы редактируете, иногда может понадобиться еще одно-два, если вы хотите в создаваемую программу скопировать какой-то текст из других файлов. Особенно будьте осторожны, открывая один и тот же файл несколько раз, в разных окнах (среда это позволяет и не предупреждает, что такой файл уже открыт). Могут возникнуть неприятности при закрытии этих окон, вы можете закрыть окно с одной из первых версий текста в последнюю очередь, следовательно, она и сохранится, а последний, правильный текст потеряется.

И наконец, в списке меню кнопки **File** находится команда **Exit** «выход из Паскаль-среды» (горячие клавиши **Alt + X**). Рекомендуется заканчивать работу с Паскалем именно с помощью этой опции меню, а не путем закрытия окна среды. При правильном, «стандартном» выходе происходит, если надо, запись в память открытых файлов.

Можно видеть, что никакой опции типа «уничтожить» в меню нет, поэтому, если какие-то файлы стали вам не нужны, удалять их надо не из Паскаль-среды, а, например, средствами Windows, с помощью Проводника.

## Компиляция и выполнение программы

Итак, мы наконец-то записали нашу программу в память, в нужную директорию. Если вы вспомните описанные выше этапы работы с программой, то сразу увидите, какие кнопки основного меню нам сейчас нужны. В меню Турбо-среды их можно видеть сверху, а в меню ABC они «спрятаны» во вкладке «Программа». Конечно, это **Компилировать** — **Compile** и **Выполнять** — **Run**. Каждую из кнопок можно нажимать, выбирая нужный пункт меню, а можно, запомнив соответствующие горячие клавиши (они написаны в меню рядом с кнопками). Нажмем кнопку **Compile** — компиляция. Компиляция такой маленькой программки происходит мгновенно, на экране появляется окошко о том, что она успешно завершена — «Compile successful», для продолжения работы надо нажать любую клавишу. После ее нажатия вы снова видите перед собой окно редактора, а в памяти компьютера появляется выполняемый файл (с расширением EXE) с вашей программой. Выполнять такой файл, смотреть, как работает программа, можно уже и без Паскаль-среды, например из Windows. Нам это пока не нужно, мы будем работать в Паскаль-среде, поэтому будем в основном пользоваться кнопкой **Run** — выполнять (**Ctrl + F9** в Турбо-среде и треугольник в ABC). Эта кнопка запускает сразу два действия: и компиляцию, и выполнение откомпилированной программы.

Нажмем ее. Если вы работаете в среде ABC, то в нижнем окошке видите результат работы нашей программы. А вот в Турбо-среде ничего не произошло. Перед нами все то же окно редактора. Что-то сломалось? Надо нажать еще раз, посильнее? Не стоит, все, что надо, уже произошло, просто слишком быстро, вы этого не заметили. Программа у нас маленькая, делает единственное действие: печатает на экран текст из двух слов. Современный компьютер делает это за доли секунды.

Почему же мы не видим текста? А мы уже говорили, что работа с редактором программ — одно из основных назначений Паскаль-среды, поэтому после выполнения программы автоматически происходит возврат в окно редактора, чтобы вы смогли дальше дописывать, корректировать программу. Как же посмотреть, что она напечатала, и где она это сделала? Результат выведен в специальном окне **User Screen** (экран пользователя), перейти в него можно клавишами **Alt + F5** (выйти — нажав любую клавишу). Это окно черного цвета, на нем вы можете видеть свой текст «Здравствуй, мир!» (над текстом могут быть выведены некоторые сообщения, не имеющие отношения к программе, они появляются при загрузке Паскаль-среды). Кстати, если вы все-таки нажали клавиши **Ctrl + F9** несколько раз, на экране увидите несколько строчек с одинаковым текстом.

## Почему транслятор «ловит» ошибки?

Давайте дополним нашу программу, чтобы она выводила две строчки текста. Перед строкой

```
Writeln ('Здравствуй, мир!')
```

вставим строку

```
Writeln ('Это моя первая программа');
```

Обратите внимание, что в конце строки стоит символ «;». Не забыв сохранить новую версию программы, запустим ее выполняться, перейдем в экран пользователя и убедимся, что теперь на печать выводятся две строчки с написанным нами текстом.

А теперь немножко «испортим» нашу программу, сделаем в ней ошибку, уберем из только что вписанной строки точку с запятой. Запустим ее на компиляцию (или на выполнение). Вместо результата или сообщения об успешной компиляции видим, что в окне редактора появляется строчка с текстом на красном фоне. Это означает, что обнаружена ошибка: в строчке сообщается номер ошибки (существует пронумерованный список, где перечислены все ошибки, обнаруживаемые компилятором) и кратко описывается ее суть. У нас текст «; expected» — ожидается точка с запятой. При этом курсор стоит в том месте, где, как считает компилятор, она и ожидается, — почему-



то не в конце первой, а в начале второй строки. Дело в том, что точка с запятой должна стоять между этими строчками, в любом месте, не обязательно точно в конце первой строки. Вот компилятор и искал (ожидал) ее вплоть до начала следующей строчки.

Исправим эту ошибку (поставим точку с запятой на место) и сделаем другую ошибку, например уберем какую-нибудь букву в слове **Writeln**. Будет выявлена ошибка «Unknown identifier» — неизвестное имя. При этом курсор четко указывает на слово, в котором мы сделали ошибку.

Мы сейчас не будем изучать весь список обнаруживаемых компилятором ошибок, обычно предоставляемое им описание достаточно понятно. В некоторых случаях, по ходу изучения языка, мы будем обращать внимание на типичные ошибки и реакцию компилятора на них.

Почему же компилятор обнаруживает ошибки? Правильно ли он указывает их суть и место? Всегда ли ему можно доверять?

Чтобы ответить на эти вопросы, надо понять, что компилятор обнаруживает ошибки не потому, что он такой «умный» или «добрый» и хочет помочь несчастному программисту правильно написать текст. Обнаруживает он их как раз потому, что не очень «умный», а является, по сути, «шифровальной машиной», которая из одного текста делает другой. Если какая-то конструкция исходного текста непонятна, не содержится в «словаре», компилятор не может ее перевести, останавливается в данном месте программы и выдает сообщение об ошибке.

Таким образом, остановиться компилятор может не точно в том месте, где ошибка в тексте допущена, а где-нибудь после нее, где он обнаружил, что текст ему непонятен. Существует еще и такая ситуация: так называемые «наведенные ошибки». Если некоторая конструкция была записана автором текста неправильно, неуместно, но по правилам языка является законной, компилятор ее саму переводит, но обнаруживает несоответствия дальше по тексту (в правильных конструкциях). Сейчас, когда вы еще не знаете языка Паскаль, мы не будем приводить соответствующие примеры. В процессе программирования вам предстоит встретиться со всякими ситуациями, в том числе и с описываемой здесь. Сейчас хочется лишь, чтобы вы поняли, что сообщения компилятора не должны быть единственной «путеводной звездой» для составителя программы, не всегда надо вносить в программу именно тот символ, который в данном месте «expected».

И еще немного об ошибках в программе. К сожалению, если ваша программа успешно прошла компиляцию, нельзя считать, что в ней нет ошибок. Все еще только начинается! Дело в том, что компилятор обнаруживает лишь так называемые синтаксические ошибки: пропущен знак препинания, неправильно записано слово и т. п. Семантические, смысловые ошибки он обнаружить не может, так как просто не «задумывается» о смысле вашей программы, а слепо выполняет то, что вы написали.

Например, если вы вдруг решите, что теорема Пифагора выглядит так:  $C = A + B$  и, основываясь на этом представлении, напишете программу вычисления гипотенузы треугольника, программа будет работать, выдавать некоторый результат. Но он, конечно, будет неправильным. Никакой компьютер эту неправильность обнаружить не может, потому что не имеет представления ни о Пифагоре, ни о треугольниках. Он умеет вычислять по заданной вами формуле, что и будет делать совершенно правильно. Точно так же компьютер не обнаружит никаких ошибок в тексте на русском языке в нашей программе (если вы напишете «Здравствуй, мир!» с ошибками), а просто выведет на экран именно то, что вы попросили.

Для обнаружения таких ошибок существуют тестирование и отладка программы, о которых мы говорили выше. Впрочем, к нашей программе это не относится, в ней нет никаких вычислений, так что если она работает, то работает правильно.

## Справочная система Паскаль-среды

В основном меню среды справа находится кнопка **Help (Помощь)**. Это вход в справочную систему. Здесь можно найти некоторые справки по работе среды, редактора, компилятора, а также по самому языку Паскаль. Например, выбрав пункт **Reserved Words**, вы получаете список всех служебных слов вашей версии языка. Причем, выбрав нужное слово из списка и нажав **Enter**, вы получаете достаточно подробную справку о его использовании (увы, на английском языке). В конце этого текста обычно находится пример — фрагмент программы или целая программа на Паскале, поясняющая использование данного объекта. Программа может быть скопирована в новый файл и выполнена.

В Турбо-среде попасть в окно справки, поясняющее значение нужного слова, можно и другим способом. Надо поставить курсор на интересующее вас слово (это должно быть, естественно, слово из языка, а не придуманное вами имя) и нажать клавиши **Ctrl + F1**. При этом, если в слове у вас допущена ошибка (программа не нашла его в списке), выдается список всех слов языка и вы можете попробовать поискать в нем то, что вам надо.

Конечно, изучить язык с помощью его справочной системы затруднительно (тем более, напомним, самое сложное в программировании все-таки не запись конструкций, а составление алгоритма), но пользоваться системой именно для справки очень полезно.

## «Горячие клавиши» среды Турбо Паскаль

Напоследок выпишем «горячие клавиши», которыми придется пользоваться особенно часто.

Клавиши	Элемент меню	Функция
F1	Help	Вывод на экран окна подсказки о работе редактора
Ctrl + F1	Help/Topic Search	Вывод окна подсказки об использовании оператора или служебного слова, на котором стоит курсор
F2	File/Save	Сохранение программы из окна редактора
F3	File/Open	Открывается окно для выбора нужного файла
Alt + F3	Window/Close	Закрывается окно редактора
F5	Window/Zoom	Меняется масштаб окна редактора
Alt + F5	Debug/User Screen	Переход в «экран пользователя» — окно, где отражена работа программы. Переход в редактор — любая клавиша
F9 Alt + F9	Compile/Make Compile/Compile	Запускается на компиляцию программа из открытого окна (в случае одной небольшой программы кнопки работают одинаково)
Ctrl + F9	Run	Программа компилируется и запускается на выполнение
F10		Переход в строку меню
Alt + X	File/Exit	Выход из Паскаль-среды

И еще одна «горячая» клавиша: **Alt + Enter** позволяет включить/выключить полноэкранный режим работы Паскаль-среды.

## Задачи 3.1–3.4. Работа в редакторе

- 3.1. Проверьте работу всех горячих клавиш. Правильно ли они работают в используемой вами Паскаль-среде?
- 3.2. Воспользуйтесь справочной системой, посмотрите, какой текст выдается к словам **Begin**, **Writeln**.
- 3.3. Введите в Паскаль-редакторе следующий текст: в первой строке — имя и фамилия, во второй строке — адрес, в третьей — телефон. Запишите этот текст в память под именем NAME1.pas. С помощью функции копирования подготовьте файл NAME2.pas, где каждая строчка указанного текста записана по 2 раза подряд.
- 3.4. С помощью копирования текста из одного файла в другой составьте программу, которая печатает текст, записанный в задаче 3.3, на экране.

## Глава 4

# Структура Паскаль-программы

---

Паскаль-программа состоит из **заголовка** и **программного блока**. Заголовок и блок отделяются друг от друга точкой с запятой, а за блоком, в конце всей программы, ставится точка. Эта точка является для транслятора признаком конца текста программы, дальше он читать ничего не будет. Именно поэтому в файл имеет смысл помещать только одну программу, если поместить несколько — компилироваться будет только первая.

Заголовок начинается со служебного слова **Program**, далее идет имя программы. Отметим сразу, что имя программы в наших примерах носит чисто информативный характер, оно не обязательно должно совпадать с именем файла, в котором хранится программа. В заголовке может содержаться еще некоторая информация, но мы ее рассматривать не будем, более того, в некоторых программах заголовок будем вообще опускать, так как в наших программах он никакой смысловой нагрузки не несет.

Блок программы состоит из **раздела описаний** и **раздела операторов**. Они отделяются друг от друга точкой с запятой.

Все объекты, используемые в программе (имена, вводимые пользователем), должны быть описаны. В разделе описаний описываются метки, константы, типы данных, переменные, процедуры и функции. Нам в первую очередь понадобятся описания переменных и констант, а затем мы научимся описывать и более сложные объекты.

Раздел описаний может быть пустым (как в нашей первой программе), а может занимать почти весь текст программы.

В разделе операторов задаются действия, которые программа должна выполнить — операторы. Операторы отделяются друг от друга точкой с запятой, а весь раздел заключается в операторные скобки **Begin-End**.

Посмотрев на нашу программу, мы обнаружим в ней заголовок (мы дали программе имя **FIRST**) и раздел операторов, состоящий из одного оператора **Writeln('Здравствуй, мир!')**, именно он и обеспечивает вывод текста на экран.

В программе есть еще одна строка, мы уже упоминали, что это — **комментарий**. Комментарием может быть любой текст, заключенный в фигурные скобки или в скобки, состоящие из двух символов: **(\*** и **\*)**. Коммен-

тарии не оказывают никакого влияния на работу программы, можно даже сказать, что и не являются частью программы, предназначены они не для компьютера, а для человека. Комментарии используются для внесения пояснений в программу, дают возможность сделать ее более понятной, облегчить дальнейшую работу с ней. В комментариях может быть указана фамилия автора программы, номер решаемой задачи, дата последних изменений программы, может поясняться алгоритм, назначение переменных и т. п. Мы настоятельно рекомендуем вам комментировать свои программы, чтобы их легче было использовать при подготовке к контрольным работам, экзаменам.

## Глава 5

# Основные типы данных. Описания переменных. Присваивание

---

Имена, вводимые программистом (в том числе переменные и константы), должны быть описаны в программе в разделе описаний.

Описания переменных начинаются служебным словом **Var** (*Variation* — изменение, *Variable* — переменная). Далее идет список переменных какого-либо одного типа через запятую, затем двоеточие и указание типа.

Описания переменных разного типа отделяются друг от друга точкой с запятой, а слово **Var** можно не повторять:

```
Var K,M      : Integer;  
    Stroka   : String;
```

Здесь описаны две переменные *K* и *M* типа **Integer** и переменная **Stroka** типа **String**. Имена переменных мы придумали сами, а вот имена использованных здесь типов определены языком, это стандартные имена, их описывать не надо.

Язык не запрещает помещать несколько описаний в одной строке, но лучше описания разных типов писать в отдельных строках. Можно добавлять комментарии, поясняющие, для чего будет использоваться та или иная переменная.

Самым существенным в описании переменной является ее тип. Тип указывает, что собой представляет значение переменной — символ, число, массив чисел и т. п. Зачем важно знать тип переменной? Во-первых, для размещения переменных разных типов в памяти компьютера нужен разный объем памяти, данные по-разному кодируются, имеют разный допустимый диапазон значений. Во-вторых, над данными разных типов возможны разные операции. Например, числа можно делить друг на друга, а применительно к символам такая операция недопустима.

Интересно, что данные разных типов при записи на бумаге и даже при выводе на экран могут выглядеть совершенно одинаково. Например, число «5» и символ «5» на экране передаются одной и той же «картинкой», однако в памяти компьютера выглядят совершенно по-разному (по-разному кодируются) и в программе имеют совершенно разный смысл.

Типы данных можно разделить на две группы: основные (простые) и производные. Переменная простого типа имеет какое-то одно значение, переменная производного типа может обозначать целую структуру, состоящую из нескольких компонент. В этом разделе предметом нашего изучения будет простой тип.

## Некоторые типы данных и работа с ними

Нам чаще всего придется иметь дело с числами, поэтому с них и начнем.

Числа в языке Паскаль подразделяются на целые и вещественные. Вещественные числа — это числа, у которых есть целая и дробная части, пусть даже равные нулю. Различия между целыми и вещественными числами весьма существенные, хотя числа, «выглядящие» совершенно одинаково на экране компьютера, могут быть как целыми, так и вещественными.

Во-первых, у них разный диапазон значений, во-вторых, разный набор допустимых арифметических операций, а главное — целый тип относится к так называемым **ординальным** типам (от *order* — порядок), а вещественный тип — не ординальный.

Разберемся сейчас с этим понятием, так как оно неоднократно понадобится нам в будущем. Ко всем значениям ординального типа применима стандартная функция **Ord**, которая дает номер значения в типе. Целые числа можно пронумеровать, для любого числа (кроме самого большого и самого маленького) найдется единственное предыдущее и последующее. Например, для числа «5» предыдущим является «4», а следующим — «6». Для вещественных чисел это совсем не так. Какое вещественное число непосредственно следует за вещественным «5»? Может быть, «5.1»? Или «5.01»? Или «5.0000001»? Эти и некоторые другие различия между целыми и вещественными числами, которые мы в обычной жизни практически не замечаем, оказываются очень важными в программировании, поэтому при изучении каждой новой темы, новых операторов, в которых встречаются числа, всегда следует обращать внимание, о каких числах идет речь.

## Целые числа

Для описания целых чисел обычно используется стандартное имя типа **Integer**. В реальной жизни диапазон целых чисел не ограничен, при работе на компьютере это совсем не так. Для хранения числа отводится фиксированный объем памяти, поэтому существует минимальное (отрицательное) и максимальное значение типа **Integer**. Они, вообще говоря, могут различаться в разных версиях языка, и значение максимального целого числа зафиксировано в константе **MaxInt** (это стандартная константа, ее не надо описывать).

**Задание.** *Посмотрите, каково значение константы **MaxInt**. Для этого составьте программу, в которой будет оператор **WriteLn (MaxInt)**.*

Обычно минимальное целое число: **-MaxInt-1**.

Диапазон значений этого типа невелик, но, так как тип в основном используется для разного рода счетчиков (например, количество учеников в классе), его обычно хватает. В разных версиях языка есть и другие описатели для целых чисел: **LongInt**, **ShortInt**, **Word**, **Byte**. Информацию о них (диапазон значений) вы можете посмотреть сами, используя «помощь» в Паскаль-среде. Наберите в редакторе слово **Integer**, подведите к нему курсор, вызовите справку нажатием **Ctrl + F1**.

Все указанные типы отличаются только диапазоном значений, поэтому, когда далее мы будем говорить о типе **Integer**, следует понимать, что все сказанное относится и к другим типам целых чисел. В программах мы будем использовать в основном тип **Integer**. Когда в отдельных случаях нам понадобится использование какого-либо другого целого типа, мы оговорим это особо.

Изображаются целые числа обычным образом: в виде последовательности цифр, причем, если перед числом записать один или несколько нулей (незначащие нули), это не будет ошибкой. Перед числом может стоять знак («+» или «-»), знак «+» можно не писать.

Примеры изображения целых чисел:

1      100      0125      -53      +3200      -0007

Пример описания данных целого типа:

```
Var K,M : Integer;
```

## Вещественные числа

Для описания вещественных чисел обычно используется стандартное имя типа **Real**. Как и в случае целых чисел, на вещественные числа, которые может использовать компьютер, наложены ограничения. Ограничения связаны с диапазоном и с точностью представления чисел (количеством верных значащих цифр в числе). Сразу скажем, что кроме **Real** в разных версиях Паскаля существуют другие типы вещественных чисел, отличающиеся друг от друга как раз этими показателями. Мы будем использовать **Real**, а посмотреть, какие еще бывают описатели, оценить характеристики описываемых ими чисел вы можете с помощью справочника Паскаль-среды.

Примеры описания данных вещественного типа:

```
Var S,D      : Real;
```

Изображаются вещественные числа двумя способами: с фиксированной и с плавающей точкой. Оба типа представления могут иметь или не иметь знак «+» или «-».



Запись с фиксированной точкой — это та запись, к которой вы привыкли в школе (только целая часть числа отделяется от дробной знаком «.», а не «,»). Сначала записываются цифры целой части, потом ставится точка, а затем пишутся цифры дробной части (без пробелов!). Ни целая, ни дробная части не могут быть пустыми, обязательно хотя бы одна цифра (например, ноль) должна присутствовать.

Примеры:

Правильные числа с фиксированной точкой	Неправильные числа
0.5 +3.141591 -123.456789 0.0000000004	.5 124. 3,14159 25h

Если число очень большое или очень маленькое (как последнее из правильных в нашем примере), запись с фиксированной точкой неудобна: приходится, напрягая глаза, подсчитывать количество нулей. В этом случае удобно применять запись с плавающей точкой.

При записи чисел с плавающей точкой число представляется в виде произведения мантиссы на степень числа 10 (показатель степени называется «порядок»); например  $1234 = 1.234 * 10^3$ ,  $0.00123 = 1.23 * 10^{-3}$ . В Паскале все записи делаются одинаковым шрифтом и в одну строку, поэтому нельзя показатель степени записать, как это принято, маленькими цифрами сверху. Для того чтобы отделить порядок от мантиссы, используется буква E (латинская, большая или маленькая), а символ умножения и число 10 не пишутся вообще. Получается вот такая непривычная запись числа: цифры (возможно, разделенные точкой), затем буква E (e), за которой следует изображение целого числа (как положительного, так и отрицательного). Например:  $12E - 5$ ,  $34.7e + 6$ ,  $0.04E21$ .

Заметим, что после буквы E стоит показатель степени, он может быть только целым; до буквы E стоит мантисса, она может быть представлена как целым числом, так и числом с фиксированной точкой. Число в любом случае понимается как вещественное. Все «части» числа обязательно должны присутствовать, поэтому вот такие числа:  $12.e45$ ,  $.34E5$ ,  $123E$  — являются неправильными.

В качестве примеров запишем одни и те же числа в разном представлении.

С плавающей точкой	Привычное представление	С фиксированной точкой
3.1415E0	$3,1415 * 10^0$	3.1415
5E-7, 5E-07, 0.5E-006	$5 * 10^{-7}$ , $0,5 * 10^{-6}$	0.0000005
4E5, 04E05, 4.0E+5	$4 * 10^5$	400000
1.2345E4, 0.12345E5	$1,2345 * 10^4$ , $0,12345 * 10^5$	12345
1.2345E-4, 123.45E-6	$1,2345 * 10^{-4}$ , $123,45 * 10^{-6}$	0.00012345

***Задание.** Напишите сами примеры неправильной записи чисел с фиксированной и плавающей точкой. Посмотрите, в чем ошибка, и больше такие числа в программах не пишите!*

## Операции над числами

Конечно, и над вещественными, и над целыми числами в Паскале определены привычные нам арифметические операции.

### Сложение, вычитание и умножение

Вычитание и сложение обозначаются теми же знаками, что и в математике, умножение обозначается звездочкой, причем, в отличие от математических формул и уравнений, где знак умножения часто опускается, в Паскале его писать обязательно (в математике запись  $5X$  допустима и понимается как умножение, в программе на Паскале нужно писать **5\*x**).

При целых аргументах результат операции получается целый, если хотя бы один из аргументов вещественный — результат вещественный. Заметим, что важен именно тип аргумента, а не его значение: при сложении вещественных чисел 2.7 и 4.3 получается число 7, которое «по внешнему виду» можно принять за целое, однако в компьютере оно будет кодироваться как вещественное (иногда, чтобы это было понятнее, такое число записывают как 7.0).

### Деление

В Паскале три операции деления.

1. «Обычное» деление обозначается знаком  $\langle / \rangle$ . Независимо от аргументов результат такого деления вещественный (**Real**). На это стоит особо обратить внимание: даже если результат выглядит целым (например,  $5/1 = 5$ ,  $20/2 = 10$ ), для компьютера он является вещественным числом с дробной частью, равной нулю.

Следующие две операции применимы только к целым числам (оба числа должны быть типа **Integer**). Результат тоже будет целым. Обе эти операции — части действия, которое в школе называется «деление с остатком». Поскольку значков для этих операций не хватило, они обозначаются с помощью служебных слов **div** и **mod**. Между названием операции и операндами (делимым и делителем) обязательно должны быть пробелы.

2. Нахождение целой части от деления числа — **div**.
3. Нахождение остатка от деления — **mod**.

Соответственно, возможны записи: **X div Y**; **35 div 21**; **A mod 4**.

Естественно, ни один из видов деления невозможен, когда делитель равен 0. Операции **mod** и **div** обычно применяются к положительным числам, но на всякий случай заметим, что для отрицательных чисел они в разных версиях Паскаля могут давать разный результат.

Деление с остатком обычно проходят в начальной школе и «с возрастом» основательно забывают. Давайте вспомним, что это такое.

Повторим, операция выполняется только над целыми числами и все результаты тоже целые. А результатов два: неполное частное и остаток. Пусть  $M$  и  $N$  положительные. Чтобы число  $M$  разделить на  $N$  с остатком, надо подобрать такое максимальное число  $K \leq M$ , которое нацело делится на  $N$ . Результат деления  $K$  на  $N$  — неполное частное, разность  $M - K$  — остаток. Заметим, что если  $M$  делится на  $N$  нацело, то остаток равен 0. Если  $M < N$ , то неполное частное равно 0, а остаток равен самому числу  $M$ . Остаток всегда меньше делителя.

Если частное обозначить **C** (**C:=M div N**), а остаток **T** (**T:=M mod N**), выполняется соотношение: **C\*N+T=M**.

Несколько примеров:

Делимое <b>M</b>	Делитель <b>N</b>	Частное <b>M div N</b>	Остаток <b>M mod N</b>
10	5	2	0
13	5	2	3
4	5	0	4

Отметим интересную особенность деления с остатком на 10. Остаток от деления числа на 10 равняется последней цифре этого числа, а частное — числу без последней цифры:

12345 mod 10 = 5;

12345 div 10 = 1234.

Первую цифру числа можно получить, если поделить его на  $10^{k-1}$ , где  $k$  — количество цифр в числе: **543 div 100 = 5** — при этом остаток от деления будет равен числу без первой цифры: **543 mod 100 = 43**.

## Функции

В языках программирования часто возникает необходимость проведения одинаковых вычислений с разными исходными данными. Для упрощения таких вычислений можно написать специальным образом оформленный программный блок и подключать его к своей программе. Такой блок называется функцией. Функции может писать программист, но есть и набор стандартных функций, входящий в состав языка.

Большую часть функций, известных из математики, можно использовать в Паскале. При записи функций надо учитывать, что их аргумент всегда пишется в круглых скобках.

Некоторые функции записываются так же, как в математике, например, можно написать: **sin(x)** ; **cos(3\*z-5)** .

Для функций, которые в математике записываются специальными знаками, в Паскале пришлось придумать буквенные названия:

- **abs** — взятие модуля (от слов *absolute value* — абсолютная величина);
- **sqr** — возведение в квадрат (*square* — квадрат);
- **sqrt** — извлечение квадратного корня (*square root* — квадратный корень).

Чтобы список математических функций был полным, остается упомянуть **arctan** (арктангенс), **ln** (натуральный логарифм, т. е. логарифм по основанию  $e$ ) и **exp** (экспоненту, степень числа  $e$ ). Возможно, эти названия вам еще незнакомы, но как только вы их изучите в школе, использование соответствующих функций в Паскале не составит труда.

Все эти функции имеют один аргумент (целый или вещественный), результат у **abs** и **sqr** того же типа, что и аргумент, а у всех остальных функций результат всегда вещественный.

В Паскале определены не только математические функции, есть еще логические, строковые, мы с ними познакомимся в дальнейшем.

Сейчас остановимся на тех функциях, которые часто используются в математике, но отсутствуют в Паскале. Это в первую очередь тангенс и возведение в степень.

Думаю, любой читатель сможет выразить тангенс через имеющиеся в Паскале функции. Кстати, аргументы тригонометрических функций в Паскале выражаются в радианах.

Для представления функции возведения в степень также существует формула (в ней используются **exp** — экспонента и **Ln** — натуральный логарифм). Кто уже знаком с этими функциями из школьного курса, легко сможет вывести эту формулу (воспользовавшись основным логарифмическим тождеством), остальным придется немного подождать, поэтому здесь мы ее приводить не будем.

Иногда возникает необходимость «перевести» вещественное число в целое, округлить. Для этого существуют специальные функции, аргументами которых являются вещественные числа, а результаты — целыми:

- **trunc** — дает целое число, которое получается отбрасыванием цифр после точки и самой точки: **trunc(1.8)=1**; **trunc(1.1)=1**; **trunc(-2.6)=-2**; **trunc(-10.4)=-10**;
- **round** — дает округленное целое число: **round(1.8)=2**; **round(1.1)= 1**; **round(-2.6)=-3**; **round(-10.4)=-10**.

Чтобы хорошо понять, как работают эти функции, полезно знать, что они связаны формулой:

$$\text{Round}(x) = \begin{cases} \text{Trunc}(x+0.5) & \text{при } x \geq 0; \\ \text{Trunc}(x-0.5) & \text{при } x < 0. \end{cases}$$

## Запись и вычисление арифметических выражений

В арифметическом выражении на Паскале могут использоваться константы и имена переменных, знаки арифметических операций, функции. Скобки можно использовать только круглые. Порядок выполнения операций привычный, как в математике: сначала вычисляются аргументы функций и сами функции, потом выполняются действия в скобках, потом слева направо операции 1-го ранга: **\***, **/**, **div**, **mod**, затем — операции 2-го ранга: **+** и **-**.

Запись арифметического выражения на Паскале должна быть линейной, из-за этого часто возникают «лишние» скобки, которые не нужны в обычной записи например, в дробях:

$$\frac{a+b}{x-y} \text{ — на Паскале будет выглядеть } (a+b) / (x-y).$$

*Пример.* Вспомним формулу вычисления дискриминанта и корней квадратного уравнения и запишем ее на Паскале.

Получим дискриминант: **sqr(b)-4\*a\*c**.

Один из корней **(-b+sqr(D))/(2\*a)** или **(-b+sqr(D))/2/a**, где **sqr(x)** — это  $x^2$ , а **sqr(x)** — это  $\sqrt{x}$ .

## Сравнение чисел

Понятно, что числа можно сравнивать по величине. В Паскале определены все известные нам из математики операции сравнения, записываются они обычным образом, только для некоторых приходится использовать два символа:

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

Со сравнением целых чисел никаких проблем не возникает, а вот о вещественных придется поговорить. Дело в том, что если два вещественных числа почти одинаковы, они могут «выглядеть» одинаково (например, будучи напечатанными на экране), но различаясь, например, в самом последнем знаке после запятой, одинаковыми (равными) не являться. Следует знать, что, если из

вещественного числа извлечь корень, а потом обратно возвести его в квадрат, может получиться величина, чуть-чуть отличная от первоначальной, т. е. `sqr (sqrt (x))` не всегда равно `x`. Эта ситуация не должна быть новой для вас, если вам приходилось считать на калькуляторе: например, иногда можно подобрать такие числа (ненулевые), что результат деления одного на другое будет равен нулю. Эта проблема может вызвать неприятности, если в программе надо проверить, не равны ли друг другу два вещественных числа. В таком случае для подстраховки проверяют числа не на точное равенство (`x=y`), а на приблизительное (`abs (x-y) < EPS`), где **EPS** (имя придумано нами) — очень маленькое число, обычно оно зависит от точности измерений и вычислений.

## Символы и строки

Для описания символьного (литерного) типа используют служебное слово **Char** (от *Character* — символ). Простейшая строка — это несколько (от 0 до 255) символов, для ее описания используется слово **String** (которое и означает «строка»).

Символьные константы и строки пишутся в апострофах. Символьной константой может являться любой символ из некоторого набора, определяемого реализацией языка. Обычно это цифры, прописные и строчные латинские буквы, знаки препинания, знаки арифметических операций, часто прописные и строчные русские буквы и много других символов. Таким образом, в качестве литер можно использовать символы любых групп, которые мы обсудили в начале, в том числе русские буквы (а также буквы любого другого национального языка, если они есть в вашей реализации).

Примеры описания данных символьного типа и строк:

```
Var Symbol      : Char;  
    Stroka      : String;
```

Большинство символов можно ввести с клавиатуры и увидеть на экране, но есть и так называемые управляющие символы, которые сами на экране не отображаются, но управляют размещением других символов на экране (например, перевод строки). Некоторые символы нельзя ввести с клавиатуры простым нажатием одной-двух клавиш, позже мы научимся «извлекать» их.

Примеры символьных (литерных) констант: `'1'`, `'@'`, `'ë'`, `'¶'`. Примеры строк здесь приводить не будем — это просто последовательность любых литер, взятая в апострофы. Именно строку мы использовали в своей программе «Здравствуй, мир!».

Обычно символов 256, они пронумерованы от 0 до 255. Эти номера (их принято называть коды) запоминать не надо (тем более они могут отличаться в зависимости от используемой таблицы кодировки — таблицы соответ-

ствия каждого символа и его номера). Скажем сразу, что номера цифр не совпадают с их значениями, т. е. код нуля вовсе не равен нулю.

Номер (код) символа `h` можно узнать с помощью функции **Ord(h)** — ее результат целого типа, число от 0 до 255. Таким образом, тип **Char** является ординальным (как и **Integer**).

И наоборот, зная код символа `n`, можно получить соответствующий символ с помощью функции **Chr(n)**. Это и есть один из способов напечатать символ, которого нет на клавиатуре. Понятно, что эти функции являются взаимно обратными: **chr(ord(h))=h** и **ord(chr(n))=n**.

### Задание

1. Напишите программу, которая печатает несколько разных символов, указанных их кодами.
2. Напишите программу, которая определяет коды некоторых символов.

Выполняя задание 1 на компьютере, вы можете заметить, что символы с некоторыми номерами не печатаются, а некоторые еще и производят с текстом на экране странные действия, например передвигают курсор. Дело в том, что символы с маленькими номерами так и называются «непечатаемые», они невидимы, являются управляющими (вы могли сталкиваться с невидимыми управляющими символами в Word'e).

Итак, символы имеют номера, их можно поставить «по порядку», например по возрастанию номеров. Как для любого ординального типа, существуют функции, определяющие для каждого символа (кроме первого и последнего) предыдущий символ — **pred** и последующий — **succ**.

```
Pred('2') = '1'; Succ('4') = '5'
```

Символы можно сравнивать, причем не только на «равно — не равно», но и на «больше — меньше». Сравнение происходит по номерам (кодам).

Мы уже говорили, что номера (коды) символов запоминать не надо, но некоторые общепринятые правила их упорядочения знать полезно. Символы-цифры пронумерованы друг за другом, символ **'0'** меньше символа **'1'**, который, в свою очередь, меньше символа **'2'** и т. п. Напомним еще раз, что символы-цифры, хоть и выглядят как числа, числами не являются, и, например, арифметические операции к ним неприменимы.

Кодировки символов в разных версиях операционных систем могут различаться. Но следующий закон сохраняется: сначала расположены цифры от 0 до 9, потом — по алфавиту большие латинские буквы, по алфавиту малые латинские буквы, и самые большие коды имеют символы национального алфавита, псевдографические символы (в том числе русские буквы). К сожалению, в некоторых кодировках русские буквы могут быть не упорядочены по алфавиту и даже могут идти не подряд.

***Задание.** Есть ли в той версии Паскаля, которой вы пользуетесь, русские буквы? Проверьте, расположены ли они подряд, упорядочены ли по алфавиту.*

Строки тоже можно сравнивать (при этом происходит последовательное сравнение соответствующих символов).

Существует достаточно много встроенных функций для работы с символами и строками, более того, строки и описывать можно несколько по-другому. Мы пока ограничимся приведенным здесь набором.

## Описания констант

Как уже говорилось ранее, константы — это изображения значений. Константы бывают числовые, логические, символьные и строковые.

Константе можно дать имя и использовать это имя для обращения к значению. Такую константу нужно описать заранее.

Описание констант производится там же, где и всех имен, т. е. в разделе описаний. Константы описываются с помощью служебного слова **Const**, после которого пишется имя константы, знак «=» и значение константы. Тип константы не указывается, о нем транслятор должен «догадаться» сам, исходя из изображения константы. Описания отделяются друг от друга точкой с запятой. Если описания нескольких констант следуют друг за другом, служебное слово можно не повторять. Далее приведены примеры.

```
Const Dlina=20.67;  
      Visota=3.567;  
Const Kolvo=100;
```

Константы отличаются от переменных тем, что в программе значение переменной можно изменить, а значение константы изменить нельзя. При попытке сделать это фиксируется ошибка.

## Оператор присваивания

Один из самых важных операторов языка — оператор присваивания — имеет непосредственное отношение к разговору о переменных, данных, типах. С помощью этого оператора переменные получают (меняют) значение.

Оператор выглядит так:

```
<переменная> := <выражение>
```

Здесь и далее, если в определении потребуется указать не конкретное слово или значение, а некоторое понятие, будем заключать его в угловые скобки.

С понятиями «переменная» и с самим знаком присваивания мы уже сталкивались в гл. 1, а понятие «выражение» имеет общепринятый смысл:



переменные и константы, связанные допустимыми знаками операций. Простейшими примерами выражений являются единичная переменная или константа.

Переменная и выражение должны быть совместимых типов. Нельзя переменной, описанной как **Char** (предназначенной для символов), присвоить значение арифметического выражения, а переменной целого типа — строку. В то же время мы не говорим, что типы обязаны совпадать. Дело вот в чем. Переменной типа **String** можно присвоить значение типа **Char**, так как символ — частный случай строки. Переменной типа **Real** можно присвоить значение типа **Integer**. Конечно, нельзя сказать, что целые числа являются частным случаем вещественных, просто при таком присваивании транслятор автоматически получившееся в правой части оператора целое число превращает в вещественное (создает ему дробную часть, равную нулю). Важно знать, что обратная операция автоматически не выполняется. Если результат какой-либо операции вещественный, его нельзя присвоить целой переменной, вначале надо его округлить с помощью специальных встроенных функций, о которых мы говорили выше.

Примеры правильных операторов присваивания мы не сможем показать, не указывая типа используемых в них переменных. Следовательно, придется написать программу, так как нужен раздел описаний. Эта программа ничего не делает, ее текст следует рассматривать лишь как пример правильных операторов присваивания с переменными разного типа.

```
Var H,C : Char;
    St : String;
    I,J : Integer;
    X,Y : Real;
Begin
    St:='123'; H:='%'; C:=H; St:=C;
    I:=ord(C); J:=I+1; H:=chr(J);
    X:=1.1; Y:=J*2;
    I:= 10 mod 3; X:=100/4
End.
```

В рамках тех же описаний неправильными операторами присваивания будут:

**St:=123** — строке нельзя присвоить число;

**H:='12'** — символьной переменной нельзя присвоить строку;

**I:=Ord(X)** — аргумент функции **Ord** должен быть целым числом;

**J:=I/2** — результат операции деления «/» всегда вещественный, его нельзя присвоить целой переменной.

На этом мы закончим список неправильных операторов присваивания — скоро вы самостоятельно начнете писать программы и сможете его пополнить.

## Пример программы с разными типами данных и операторами присваивания

Все представленные выше сведения о языке полезно проверить на практике, с помощью соответствующих программ. Приведем пример.

```
Program Types;
Const Stroka='#####';
      Ot='Ответ таков:';
Var M,N, Celoe,Ost : Integer; {Делимое, делитель, целая
                               часть и остаток от деления}
Var A,B,C:Real; {катеты и гипотенуза треугольника}
Begin Writeln (Stroka);
      M:=150; N:=45;
      Celoe:=M div N;
      Ost:=M mod N;
      Writeln(Ot);
      Writeln(Celoe);
      Writeln(Ost);
      Writeln(Stroka);
      A:=1.5; B:=2.9;
      C:=Sqrt (A*A+B*B); {Теорема Пифагора}
      Writeln(Ot);
      Writeln(C)

End.
```

В этой программе мы использовали две строковые константы (*Stroka* и *Ot*), что позволяет не выписывать каждый раз текст, который надо напечатать.

С помощью операторов присваивания сначала переменным придаются начальные значения, потом вычисляются значения арифметических выражений.

### Задание

1. Известно, что на нуль делить нельзя. А что сделает компьютер, если ему приказать поделить на нуль? Напишите программу, которая проверяет, что получится в этом случае. Перепишите и запомните сообщение компьютера. Когда была обнаружена ошибка — во время трансляции или во время выполнения программы?
2. Было сказано, что при делении с помощью операции «/» результат получается всегда вещественный. Попробуйте написать оператор присваивания, в котором результат такого деления присваивается целой переменной. Как реагирует компьютер? Перепишите сообщение, запомните его. На каком этапе компьютер обнаружил ошибку?

Присмотримся еще раз к нашей программе. В ней несколько операторов присваивания. Сначала некоторым переменным присваиваются начальные значения. В следующем операторе присваивания вычисляется значение другой переменной, которое зависит от начальных значений. А что случится, если мы вдруг забудем переменным присвоить начальные значения? Кто обнаружит ошибку? Будет ли компьютер производить вычисления с переменными, которым программист ничего не присвоил?

К сожалению, будет. Конечно, работа с переменной без значения является серьезной ошибкой, но не все трансляторы в состоянии ее обнаружить. Дело в том, что в памяти компьютера не бывает «пустоты», в каждом мельчайшем ее элементе обязательно хранится ноль или единица, не бывает так, чтобы там «не было ничего». Поэтому в том месте памяти, которое было отведено для хранения значения некоторой переменной, обязательно находится какое-нибудь значение. Часто вновь описанная переменная именно его и приобретает. Компьютер не отличает случайно приобретенного значения от того, которое дает переменной пользователь, и выполняет вычисления обычным образом. Естественно, результаты вычислений с такими случайными данными совершенно не похожи на правду.

Некоторые из используемых сейчас трансляторов часто числовым переменным без значения по умолчанию присваивают 0, что, естественно, не всегда совпадает с начальным значением, нужным программисту. Даже если начальные значения ваших переменных равны нулю, не присваивать им этот ноль «вручную», надеяться на транслятор совершенно неправильно.

Таким образом, не всегда надейтесь на помощь транслятора в поиске ошибок, тестируйте программу, проверяйте, правильные ли результаты она дает.

**Задание.** Составьте программу, в которой описывается несколько переменных разных типов. Выведите их, не присваивая им никаких значений. Что получилось?

## Задачи 5.1–5.17. Числа и формулы

5.1. Правильно или неправильно записаны числа?

0008; -0; 6,0; 5.; +0.2; .9;  $\frac{3}{4}$ ; E-1; 4E0; 0E-3; 3\*E7; pi.

5.2. Записать арифметические выражения на Паскале

$$\frac{ab}{c} + \frac{c}{ab}; \sin^2 x + \cos^2 x; \operatorname{tg} 2x; \operatorname{ctg} x/2; \sin \frac{x+y}{2};$$

$$\frac{a}{b \frac{c}{d}} - 0.0000003; |3x-2|; \sqrt{\cos x}; 35/2x; \sin x^2.$$

5.3. Выражение  $1E3+beta/(x2-gamma*delta)$  записать в общепринятой форме.

5.4. Посчитать результат арифметического выражения, записанного на Паскале:

$$24/(3*4)-24/3/4+24/3+4.$$

5.5. Вычислить:

```
(25 mod 5) div 2+ 100 div 2 mod 33;  
1234 div 1000; 1234 div 100; 1234 mod 10;  
12345 div 100 mod 10.
```

5.6. Концерт длился 245 минут. Написать на Паскале формулы, вычисляющие длительность концерта в часах и минутах.

5.7. Пусть задано 5-значное число. Написать на Паскале формулы, позволяющие получить по отдельности каждую из его цифр.

5.8. Задано некоторое вещественное число. Написать выражения для вычисления его целой и дробной частей.

5.9. Вспомнить формулу перевода радианной меры угла в градусную и наоборот. Написать соответствующие выражения на Паскале. Написать программу, вычисляющую тригонометрические функции от 30, 60, 90 градусов. Проверить, правильно ли она это делает.

5.10. Записать на Паскале формулы для вычисления в прямоугольном треугольнике:

- катета, если известна гипотенуза и другой катет;
- гипотенузы, если известен один из катетов и один из острых углов;
- периметра, если известна гипотенуза и один из катетов.

5.11. Записать на Паскале формулы для вычисления длины окружности и площади круга.

5.12. Написать программу, которая присваивает некоторой переменной среднее арифметическое двух чисел.

5.13. Написать программу, которая печатает среднее арифметическое трех чисел.

5.14. С помощью минимального количества операторов присваивания переменной  $A$  присвоить значение  $x^8$ .

5.15. Написать программу, печатающую символы с кодами 100, 150, 200 и 250.

5.16. Написать программу, печатающую коды цифр. Убедиться, что код цифры не равен самой цифре. Убедиться, что коды цифр идут подряд.

5.17. Написать программу, печатающую коды некоторых русских и латинских букв. Найти формулу, связывающую большую латинскую букву и соответствующую ей маленькую.

## Глава 6

# Ввод с клавиатуры и вывод на экран

---

Ввод и вывод служат для связи программы с внешним миром: таким образом она получает начальные данные и выдает ответ. Программа без вывода (под выводом понимается не только отображение результатов на экране) бессмысленна: зачем же что-то считать, если результаты работы никто не видит и они пропадают. Вот и в нашей первой программе был оператор вывода.

Операторы ввода и вывода в Паскале правильнее было бы называть «процедуры», но мы пока не знаем точного значения этого термина в языках программирования, так что оставим такое, не совсем корректное название. Операторы ввода-вывода служат для обмена информацией между программой и какими-либо внешними устройствами. Можно ввести данные из файла, с клавиатуры, можно вывести в файл, на принтер, на экран. Пока нас будет интересовать только ввод с клавиатуры и вывод на экран. Позже мы узнаем, как с помощью этих операторов работать с файлами.

Операторы выглядят так.

Оператор ввода:     **Read** и **ReadLn**     (*Read* по-английски означает «читать»).

Оператор вывода:   **Write** и **WriteLn**   (*Write* по-английски означает «писать»).

Дополнение «**ln**» в операторах происходит от слова *line* — линия, строка. Оператор с этим окончанием после своих действий переводит строку в отличие от своего «собрата». Подробнее: после работы **Write** (печати на экране) курсор остается справа от последнего напечатанного символа, а после **WriteLn** будет переведен на новую строку, в ее начало. После работы **Read** очередные данные будут считываться из той же строки, а после **ReadLn** будут считываться с новой, игнорируя незадействованные значения в первой строке.

Операторы могут использоваться без параметров (в Турбо Паскале пишется просто соответствующее слово и ничего больше, а во Free Паскале еще добавляются скобочки) и с параметрами. Параметры — список ввода (вывода) — записываются после оператора в круглых скобках. Друг от дру-

га параметры в списке отделяются запятыми. Количество параметров, вообще говоря, может быть любым. Например:

```
Read(X,Y,Z); Readln(Alfa); Write(X, Y, Z);  
Writeln (Alfa); Writeln
```

## Оператор ввода

В списке ввода могут находиться только переменные. Встретив в программе оператор ввода, компьютер останавливается и ждет, когда с клавиатуры будут введены данные (цифры, символы). Если вы работаете в Турбо Паскале, вводимые символы сразу попадают на экран. В ABC-Паскале они сначала печатаются в специальном окошке «Ввод данных», а потом помещаются на экран. Ввод завершается нажатием **Enter**, работа программы продолжается. Введенные значения последовательно присваиваются переменным, перечисленным в скобках. Если переменная до этого имела какое-то значение, оно утрачивается, заменяется новым, введенным.

Тип вводимого значения должен совпадать с типом переменной, которой он присваивается: **Integer**, **Real** или **Char**, **String**.

Например, если переменная *X* описана как **Real**, при вводе **Read(X)** можно набирать только числа, как вещественные, так и целые (они понимаются как вещественные с нулевой дробной частью). Если же ввести символ, при попытке присвоить его значение переменной *X* возникнет ошибка, программа прекратит работу. На экране появится окно редактора, а в нем — красная строка с сообщением об ошибке «неверный числовой формат».

Если переменная описана как **Integer**, вводить можно только целые числа. А вот в случае, когда переменная имеет тип **Char**, вводить можно и буквы, и цифры. Только надо иметь в виду, что все введенное будет понято как символ. Для типа **String** можно вводить несколько любых символов.

С помощью одного оператора **Read** можно ввести несколько значений: **Read(A, C, N)**. При вводе числа должны отделяться друг от друга пробелами, символы записываются подряд. Вот здесь возникают некоторые неприятности. Во-первых, очередность вводимых значений легко перепутать. Во-вторых, можно ошибиться и ввести несколько лишних значений — какие из них будут приняты компьютером — первые или последние? Неприятности особого рода возникают при вводе одним оператором **Read** нескольких символьных величин (типа **Char**). Если при этом по ошибке ввести символы не подряд (в одну строку), а по одному (нажимая **Enter** после каждого символа) или разделяя их чем-либо (пробелом, запятой), то разделитель или **Enter** будет принят программой за очередной вводимый символ. Соответственно, она работать будет не так, как хотелось бы автору.

Чтобы подобных проблем не возникало, рекомендуем на первых порах каждым оператором ввода вводить только одно значение и пользоваться оператором **Readln**. Позже, по мере приобретения опыта программирова-

ния, мы будем отмечать случаи, когда удобно (или даже необходимо) ввести сразу несколько значений, а также более точно проясним разницу между **Read** и **Readln**.

Выше говорилось, что оператор ввода может использоваться без параметров: **Readln**. Зачем это нужно, если никакой информации в компьютер не поступает? Вспомним, что при выполнении этого оператора программа останавливается и ждет ввода. В данном случае, так как список ввода пуст, она ждет просто нажатия клавиши **Enter**. Оператор **Readln** без параметров может использоваться для организации задержки — программа не выполняется, картинка на экране не меняется, пока человек не нажмет на клавиатуре **Enter**. При работе в Турбо Паскале часто такой оператор ставят в конце программы, например для ее отладки. Тогда до тех пор, пока не нажата **Enter**, выполнение программы не заканчивается, переход в окно редактора не происходит, и можно, пока не надоест, любоваться результатами работы своей программы.

## Оператор вывода

В списке вывода должны быть объекты только тех типов, которые можно выводить. Да, да, бывают типы данных, значения которых нельзя выводить на экран. Но мы с вами такие пока не изучали. То, о чем мы уже говорили, — числа, символы и строки — на экран выводить можно.

В качестве элемента списка вывода могут фигурировать:

- переменные — `Writeln(Summa);`
- константы (числовые, символьные, строковые) — `Writeln(25);`  
`Writeln('X'); Writeln('Ответ');`
- выражения, в частности арифметические — `Writeln(sin(2*X) / 35+1);`  
`Writeln(M div N mod 3).`

Как уже говорилось, операторы **Writeln** и **Write** различаются тем, что первый, напечатав на экране все элементы своего списка вывода, осуществляет перевод строки (следующий оператор вывода будет печатать значения уже на новой строчке), а второй оставляет курсор в той позиции, где закончился вывод (следующий оператор вывода будет писать свои значения на той же строчке, рядом с напечатанными ранее). Два подряд оператора **Write(X); Write(Y)** можно заменить одним **Write(X,Y)** — работать они будут совершенно одинаково.

Оператор вывода может не иметь параметров. В этом случае пишутся только сами слова **Writeln** и **Write** (в Турбо Паскале скобки не ставятся, а во Free и ABC-Паскале ставятся «пустые» скобки). **Writeln** без списка вывода переводит курсор на следующую строку (таким образом, выполнив 20 раз **Writeln** можно очистить экран). **Write** без списка вывода не делает ничего, «печатает пустоту». Зачем он нужен? Можно таким образом пометить, что потом в этом месте будет какой-то вывод.

Итак, количество параметров в списке вывода может быть от нуля до бесконечности. Ну, конечно же, не до бесконечности, текст программы не может быть бесконечным, но, строго говоря, их количество никакими правилами не ограничено.

Посмотрим, как работает оператор вывода, у которого в списке вывода несколько элементов:

Оператор	Напечатает
<code>Writeln (1,2,3)</code>	123
<code>Writeln(100,200)</code>	100200
<code>Writeln(10, 'умножить', 'на', '5')</code>	10умножитьна5

Почему в последнем примере получается так некрасиво и непонятно, почему элементы списка вывода «склеиваются» между собой? Да потому, что это мы сами так приказали компьютеру их напечатать. Если мы хотим, чтобы элементы вывода разделялись, например, пробелом, надо этот пробел явно указывать (выводить его как символ внутри кавычек). Например (для наглядности заменим здесь пробел знаком «нижнее подчеркивание»):

```
Writeln(1, '_', 2, '_', 3)
Writeln(10, '_умножить_на_', '5')
```

## Форматный вывод

Для того чтобы сделать текст на экране понятным, а числа удобочитаемыми, хорошо расположенными, существует так называемый **форматный вывод**. В форматном выводе указывается количество позиций, которые при выводе надо отвести под выводимое число, символ или строку.

Для вывода символов, строк и целых чисел формат — это одно целое число. Оно указывается после выводимого элемента через двоеточие. Именно столько позиций будет отведено под выводимое значение. Выравнивание происходит по правому краю. То есть если остаются «лишние» позиции (число трехзначное, а под него отведено 7 позиций), то сначала выводится 4 пробела, а потом 3 цифры числа. Если по ошибке под выводимый элемент отводится меньше позиций, чем ему минимально необходимо, формат игнорируется.

### Примеры

### Будет напечатано

```
A:=12345; S:='abcde';
Writeln(A:5;S:10);
Writeln(A:10,S:5);
```

1	2	3	4	5						a	b	c	d	e
					1	2	3	4	5	a	b	c	d	e

Для вывода вещественных чисел используется формат из двух чисел, разделенных двоеточием. Например, **х:10:2**. Первое число — сколько



позиций отводится для всего числа, второе число — количество цифр после точки. Обратите внимание, что никакого округления при отбрасывании «лишних» цифр не происходит, они просто «отрезаются». Понятно, что для правильного вывода (если формат неверный, он игнорируется) необходимо, чтобы первое число формата было больше второго, ведь кроме дробной части в числе есть целая часть, точка, возможно, знак числа.

Если разные вещественные числа выводить в нескольких строках в одинаковом формате, они будут выводиться, как это принято при написании их в столбик: точка под точкой, соответствующие разряды друг под другом (см. две последние строчки примера).

Примеры

Будет напечатано

```
X:=123.456789; Y:=9.8;  
Writeln(X:10:1) .  
Writeln(X:10:0);  
Writeln(X:10:4);  
Writeln(Y:10:4);
```

					1	2	3	.	4
							1	2	3
		1	2	3	.	4	5	6	7
				9	.	8	0	0	0

Для вывода вещественных чисел формат применяется особенно часто, уж очень «некрасиво» они выводятся по умолчанию. Если при выводе вещественного числа не указать формат, оно будет напечатано в виде с плавающей запятой. Например, `Writeln(300/3)` напечатает нам не 100 и не 100.0, а 1.000000000000E+02. Использование формата значительно увеличивает наглядность: `Writeln(300/3:10:1)` печатает 100.0.

# Грамотное использование операторов ввода и вывода

Мы уже говорили, что будем пристальное внимание уделять «красоте» программы, ее понятности. Результат работы программы — то, что остается на экране после ее работы, — должен выглядеть так, чтобы было понятно, что программа делала. Хорошо выглядит программа, работа которой начинается с оператора вывода, поясняющего ее назначение. Например: «Решение квадратного уравнения». В таком случае и текст программы становится понятнее (ведь строку для вывода мы видим и в тексте при редактировании, и на рабочем экране при выводе).

Хорошая программа не только правильно производит вычисления, но и еще красиво их распечатывает. Плохо, если результатом программы является просто набор чисел (даже разделенных запятыми или пробелами), — вы вскоре забудете, что эти числа означают: скорость и расстояние, объем и площадь, синус и косинус? Выводимые значения следует пояснять: что каждое из них означает, для каких параметров подсчитано и т. п.

Особое внимание следует уделить «понятности» ввода. Если начать программу с оператора ввода (что часто, по сути, и бывает нужно), человек, сидящий за компьютером, может просто не понять, что от него требуется. Компьютер, увидев оператор **Readln**, остановится и будет ждать ввода нужного количества значений, а человек будет сидеть перед экраном и ждать, когда же на нем что-нибудь появится. Так и будут «смотреть» друг на друга... В конце концов, может быть, человек догадается, что он должен что-то ввести. Но что? Числа? Символы? В каком порядке? Поэтому хорошо, когда операторы ввода предваряются подсказкой: что требуется ввести. Вводить каждым оператором желательно одно значение. Отступление от этих правил может привести к ошибкам ввода (человек может перепутать, в каком порядке надо вводить значения, ввести вместо чисел символы и т. п.).

Пояснения к вводу, как и пояснения к ответу, должны быть понятными, совпадающими с условиями задачи, а не содержать имена придуманных вами переменных, которые используются внутри программы. Например, если в программе по двум сторонам треугольника и углу ищется третья сторона, а вы назвали стороны *X*, *Y*, *Z*, не следует писать подсказку «**Введите X**» или выводить результат «**Ответ Z=**». Работающему с программой (в том числе и ее автору — спустя несколько дней после написания) для правильного ввода придется смотреть в текст программы — что же там этими буквами обозначено. Буквы со знаком равенства в подсказке (**Write('R=')**) уместны, когда их назначение пояснено предыдущими операторами вывода. Например, если вы напечатали название задачи «**Решение линейного уравнения  $KX+B=0$** », то после этого можно использовать подсказки «**K=**» и «**B=**».

Подсказку к вводу обычно выводят с помощью оператора **Write**, а не **Writeln**, в этом случае текст подсказки и введенное значение находятся рядом, на одной строчке.

В операторах вывода строк используйте пробелы в конце строки (ставится внутри апострофов), чтобы последующий объект не «приклеился» к выведенной строке.

## Примеры программ с вводом-выводом

*Пример 6.1. Найти сумму цифр трехзначного числа (число целое, положительное).*

Кроме ввода-вывода для решения этой задачи нужен алгоритм, который не настолько очевиден, чтобы не сказать о нем хоть немного. Цифры числа можно искать с помощью операций целочисленного деления **mod** и **div**. Остаток от деления любого целого положительного десятичного числа на 10 дает его последнюю цифру (операция **mod**). Неполное частное при делении этого числа на 10 — первоначальное число без последней цифры.

Вторую цифру исходного числа можно искать, используя такое «усеченное» число, определяя его последнюю цифру. Чтобы узнать число сотен в числе (первую цифру), надо найти неполное частное от деления его на 100.

Можно действовать и другими способами. Например, сначала найти число сотен, а потом уменьшить число, отняв от него сотни, оставив две последние цифры.

```
Program Summa_Cifr;
Var K, C1, C2, C3 : Integer; {число и цифры слева направо}
    S : Integer; {сумма цифр}
Begin Writeln('сумма цифр трехзначного числа');
    Write('Введите трехзначное число '); Readln(K);
    C3:=K mod 10;
    C1:=K div 100;
    C2:=(K div 10) mod 10; {Скобки здесь необязательны,
                           они для большей наглядности}
    S:=C1+C2+C3;
    Writeln(C1, '+', C2, '+', C3, '=', S)
End.
```

Отметим, что эта программа будет правильно работать, только если вводить именно трехзначное число. Позже мы напишем похожую программу, которая будет работать и с другими числами.

Давайте повнимательнее посмотрим, как организован список вывода у последнего оператора **Writeln**. В нем 7 элементов, все они отделяются друг от друга запятыми. Некоторые из них в апострофах, некоторые — без. Как понять, что писать в апострофах, а что без них? Если сразу разобраться сложно, советуем поступить вот так: выпишите, что получится при выводе для каких-нибудь заданных значений. Например, если ввести число 203, выведено должно быть  $2 + 0 + 3 = 5$ . Если ввести другое число, будет выведена другая строка (при 162 получится  $1 + 6 + 2 = 9$ ). Что в ней изменится, а что останется таким же, как и в первой строке?

Подчеркнем те элементы, которые изменяются, волнистой чертой (это переменные), а те, которые постоянны, — прямой чертой (это константы). Ну, а теперь все совсем просто. Переменные в списке вывода записываются без апострофов — тогда на экран выведется не имя переменной, а ее значение, а вот символьные и строковые константы, содержащие текст для вывода, — в апострофах. Мы запишем это в табличке, чтобы было ясно видно, что как подчеркнуто:

2	+	0	+	3	=	5
1	+	6	+	2	=	9
~	—	~	—	~	—	~

**Пример 6.2.** Дана площадь круга. Определить радиус этого круга и длину окружности.

Обратите внимание, что в этой задаче все величины — вещественные. Для написания программы понадобится вспомнить формулы для вычисления площади круга и длины окружности, а также число  $\pi$ . В Турбо Паскале есть такая стандартная константа, она называется **Pi**.

```
Program Krug;  
Var R, C, S : Real; {радиус, длина окружности  
                    и площадь круга}  
Begin Write ('Введите площадь круга '); Readln(S);  
      R:=SQRT(S/Pi);  
      C:=2*Pi*R;  
      Writeln('Радиус круга площади', S:8:4, ' равен ',  
              R:5:2);  
      Writeln('Длина окружности радиуса', R:5:2, ' равна',  
              C:6:2)  
End.
```

**Пример 6.3.** По одной букве ввести слово из четырех букв. Составить новое «слово» из 3-й, 4-й и 1-й букв (именно в таком порядке), напечатать его. Например, если ввести слово «кора», напечатается «рак».

```
Program Slovo;  
Var B1,B2,B3,B4 : Char; {буквы слова}  
Begin Writeln('Введите слово из 4 букв по одной букве');  
      Write ('Введите 1-ю букву '); Readln(B1);  
      Write ('Введите 2-ю букву '); Readln(B2);  
      Write ('Введите 3-ю букву '); Readln(B3);  
      Write ('Введите 4-ю букву '); Readln(B4);  
      Writeln('Новое слово');  
      Writeln(B3:10,B4,B1)  
End.
```

Ввод в этой программе можно оформить и короче.

```
Program Slovo1;  
Var B1,B2,B3,B4 : Char; {буквы слова}  
Begin Write('Введите слово из 4 букв ');  
      Readln(B1,B2,B3,B4);  
      Writeln('Новое слово');  
      Writeln(B3:10,B4,B1)  
End.
```

Смотрится это гораздо эффектнее: и текст, и работа программы. Ведь человек вводит слово целиком, а компьютер как бы сам разбивает его на буквы. В первом случае вводятся именно 4 буквы (у нас в программе

4 оператора ввода, человек каждый раз знает, какую букву надо вводить). Во втором случае человек может ввести любое слово. Попробуйте! Если букв в слове будет больше четырех, компьютер отберет 4 первые и будет с ними работать, а вот если меньше четырех, он остановится, ничего не напишет, а просто будет ждать ввода недостающих букв.

## Задачи 6.1–6.27. Ввод и вывод

- 6.1. Во дворе гуляют кролики и куры (количество тех и других вводится с клавиатуры). Все они здоровы и полноценны. Сколько ног топчет двор?
- 6.2. Известно время начала и конца митинга. Время в часах и минутах вводится с клавиатуры. Сколько минут длился митинг? (Митинг проходил в течение одних суток, часы считаются от 0 до 23.)
- 6.3. Вводится скорость ветра в метрах в секунду. Сколько это в километрах в час?
- 6.4. Вводятся длины сторон прямоугольника. Найти его площадь и периметр.
- 6.5. Известна площадь квадрата. Найти сторону и периметр.
- 6.6. Вводятся длины трех сторон треугольника (внимание — вводите «правильные числа» — треугольник можно построить не всегда). Найти площадь и периметр такого треугольника.
- 6.7. Вводится величина угла в градусах. Найти его  $\sin$  и  $\cos$ . (Напомним, что в Паскале аргумент тригонометрических функций задается в радианах.)
- 6.8. Дана гипотенуза равнобедренного прямоугольного треугольника. Напечатать значения всех его сторон, площадь и периметр.
- 6.9. В магазине купили несколько пакетов сахара и несколько коробок конфет. Цены, вес, количество — все вводится с клавиатуры (продумать, какого типа переменные надо использовать). Требуется вычислить стоимость покупки и общий вес всех товаров.
- 6.10. Автомобиль ехал несколько часов со скоростью  $V_1$  км/ч и потом несколько минут со скоростью  $V_2$  км/ч. Значения времени и скорости вводятся с клавиатуры. Сколько километров он проехал? Сколько времени ушло на дорогу? Какова средняя скорость?
- 6.11. Решить уравнение  $kX + B = 0$  ( $k$  и  $B$  не равны 0). Коэффициенты вводятся с клавиатуры. Напечатать получившееся уравнение (вместо коэффициентов  $k$  и  $B$  подставить введенные числа). Найти  $X$ .
- 6.12. Найти дискриминант квадратного уравнения  $aX^2 + bX + c = 0$  ( $a < > 0$ ;  $b, c > 0$ ). Напечатать получившееся уравнение (вместо коэффициентов  $a, b$  и  $c$  подставить введенные числа).
- 6.13. Из зарплаты работника вычитается  $K$  процентов в качестве налога. (Размер зарплаты в рублях и  $K$  вводятся с клавиатуры.) Сколько денег он получит на руки?

- 6.14. Известно, сколько в доме этажей и сколько квартир на каждой площадке в подъезде (значения вводятся с клавиатуры). Определить, в каком подъезде и на каком этаже находится заданная квартира.
- 6.15. С дерева собрали некоторое количество яблок. Их разделили поровну между детьми, гулявшими в саду (яблоки не резали), а оставшиеся отдали, чтобы сварить компот. Сколько яблок досталось каждому ребенку? Сколько пошло на компот?
- 6.16. Известно, что 1 января пришлось на понедельник (1-й день недели). С клавиатуры вводится дата — номер дня в январе ( $\leq 31$ ). На какой по счету день недели он придется?
- 6.17. Вводится четырехзначное натуральное число. Найти произведение его цифр.
- 6.18. Вводится трехзначное натуральное число. Составить число из его цифр в обратном порядке (не напечатать цифры рядом, а именно получить нужное число, а затем напечатать его).
- 6.19. Вводится четырехзначное натуральное число. Выписать, сколько в нем тысяч, сотен, десятков, единиц.
- 6.20. Вводится трехзначное число и цифра. Заменить вторую цифру числа на введенную. Пример: 123 и 6  $\rightarrow$  163.
- 6.21\*. Вводится двузначное число и цифра. Создать пятизначное число, добавив цифру в начало, в конец числа и между цифрами. Пример: 12 и 3  $\rightarrow$  31323. Для числа-ответа используйте тип `LongInt`.
- 6.22\*. Вводится четырехзначное число. Создать два трехзначных числа путем удаления из введенного числа сначала второй, а потом третьей цифры. Пример: 1234  $\rightarrow$  134 и 124.
- 6.23. Вводится символ. Напечатать его номер (код), предыдущий и следующий за ним символ.
- 6.24. Вводится слово из четырех букв. Составить «слово» из символов, каждый из которых является следующим по порядку для букв введенного слова.
- 6.25. Вводится слово из четырех букв. Напечатать «перевернутое» слово.
- 6.26. Компьютер спрашивает человека, как его зовут, а потом печатает, обращаясь по имени:
- «Здравствуй, <Имя>!», «<Имя> пишет программу»
- 6.27\*. С клавиатуры вводятся 4 символа-цифры (в процедуре ввода используются переменные типа **Char**, ввод всегда «правильный»). Составить из введенных цифр 2 различных числа (любых) и найти их разность.

# Глава 7

## Разветвления

---

### Условный оператор

Записывая алгоритмы с помощью блок-схем, мы видели, что часто разветвление выглядит так: указываются действия, которые надо делать в случае выполнения условия, и действия, которые надо сделать, если условие не выполнено (рис. 7.1).

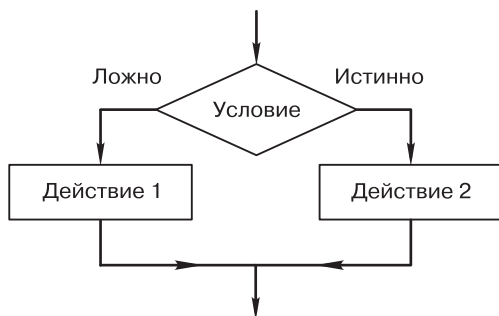


Рис. 7.1

В Паскале для реализации такой блок-схемы служит условный оператор, который в общем виде выглядит так:

```
If <условие> Then <Оператор1> Else <Оператор2>
```

Обратите внимание, что никаких знаков (в том числе «;» перед **Else**) в этой конструкции нет, «;» ставится между операторами, а здесь вся строка — один оператор.

В операторе используются служебные слова **If** — «если», **Then** — «то», **Else** — «иначе». Если условие истинно, выполняется **<Оператор1>**, если нет — **<Оператор2>**.

Условие (или логическое выражение) — это некоторое выражение, которое может принимать значение «истина» или «ложь». Пока в качестве логических выражений будем использовать отношения: выражения, в которых две части (символы, константы или выражения), связаны одной из операций логического отношения (в математике это операции сравнения). Позже

мы узнаем, что логические выражения могут быть достаточно длинными и сложными.

Значения обеих частей в отношении должны быть либо числового типа (**Integer** или **Real**), либо символьного (**Char**, **String**). Сравнивать символ с числом нельзя! В отношении может присутствовать только один знак сравнения:

$X \geq 1$ ,  $Y/2 < 3 \cdot X - 1$ ,  $A = B$ ,  $H < \#$  — правильные отношения;  
 $-1 < X < 1$ ,  $A = B = C$ ,  $5 = '5'$  — неправильные отношения.

**<Оператор1>** и **<Оператор2>** — это любые операторы языка. Например, операторы присваивания или вывода или даже опять условный:

```
If X <> 0 Then Y := 1/X Else Writeln('Нет решения');  
  
If X > 0 Then Writeln(X, '- положительное')  
           Else Writeln(X, '- отрицательное или 0');  
  
If A > B Then If A > C Then Writeln (A, '- наибольшее')  
              Else Writeln (C, '- наибольшее')  
           Else If B > C Then Writeln (B, '- наибольшее')  
              Else Writeln (C, '- наибольшее');
```

Как видите, **Else** в программах часто бывает удобно писать под **Then**.

Посмотрим, какие задачи решаются в предложенных выше примерах. В первом примере вычисляется число, обратное к  $X$ , причем учитывается, что сделать это можно только в том случае, когда  $X$  не равно 0. Во втором случае определяется, является ли  $X$  положительным числом. А в третьем примере находится наибольшее из трех чисел. Этому оператору соответствует следующая блок-схема (рис. 7.2)

Оба оператора (и после **Then**, и после **Else**) могут быть пустыми. Пустой оператор — это «пустота» и есть, он никак не записывается и не выполняет никаких действий. Но так как в Паскале между операторами ставится знак «;», то, если в тексте есть пустые операторы, иногда появляются дополнительные знаки «;». Например  **$X := 1$ ;  $Y := 2$**  — здесь 3 оператора. Таким образом, запись:

```
If X > 0 Then Else X := -X
```

совершенно правильная, между **Then** и **Else** находится пустой оператор.

Следующая запись тоже совершенно правильная:

```
If X < 0 Then X := -X Else; Writeln(X) (1)
```

А означает она вот что: если  $X$  отрицательное, у него изменится знак, в противном случае никаких действий сделано не будет. После условного оператора стоит оператор вывода, поэтому значение  $X$  выведется в любом случае, какое бы оно ни было.



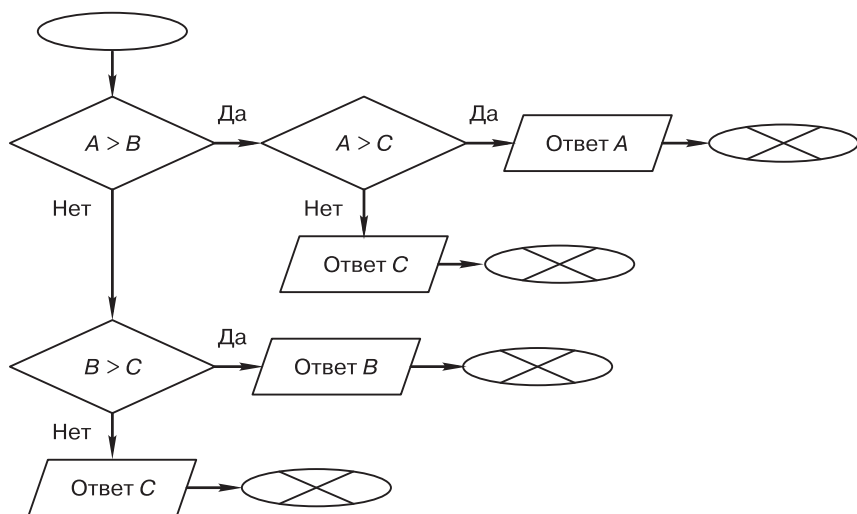


Рис. 7.2

Сравните эту запись с очень похожим фрагментом (отсутствует всего один знак — какой?)

If  $X < 0$  Then  $X := -X$  Else Writeln( $X$ ) (2)

Здесь работа будет вестись по-другому: если  $X$  отрицательное, у него изменится знак, а если неотрицательное — оно напечатается. Отрицательное  $X$  печататься не будет — здесь, в отличие от первого фрагмента, оператор **Writeln( $X$ )** находится внутри условного оператора, как действие после **Else**. В первом фрагменте после **Else** — пустой оператор, а оператор **Writeln** — самостоятельный оператор, который стоит после условного.

Посмотрите (рис. 7.3), как различаются блок-схемы для первого и второго рассмотренных фрагментов (слева и справа соответственно):

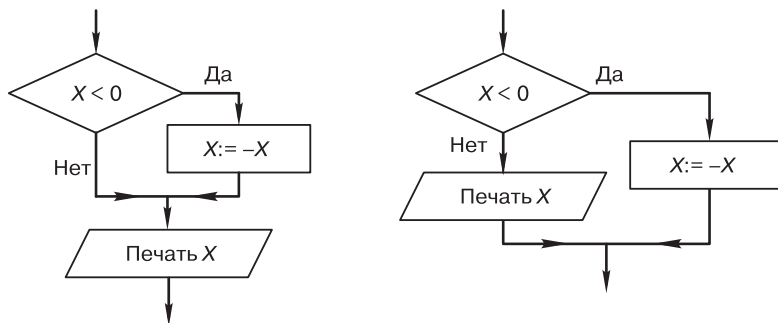


Рис. 7.3

Впрочем, для тех случаев, когда «иначе» (**Else**) делать ничего не надо (при изучении блок-схем мы называли такие ветвления «обходом»), существует неполная форма условного оператора. Последняя часть **Else <Оператор2>** опускается:

```
If X<0 Then X:=-X; Writeln(X)
```

Это аналог фрагмента 1 с неполной формой условного оператора. Работать он будет точно так же — напечатает  $|X|$ .

## Составной оператор

Что произойдет, если вместо одного оператора ничего не написать, мы разобрались. А вот как быть, если в том месте, где «по закону» положен только один оператор, нужно написать несколько? Ведь по правилам языка в условном операторе и после **Then**, и после **Else** допускается писать только один оператор!!! Несложно придумать соответствующую задачу, и можно видеть на примерах блок-схем из первого раздела книги, что нередко и на одной, и на другой «веточке» «висит» несколько блоков. В этом случае надо несколько операторов «превратить» в один. Это делается с помощью так называемых «операторных скобок» **Begin—End**.

Получается конструкция:

```
Begin  
    Оператор1;  
    Оператор2;  
    и т.д.  
End
```

Сколько бы внутри такой конструкции операторов ни было, для языка она превращается в один так называемый **составной оператор**. Превращать много операторов в один требуется не только для использования этой конструкции в условном операторе, так что составной оператор бывает нужен очень часто.

Записывать составные операторы удобно «лесенкой»: **Begin** и **End** писать друг под другом, а операторы, входящие в составной, — с небольшим отступом вправо. Так будет видно, где оператор закончился.

Паскаль-редактор позволяет удобно вводить составные операторы в тексте программы. Можно написать **Begin** и **End**, поставить курсор между ними, нажать **Enter** — **End** окажется на следующей строчке под **Begin**. После этого поставить курсор после **Begin** и вставлять операторы между операторными скобками. Кстати, такой способ ввода текста позволит сократить количество ошибок, связанных с несоответствием количества слов **Begin** и **End**.

В условном операторе после **Then** и **Else** могут стоять любые операторы, в том числе и условные.

```
If X<>0 Then If X>0 Then Writeln ('положительное')  
                Else Writeln ('отрицательное')
```

В этом фрагменте по два **If** и **Then** и всего один **Else**. В Паскале четко определено (чтобы не было никаких неоднозначностей), что в этом случае **Else** относится к идущей непосредственно перед ним конструкции **If-Then**, как это и изображено с помощью отступов во фрагменте. Напомним, кстати, что отступы при записи текста программы мы делаем для себя, а не для транслятора, неправильную конструкцию правильные отступы не спасут.

Этот фрагмент можно записать и по-другому. В подобных конструкциях внутренний условный оператор часто заключают в операторные скобки, несмотря на то, что он всего один. Это делают, чтобы не перепутать внешний оператор с внутренним (особенно легко запутаться, если один из операторов укороченный).

```
If X<>0 Then Begin If X>0 Then Writeln ('положительное')  
                    Else Writeln ('отрицательное')  
                End
```

Этим фрагментам соответствует одна и та же блок-схема (рис. 7.4):

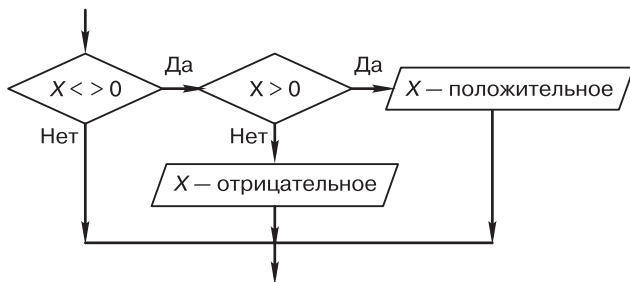


Рис. 7.4

Если же мы хотим, чтобы блок-схема была другой (рис. 7.5), чтобы **Else** соответствовал первому **Then**, операторные скобки надо поставить в другом месте:

```
If X<>0 Then Begin If X>0 Then Writeln ('положительное')  
                End  
                Else Writeln ('нулевое')
```

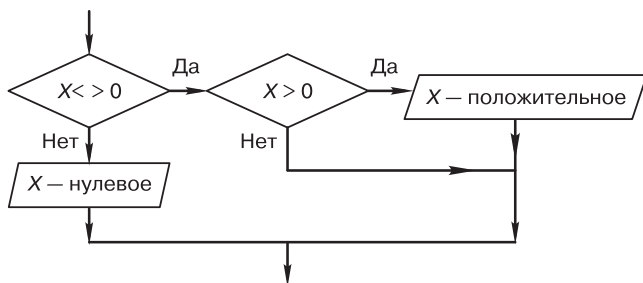


Рис. 7.5

Обратите внимание, и в том, и в другом случае существуют значения  $X$ , при которых не выводится никакого «ответа»: в первом случае это  $X = 0$ , во втором это  $X < 0$ .

## Решение задач с условным оператором

*Пример 7.1. Решим задачу про треугольник (пример 1.2 из гл. 1).*

Посмотрев на блок-схему, видим, что фразу «Треугольник построить нельзя» придется выводить три раза. Заведем строковую константу с таким текстом.

```

Program Treug;
  Const Net='Треугольник построить нельзя';
  Var  A,B,C: Real;
Begin Writeln ('Можно ли построить треугольник с такими
                                                    сторонами');
  Writeln ('Введите 3 числа');
  Write('1-е число '); Readln(A);
  Write('2-е число '); Readln(B);
  Write('3-е число '); Readln(C);
  If A<B+C Then
    If B<A+C Then
      If C<A+B Then Writeln ('Треугольник построить
                                                    можно')
                    Else Writeln(Net)
                    Else Writeln (Net)
                    Else Writeln(Net)
  End.

```

**Задание.** Программу для примера 1.1 из гл. 1 напишите самостоятельно.

**Пример 7.2.** С клавиатуры вводится вещественное значение  $X$ . Напечатать значение  $Y$ , если оно определено.

$$Y := \begin{cases} 1/X^8 & \text{при } X > 0; \\ -1/X^8 & \text{при } X < 0. \end{cases}$$

```

Program Func;
Var  X,Y,Z: Real;
Begin Writeln ('Вычисление значения функции ');
      Write ('Введите аргумент '); Readln(X);
      If X=0 Then Writeln('В этой точке функция не определена')
      Else Begin  Z:=SQR(SQR(SQR(X)));
                  Y:=1/Z;
                  If X<0 Then Y:=-Y;
                  Writeln('при X=',X:4:1,
                           ' значение функции =', Y:30:20)
      End
End.

```

**Задание.** Начертите блок-схему этого алгоритма.

Для того чтобы найти  $X^8$  (напомним, в Паскале нет операции «возведение в степень»), введем дополнительную переменную  $Z$ .

Сначала «разделаемся» со случаем, когда функция не существует, — это будет одна из ветвей условного оператора. В другой ветви будет составной оператор, внутри которого мы подсчитаем значение функции и напечатаем его.

Обратите внимание на формат вывода. Для  $Y$  надо оставить больше позиций для цифр после десятичной точки, ведь число может получиться достаточно маленьким. Попробуйте ввести число 500 — в ответе получится 0. Почему? Ведь наша дробь никогда не может равняться нулю. Она и не равна, это так называемый «машинный нуль». Мы выводим только 20 цифр после запятой, а цифры, не равные нулю, в нашем числе оказались дальше, вот мы и видим одни нули. Как быть? Если предполагается, что могут получаться такие маленькие числа, следует использовать соответствующий формат вывода.

А с маленькими аргументами получается еще хуже! Введите 0.00001. В программе возникает ошибка — так называемая «ошибка времени вы-

полнения». На рабочем, «черномом» экране появляется надпись «Run-time error ...», а в окне редактора — красная строка-предупреждение «Floating point overflow» — переполнение при работе с вещественными числами. Число  $X^8$  при введенном значении оказалось очень маленьким и при выполнении операции деления произошло так называемое «переполнение», получилось слишком большое число. Как быть? Если такие числа действительно нужны для вычислений, придется воспользоваться не типом **Real**, а другим вещественным типом, диапазон значений которого больше, например типом **Double** или **Extended**. Если же работа с такими числами не нужна, можно считать, что функция не определена не только при  $X=0$ , но и при всех  $X$ , достаточно близких к 0. То есть первый условный оператор будет выглядеть, например, так:

```
If Abs(X) < 0.00002 ...
```

А чтобы красиво напечатать получившееся число, желательно при больших и маленьких ( $>1$  и  $<1$ ) значениях аргумента пользоваться разными форматами вывода (для этого придется написать целиком другой оператор вывода). *Сделайте это самостоятельно.*

**Пример 7.3.** Пусть зависимость  $Y$  от  $X$  задана графиком (рис. 7.6). Написать программу, в которой вводится с клавиатуры значение  $X$ , а выводится соответствующее ему значение  $Y$ .

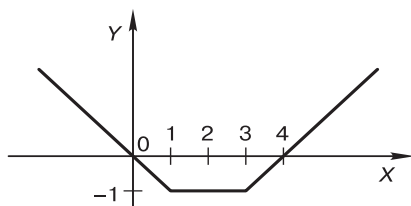


Рис. 7.6

В данном случае совершенно все равно, в каком порядке рассматривать части функции, но все же будем это делать слева направо.

Обратите внимание (рис. 7.7): во втором блоке-условии мы не проверяем принадлежность  $X$  всему промежутку  $[1, 3]$ , нам важно проверить, что  $X < 3$ . Условие  $X \geq 1$  на этой ветви уже выполнено. А для третьей части функции мы вообще не проверяем  $X$ , все лишние значения мы уже «отмели».

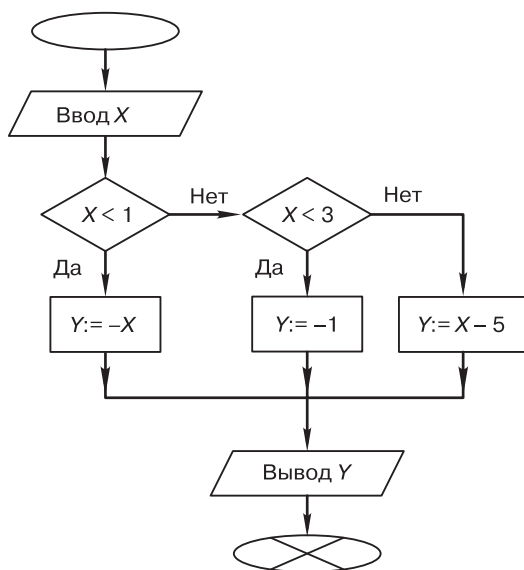


Рис. 7.7

**Пример 7.4.** Дано уравнение  $AX^2 + BX + C = 0$  (коэффициенты целые, вводятся с клавиатуры). Определить, имеет ли оно корни; если имеет, то сколько. Напечатать их.

```

Program Kvur;           {Решение уравнения}
  Var A, B, C : Integer; {Коэффициенты}
  X1,X2,D : Real; {Корни, дискриминант}
Begin Writeln ('Решение квадратного уравнения
               A*X*X + B*X + C=0');
  Writeln ('Введите коэффициенты - целые числа');
  Write('A='); Readln(A);
  Write('B='); Readln(B);
  Write('C='); Readln(C);
  D:=B*B-4*A*C;
  If D<0 Then Writeln('Корней нет')
  Else Begin If D=0 Then Begin Writeln('1 корень');
                             X1:=-B/(2*A);
                             Writeln('X=',X1:8:3)
                           End
        Else Begin Writeln('2 корня');
                  D:= Sqrt(D);
                  X1:=(-B-D)/(2*A);
                  X2:=(-B+D)/(2*A);
                  Writeln('X1=',X1:8:3,
                          ' X2=',X2:8:3)
                End
        End
  End.
End.

```

После вычисления дискриминанта в программе написан один большой условный оператор, который и завершает решение задачи. Посмотрите: в случае  $D < 0$  после оператора, печатающего, что корней нет, программа завершается, несмотря на то что в тексте ниже еще много строчек. Никаких завершающих действий здесь не нужно, потому что весь текст ниже рассматриваемой строки относится к другой ветви ( $D \geq 0$ ). А мы знаем, что в условном операторе выполняется только одна ветвь, после чего выполняется оператор, следующий за условным. У нас в программе больше операторов нет, поэтому она и заканчивается.

### **Задание**

1. Дополните программу, чтобы после ввода коэффициентов она печатала получившееся уравнение (с конкретными коэффициентами). Не думайте, что это очень простая задача и для ее решения достаточно написать всего лишь один оператор вывода. Добейтесь, чтобы уравнение красиво печаталось и в тех случаях, когда какие-то коэффициенты отрицательные (чтобы не было двух знаков подряд). А если какие-то коэффициенты равны нулю, соответствующую часть уравнения вообще писать не надо.
2. Кстати, о нуле. При всех ли значениях коэффициентов наша программа работает правильно? Что случится, если  $A = 0$ ?

Дело в том, что мы пользовались формулами для вычисления корней квадратного уравнения, но при  $A = 0$  уравнение перестает быть квадратным, и решать его в этом случае надо по другим формулам. Сделайте соответствующую вставку в программу. Не забудьте рассмотреть и случай, когда нулю равны все коэффициенты одновременно.

Обе предыдущие программы состоят из нескольких частей, «веточек», в которых описаны действия, выполняемые при соблюдении определенных условий. В зависимости от условий выполняется одна из «веточек» программы. Можно ли рассматривать условия в другом порядке? Да. В первом случае мы выбрали порядок исходя из того, что график привычнее рассматривать слева направо, во втором просто сначала описали наиболее легкий случай. Однако бывают ситуации, когда порядок условия важен и даже строго определен. Посмотрим это на следующем примере.

**Пример 7.5.** С клавиатуры вводится число  $X$ . Найти значение выражения: в числителе квадратный корень из разности  $X^2 - 15$ , в знаменателе — сумма  $\operatorname{tg} X + 2 \operatorname{ctg} X$ . Если при введенном  $X$  выражение не существует, сообщить об этом.

Запишем выражение и хорошенько его рассмотрим. Конечно, вычислить его значение можно не при всех значениях  $X$ . Какова область допустимых значений (ОДЗ) этой переменной? Конечно, подкоренное выражение должно быть неотрицательным и выражение в знаменателе не должно равняться



нулю. Но, увы, написать в программе только эти условия — недостаточно. Ведь чтобы проверить, что выражение не равно нулю, компьютер должен подсчитать его значение для заданного  $X$ , но выражение  $\operatorname{tg} X + 2 \operatorname{ctg} X$  само существует не при всех значениях  $X$ , значит, мы можем заставлять компьютер подсчитывать его значение только в том случае, когда уверены, что это значение существует. То есть нам надо проверить, не равны ли нулю синус и косинус  $X$  (напомним  $\operatorname{tg}(x) = \sin(x)/\cos(x)$ ;  $\operatorname{ctg}(x) = \cos(x)/\sin(x)$ ). Таким образом получается, что надо сделать не две, а целых четыре проверки. И вот здесь порядок проверок очень важен!

Обратите внимание, подобные действия мы делаем, когда, выполняя задания по алгебре, определяем ОДЗ функции. И там, выписывая условия, которым должны удовлетворять аргументы, мы об их порядке не задумываемся. Почему же здесь, в аналогичной задаче, мы говорим, что соблюдение порядка проверок имеет решающее значение? Дело в том, что, решая задачу по алгебре, мы в большинстве случаев лишь указываем, какие условия нужно проверить, и, возможно даже не проверяем их целиком, а лишь подставляем ответ (определяем, удовлетворяет он ОДЗ или нет). Здесь же мы не просто выписываем условия существования функции, а пишем команды, в процессе выполнения которых компьютер должен будет произвести необходимые вычисления, и мы должны позаботиться о том, чтобы эти вычисления можно было сделать.

С порядком проверок определились, можно писать программу. Но хорошо бы еще иметь в виду вот что: для того чтобы произвести проверки, нам (точнее, компьютеру) придется выполнять достаточно сложные вычисления (например, подсчитывать значения тригонометрических функций). При подсчете значения конечного выражения понадобятся результаты этих вычислений. Давайте не будем заставлять компьютер вычислять одно и то же несколько раз, запомним вычисленные значения. Получим следующую блок-схему (рис. 7.8).

В блок-схеме у нас несколько одинаковых блоков — печать сообщения о том, что выражение не имеет смысла. Объединять все эти блоки в один не стоит — в этом случае программа, соответствующая блок-схеме, получится «не структурная», в ней придется использовать оператор перехода. Чтобы в программе не писать несколько раз одно и то же, удобно ввести строковую константу (мы так уже делали).

И еще одно замечание перед тем, как мы приступим к написанию программы. Мы говорили, что вещественные числа не принято сравнивать на точное равенство, а у нас присутствуют сравнения с нулем для переменных  $Y$  и  $S$ , которые являются вещественными. В блок-схеме для наглядности удобна именно такая запись, в программе не будем сравнивать число на равенство с нулем, а напомним, что оно должно отличаться от нуля на некоторое небольшое значение (назовем его  $\epsilon$ ), т. е. по модулю должно превосходить это  $\epsilon$ .

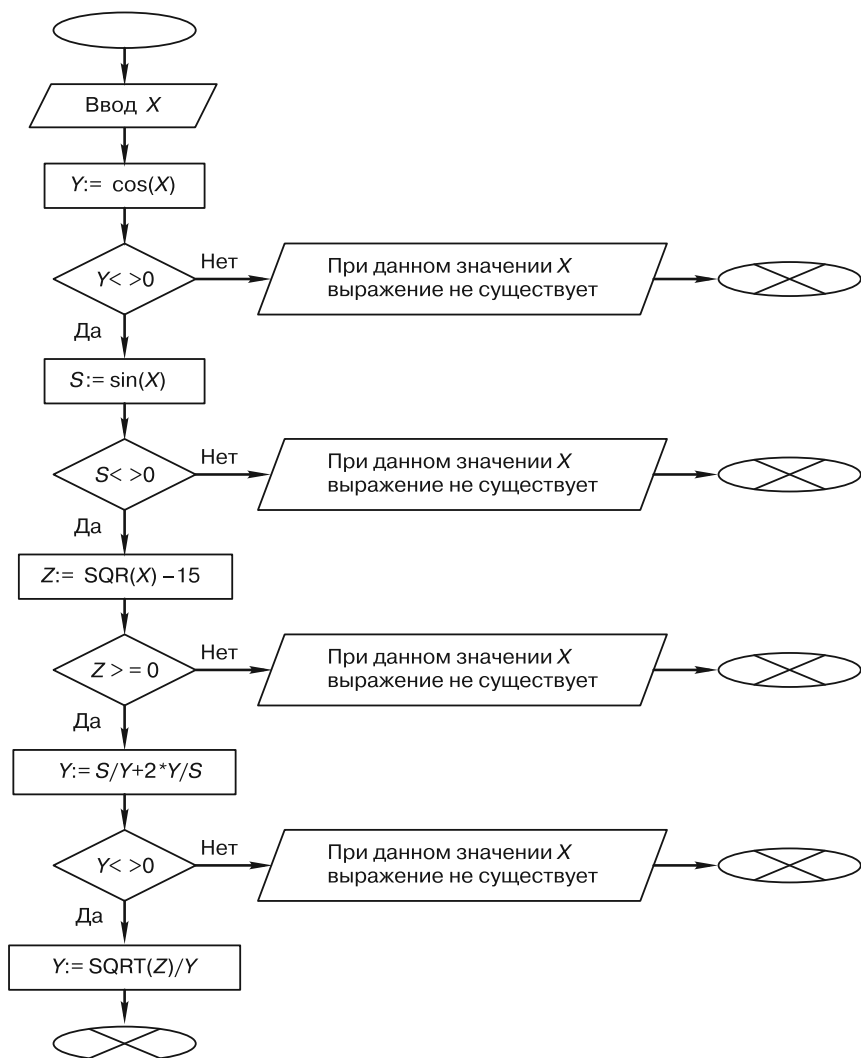


Рис. 7.8

```

Program Func;
Const Eps=1E-8;
      Net='Выражение не существует';
var X,Y,Z,S : Real;
Begin
  Write('X='); Readln(X);
  Y:=cos(X);
  If Abs(Y)>Eps Then
    Begin S:=Sin(X);
          If Abs(S)>Eps Then
            Begin Z:=Sqr(X)-15;
  
```

```

      If Z>=0 Then
        Begin Y:=S/Y+2*Y/S;
              If Abs(Y)>Eps Then
                  Writeln(Sqrt(Z)/Y:10:4)
                  Else Writeln(Net)
              End
            Else Writeln(Net)
        End
      Else Writeln(Net)
    End
  Else Writeln(Net)
End.

```

Обратите внимание на структуру получившейся программы. На ее верхнем уровне всего 4 оператора: вся «лесенка» — это один большой условный оператор. В программе, как и в блок-схеме, 4 веточки с текстом «**Else Writeln(Net)**», и убрать ни одну из них никак нельзя: если это сделать, то при каких-то значениях  $X$  программа не будет печатать никаких сообщений.

В отношении в условном операторе могут сравниваться не только числа, но и символы.

**Пример 7.6, а.** С клавиатуры вводятся два числа и знак операции между ними: «+» или «-». Выполнить над числами соответствующее действие и напечатать результат.

```

Var  A,B,C : Real;
      Z : Char;
Begin Write('Введите первое число '); Readln(A);
      Write('Введите второе число '); Readln(B);
      Write ('Введите знак операции (+ или -) ');
                                          Readln(Z);

      If Z='+' Then C:=A+B
        Else C:=A-B;
      Writeln(A:4:1,Z,B:4:1,'=',C:6:2)
End.

```

Посмотрите, как будет вести себя наша программа, если человек введет не «+», не «-», а какой-то другой символ? Неправильно! Как с этим бороться, мы узнаем чуть позже, а сейчас немного усложним задачу. Пусть операций будет теперь не две, а четыре.

**Пример 7.6, б\*.** С клавиатуры вводятся два числа и знак операции между ними: «+», «-», «\*» или «/». Выполнить над числами соответствующее действие и напечатать результат.

```
Var  A,B,C : Real;
      Z : Char;
Begin Write('Введите первое число '); Readln(A);
      Write('Введите второе число '); Readln(B);
      Write ('Введите знак операции (+,-,* или /) ');
      Readln(Z);
      If Z='+' Then C:=A+B
        Else If Z='-' Then C:=A-B
          Else If Z='*' Then C:=A*B
            Else C:=A/B;

      Writeln(A:4:1,Z,B:4:1,'=',C:6:2)
End.
```

**Задание.** Всегда ли наша программа работает правильно? Подумайте, какими способами можно ее исправить, чтобы не было неприятностей при  $B = 0$ .

Как хорошо, что у нас всего четыре арифметических действия! Как бы выросла наша программа, какую большую «лестницу» из **Then-Else** пришлось бы писать, если бы их было, скажем, 10! Оказывается, не пришлось бы. В Паскале такие ситуации предусмотрены.

## Оператор выбора

Этот оператор создан как раз для реализации таких «широких» разветвлений, когда вычислительный процесс разделяется не на две, а на большее количество ветвей.

Запишем, как будет выглядеть оператор выбора в нашей программе, если поставить его вместо условного оператора.

```
Case Z of
  '+' : C:=A+B;
  '-' : C:=A-B;
  '*' : C:=A*B;
  '/' : C:=A/B
End;
```

Как видите, программа становится гораздо короче и нагляднее! Как же выглядит оператор выбора в общем виде?

**Case** и **of** — это служебные слова, они присутствуют в любом операторе выбора. Заканчивается оператор словом **End** (как видим, **End** может «закрывать» не только **Begin**, но еще и **Case**). Между **Case** и **of** должно быть выражение, от значения которого зависит, какую ветвь следует выбирать. В нашем случае выражение простейшего вида — переменная. Это выражение должно быть строго ординального типа! Вспомним, пока мы знаем два таких типа — **Integer** и **Char**.

Далее перечисляются «метки выбора» — все значения, которые может принять данное выражение, и для каждого значения через двоеточие записывается оператор, который надо выполнить в этом случае. Обратите внимание — один оператор! Как быть, если для решения задачи одного не хватает? Мы это уже знаем — любое количество операторов превращается в один, если их поместить в операторные скобки **Begin–End**. Пары «значение : оператор» (альтернативы) отделяются друг от друга точкой с запятой. Порядок, в котором перечисляются эти пары, значения не имеет.

Выше было сказано, что должны быть перечислены все возможные значения выражения, которое используется в операторе выбора. А что будет, если «забыть» какое-то значение? В разных версиях языка эта часть оператора «устроена» по-разному. В «стандартном» языке, описанном его создателем (Н. Виртом), определено, что, если в операторе выбора выражение приняло значение, для которого не написан оператор, — это ошибка, программа прекращает работу. В большинстве реализаций такого не происходит, просто не выполняется ни один из операторов, составляющих ветви оператора выбора. А в некоторых реализациях даже есть специальная ветвь **Else**. Она записывается в качестве последней ветви, после всех значений (но перед **End**). Интересно, что здесь перед **Else** ставится точка с запятой (в отличие от условного оператора).

Далее приведен фрагмент программы, которая по номеру дня в месяце ( $n$ -целое, от 1 до 31) печатает, на какой день недели он приходится (считаем, что 1-е число этого месяца пришлось на понедельник).

```
Case N mod 7 of
  1 : Writeln('Понедельник');
  2 : Writeln('Вторник');
  3 : Writeln('Среда');
  4 : Writeln('Четверг');
  5 : Writeln('Пятница');
  6 : Writeln('Суббота');
  0 : Writeln('Воскресенье')
End;
```

Записанное в этом операторе арифметическое выражение **N mod 7** может принимать одно из семи значений, мы их все описали.

Иногда бывает, что для нескольких разных значений выражения надо выполнить одни и те же действия, тогда можно действия дважды не выписывать, а соответствующие метки варианта записать через запятую.

С днями недели можно написать такой фрагмент:

```
Case N mod 7 of
  1,2,3,4,5 : Writeln('Рабочий день');
  6,0       : Writeln('Выходной день')
End;
```

В некоторых реализациях допускается даже не перечислять все значения через запятую, а записать целый диапазон, если для всех значений из этого диапазона требуется выполнить одинаковые действия. Тогда следующий фрагмент будет тоже правильным (**Ch** — типа **Char**, любой символ):

```
Case Ch of
  'A'..'Z', 'a'..'z': WriteLn('Латинская буква');
  '0'..'9':          WriteLn('Цифра');
  '+', '-', '*', '/': WriteLn('Знак арифметической
                           операции');
  Else              WriteLn('Специальный символ')
End;
```

Вспомним, мы уже говорили, что латинские буквы, большие и маленькие, пронумерованы подряд (имеют коды, идущие друг за другом). Запись **'A'..'Z'** означает диапазон (отрезок) всех значений от **'A'** до **'Z'**, включая концы. Цифры тоже идут подряд, а вот про арифметические знаки мы ничего такого не знаем, поэтому их пришлось перечислить.

А напоследок напомним, что в любой версии Паскаля, как бы широки возможности оператора **Case** в ней ни были, оператор выбора работает только с ординальными типами. Например, при решении квадратного уравнения у нас имеется три веточки-альтернативы (для  $D = 0$ ,  $D < 0$ ,  $D > 0$ ), но оформить эту часть программы с помощью оператора выбора не удастся, так как  $D$  — вещественное. (*Подумайте, если бы вдруг  $D$  стало целым, удалось бы тогда записать соответствующий оператор выбора?*)

**Задание.** Посмотрите, какой оператор **Case** реализован в том варианте Паскаля, которым вы пользуетесь? Какие из приведенных здесь фрагментов у вас работают правильно?

## Задачи 7.1–7.35. Программы с разветвлениями

- 7.1. Напечатать, является ли заданное число четным.
- 7.2. Вводятся 2 числа. Вывести наибольшее.
- 7.3. Вводятся 3 числа. Найти наибольшее.
- 7.4. Вводятся 3 символа. Напечатать их в порядке возрастания кодов.
- 7.5. Задана сторона квадрата и радиус некоторого круга. Найти периметр фигуры, у которой площадь больше.
- 7.6. Заданы 2 целых числа. Напечатать, делится ли первое на второе нацело (учесть, что на 0 делить нельзя).
- 7.7. Заданы 2 целых числа. Напечатать, делится ли большее на меньшее нацело.
- 7.8. С клавиатуры вводится символ. Проверить, является ли он маленькой латинской буквой.

- 7.9. С клавиатуры вводится, сколько сейчас времени (в часах, от 0 до 23) и через сколько часов должен зазвонить будильник. В какое время будильник зазвонит? (Не забыть рассмотреть ситуацию, когда время вечернее, а будильник зазвонит утром следующего дня, или когда будильник зазвонит через несколько дней — такой он у нас «долгоиграющий».)
- 7.10. Известно время начала и конца некоторого мероприятия (в часах и минутах, часы задаются в диапазоне от 0 до 23). Мероприятие длилось не более 12 часов, но могло начаться вечером, а закончиться на следующий день. Подсчитать, сколько времени длилось мероприятие (в часах и минутах).
- 7.11. Функция задана в виде дроби. В числителе  $|\cos x^3| * \sqrt{3x-5}$ , в знаменателе  $x^2 * (x - 1)$ . Вводится значение  $x$ , на выходе — значение функции, если его можно вычислить, или сообщение о том, что  $x$  не входит в область определения.
- 7.12. Вспомним, что координатная плоскость делится на 4 четверти, и известно, какие знаки имеют  $X$  и  $Y$  в этих четвертях. Написать программу, в которой с клавиатуры вводятся координаты некоторой точки и печатается, в какой четверти она находится.
- 7.13. Вводятся 3 числа. Можно ли построить треугольник с такими длинами сторон? Если можно, то определить, является ли он равносторонним, равнобедренным, прямоугольным.
- 7.14. Известны длины сторон и углы выпуклого четырехугольника (надо ли вводить все углы?). Является ли он квадратом, ромбом, прямоугольником, параллелограммом? Причем, если четырехугольник является квадратом, не надо писать, что он к тому же ромб, прямоугольник и параллелограмм (то же для прямоугольника — не надо писать, что он параллелограмм).
- 7.15. Прямую можно задать уравнением  $Y = kX + b$ . С клавиатуры вводятся коэффициенты  $k$  и  $b$  двух прямых. Определить их взаимное расположение (совпадают, параллельны, пересекаются, а, может быть, и вовсе не существуют).
- 7.16. Решить уравнение  $kX + B = 0$ . (Учесть случай  $k = 0$ , в том числе одновременно с  $B = 0$ .) Коэффициенты вводятся с клавиатуры. Выписать получившееся уравнение (вместо коэффициентов  $k$  и  $B$  подставить введенные числа). Найти  $X$ .
- 7.17. Задано трехзначное положительное целое число. Есть ли среди его цифр 3? Если есть, на каком месте стоит?
- 7.18. Заданы 3 числа. Сколько из них одинаковы?
- 7.19. Заданы 4 числа. Сколько среди них одинаковых? (Не забыть рассмотреть случай, когда заданы 2 пары одинаковых чисел, например: 1, 1, 2, 2.)
- 7.20. Вводится слово из 4 букв. Есть ли в нем одинаковые буквы?
- 7.21. С клавиатуры вводится символ. Если это маленькая латинская буква, напечатать ее, если это большая латинская буква — напечатать соответствующую ей маленькую. В остальных случаях напечатать сообщение об ошибке.

- 7.22. Человеку задается некоторый вопрос, на который возможен ответ «да» или «нет». В качестве ответа он вводит одну букву. Понять и напечатать ответ человека, учитывая, что, имея в виду «да» («yes»), он может ввести большую или маленькую, русскую или латинскую букву «д» («у»), а вместо «нет», соответственно, те же «вариации» буквы «н» («п»). Человек может и ошибиться — об этом надо напечатать сообщение.
- 7.23. Вводится номер месяца. Напечатать его название.
- 7.24. Вводится номер месяца. Напечатать название следующего и предыдущего месяцев (аккуратнее с январем и декабрем!).
- 7.25. Вводится номер месяца. Напечатать, какое это время года.
- 7.26. С клавиатуры вводится символ. Напечатать, является ли он цифрой, маленькой или большой латинской буквой.
- 7.27\*. Составить программу, которая грамотно печатает фразу «Я решил  $K$  задач».  $K$  вводится с клавиатуры. Например, для  $K = 1$  должно напечататься «Я решил 1 задачу», для  $K = 24$  — «Я решил 24 задачи», для  $K = 100$  — «Я решил 100 задач». Программа должна правильно работать для любого натурального  $K$  ( $0 < K < 32\,000$ ).
- 7.28\*. Вводятся два целых числа. Вывести наибольшее четное число или сообщение, что четных чисел нет.
- 7.29\*. Вводятся 3 вещественных числа. Вывести 2 наибольших отрицательных, если столько отрицательных среди введенных чисел есть. В противном случае вывести одно отрицательное число или сообщение, что отрицательных чисел нет.
- 7.30\*. С клавиатуры вводятся 4 символа. Является ли введенная последовательность целым числом (число может начинаться со знака  $+$  или  $-$  и все остальные символы должны быть цифрами)?
- 7.31\*. С клавиатуры вводятся 4 символа. Является ли введенная последовательность правильным вещественным числом по правилам Паскаля?
- 7.32\*. Вводится четырехзначное число. Сформировать из его цифр другое четырехзначное число, такое, что в нем цифры будут идти в порядке убывания.
- 7.33\*. Вводится четырехзначное число, все цифры в числе разные. Поменять в нем местами минимальную и максимальную цифры. Вывести полученное число (вывести ОДНО число типа INTEGER).
- 7.34\*. С клавиатуры вводится трехзначное 16-ричное число (используется символьный ввод). В качестве дополнительных цифр используются большие буквы от A до F. Перевести это число в системы счисления с основаниями 10, 2, 8.
- 7.35\*. Пусть число вводится следующим образом (используется только символьный ввод): первый символ — буква b или d, обозначающая систему счисления (b — двоичная, от *binary* или d — десятичная, от *decimal*), следующие 3 символа — цифры числа. Проверить правильность ввода (в двоичной системе счисления число может состоять только из цифр 0 и 1) и вывести введенное число в десятичной системе счисления.



## Глава 8

# Тип Boolean.

# Логическое выражение

---

Мы научились писать программы с условным оператором, но сам условный оператор до конца не изучили. В предыдущей главе было сказано, что после служебного слова **If** должно стоять условие — логическое выражение. Что же это такое?

Это выражение так называемого логического типа **Boolean**. Константы этого типа могут принимать всего два значения: истина и ложь. На Паскале они записываются **True** и **False**. Всего две константы, а мы их изучению посвящаем целый раздел! Дело в том, что, во-первых, над значениями этого типа в Паскале определено несколько операций. Во-вторых, правильное составление логических выражений очень важно для программирования. Вы можете записать правильные формулы, но, если неверно сформулировано условие (неправильно записано логическое выражение), объясняющее, когда какие формулы применять, программа будет работать неправильно.

Поэтому приступим.

## Логические значения, логические константы

Итак, значениями логических выражений могут быть **True** и **False**. Логических констант всего две, тип ординальный. **Ord(False)=0**, **Ord(True)=1**, таким образом **False<True**, **Succ(False)=True**, **Pred(True)=False**.

Логические значения можно сравнивать на равно—не равно и на больше—меньше. Последнее используется очень редко, так как если логические **A<B**, то **A=False**, а **B=True**, поэтому проще именно так и записать.

Логические константы, значения логических переменных и выражений можно выводить с помощью операторов **Write** и **Writeln**: **Writeln(True)** именно **True** и напечатает, **Writeln(X<5)** напечатает, например, **True** при **X=0** и **False** при **X=5**.

А вот вводить значения логических переменных с клавиатуры нельзя. Как же быть? Ну, во-первых, в «реальных» задачах такое встречается довольно редко, во-вторых, можно вместо логического значения ввести целое или символьное, а в программе в зависимости от того, что введено, присвоить логической переменной нужное значение:

```
Var    B : Boolean;  
        C : Char;  
Begin Write('Введите "t", если надо ввести True, и "f",  
                                     если False ');  
        Readln(C);  
        B:=C='t';  
        ...  
End.
```

В операторе присваивания в левой части — логическая переменная **B**, а в правой — логическое выражение **C='t'**. Его значение зависит от введенного значения **C**, выражение вычисляется и результат присваивается **B**. Конечно, при таком вводе мы должны быть уверены во внимательности пользователя: если он по ошибке введет не «**t**», не «**f**», а что-то третье, например «**T**», логической переменной будет присвоено **False**. Но это преодолимая проблема, позже мы ее решим.

## Булева алгебра, алгебра логики

Сначала о названии. Оно происходит от фамилии английского математика, основавшего так называемую алгебру логики. Джордж Буль (George Boole) жил в середине XIX века. Как видите, при своем зарождении эта наука не могла иметь никакого отношения ни к программированию, ни к языку Паскаль!

Алгебра логики — это наука, которая занимается операциями над высказываниями. Высказывание — повествовательное предложение, утверждение, относительно которого можно сказать, что оно или истинно, или ложно. «Москва — столица России», «Волга впадает в Каспийское море», « $2 * 2 = 4$ » — истинные высказывания. «От Земли до Луны дальше, чем до Солнца», «Все крокодилы на земле вымерли», « $25 < 0$ » — ложные высказывания. Высказывания « $X > 0$ », «Петя — отличник» могут быть истинными или ложными в зависимости от значения  $X$ , от того, какой именно Петя имеется в виду. Предложения: «число  $1/999$  — очень маленькое», «Спартак — хорошая команда» — не являются высказываниями, так как приведенные характеристики — понятия относительные. Одно и то же число в одних случаях может считаться маленьким, а в других — большим; «Спартак» по каким-то параметрам хорош, а по каким-то, может, и не очень.

Джордж Буль предложил операции над высказываниями записывать в символьном виде, как это делается в алгебре, применяя буквенные обозначения, знаки операций и т. п. Для записанных таким образом высказываний оказалось возможным формулировать и доказывать теоремы, решать уравнения и т. п. Основанная Джорджем Булем наука называется алгебра логики или Булева алгебра.

В повседневной жизни мы постоянно пользуемся высказываниями и даже производим операции над ними: соединяем несколько простых высказываний в более сложные, в логические выражения. Просто мы раньше не знали, что это так называется. При рассмотрении условий возможности или невозможности некоторого события, при формулировке и доказательстве теорем мы пользуемся логическими связками: «и», «или», «следовательно», «необходимо и достаточно», многими другими.

В алгебре логики порядка десяти различных логических операций, мы изучим и будем использовать в программах всего три из них.

## Логические операции

### Операция NOT

Достаточно часто в быту, в математике мы пользуемся операцией «логическое отрицание»: «*X* не является положительным числом». В логике она обозначается знаком «¬» или чертой над отрицаемым выражением, в Паскале — служебным словом **NOT**.

Эта операция имеет один операнд: **not A, not (X>0)**.

Результат операции: **True** превращается в **False**, а **False** — в **True**.

### Операция AND

А вот еще одна логическая конструкция, достаточно часто используемая в повседневной жизни. Вспомним, что нужно, чтобы дробь равнялась нулю? «Чтобы знаменатель был отличен от нуля, а числитель равен нулю». Два высказывания: «Знаменатель равен нулю» и «Числитель отличен от нуля» мы соединили в одно, причем для выполнения условия они оба должны быть истинными; если хоть одно ложно — ложно и все условие. В повседневной речи мы обычно высказывания для таких условий объединяем союзом «и». «Я пойду в кино, если там идет интересующий меня фильм **И** у меня хватит денег на билет».

В алгебре логики такая операция называется «логическое И», «логическое умножение» или «конъюнкция», обозначается знаком  $\wedge$  или **&**. В Паскале эта операция обозначается словом **and** (что по-английски означает И). Таким образом, если *A* и *B* — высказывания, то можно написать **A and B**. Представим зависимость значения логического выражения от значений входящих в него высказываний в виде таблицы.

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Как видно из таблицы (и следует из нашего опыта использования этой логической конструкции в повседневной жизни), для этой операции действует переместительный закон: **A and B** то же самое, что **B and A**.

Обратите внимание, что при ложности одного из высказываний вся конструкция будет ложной. В этом случае все выражение целиком можно не анализировать. Если мы установили, что первое высказывание ложно, второе и проверять не надо. В нашем случае с кино: если нужного фильма в расписании нет, не надо считать, хватит ли денег на билет — все равно в кино не пойдешь. И наоборот — если нет денег, нечего и расписание смотреть.

При программировании на Паскале это совсем не так, способ вычисления логических выражений, очередность вычисления его составляющих зависит от версии языка. Мы должны быть готовы к тому, что будут вычисляться оба выражения, составляющих конструкцию, и только потом посчитается значение всего логического выражения. Например, в некоторых версиях вычисление выражения **(X > 0) and (1/X > 0)** при  $X = 0$  не будет закончено, так как обнаружится ошибка «деление на 0».

Операция OR

Название этой операции в алгебре логики: «логическое ИЛИ», «логическое сложение», «дизъюнкция»; обозначается она знаком  $\vee$ . В Паскале операция обозначается служебным словом **or**.

Чтобы логическое выражение **A or B** было истинным (равнялось **True**), достаточно, чтобы равнялось **True** одно из составляющих его высказываний. Выражение принимает значение **False**, только если оба высказывания имеют значение **False**.

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Пример: «Я смогу пойти в кино, если в кармане найдутся деньги на билет ИЛИ мне мама даст денег».

Как видим, эта операция, как и логическое умножение, обладает переместительным свойством. Общее с логическим умножением и то, что Паскаль будет сначала вычислять оба операнда, входящих в выражение, и только потом значение всего выражения.

## Составление логических выражений

Простейшими логическими выражениями являются логические константы, логические переменные, отношения, логические функции. Например: **True**, **A**, **Bul**, **X>0**, **Y=Z**.

С логическими функциями мы пока не встречались. Это функции, результат которых имеет тип **Boolean**.

Логическая функция **Odd** определена для целочисленного аргумента. Она принимает значения **True**, если аргумент нечетный, и **False**, если четный: **Odd(5)** равно **True**, а **Odd(0)** — **False**.

С помощью логических операций из более простых, коротких логических выражений можно составлять более сложные. При вычислении логического выражения надо помнить о приоритете операций: самый высокий приоритет у операции **not**, она выполняется в первую очередь; приоритет логического умножения, как и у обычного умножения, выше, чем у логического сложения. В выражении **A or not B and C** сначала вычисляется **not B**, результат логически умножается на **C** и в последнюю очередь выполняется **Or**.

В логических выражениях часто используются операции отношения; их приоритет самый низкий. Для изменения порядка выполнения операций используются скобки. Выражение

$(X \geq -1) \text{ and } (X \leq 1)$

истинно для всех  $X$  из отрезка  $[-1, 1]$  и ложно для  $X$  вне этого отрезка. Если в программе на Паскале записать это выражение без скобок, транслятор обнаружит ошибку «несоответствие типов» (а не «отсутствие скобок», как хотелось бы). Почему? **and** — операция с самым высоким приоритетом в этом выражении, поэтому при отсутствии скобок компьютер попытается выполнять действие **-1 and X**. Но операндами **and** могут быть только логические величины, вот и несоответствие типов.

***Задание.** Может ли выражение **A<B or C** быть правильным?*

Может, если все переменные — логические. Сначала выполнится **B or C**, а потом будет осуществляться сравнение с **A**. Это выражение истинно, если **A=False**, а **B or C=True**.

***Задание.** Может ли выражение **A<B or A<C** быть правильным? Какого типа тогда должны быть переменные?*

Это выражение является неправильным при любых  $A, B, C$ . Здесь сначала должна выполняться операция **or**: **B or A**, а потом как-то вычисляться получившееся отношение. Но мы уже говорили, что «двойные», «тройные» и т. п. отношения в Паскале не допускаются.

Так как в логическом выражении могут использоваться операции разных типов: и арифметические, и логические, и операции сравнения, — приведем порядок выполнения действий по мере убывания приоритета.

1. Вычисление аргументов функций (они стоят в скобках после имени соответствующей функции), вычисление значений функций.
2. Вычисление выражений в скобках.
3. Операция «логическое отрицание»: `not`.
4. Операции «типа умножения»: `*`, `/`, `div`, `mod`, `and`.
5. Операции «типа сложения»: `+`, `-`, `or`.
6. Операции сравнения: `=`, `<>`, `>`, `<`, `>=`, `<=`.

Операции одного приоритета выполняются слева направо. Так, в выражении

`(X<A+D div K) or not Odd(N+L)`

сначала посчитается значение **N+L** и будет выяснено, является ли это число нечетным (функция **Odd**), затем результат целочисленного деления **D div K** прибавится к **A**, потом результат этих действий будет сравниваться с **X**. Затем будет выполняться операция **not** и, в последнюю очередь, **or**.

## Задачи с логическими выражениями

*Пример 8.1. Записать логическое выражение, которое истинно тогда и только тогда, когда:*

- а) все три числа  $X$ ,  $Y$  и  $Z$  больше 10;
- б) хотя бы одно из трех чисел  $X$ ,  $Y$ ,  $Z$  больше 10;
- в) ровно одно число из  $X$ ,  $Y$ ,  $Z$  больше 10.

При решении этой задачи надо очень внимательно следить, с помощью какой логической связки — **and** или **or** — соединять отношения. Если требуется одновременное выполнение всех условий, ставим **and**, если достаточно выполнения одного условия — **or**. Получаем:

- а)  $(X > 10) \text{ and } (Y > 10) \text{ and } (Z > 10);$
- б)  $(X > 10) \text{ or } (Y > 10) \text{ or } (Z > 10);$
- в)  $(X > 10) \text{ and } (Y \leq 10) \text{ and } (Z \leq 10) \text{ or}$   
 $(Y > 10) \text{ and } (X \leq 10) \text{ and } (Z \leq 10) \text{ or}$   
 $(Z > 10) \text{ and } (X \leq 10) \text{ and } (Y \leq 10).$

*Пример 8.2. Написать логическое выражение, которое истинно тогда и только тогда, когда  $X$  — двузначное четное число.*

Двузначное число — это число от 10 до 99, а признак четности — деление на 2. Получаем:

$(X \geq 10) \text{ and } (X \leq 99) \text{ and } (X \bmod 2 = 0) \quad \text{или}$   
 $(X \geq 10) \text{ and } (X \leq 99) \text{ and } \text{not}(\text{Odd}(X)).$

**Пример 8.3.** Записать логическое выражение (зависящее от значения натурального  $M$ ), которое истинно, если число  $M$  трехзначное и цифра 7 входит в его десятичную запись, и ложно в противном случае.

Из условия, что число трехзначное, получаем:

$$(M > 99) \text{ and } (M < 1000)$$

Мы уже знаем, что с помощью операций **div** и **mod** можно выделить цифры числа. Остается только записать, что одна из них должна быть равна 7:

$$(M \bmod 10 = 7) \text{ or } (M \div 100 = 7) \text{ or } (M \div 10 \bmod 10 = 7)$$

Получившиеся части логического выражения надо соединить операцией **and** (так как в условии стоит «и»):

$$((M > 99) \text{ and } (M < 1000)) \text{ and } ((M \bmod 10 = 7) \text{ or } (M \div 100 = 7) \text{ or } (M \div 10 \bmod 10 = 7))$$

Обратите внимание: каждую из частей мы заключили в скобки, но заметим, что первое выражение можно в скобки не заключать, от их отсутствия ничего не изменится, второе же выражение обязательно должно быть в скобках. Дело в том, что **or** по приоритету операция более слабая, чем **and**, и при отсутствии скобок действия будут выполняться в другом порядке, значение выражения не всегда будет правильным.

### Задание

1. Подберите  $M$ , при которых можно увидеть, что выражение без скобок имеет неправильное значение.
2. Сформулируйте задачу, для которой выражение без указанных скобок будет правильным.

**Пример 8.4.** Дано логическое выражение  $(X \geq A) \text{ and } (X \leq B)$ . Можно ли подобрать такие числа  $A$  и  $B$ , что выражение будет ложно при любом вещественном  $X$ ?

Если сформулировать это высказывание по-другому, получится, что  $X$  лежит между  $A$  и  $B$  или что выполняется двойное неравенство  $A \leq X \leq B$ . Получается, что, если  $A > B$ , выражение будет ложно при любых  $X$ . Например,  $(X \geq 0) \text{ and } (X \leq -3)$ .

При  $A = B$  выражение истинно только при  $X = A$ .

В остальных случаях (при  $A < B$ ) выражение истинно, если  $X \in [A, B]$  (рис. 8.1.).

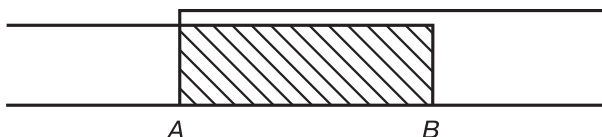


Рис. 8.1

**Пример 8.5.** Изобразим теперь на числовой прямой все значения  $X$ , при которых истинно выражение  $(X < A) \text{ or } (X > B)$ .

При  $A \leq B$  (рис. 8.2) получаются две области (обратите внимание, сами точки  $A$  и  $B$  не закрашены), а при  $A > B$  — вся числовая прямая, т. е. выражение всегда истинно, для любого  $X$ . А вот подобрать такие  $A$  и  $B$ , чтобы выражение было всегда ложно, не удастся.



Рис. 8.2

**Пример 8.6.** Составим отрицание к логическому выражению из примера 8.4. Получим  $\text{not}((X \geq A) \text{ and } (X \leq B))$ . Попробуем догадаться, при каких  $X$  оно истинно.

Пусть  $A < B$ . Это должны быть  $X$ , которые не принадлежат отрезку  $[A, B]$ . Изобразим их на числовой прямой и получим решение примера 8.5!

Таким образом, мы получили:

$$\text{not}((X \geq A) \text{ and } (X \leq B)) = (X < A) \text{ or } (X > B).$$

Обозначим отношения логическими переменными  $R := X \geq A$ ;  $P := X \leq B$ . Тогда, сделав замены в логическом выражении, получим:

$$\text{not}(R \text{ and } P) = \text{not } R \text{ or } \text{not } P.$$

Если мы сделаем аналогично отрицание к выражению из примера 8.5, получим очень похожее утверждение:

$$\text{not}(R \text{ or } P) = \text{not } R \text{ and } \text{not } P.$$

Эти утверждения доказываются в курсе математической логики (вам предстоит изучать эту науку как часть информатики). Здесь мы их доказывать не будем, но полезно помнить о существовании таких соотношений и применять их.

**Пример 8.7.** Нарисовать на плоскости  $(X, Y)$  область, в которой и только в которой истинно логическое выражение:

a)  $(X^2 + Y^2 \leq 4) \text{ and } (Y < X);$

b)  $(X^2 + Y^2 \leq 4) \text{ or } (Y < X).$

Для решения задачи изобразим сначала линии и области, описанные в отношениях (рис. 8.3).



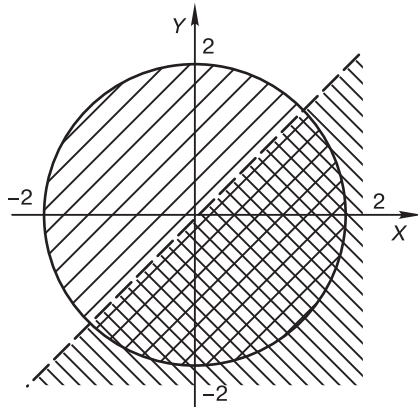


Рис. 8.3

$X^2 + Y^2 = 4$  — это окружность радиуса 2 с центром в начале координат.  
 $X^2 + Y^2 \leq 4$  — это область внутри окружности, т. е. круг (заштрихуем его линиями с наклоном вправо).

$Y = X$  — прямая, а область  $Y < X$  лежит под этой прямой. Мы заштрихуем ее линиями с наклоном влево, причем, чтобы показать, что сама прямая в область не входит, начертим ее пунктиром.

Решение задачи  **$(X^2 + Y^2 \leq 4)$  and  $(Y < X)$**  — пересечение областей, т. е. полукруг, находящийся под прямой (то, что заштриховано обоими видами штриховок одновременно, «в клеточку»).

Решение задачи  **$(X^2 + Y^2 \leq 4)$  or  $(Y < X)$**  — объединение областей, т. е. весь круг и вся полуплоскость под прямой (все, что заштриховано хоть какими-нибудь линиями).

**Пример 8.8.** Записать на Паскале логическое выражение, зависящее от  $X$  и  $Y$ , которое принимает значение **True**, если точка с координатами  $(X, Y)$  не принадлежит заштрихованной области, и **False** в противном случае (см. рис. 8.4).

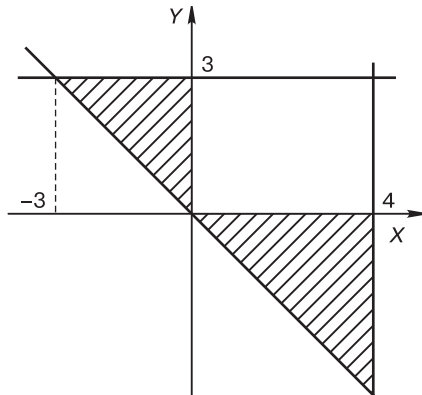


Рис. 8.4

Описывать незаштрихованную область в данном случае достаточно сложно. Составим выражение для заштрихованной области, а решением будет отрицание этого выражения.

Заштрихованная область состоит из двух треугольников. Верхний треугольник ограничен прямыми  $Y = -X$ ,  $Y = 3$  и  $X = 0$ . Эти отношения в логическом выражении надо объединить операцией **and**:

$$(Y > -X) \text{ and } (Y \leq 3) \text{ and } (X \leq 0) .$$

Второй треугольник — это пересечение областей  $Y \geq -X$ ,  $Y \leq 0$  и  $X \leq 4$ . Аналогично составляем выражение для второго треугольника:

$$(Y > -X) \text{ and } (Y \leq 0) \text{ and } (X \leq 4) .$$

В общее выражение объединяем их операцией **or**:

$$(Y > -X) \text{ and } (Y \leq 3) \text{ and } (X \leq 0) \text{ or}$$

$$(Y > -X) \text{ and } (Y \leq 0) \text{ and } (X \leq 4) .$$

А теперь, вспомнив, что нам нужна незакрашенная область, получаем:

$$\text{not}((Y > -X) \text{ and } (Y \leq 3) \text{ and } (X \leq 0) \text{ or}$$

$$(Y > -X) \text{ and } (Y \leq 0) \text{ and } (X \leq 4)) .$$

Эту задачу можно было решить и по-другому, заметив, что оба треугольника находятся выше прямой  $Y = -X$ . Тогда закрашенная область описывается чуть короче:

$$(Y > -X) \text{ and } ((Y \leq 3) \text{ and } (X \leq 0) \text{ or } (Y \leq 0) \text{ and } (X \leq 4)) .$$

А применив операцию отрицание и правила, полученные в примере 8.6, получим:

$$(Y < -X) \text{ or } ((Y > 3) \text{ or } (X > 0)) \text{ and } ((Y > 0) \text{ or } (X > 4)) .$$

Здесь общая часть логических слагаемых выносится за скобки, скобки раскрываются после отрицания — такие правила преобразования логических выражений изучаются в алгебре логики. Здесь мы только заметим, что, если вы эти правила знаете, их можно применять при составлении логических выражений.

**Пример 8.9.** Заштриховать на числовой прямой область, где заданные логические выражения равны (одновременно принимают одинаковые значения):

$$(X > 0) \text{ and } (X < 15) \quad \text{и} \quad (X \geq 20) \text{ or } (X \leq 10) .$$

Логические выражения равны в двух случаях: когда они оба истинны и когда оба ложны.

Сначала изобразим область, в которой оба логических выражения истинны (рис. 8.5).

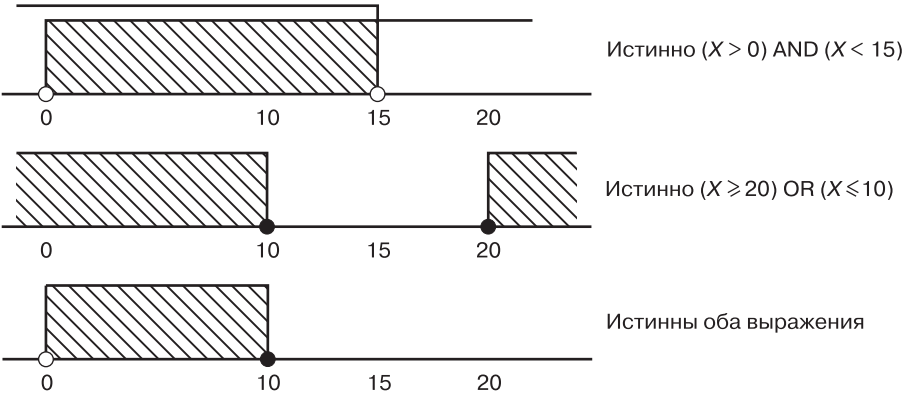


Рис. 8.5

Теперь построим область, в которой оба выражения ложны (рис. 8.6).

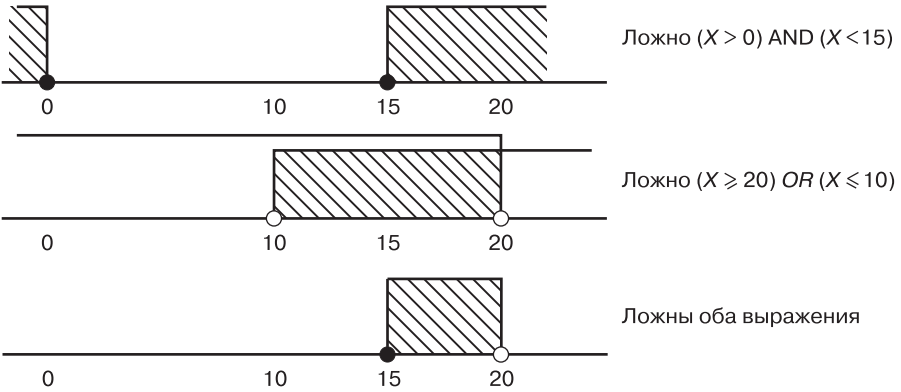


Рис. 8.6

В первом случае ответ — область  $(0, 10]$ , во втором — область  $[15, 20)$ . Они оба входят в решение задачи. Ответ представлен на рис 8.7.

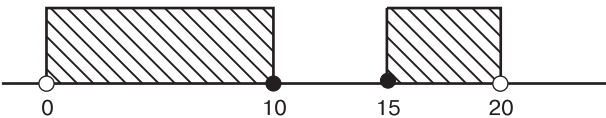


Рис. 8.7

Остается только заметить, что правильность решения всех этих задач можно проверить на компьютере. Для этого надо написать программы, в которые будут вводиться данные (например,  $X$  и  $Y$ ), а выводиться результат вычисления логического выражения. Программы будут выводить **True** или **False**, а наше дело — проверить, соответствуют ли эти ответы условиям задач.

## Программы с логическими выражениями

Для начала приведем несколько примеров использования «длинных» логических выражений в условных операторах. Часто бывает необходимо установить, является ли заданный символ цифрой или буквой. Это можно сделать, например, так (переменная **C** имеет тип **Char**):

```
If (C<'0') or (C>'9') Then Writeln(C, 'не является  
                                цифрой');  
If (C>='a') and (C<='z') Then  
    Writeln(C, '- маленькая латинская буква').
```

Здесь мы пользуемся, во-первых, тем, что символы можно сравнивать, во-вторых, тем, что цифры, а в большинстве современных реализаций и латинские буквы закодированы подряд.

***Пример 8.10.** Пусть целое число  $K$  вводится с клавиатуры. Вывести значение логического выражения, которое истинно тогда и только тогда, когда  $K$  четное и не лежит внутри отрезка  $[-20, 10]$ .*

Программа будет состоять из двух частей: ввод данных и вывод значения логического выражения. Мы здесь воспользуемся тем, что логические значения можно выводить с помощью оператора **Write (WriteLn)**.

Составим логическое выражение, условие « $K$  — четное» записываем в виде: **not Odd(K)**, а то, что « $K$  не лежит внутри отрезка  $[-20, 10]$ », можно представить в виде сочетания двух условий:  $K$  лежит либо до отрезка, либо после него, т. е. **(K<-20) or (K>10)**. «Четность» и «расположение» в условии объединены союзом «и». Получаем логическое выражение: **not Odd(K) and ((K<-20) or (K>10))**.

Это не единственное правильное логическое выражение, отвечающее условиям задачи, *попробуйте самостоятельно написать другие варианты*.

Составим программу.

```
Var K:Integer;  
Begin Write('K='); Readln(K);  
      Writeln(not Odd(K) and ((K<-20) or (K>10)))  
End.
```

При вводе чисел, удовлетворяющих условию задачи (например, 100, -30, 40), программа будет печатать **True**, а для чисел, не удовлетворяющих условию (например, 111, 0, 5), — **False**.

**Пример 8.11.** Написать программу, в которой с клавиатуры вводятся координаты точки на плоскости ( $x, y$  — действительные числа) и выводится сообщение, принадлежит ли эта точка заданной заштрихованной области (включая границы) — рис. 8.8. (В этом и следующем примере чертежи взяты из демо-версий заданий ЕГЭ.)

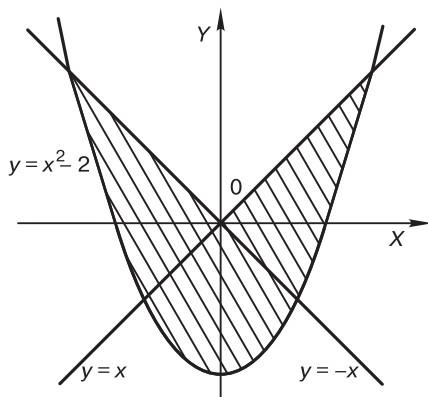


Рис. 8.8

Напишем логическое выражение, как мы это делали в предыдущих примерах. Закрашенная область снизу ограничена параболой (неравенство  $y \geq x^2 - 2$ ), а сверху — двумя прямыми. Однако если мы напишем  $(y < x)$  and  $(y < -x)$ , то получим область гораздо меньше заданной (рис. 8.9).

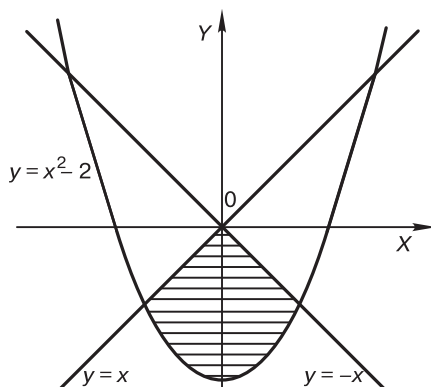


Рис. 8.9

Как же быть? Можно исходную область разделить на части и описать отдельно каждую часть, соединив их потом операцией **or**. Например, если мы разделим область осью ординат (прямая  $OY$ ), то получившиеся части легко

описываются логическими выражениями (добавляется еще одна граница  $X = 0$ ). Получаем следующее логическое выражение:

$$(y > x^2 - 2) \text{ and } (y \leq x) \text{ and } (x \geq 0) \text{ or} \\ (y > x^2 - 2) \text{ and } (y \leq -x) \text{ and } (x \leq 0).$$

Некрасивое? Очень длинное и лишняя граница появилась? Зато правильное, это важнее. Конечно, можно написать и по-другому. «Хорошо натренированный глаз» (а он у вас будет таким, когда сделаете несколько заданий) может заметить, что область под прямыми описывается выражением  $(y \leq x) \text{ or } (y \leq -x)$ . Соединив это выражение с выражением для параболы (и не забыв про скобки!), получаем  $(y > x^2 - 2) \text{ and } ((y \leq x) \text{ or } (y \leq -x))$ . Могут быть и другие варианты правильного логического выражения для этой задачи.

Программа может быть, например, такая:

```
var x , y : real ;
begin
  readln ( x , y );
  if (y >= x*x-2) and ((y <= x) or (y <= -x))
    then write ('принадлежит')
    else write ('не принадлежит')
end.
```

Но можно эту задачу решить и совсем другим способом, без использования «длинного» логического выражения. Посмотрим еще раз на рисунок. Заданными линиями и осями координат плоскость делится на некоторое количество частей. Дадим каждой части номер. Частям, которые закрашены, дадим номера с буквой  $Z$ , а тем, которые не закрашены, — с буквой  $N$ . Вот какой рисунок получим (рис. 8.10).

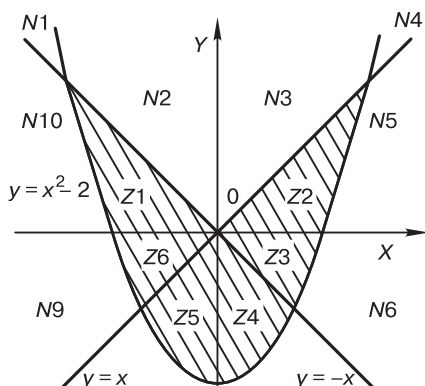


Рис. 8.10

Напишем условный оператор с простыми условными выражениями (но сам оператор получится достаточно сложный — он будет содержать вложенные условные операторы). После каждого условия будем выписывать, какие области описываются им.

```
if y>=x*x-2 then {N2 N3 Z1 Z2 Z3 Z4 Z5 Z6 }
  if y<=x then {Z2 Z3 Z4 Z5} write('принадлежит')
    else {N2 N3 Z1 Z6}
      if y<-x then {Z1 Z6} write('принадлежит')
        else {N2 N3} write('не принадлежит')
    else {N1 N4 N5 N6 N7 N8 N9 N10} write('не принадлежит')
```

Проверим, не забыли ли мы чего-нибудь. Первоначально наша плоскость состояла из областей  $Z1 - 6$  и  $N1 - 10$ . Первое условие ( $y \geq x^2 - 2$ ) разделило ее на часть, в которой оказались  $N2 - 3$  и все закрашенные части, и на часть, где оказались  $N1, N4 - 10$ . Все кусочки из этой второй части не закрашены, их можно больше не рассматривать, а вот кусочки из первой части разделим еще раз, теперь уже прямой  $Y = X$ . Выделяется область с кусочками  $Z2 - 5$  — они все закрашены, для них можно писать ответ, и область  $N2 - 3, Z1, Z6$ , ее опять надо делить на части. Здесь нам поможет прямая  $Y = -X$ . При выписывании названий кусочков внимательно следим, чтобы никакой из них не потерялся — это можно каждый раз проверять, складывая их количество в веточках **Then** и **Else**.

Как видите, это несложно. Возможно, кому-то именно этот метод со временем понравится больше. По нашему мнению, он требует большей аккуратности, соответственно, освоив его, вы допустите меньше ошибок.

Метод деления координатной плоскости на части очень хорош и для проверки правильности логического выражения или программы — надо протестировать программу для точек из каждой пронумерованной части плоскости. Заметим, что если вы разделите плоскость на большее количество частей, чем это нужно, ничего страшного не случится, придется лишь проверить больше точек, если же «забыть» про какую-то часть, это может привести к ошибке.

Проверить правильность работы программы с помощью компьютера легко, а вот как это сделать, если компьютера нет?

**Пример 8.12.** *Требовалось написать программу, при выполнении которой с клавиатуры считываются координаты точки на плоскости ( $x, y$  — действительные числа) и определяется принадлежность этой точки заданной заштрихованной области, включая границы (рис. 8.11). Найти ошибки в данной программе (привести пример таких чисел  $x, y$ , при которых программа неверно решает поставленную задачу) и исправить программу, чтобы она для всех чисел давала верный ответ.*

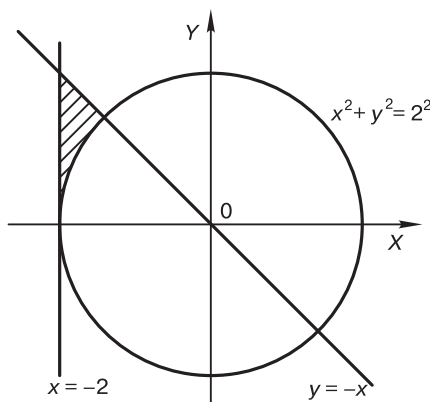


Рис. 8.11

```

Var x,y: Real;
Begin
  Readln(x,y);
  If x*x+y*y>=4 then
    If x>= -2 Then
      If y<= -x Then
        Write('принадлежит')
      Else
        Write('не принадлежит')
      End.

```

Сначала переписываем программу с правильными отступами, чтобы разбраться с соответствиями **If-Then-Else**.

```

Var x,y: Real;
Begin
  Readln(x,y);
  If x*x+y*y>=4 Then
    If x>= -2 then
      If y<= -x Then Write('принадлежит')
      Else Write('не принадлежит')
      {К этому then нет Else}
    {К этому then нет Else}
  End.

```

Сразу видим, что, если взять некоторые пары точек, программа для них не даст никакого ответа. Так, в случае невыполнения неравенства  $x*x+y*y \geq 4$  никаких действий программы не предусмотрено, поэтому, например, для точек (0, 0), (1, 1) программа не выдаст никакого ответа. Не будет никакого ответа и в случае, когда  $x*x+y*y \geq 4$  выполняется, а  $x \geq -2$  не выполняется.



Посмотрим, только ли для точек, которым «не хватает **Else**», программа дает неправильный ответ. Обозначим на рисунке искомую (закрашенную) область буквой *Z*, внутренность окружности буквой *O*, всем остальным частям плоскости дадим номера от 1 до 6. Получается вот такой рисунок — рис. 8.12.

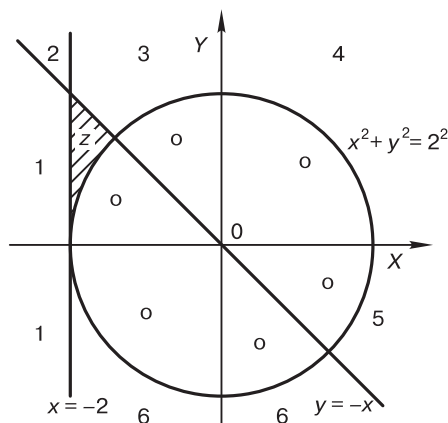


Рис. 8.12

Расставим в программе числа, пометив, какие части операторов соответствуют номерам областей:

```
If x*x+y*y>=4 Then {Z 1 2 3 4 5 6}
  If x>= -2 Then {Z 3 4 5 6}
    If y<= -x Then {Z 6} Write('принадлежит')
      Else {3 4 5} Write('не принадлежит')
```

Мы видим, что два первых условия  $x*x+y*y \geq 4$  и  $x \geq -2$  отсекают области *Z*, 3, 4, 5, 6. Чтобы получить закрашенную область *Z*, нам надо из нее удалить части 3, 4, 5, 6. Однако в программе отделяются только части 3, 4, 5, лежащие над прямой  $y \leq -x$ , а часть 6 остается.

Таким образом, «хитрость» этой задачи в том, что область ограничена тремя линиями и хочется, соответственно, описать ее логическим выражением из трех частей:

$(x*x+y*y \geq 4) \text{ and } (x \geq -2) \text{ and } (y < -x)$ .

Это логическое выражение неправильное, в него, как мы видели, кроме искомой области попадает еще и область 6. Вот такие «вредные» бывают области: для их описания нужно больше логических условий, чем это кажется с первого взгляда. Как уберечься от ошибок? Например, написав логическое выражение, можно аккуратно решить обратную задачу: нарисовать область, которую оно описывает, и посмотреть, совпадает ли она с заданной.

Подходит здесь и предложенный нами выше способ написания программы «без длинного логического выражения». Нам нужно разделить области  $Z$  и  $6$ . Чем они различаются? От какой прямой на чертеже они лежат по разные стороны? Это прямая  $y = 0$ . Вставим соответствующее неравенство в программу:

```
if x*x+y*y>=4 then {Z 1 2 3 4 5 6}
  if x>= -2 then {Z 3 4 5 6}
    if y<= -x then {Z 6}
      if Y>=0 then {Z} write('принадлежит')
      else {6} write('не принадлежит')
    else {3 4 5} write('не принадлежит')
  else {1 2} write('не принадлежит')
else {0} write('не принадлежит')
```

Таким образом, чтобы проверить программу без компьютера, во-первых, полезно подсчитать количество частей условного оператора — **If-Then** и **Else** должно быть поровну. Для того чтобы программа давала ответ для любой точки плоскости, все условные операторы должны быть полными. Во-вторых, следует проверить, не забыта ли какая-то часть области. Всего у нас их 8. Первый **If** разделил плоскость на 2 части: ( $Z$ , 1, 2, 3, 4, 5, 6) и  $O$ . Вторым удалит части 1, 2, которые лежат левее прямой  $x = -2$ . (Части плоскости, которые удаляет очередной условный оператор, записаны в его **Else**.) Аналогично далее прослеживаем, что у нас ничего не потеряно: для каждой части плоскости будет выдан ответ, причем правильный.

Нам кажется, в данной задаче способ «без длинного логического выражения» вполне уместен, хотя программа и получается некомпактной.

И напоследок все же запишем «длинное» логическое выражение, описывающее данную область. Оно получается, если соединить все неравенства, которые стоят в нашей исправленной программе после **If**:

$(x*x+y*y>=4) \text{ and } (x>= -2) \text{ and } (y<= -x) \text{ and } (y>=0)$ .

## Задачи 8.1–8.11. Логическое выражение

- 8.1. Написать логическое выражение, которое истинно тогда и только тогда, когда целое  $K$  удовлетворяет следующему условию:
- а)  $K$  — трехзначное положительное число с 0 в конце;
  - б)  $K$  — нечетное, делится на 3 или на 5;
  - в)  $K$  принадлежит отрезку числовой прямой  $[2, 6]$ .
- 8.2. Написать логическое выражение из задачи 1, которое истинно, если целое  $K$  не удовлетворяет условию, и ложно в противном случае:
- а) с использованием операции **not**;
  - б) без использования операции **not**.
- 8.3. Написать логическое выражение, которое истинно, если все цифры натурального трехзначного числа одинаковые.

- 8.4. Написать логическое выражение, истинное при выполнении указанного условия и ложное в противном случае:
- а) ни одно из трех чисел  $X, Y, Z$  не является положительным;
  - б) только одно из трех чисел  $X, Y, Z$  положительно;
  - в) два целых числа  $X$  и  $Y$  не являются четными или нечетными одновременно;
  - г) все три числа  $X, Y, Z$  равны между собой;
  - д) только два числа из тройки  $X, Y, Z$  равны между собой;
  - е) среди чисел  $X, Y, Z$  нет равных;
  - ж) три числа в порядке неубывания располагаются вот таким образом:  
 $X \leq Y \leq Z$ ;
  - и) вещественные числа  $X, Y, Z$  являются сторонами равнобедренного прямоугольного треугольника;
  - к) целое число  $a$  четырехзначное и в нем есть цифра 0;
  - л) целое число  $a$  трехзначное и в нем нет цифры 5;
  - м) целое число  $a$  трехзначное и все его цифры равны.
- 8.5. Написать логическое выражение, в котором используются числа 3 и 5, переменная  $X$  и операция **and**:
- а) выражение, истинное при любом  $X$ ;
  - б) выражение, ложное при любом  $X$ .
- 8.6. Написать логическое выражение, в котором используются числа 10 и  $-10$ , переменная  $X$  и операции **or** и **not**:
- а) выражение, истинное при любом  $X$ ;
  - б) выражение, ложное при любом  $X$ .
- 8.7. Нарисовать на плоскости  $(X, Y)$  область, в которой и только в которой истинно логическое выражение:
- а)  $(X * X + Y * Y \leq 9) \text{ and } (Y < 2 * X)$ ;
  - б)  $(\text{SQR}(X + 1) \leq Y) \text{ or } (Y < X + 1)$ ;
  - в)  $(2 * X + Y \leq 1) \text{ and } (Y - 3 * X > -1)$ ;
  - г)  $(X + Y \leq 1) \text{ or } (X - Y \geq 2)$ ;
  - д)  $(y * y < 4 - x * x) \text{ or } (y \geq x) \text{ and } (y < 2 - x / 2)$ ;
  - е)  $(\text{abs}(x) > 3) \text{ and } (\text{abs}(y) < 3)$ ;
  - ж)  $(\text{abs}(x) < 3) \text{ or } (\text{abs}(y) > 3)$ ;
  - и\*)  $\text{abs}(x) > \text{abs}(y)$ .
- 8.8. Записать на Паскале логическое выражение, зависящее от  $X$  и  $Y$ , которое принимает значение **True**, если точка с координатами  $(X, Y)$  принадлежит закрашенной области, и **False**, если не принадлежит (рис. 8.13, а–м). Границы, которые принадлежат области, проведены сплошной линией, а те, которые не принадлежат, — пунктирной.

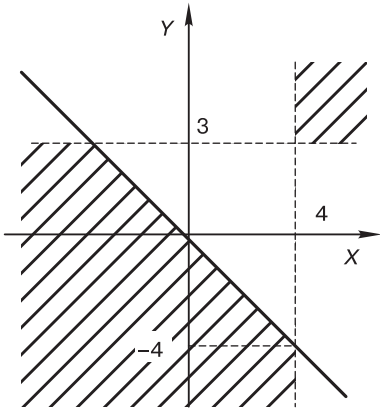


Рис. 8.13, а

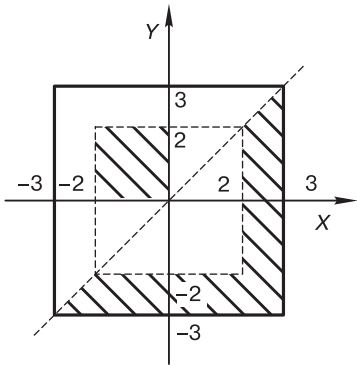


Рис. 8.13, б

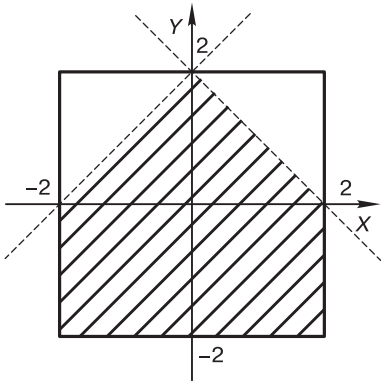


Рис. 8.13, в

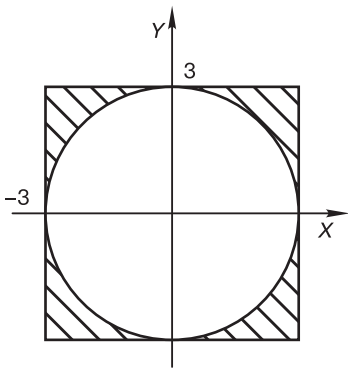


Рис. 8.13, г

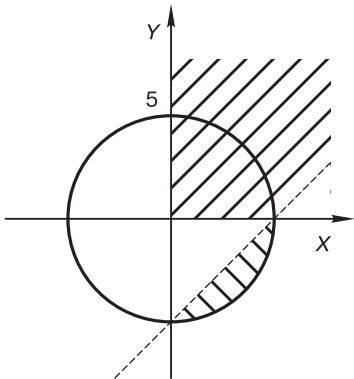


Рис. 8.13, д

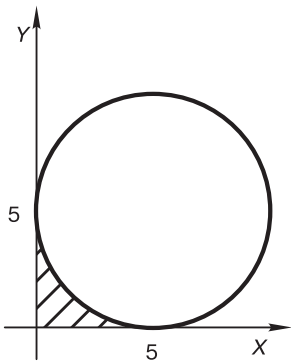


Рис. 8.13, е

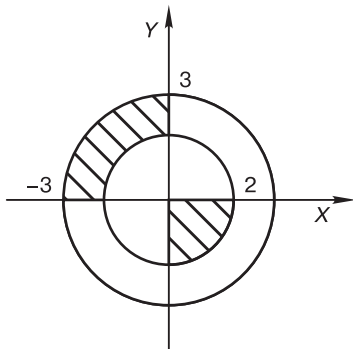


Рис. 8.13, ж

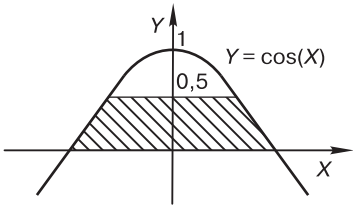


Рис. 8.13, з

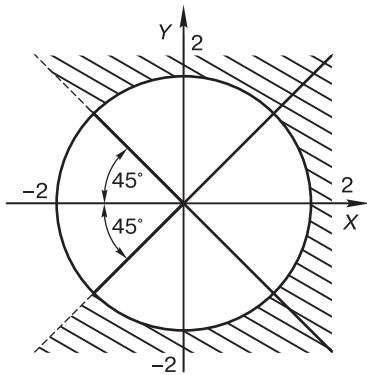


Рис. 8.13, и

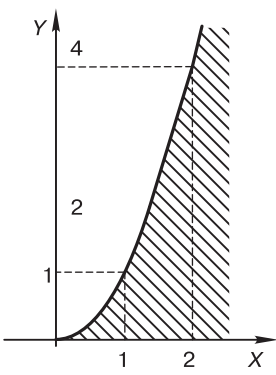


Рис. 8.13, к

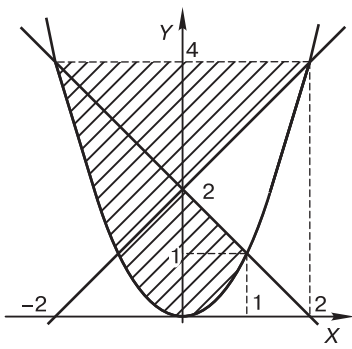


Рис. 8.13, л

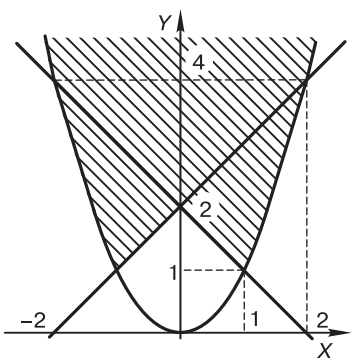


Рис. 8.13, м

Написать программу, в которой вводятся координаты точки и выводится «да», если точка принадлежит закрашенной области, и «нет», если не принадлежит. Программу написать двумя способами: с использованием логических операций и без них.

8.9. Заштриховать на координатной плоскости область, где логические выражения равны между собой:

а)  $X * X + Y * Y \leq 16$  и  $Y < 2 - X$ ;

б)  $SQR(X) > Y$  и  $Y < -3 * X + 1$ .

Описать эту область с помощью логического выражения. Написать программу, в которой вводятся координаты точки и выводится «да», если точка принадлежит закрашенной области, и «нет», если не принадлежит.

Учтите, что при решении этих задач надо рассмотреть два случая (см. пример 8.9 в тексте).

8.10. Заштриховать на координатной плоскости область, где логические выражения не равны друг другу:

а)  $2 * X - Y < 1$  и  $Y - X / 3 >= 1$ ;

б)  $X + Y <= 1$  и  $X - Y >= 2$ .

Описать эту область с помощью логического выражения. Написать программу, в которой вводятся координаты точки и выводится «да», если точка принадлежит закрашенной области, и «нет», если не принадлежит.

8.11. Придумайте интересную область, которую можно красиво описать с помощью логического выражения.

## Глава 9

# Операторы цикла

---

Операторы цикла — пожалуй, самые важные для программирования. Без них можно реализовать только простейшие вычисления, где описанные действия выполняются один раз (либо не выполняются вообще). Оператор цикла дает нам мощный инструмент для написания алгоритмов, в которых действия не только повторяются заданное число раз, но и число повторений может определяться автоматически.

В Паскале три оператора цикла. В одном из них число повторений должно быть известно до начала цикла, в двух других определяется в процессе работы. Последние два годятся для решения более общих задач, о них вначале и поговорим.

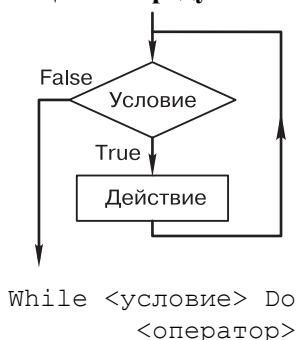
### Циклы с предусловием и с постусловием

Мы уже говорили, что в цикле должно присутствовать:

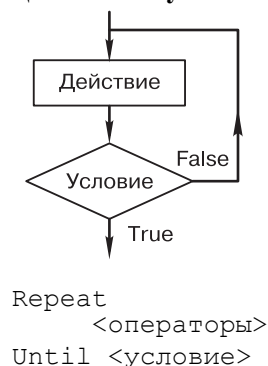
- действие (то, что надо повторять несколько раз, — «тело цикла»);
- условие, в зависимости от выполнения которого цикл будет либо продолжаться, либо закончится.

Взаимное расположение этих двух составляющих и определяет, каким является цикл: с предусловием (сначала проверяется условие, а потом производится действие) или с постусловием (выполняется действие, а потом проверяется условие). Схема работы циклов и общий вид записи этих циклов на Паскале представлены на рис. 9.1.

**Цикл с предусловием**



**Цикл с постусловием**



**Рис. 9.1**

Здесь **While** — пока; **Do** — делать, выполнять; **Repeat** — повторять; **Until** — до тех пор; условие — логическое выражение. В операторе **While** в цикле допускается один оператор; если надо написать несколько операторов, превращаем их в один с помощью операторных скобок. В операторе **Repeat** можно записать любое количество операторов внутри цикла.

Обратите внимание на форму записи. Операторы, выполняющиеся в цикле (тело цикла), мы будем писать с некоторым сдвигом вправо, чтобы отличать их от остальных, выполняющихся не более одного раза.

Пример	R:=1; While R>0.3 Do Begin Write(R:6:3); R:=R/2 End;	R:=1; Repeat Write(R:6:3); R:=R/2 Until R<=0.3;
Пример	Пока $R > 0.3$ выполнять: печатать значение $R$ ; $R$ уменьшить в 2 раза	Повторять действия: печатать значение $R$ ; $R$ уменьшить в 2 раза до тех пор, пока $R$ не станет $\leq 0.3$
Результат работы	Оба этих оператора делают одно и то же — печатают строку: 1.000 0.500	
Работа	1. Проверка $R > 0.3$ ? 2. Да. Следовательно, выполняются действия: 2.1. Печать 1.000 2.2. $R = R/2$ {теперь $R = 0.5$ } 3. После выполнения операторов, записанных в теле цикла, опять проверяется условие $R > 0.3$ ? 4. Да. Опять выполняются действия: 4.1. Печать 0.500 4.2. $R = R/2$ ; {теперь $R = 0.25$ } 5. И снова проверка условия $R > 0.3$ ? 6. Нет. Работа оператора заканчивается	1. Выполняются действия из тела цикла: 1.1. Печать 1.000 1.2. $R = R/2$ ; {теперь $R = 0.5$ } 2. После выполнения тела цикла проверяется условие $R \leq 0.3$ ? 3. Условие не выполняется, поэтому повторяется тело цикла: 3.1 Печать 0.500 3.2. $R = R/2$ ; {теперь $R = 0.25$ } 4. Снова проверяется условие $R \leq 0.3$ ? 5. Условие выполнено, работа оператора заканчивается



Действия оператора **While** можно интерпретировать так: «Пока условие выполняется, делай нечто». Оператор как бы «разрешает» что-то делать еще и еще раз. «Пока светит солнце, загорай на пляже».

Оператор **Repeat** выполняется до тех пор, пока значение условного выражения не станет истиной, как будто мы делаем что-то, пытаясь добиться выполнения условия: «Читай стихотворение, пока не выучишь его наизусть», «тренируйся до тех пор, пока не выполнишь норматив».

Операторы различаются не только тем, что раньше идет: действие или условие, но и тем, при каком значении условного выражения продолжается выполнение цикла, а при каком заканчивается.

Если значение условного выражения **True**, тело цикла в операторе **While** выполняется, а работа **Repeat**, наоборот, при истинном значении условного выражения заканчивается. В наших операторах условия взаимно противоположны: **NOT (R>0.3)** — это то же самое, что **R<=0.3**.

Всегда ли эти операторы цикла будут делать одно и то же, если написать в них взаимно противоположные условия? Рассмотрим два фрагмента:

<pre>C:='a'; While C&lt;&gt;'a' Do     Writeln('=====')</pre>	<pre>C:='a'; Repeat     Writeln('=====') Until C='a'</pre>
---	--

Оператор **While** здесь не выполнится ни разу, так как условие ложно, а вот оператор **Repeat** один раз выполнится, и только после этого проверится условие. Таким образом, нельзя написать оператор **Repeat**, который бы не выполнялся ни разу, ведь условие, которое «решает», выполнять — не выполнять, проверяется только после того, как действия уже один раз сделаны!

Итак, оператор

`While False Do Writeln ('+')` — не выполнится ни разу,

а оператор

`Repeat Writeln('+') Until True` — выполнится один раз.

Посмотрим теперь, как будут работать и сколько раз выполнятся такие операторы:

`While True Do` и `Repeat Until False`

(операторы здесь написаны совершенно правильно, тело цикла у них пустое, т. е. в теле цикла находится пустой оператор, правилами Паскаля такое допускается). Оба оператора будут выполняться бесконечное число раз. Это

называется закикливанием. Несмотря на то что в теле цикла никаких операторов нет и никакие действия в нем не выполняются, если в программу вставить один из таких операторов, она будет работать вечно (пока ее не прервут извне, например, закрыв окно Паскаль-среды, выключив компьютер и т. п.).

Конечно, никто не захочет специально портить свою программу и вставлять в нее такие «нехорошие» операторы, однако довольно часто случается, что программа закикливается из-за ошибок в операторах **Repeat** и **While**. Как этого избежать? Особенно тщательно проверять условие, которое стоит в цикле. В нем, как правило, должна быть переменная, и эта переменная в теле цикла должна меняться (иначе условное выражение не будет менять своего значения). Программу перед выполнением надо сохранить на диск, чтобы в случае закикливания можно было сразу посмотреть текст, а не набирать его заново (если программа не была сохранена, при аварийном выходе из Паскаль-среды она пропадет).

Вот несколько фрагментов программ, в которых из-за ошибок происходит закикливание:

Текст фрагмента	Описание ошибки
K:=10; While K>0 Do; Begin Writeln(K); K:=K-1	Из-за «вкравшегося» после слова <b>Do</b> знака «;» цикл оказался пустым, Значение переменной <i>K</i> в теле цикла не меняется (оно изменится только после того, как цикл закончится), а значит, не меняется и условие, оно всегда истинно
K:=10; While K>0 Do Begin Writeln(K); K:=K+1 End	Значение переменной <i>K</i> меняется, но «не туда», оно увеличивается и всегда остается положительным
K:=0.2; Repeat Writeln(K); K:=K+0.3 Until K=1	Значение переменной <i>K</i> изменяется вроде бы «куда надо», в сторону увеличения, но оно никогда не станет равно 1, «перепрыгнет» через единицу. Переменная <i>K</i> здесь принимает значения 0.2, 0.5, 0.8, 1.1 и т. д. Причем для исправления ошибки нежелательно изменять условие на $K = 1.1$ . Мы уже говорили о том, что вещественные числа могут оказаться не одинаковыми по значению, а достаточно близкими, в этом случае условие равенства выполняться не будет. Чтобы цикл остановился, условие может быть, например, $K \geq 1$

## Решение задач с помощью циклов с постусловием и с предусловием

**Пример 9.1.** Даны целые числа  $T, R$  ( $T > 1$ ). Напечатать все члены бесконечной последовательности  $T, T^2, T^3, T^4, T^5, \dots$ , меньшие числа  $R$ .

Значение очередного члена последовательности обозначим переменной  $P$ , будем его печатать и вычислять следующее значение. Первый член последовательности равен  $T$ , а каждый следующий будет получаться из предыдущего умножением на  $T$ . Понятно, что надо использовать цикл: членов последовательности может быть много. В цикле будет вывод и умножение. Когда закончить цикл? Об этом сказано в условии задачи: члены последовательности надо печатать, пока они меньше числа  $R$ . Напишем программу с циклом с предусловием (блок-схема представлена на рис. 9.2):

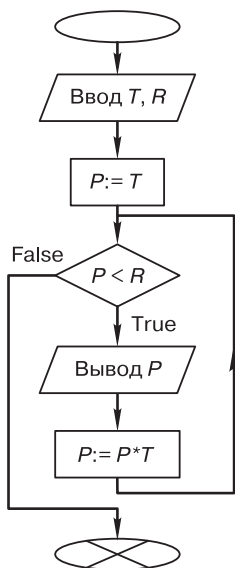


Рис. 9.2

```

Var  T,R,P   : Integer;
Begin Write('Введите число, которое будем возводить
                                     в степень ');
      Readln(T);
      Write('Введите число-ограничитель '); Readln(R);
      P:=T;
      While P<R Do
      Begin Write (P:5);
            P:=P*T
      End; Writeln
End.
  
```

Здесь принципиально важно использование именно оператора **While**. Ведь если изначально  $T > R$ , печатать ничего не нужно, т. е. цикл не должен работать ни разу. А мы знаем, что оператор **Repeat** всегда хотя бы один раз работает, поэтому он здесь не подходит. Можно, конечно, написать перед ним условный оператор, который и будет определять, работать циклу или нет, но от этого программа станет несколько длиннее.

**Пример 9.2.** Напишем «клавиатурный тренажер». Точнее, его маленький кусочек — программу, которая просит человека ввести некоторую букву (ее мы зададим сами) и считает, с какой попытки он сделал это правильно.

Букву (символ), которую потребуем ввести, поместим в переменную  $B$ . Далее надо печатать предложение «Введите букву ...», вводить с клавиатуры символ и проверять правильность ввода. Цикл закончится, как только будет введен нужный символ. Подсчитывать количество попыток будем в переменной  $K$ . Блок-схема изображена на рис. 9.3.

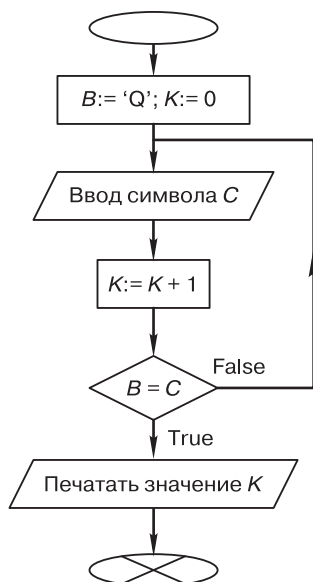


Рис. 9.3

Как видно из рисунка, у нас получился цикл с постусловием, поэтому пишем программу с **Repeat**:

```
Program Tren;
Var B,C : Char;
    K   : Integer;
```

```
Begin B:='Q'; {Попросим ввести эту букву}
      K:=0;
      Repeat
        Write('Введите символ ',B,' - '); Readln(C);
        K:=K+1;
      Until B=C;
      Writeln('Правильно введено с ',K,' попытки')
End.
```

Принципиально ли здесь использование цикла с постусловием? Попробуем переписать программу с циклом **While**. Заголовок цикла будет иметь вид: **While B<>C**. Но ведь когда мы входим в цикл первый раз, переменная *C* не имеет значения, она его получит только в цикле. Значит, это неверное решение.

Мы уже говорили, что работа с переменной, которой не присвоено значение, может привести к ошибке. Правда, работать такая программа, возможно, будет, и иногда даже правильно. Мы говорили, что многие версии трансляторов Паскаля переменным без значения какие-то значения присваивают. Если это «свыше данное» значение вдруг окажется «Q», цикл выполняться не будет, если же какое-то другое, программа проработает нормально, ошибку можно и не заметить. Однако не стоит забывать, что использовать переменные, которым ничего не присвоено, нельзя. На экзамене такая ошибка будет обнаружена и соответствующим образом повлияет на оценку.

Можно, конечно, перед циклом переменной *C* «вручную» присвоить какое-нибудь значение (лишь бы не «Q»), но проще использовать оператор **Repeat**.

### *Пример 9.3. Выписать значения цифры целого положительного числа.*

Казалось бы, чего проще, для детского сада задачка — назвать, из каких цифр состоит число, например 365! Однако для компьютера это совсем не так (может, потому, что они в детский сад не ходят?). Ведь в памяти хранится не число 365, а его двоичный код. Нам легко видеть цифры этого числа потому, что мы имеем в виду десятичные цифры и выписываем число, представляя его именно в десятичной системе счисления. Представьте себе, что число у нас не записано, а представлено «в натуре» — лежит на столе кучка из 365 камешков. Задача сразу становится не такой уж простой.

Стандартный прием для решения такой задачи — применение операций выделения неполного частного и остатка от деления.

Мы уже выделяли цифры из числа, разница в том, что раньше в задачах задавалось, сколько цифр в числе. Пусть *N* — целое число. Тогда ***N mod 10*** дает последнюю цифру числа, а ***N div 10*** — число без последней цифры (проверим: **365 mod 10 = 5**, **365 div 10 = 36**).

Таким образом, мы можем последовательно отделять от числа цифры (начиная справа), последнюю цифру использовать «по назначению» в зависимости от условия задачи (запоминать, печатать и т. п.), а с оставшимися цифрами, усеченным числом, продолжать работать. Когда остановиться? Количество цифр в числе все время будет уменьшаться, и наконец, мы получим число из одной цифры (т. е. число, меньшее 10), с этим числом еще надо поработать, это последняя (самая левая) цифра заданного числа. Если число из одной цифры разделить нацело на 10, получится 0, — вот здесь надо прекратить работу, все цифры числа получены.

В рассуждениях мы считали, что изначально задано число из нескольких цифр. А правильным ли будет алгоритм, если будет введено число из одной цифры? Да, ведь для такого числа операция нахождения остатка при делении на 10 дает само число, оно и будет выведено, при делении нацело получится 0, программа остановится. Блок-схема изображена на рис. 9.4.

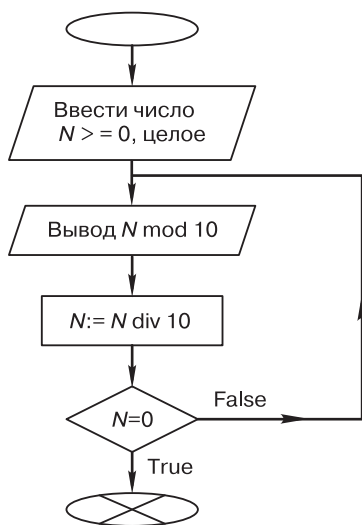


Рис. 9.4

```

Program Cifr;
Var N : Integer;
Begin Write('Введите натуральное число '); Readln(N);
  Repeat
    Write(N mod 10:3); {3 - формат, чтобы между цифрами
                        были пробелы}
    N:=N div 10
  Until N=0;
  Writeln
End.
  
```

С оператором **While**:

```
While N>0 do  
Begin Write(N mod 10:3);  
      N:=N div 10  
End;
```

При условии  $N > 0$  (как это и есть в нашей задаче) программа с циклом **While** будет работать точно так же, как и с циклом **Repeat**.

А если бы в условии такого ограничения не было? В случае  $N=0$  цикл **Repeat** один раз проработает, напечатает 0; цикл **While** не проработает ни разу, и для 0 ничего напечатано не будет. В случае  $N < 0$  цикл **While** также выполняться не будет, а вот **Repeat** будет работать. Правильно ли? Это зависит от версии транслятора Паскаля (в разных версиях операции **div** и **mod** для отрицательных чисел реализованы несколько разными способами).

Интересная вещь — математика! Решаешь некоторую частную задачу, потом смотришь, что получилось, и вдруг обнаруживаешь, что сделано-то на самом деле нечто большее! Мы делили введенное число на 10 и получили его цифры в десятичной системе счисления. А если бы мы поделили на 2, на 5, на  $Q$ ? Мы бы получили цифры числа соответственно в двоичной, в пятиричной и в какой угодно — в  $Q$ -ичной системе счисления (только не следует забывать, что цифры у нас получаются справа налево). Как же это так вышло? Да мы ведь тот самый алгоритм и использовали — вспомните, как переводят в двоичную систему: делят на 2, выписывают остаток. А потом «собирают» цифры с конца.

Таким образом, мы создали алгоритм перевода числа из десятичной системы счисления в любую другую.

***Пример 9.4.** Игра. Первый игрок задумывает целое число, сообщает, из какого оно диапазона. Пусть  $X \in [A, B]$  ( $A$  и  $B$  целые). Второй игрок должен как можно быстрее угадать задуманное. Он будет называть числа, а первый говорить, названное число меньше или больше задуманного (или равно — тогда игра окончена).*

Как, в каком порядке второму игроку надо выбирать числа? Можно, конечно, называть все подряд, начиная с первого. Но это слишком долгий метод, он совершенно не учитывает ответы первого игрока. Действуем так, чтобы каждым ходом уменьшать диапазон, в котором будет находиться  $X$ , в 2 раза. Тогда в качестве первого числа надо назвать среднее арифметическое границ диапазона  $(A+B)/2$ . Если загадано число меньше этого, то оно находится в левой половине диапазона. И следующий поиск надо производить в диапазоне  $[A, (A+B)/2]$ . Если же загаданное число больше, то оно лежит в правой половине диапазона, который можно описать так  $[(A+B)/2, B]$ .

Мы здесь для наглядности использовали знак деления  $/$ . На самом же деле, так как в нашем случае числа предполагаются только целые, надо использовать целочисленное деление (операцию **div**).

**Задание.** Посчитайте, какое наибольшее число шагов понадобится для угадывания числа по этому алгоритму при  $A=0$ ,  $B=100$ . Какое число придется отгадывать дольше всех?

Напишем программу, в которой мы будем задумывать число, а компьютер его отгадывать. Чтобы нам не «перетрудиться», отвечая компьютеру, меньше или больше его число, чем наше загаданное, переложим эту работу на него. Для этого вначале сообщим компьютеру число, которое мы загадали. Наш компьютер «честный», он «подглядывать» не будет.

Какой цикл использовать? Очевидно, с постусловием, мы в этой задаче хотим достичь определенной цели: угадать число, как только угадаем, цикл надо прекращать.

Чтобы было видно, как идет угадывание, будем выписывать номер хода  $K$  и число, которое «называет» компьютер.

```
Var A,B,C, X, K:Integer;
Begin Write('Введите диапазон (целые числа, A<B)  ');
      Readln(A,B);
      Write('Задумано число  '); Readln(X);
      K:=1;
      Repeat
        C:=(A+B) div 2;
        Writeln(K:2, C:5);
        K:=K+1;
        If X<C then B:=C-1
           else if X>C then A:=C+1
      Until X=C
End.
```

Обратите внимание, задуманное число обязательно должно входить в указанный диапазон, иначе программа заикнется.

Использованный здесь метод выбора очередного числа для угадывания называется методом половинного деления. Он годится не только для целых чисел, этим методом в некоторых случаях можно найти корень довольно сложного уравнения.

**Пример 9.5\*.** Пусть некоторая функция  $F(x)$  на отрезке  $[a, b]$  ( $a < b$ ) непрерывна, возрастает, причем на концах отрезка значения функции имеют разные знаки:  $F(a) < 0$ , а  $F(b) > 0$ . Тогда внутри этого отрезка существует единственная точка  $x_0$ , такая что  $F(x_0) = 0$  (т. е.  $x_0$  — корень функции). Написать программу, которая отыскивает приближенное значение корня этой функции.

Способов поиска корней функции существует много, попробуем здесь применить только что изученный нами метод половинного деления. В предыдущем примере мы имели дело с целыми числами, поэтому ответ полу-



чался точный, здесь же у нас числа — вещественные, ответ будем находить приближенный, с некоторой точностью. Будем говорить, что ответ получен с точностью *Eps*, если найденное (приближенное) значение отличается от точного не более чем на *Eps*.

Метод состоит в делении отрезка пополам и изменении его границ (рис. 9.5). Найдем середину заданного отрезка:  $C := (A+B) / 2$  — и посчитаем значение функции  $F$  в этой точке (обозначим его  $F_C$ ). Если  $F_C < 0$ , то корень должен находиться на отрезке  $[C, B]$ , если  $F_C > 0$  — корень на отрезке  $[A, C]$ .

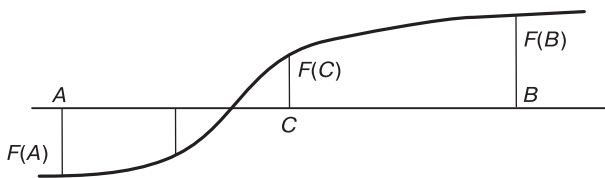


Рис. 9.5

Изменив границы отрезка, можно продолжить поиски корня тем же способом. Как долго это делать? А до тех пор, пока отрезок не станет достаточно маленьким, например пока его длина не станет меньше *Eps*, — в этом случае требуемая точность будет достигнута.

Напишем программу для функции  $F(x) = x * x - 5$ . Для получения правильного ответа надо ввести границы  $A$  и  $B$ , такие чтобы функция на этом отрезке удовлетворяла условию задачи. У нас это могут быть, например, числа 0 и 4. Если ввести неверные границы или взять функцию, не соответствующую заданным условиям, программа даст неправильный ответ или заикнется! В нашем случае корнем функции является  $\sqrt{5}$ .

```

Program Kor;
  Const Eps=0.00001;
  Var  A,B,C, Fc: Real; {Границы, середина отрезка,
                        значение функции}

  Begin
    Write('A= '); Readln(A);
    Write('B= '); Readln(B);
    While B-A > Eps Do
      Begin C:=(A+B)/2;
            Fc:=C*C-5;
            If Fc<0 Then A:=C
                  Else B:=C
            End;
    Writeln(C:10:2)
  End.

```

В наших рассуждениях мы не учли возможность  $Fc=0$ . Скажем сразу, такое «везение» при приближенном решении уравнений возможно достаточно редко, например для линейной функции. Посмотрите, правильно ли будет работать программа в таком случае? Какая ветвь условного оператора будет выполняться?

### **Задание**

1. Напишите программу, в которой учитывается возможность  $Fc=0$ . Так как числа вещественные, лучше проверять  $Fc$  не на точное равенство, а на «малость» ( $abs(Fc) < Eps1$ ). При этом должен происходить выход из цикла (возможно, в этом случае лучше воспользоваться циклом **Repeat**). Посмотрите, насколько такая программа эффективнее для разных функций: для этого надо посчитать, сколько раз выполняется цикл.
2. Напишите программу, которая бы искала корень убывающей функции. Подберите подходящую функцию и отрезок  $[A, B]$ , удовлетворяющие условиям. Найдите корень. Проверьте результат.

## **Задачи 9.1–9.12. Циклы While и Repeat**

*Предлагаемые ниже задания сделать 2 раза: с использованием оператора **While** и с использованием **Repeat**. Подумайте, какой оператор для данной задачи удобнее? Или в данном случае они равноценны?*

- 9.1. Напечатать квадраты всех чисел от 1 до  $N$ .
- 9.2. Напечатать кубы всех четных чисел из отрезка  $[M, N]$ .
- 9.3. Напечатать все трехзначные числа, делящиеся на 3. Операцию деления не использовать!
- 9.4. С клавиатуры вводится целое положительное число  $K$ . Напечатать степени двойки (начиная от  $2^0$ ), не превышающие  $K$ . Пример вывода для числа 10: 1 2 4 8.
- 9.5. Человек вводит символы, заканчивая ввод символом «\*». Сколько символов он ввел?
- 9.6. Найти  $N!$ .

*В следующих задачах задано целое неотрицательное число. Надо работать с его цифрами.*

- 9.7. Найти произведение цифр.
- 9.8. Найти количество цифр.
- 9.9. Найти старшую (левую) цифру.
- 9.10. Найти максимальную и минимальную цифры в числе.
- 9.11. Входит ли заданная цифра в число?
- 9.12. Подсчитать, сколько раз в числе встречается заданная цифра.

## Оператор цикла с параметром

Оператор цикла с параметром — так в Паскале называется цикл с известным числом повторений. Конечно, его область применимости значительно уже, чем у циклов **Repeat** и **While**. Этот оператор можно использовать, только когда число повторений известно до входа в цикл. Однако, во-первых, количество повторений бывает известно достаточно часто, а, во-вторых, при использовании этого вида цикла программа получается более наглядной.

В Паскале есть два вида оператора цикла с параметром:

- с увеличивающимся параметром:

```
For <параметр> := <выражение1> To <выражение2> Do  
<оператор>
```

- с уменьшающимся параметром:

```
For <параметр> := <выражение1> Downto <выражение2> Do  
<оператор>
```

Здесь **For** (для), **To** (увеличиваясь к), **Downto** (уменьшаясь до), **Do** — служебные слова; **<параметр>** — переменная любого ординального типа (т. е. из известных нам — **Integer**, **Char**, **Boolean**, но не **Real**).

Выражения должны быть того же типа, что и переменная. **<Выражение1>** определяет начальное значение параметра, а **<выражение2>** — конечное. Оператор является телом цикла, именно он будет повторяться заданное число раз. Заметим, что по определению в цикле стоит один оператор, поэтому для того, чтобы повторять последовательность операторов, ее надо превратить в составной оператор, воспользовавшись операторными скобками. Также стоит обратить внимание на то, что в операторе цикла нет никаких знаков препинания, в том числе и после служебного слова **Do**. Точка с запятой в этом месте будет означать, что телом цикла является пустой оператор, т. е. такой цикл ничего делать не будет.

Опишем работу оператора цикла с параметром.

Вначале вычисляются значения выражений в заголовке цикла, именно они определяют количество повторений тела цикла. Параметру присваивается значение первого выражения, выполняется тело цикла. Далее параметр изменяется: увеличивается (если в заголовке стоит **To**) или уменьшается (если в заголовке стоит **Downto**). Изменения происходят с помощью функций **Succ** или **Pred** (именно поэтому тип параметра и выражений должен быть ординальным, для которого определены эти функции). Снова выполняется тело цикла. Так происходит до тех пор, пока цикл не выполнится для параметра, равного значению **<выражения2>**. На этом работа цикла заканчивается.

На значения выражений не накладывается никаких ограничений (кроме того, что они должны принадлежать к одному и тому же ординальному типу). Количество повторений цикла можно вычислить так:

- для цикла с **To**:  $\langle \text{выражение2} \rangle - \langle \text{выражение1} \rangle + 1$ ;
- для цикла с **Downto**:  $\langle \text{выражение1} \rangle - \langle \text{выражение2} \rangle + 1$ .

Таким образом, если выражения равны, цикл выполнится 1 раз, а если разность выражений отрицательна — не выполнится ни разу. Заметим, что написать такие выражения, чтобы цикл **For** повторялся бесконечно много раз, нельзя.

В качестве примеров рассмотрим несколько операторов цикла, которые печатают значение своего параметра:

Описание параметра цикла	Оператор	Напечатается на экране
Var I:Integer;	For I:=1 to 5 do Write(I, ' ');	1 2 3 4 5
Var K:Integer;	For K:=1 downto -1 do Write(K, ' ');	1 0 -1
Var C:Char;	For C:= 'a' to 'd' do Write(C)	abcd
Var T:Boolean;	For T:=False to True do Write(T)	FalseTrue

Казалось бы, цикл **For I:=E1 To E2 Do <оператор>** можно записать с помощью циклов **While** и **Repeat** следующим образом:

I:=E1; While I<=E2 DO Begin   <оператор>; I:=succ(I) End	I:=E1; If E1<=E2 Then Repeat <оператор>; I:=succ(I) Until I>E2
--	---

Однако это не совсем так. Нельзя сказать, что цикл **For** совершенно идентичен какому-то из этих циклов. Дело в том, что его реализация имеет некоторые особенности.

1. Число повторений вычисляется при входе в цикл и потом в ходе выполнения цикла уже не меняется даже при изменении значений переменных:  
I:=5; N:=11;  
For I:=2\*I To N Do  
  Begin   N:=N+1;  
          Write(I:3,N:3),  
End;

Этот цикл выполнится 2 раза. Вначале  $I$  в нем равняется 10, а  $N$  равно 11. Напечатается 10 12 11 13. То есть  $N$  меняться будет, как это и предписано оператором  $N:=N+1$ , но цикл будет выполняться до того первого значения, которое было в заголовке цикла.

2. После выполнения цикла значение параметра цикла не определено. Значит, использовать это значение нельзя: такая программа может по-разному работать на разных компьютерах. А впрочем, это значение по окончании цикла и не нужно, параметр используется в теле цикла, все значения, которые он должен принимать, известны из заголовка.
3. Параметр цикла внутри цикла менять нельзя! Почему нельзя? Кто запретил? А вот возьмем и поменяем: иногда ведь так хочется, чтобы, например, приращение параметра было не 1, а 2 или 10! Работа такой программы совершенно непредсказуема: некоторые трансляторы обнаруживают ошибку и «отказываются» компилировать программу, другие ошибку не замечают, но работают «по-своему». Правилами языка работа в таких случаях запрещена. Программа может зациклиться, может проработать «нормально», столько раз, сколько нам бы и хотелось, а может столько раз, сколько решит «сама». Причем на разных компьютерах в разное время эти эксперименты могут закончиться по-разному. Поэтому даже и пробовать не будем, просто запомним, что так делать нельзя.
4. Если цикл ни разу не работает, никаких присваиваний не происходит, хотя проверки, необходимые для выяснения, сколько раз цикл должен выполняться, производятся. Например, после выполнения фрагмента:

```
N:=2; I:=10;  
for I:=1 to N-5 do;
```

значение переменной  $N$  останется прежним (2), а значение  $I$ , вообще говоря, будет не определено (см. п. 2 выше). Присваивание  $I:=1$  из заголовка цикла не «сработает».

## Решение задач с помощью оператора цикла с параметром

*Пример 9.6. Найти  $N!$  ( $N$  натуральное).*

Вспомним, что факториал числа определяется формулой:

$$N! = 1 * 2 * 3 * 4 * \dots * (N-1) * N, \quad N \geq 1$$

И напишем программу, реализующую эту формулу:

```
Program Factorial1;
  var N,I,F:integer; {Заданное число, параметр цикла,
                                                              факториал}
Begin
  Writeln('Нахождение N!');
  Write('N='); Readln(N);
  F:=1;
  for I:=1 to N do
    F:=F*I;
  Writeln('Ответ: ',N,'!=',F)
End.
```

А можно сделать и с уменьшающимся параметром:

```
Program Factorial2;
  var N,F:integer; {Заданное число, факториал}
Begin
  Writeln('Нахождение N!');
  Write('N='); Readln(N);
  F:=1;
  for N:=N downto 2 do
    F:=F*N;
  Writeln('Ответ: ',F)
End.
```

В этой программе мы немножко «сэкономили»:

- 1) на одном шаге цикла, так как вспомнили, что на 1 умножать не обязательно;
- 2) на переменных — обошлись без переменной *I* (правда, из-за этого мы испортили *N*, и теперь ее нельзя использовать в ответе после цикла).

**Замечание.** Посчитайте факториал для разных значений *N*. Вы обязательно обнаружите, что, начиная с какого-то значения (для разных версий Паскаля оно может быть разным, например, для Турбо Паскаля это 8), ответ получается неправильным, отрицательным. В чем дело? Мы уже говорили, что в компьютере не бывает бесконечно больших чисел, и тип **Integer**, как и все остальные, ограничен. Как только получается значение, большее **MaxInt** (см. гл. 5, раздел «Целые числа»), вычисления ведутся неправильно: получающиеся значения просто не помещаются в отведенные для них ячейки. Как быть? Можно, например, использовать тип **LongInt**, который есть в некоторых версиях Паскаля (это несколько увеличит диапазон значений, для которых получается правильный ответ, но, конечно же, бесконечным его тоже не сделает). Важно знать, что ответ получается неверный не

из-за того, что алгоритм в принципе неправильный, а из-за того, что программа в данной версии Паскаля применима только к достаточно небольшим значениям  $N$ .

**Пример 9.7.** Написать программу возведения вещественного числа в натуральную степень.

Число можно возвести в натуральную степень путем умножения.

```
Program Stepen;
var A,P : Real; {Основание степени, ответ}
    N,I : Integer; {Показатель степени, параметр цикла}
Begin
    Write('Введите основание степени '); Readln(A);
    Write('Введите показатель степени '); Readln(N);
    P:=1;
    for I:=1 to N do
        P:=P*A;
    Writeln('Ответ: ',A,'^',N,'=',P)
End.
```

Не пугайтесь, если программа будет давать неверные ответы для слишком больших чисел (см. замечание к предыдущей задаче).

**Замечание.** Можно ли написать более экономичный алгоритм? Подумайте, как это сделать (например, получить  $a^4$  можно так:  $a^4 = (a^2)^2$ ).

Вспомним, что параметр цикла и выражения в заголовке цикла могут быть не только целыми, а символьными или логическими.

**Пример 9.8.** Выписать все маленькие латинские буквы по алфавиту.

Задача решается в одно действие. Приведем здесь только оператор цикла (C типа **Char**):

```
for C:= 'a' to 'z' do
    Write(C:2)
```

**Пример 9.9.** Напечатать таблицу значений функции  $Y = \text{Not}(X)$ .

```
Program FNot ;
var X:Boolean;
Begin
    Writeln('          X          Not(X) ');
    for X:=False to True do
        Writeln(X:10, Not(X):10);
End.
```

Обратите внимание, назвать программу **Not** нельзя, так как **Not** — служебное слово, его нельзя использовать как имя.

Шаг изменения параметра в цикле **For** для целых чисел равен 1. А можно ли пользоваться этим циклом, если в задаче нужен другой шаг? Можно, если число повторений заранее известно (например, его можно посчитать). Параметр в таком цикле будет, как и положено, меняться с шагом 1, а вот какое-то другое значение можно будет менять с заданным в задаче шагом.

***Пример 9.10.** Напечатать таблицу значений синуса и косинуса для всех углов от  $0^\circ$  до  $180^\circ$  с шагом  $30^\circ$ .*

$(180 - 0)/30 + 1 = 7$  — столько значений углов с шагом 30 поместится в отрезке  $[0, 180]$  (включая границы). Поэтому здесь можно использовать любой цикл **For**, обеспечивающий 7 повторений, например от 1 до 7 или от 0 до 6. Угол вначале равен своему начальному значению — 0, в цикле он увеличивается на 30. При этом мы не забываем, что в Паскале тригонометрические функции вычисляются в радианах, и переводим значение нашего угла из градусов в радианы.

```
Program SinCos;
var I,X:Integer;
    Y  : Real;
Begin
    X:=0;
    Writeln('N      угол      sin      cos');
    for I:=1 to 7 do
    Begin Y:=Pi*X/180;
        Writeln(I:2,X:8, Sin(Y):12:2, Cos(Y):12:2);
        X:=X+30
    End;
End.
```

Можно было рассчитывать  $X$  и по-другому: учесть его зависимость от  $I$ . Если взять  $I$  от 0 до 6, то  $X = I * 30$ .

## Задачи 9.13–9.21. Цикл For

- 9.13. Напечатать строчку звездочек.
- 9.14. Напечатать все большие латинские буквы в порядке, обратном алфавитному.
- 9.15. Напечатать все трехзначные числа, в которых нет цифры 5.
- 9.16. Выписать все делители заданного числа
- 9.17. С клавиатуры вводятся целые  $N \leq K$ . Найти сумму всех целых чисел от  $N$  до  $K$  включительно (сделайте это двумя способами: с помощью оператора цикла и по формуле).



- 9.18. Напечатайте  $N$  строчек таблицы значений функции  $F(x) = (1 - x)/x$ . Количество строчек, начальное и конечное значения  $x$  вводятся с клавиатуры.
- 9.19. Напечатать символы и соответствующие им коды от символа с кодом  $K1$  до символа с кодом  $K2$ .
- 9.20. Не используя операцию умножения, выписать 10 строчек таблицы умножения на 5.
- 9.21. Подсчитать сумму  $N$  слагаемых ( $N$  вводится с клавиатуры). Выписать получившиеся слагаемые, все промежуточные суммы и полную сумму.

а)  $-1 + 2 - 3 + 4 - 5 + \dots;$

б)  $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \dots$

в)  $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \dots$

г)  $\frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \frac{4}{5} + \frac{5}{6} + \dots$

д)  $\frac{1}{3} - \frac{2}{5} + \frac{3}{7} - \frac{4}{9} + \frac{5}{11} - \dots$

## Задачи 9.22–9.24. Разные циклы

- 9.22. Заполнить таблицу (написать примеры соответствующих циклов).

	цикл <b>While</b>	цикл <b>Repeat</b>	цикл <b>For</b>
Выполняется 5 раз			
Выполняется 1 раз			
Не выполняется ни разу			
Выполняется бесконечно много раз			
Выполняется $K$ раз			

9.23. Что будет напечатано в результате выполнения следующего фрагмента программы? Напишите фрагменты для выполнения тех же действий с двумя другими операторами цикла.

- а) For C:='b' to 'h' do Write (pred(C));
- б) For C:='F' downto 'K' do Write (succ(C));
- в) For I:=-2 downto 3 do Write(I);
- г) For I:=20 to 30 do Write(I/2);
- д) For I:=0 downto -3 do Write(2\*I);
- е) N:=1; Repeat Write(N); N:=N-1 Until N=1;
- ж) N:=100; Repeat Write(N); N:=N div 10 Until N=1;
- з) P:=0.5; Repeat P:=P\*2; Write(P) Until P>3.3;
- и) P:=0.2; Repeat Write(P); P:=P+0.2 Until P<0.3;
- к) K:=-2; While K<0 do K:=K+1; Write(K);
- л) K:=-1; While K>0 do Begin K:=K+0.1; Write(K) End;
- м) L:=81; While L>0.5 do Begin Write(L); L:=L/3 End;
- н) L:=1.1; While L<>2 do Begin Write(L); L:=L+0.2 End;
- о) L:=2; While L<8 do Begin L:=L\*2; Write(L) End;
- п) L:=1000; While L<1001 do Write(L); L:=L+1.

9.24. Выписать все делители заданного числа. Решить задачу тремя способами — с использованием разных операторов цикла.

## Цикл со сложным условием. Досрочный выход из цикла

*Пример 9.11. Составить программу, отвечающую на вопрос: является ли заданное натуральное число  $N$  простым?*

Напомним, простое число делится без остатка только на 1 и само на себя. Именно это в программе и проверим: будем смотреть, делится ли число на 2, 3, 4, 5 и т. д. Когда остановимся? Можно, конечно, проверить все предполагаемые делители до  $N-1$ , но это лишнее. Достаточно проверить до  $\sqrt{N}$ .

Для того чтобы «запомнить», разделилось ли наше число хотя бы на один из делителей, мы введем логическую переменную  $T$ . Ее значение в нашей программе равносильно значению высказывания « $N$  — простое число». Как только найден делитель, становится ясно, что число простым не является, присваиваем **T:=False**. Блок-схема представлена на рис. 9.6 (действия, происходящие внутри цикла, мы изобразили в шестиугольнике).

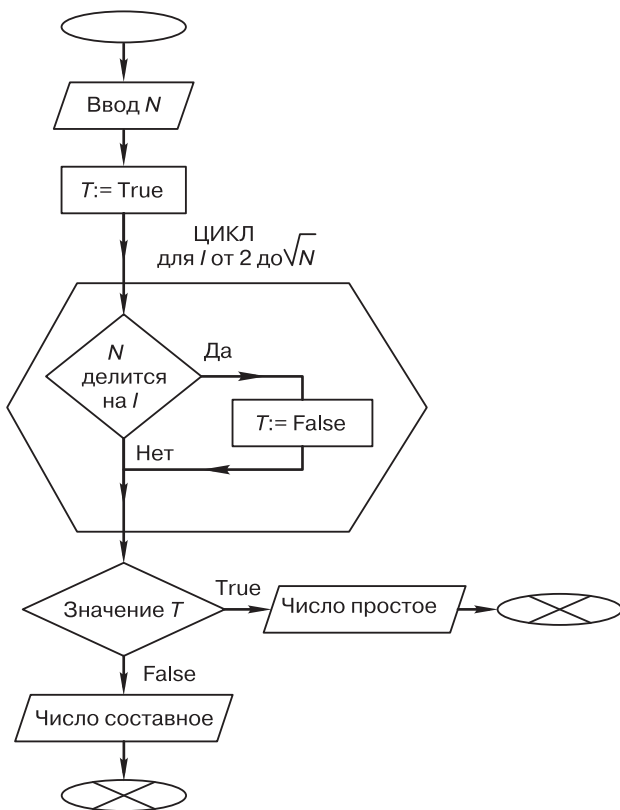


Рис. 9.6

Внимание!  $\sqrt{N}$  — всегда вещественное число, в Паскале для использования в заголовке оператора **For** его надо превращать в целое, взять его целую часть.

```

Program Prost;
var I,N:Integer;
    T :Boolean;
Begin
    Write('Введите число '); Readln(N);
    T:=True;
    for I:=2 to Round(Sqrt(N)) do
        If N mod I =0 Then T:=False;
    Write ('Число ',N, ' - ');
    If T Then Writeln ('простое')
    Else Writeln ('составное')
End.
  
```

Рационален ли наш алгоритм? Допустим, введено число 6842 — четное, на первом же шаге цикла мы выясняем, что оно простым не является, однако продолжаем его исследовать.

Значит, можно выходить из цикла раньше, как только обнаружится, что число простым не является. Таким образом, число повторений сразу посчитать нельзя, здесь более удобен цикл с предусловием, из которого надо выйти, как только обнаружится, что число имеет делитель. Однако и про выход из цикла, когда  $I > \text{Round}(\text{Sqrt}(N))$ , забывать нельзя — по этому условию мы выйдем из цикла, если число является простым. Также не забудем, что теперь придется написать оператор увеличения счетчика: в цикле **For** это происходило автоматически. Приведем здесь только фрагменты блок-схемы (рис. 9.7) и программы, которые изменились.

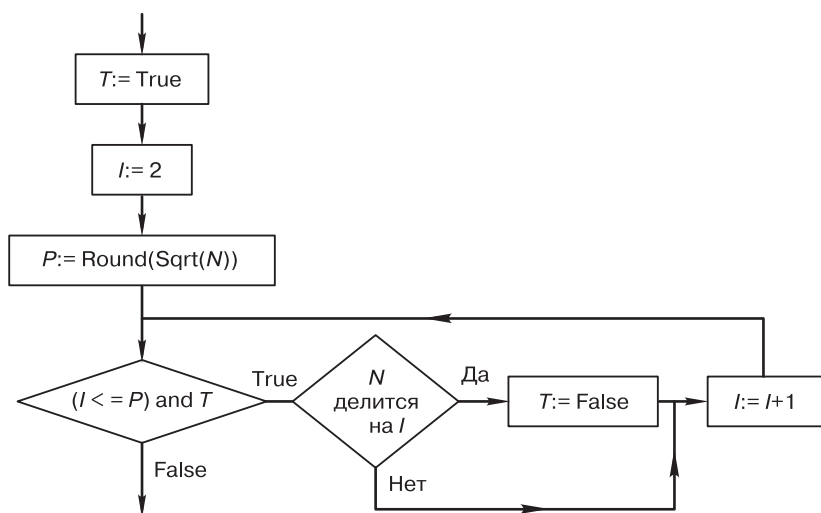


Рис. 9.7

```
T:=True; I:=2; P:=Round(Sqrt(N));
While (I<=P) and T do
Begin If N mod I =0 Then T:=False;
      I:=I+1
End;
```

Обратите внимание: вычисление квадратного корня из числа мы вынесли из цикла, присвоили это значение специальной переменной. Зачем? Чтобы вычислить его один раз, до цикла, запомнить в переменной  $P$ , а потом использовать в цикле, а не вычислять каждый раз в цикле заново. Можно подумать, что это ненужная экономия, ведь мы никак не сможем заметить,

что такая программа работает быстрее. Да, пока программа маленькая — ничего не заметим (уж слишком быстро она работает), но учиться писать эффективные программы надо сразу.

Подумайте, почему в предыдущем варианте программы с циклом **For** мы не вынесли вычисление квадратного корня из заголовка цикла?

Можно написать эту программу и с циклом **Repeat** — условие будет противоположным: **(I>P) or not T**.

А можно обойтись без логической переменной *T* — ведь условие теперь можно проверять и в заголовке цикла. Только получается, что, избавившись от переменной, мы сэкономили очень немного. Из цикла мы выходим в двух случаях. Чтобы дать правильный ответ, надо знать, какое из условий привело к выходу из цикла. Какое же из двух условий проверять? Если оба условия выполняться одновременно не могут, то все равно какое. В нашем случае оба условия могут выполняться только для числа 2 (это вообще достаточно уникальное число — единственное одновременно и четное, и простое). Число 2 (смотрим наши условия выхода из цикла) делится на 2 (это первый делитель, который мы проверяем) и этот делитель больше  $\sqrt{2}$ . Никакое другое число таким условиям удовлетворять не может. Значит, ради числа 2 надо проверять первое условие. Запишем фрагмент программы с обоими циклами:

<pre>I:=2;P:= Round(Sqrt(N)); While(I&lt;=P) and (N mod I&gt;0)do     I:=I+1;</pre>	<pre>I:=1; P:= Round(Sqrt(N)); Repeat  I:=I+1 Until(I&gt;P) Or (N mod I =0);</pre>
---	--

---

```
Write ('Число ',N, ' - ');
If I>P Then Writeln ('простое')
Else Writeln ('составное')
```

Обратите внимание на разные начальные значения переменной *I* в разных операторах цикла. Так получается потому, что в **While** сначала проверяется делимость, а потом увеличивается *I*, а в **Repeat** — наоборот.

Какой из приведенных здесь вариантов этой программы лучше? Явно проигрывает самый первый — с циклом **For**, где может быть выполнено слишком много лишних проверок. Какой выбрать из остальных — дело личных предпочтений. С одной стороны, в программе с логической переменной нет повторной проверки условия. С другой стороны, программы без переменной для кого-то понятнее (не надо запоминать, что логическая переменная означает).

## Процедура Break

Но существует еще один способ. В большинстве версий Паскаля есть специальная встроенная процедура для досрочного выхода из цикла — **Break**.

Эта процедура может применяться только внутри оператора цикла любого вида, ее использование вне цикла приведет к ошибке. Процедура производит выход из цикла, после ее применения выполняется следующий за циклом оператор (или программа заканчивается, если никаких операторов больше нет).

Нашу программу с циклом **For** можно переписать, применив **Break**, как только обнаружится, что число не является простым.

```
For I:=2 to Round(Sqrt(N)) do
    If N mod I =0 Then Break
{Сюда попадаем в случае выполнения процедуры Break
и в случае нормального завершения цикла}
```

Выиграли ли мы что-нибудь от этого по сравнению с программами с циклами **Repeat** и **While**? Вряд ли. Ведь мы опять получили два случая выхода из цикла: при выполнении процедуры и «нормальный». Значит, для того, чтобы выдать ответ, придется либо пользоваться все той же логической переменной, либо лишний раз проверять одно из условий. Попробуем определить какое. В обоих условиях фигурирует параметр цикла  $I$ , а мы знаем, что по завершении цикла его значение не определено. Таким образом, условное выражение будет «устойчиво» вычисляться только при досрочном выходе из цикла (в этом случае значение параметра определено — такое, до которого успел «дойти» цикл). К тому же при  $N < 4$  цикл не работает,  $I$  не определено вообще.

Получается, что надо использовать логическую переменную. Программа не будет сильно отличаться от программы с **While** (второй из предложенных нами):

```
T:=True;
for I:=2 to Round(Sqrt(N)) do
    If N mod I =0 Then Begin T:=False; Break End;
Write ('Число ',N, ' - ');
If T Then Writeln ('простое')
    Else Writeln ('составное')
```

Исходя из этого, можно сказать, что лучше использовать циклы **While** и **Repeat**, чем цикл **For** с принудительным выходом. Процедурой **Break** следует пользоваться только в случае, когда это действительно оправдано и дает ощутимые преимущества.

Часто вместо **Break** ошибочно используют процедуры **Exit** и **Halt**, считая их ее аналогами. Это неправильно, процедуры работают по-разному. **Break** осуществляет выход из цикла, **Exit** — из процедуры, **Halt** — из всей программы. Если программа маленькая, состоит из одного оператора цикла, работа процедур выглядит одинаково, но привыкать к их неправильному использованию не стоит.

Мы выяснили, что применение цикла **For** с досрочным выходом не всегда оправдано, но иногда дает хороший результат. А есть ли смысл применять процедуру **Break** в циклах **While** и **Repeat**? Синтаксис языка это позволяет. Пожалуй, существует немного задач, в которых применение **Break** в циклах с пред- и постусловием сделает программу более эффективной, а вот менее наглядной она станет обязательно!

Человек, читающий программу, обращает внимание на условие выхода из цикла, записанное в заголовке. Условие выхода по **Break** спрятано в теле цикла, его нужно рассматривать отдельно. К тому же сам программист при написании программы должен помнить, что к оператору, стоящему после цикла, он может подойти не только при нормальном завершении цикла (условие которого записано в заголовке), но и при «аварийном» выходе, т. е. при каком-то другом условии.

В справочной системе Турбо Паскаля в качестве примера применения процедуры **Break** в цикле **While** приводится вот такой текст:

```
While True do
    Begin Readln(S);
        If S='*' Then Break;
    End
```

В заголовке «заявлен» бесконечный цикл, а потом в теле цикла имеется «тайный» выход из него. Такой вот «обман», чтобы враги не догадались. Можно, конечно, в таком стиле писать все программы: бесконечный цикл, а все условия выхода — в виде условных операторов с **Break** в теле цикла. Программа будет с трудом понимаема всеми, в том числе и самим автором. Так как нашей целью является научиться писать программы, не только правильно работающие, но и «хорошо выглядящие», красивые и понятные, не стоит злоупотреблять процедурой **Break**. Особенно следует задуматься, если возникает желание использовать ее в циклах **While** и **Repeat**.

## Обработка последовательностей

Последовательность — набор (иногда бесконечный) однотипных элементов, который обычно обрабатывается с помощью цикла. Элементы последовательности могут вычисляться по каким-либо правилам, а могут вводиться извне (например, с клавиатуры).

## Элементы последовательности задаются с помощью формулы

Если элементы последовательности задаются с помощью формулы, описывающей зависимость элемента от его номера, то во многих случаях задача решается простым переписыванием на языке Паскаль заданной формулы и с помощью одного из циклов. Например, если  $a_i = 1/i^2$ , то первые  $N$  членов последовательности выписываются с помощью цикла:

```
For I:=1 to N do Write(1/Sqr(I):7:3);
```

а все элементы, большие, например, 0.05, — с помощью цикла:

```
I:=1; X:=1;  
While X>0.05 do Begin Write(X:7:3);  
                      I:=I+1;  
                      X:=1/Sqr(I)  
End
```

Последовательность может задаваться **рекуррентной** формулой: когда очередной член последовательности выражается через один или несколько предыдущих.

***Пример 9.12.** Числа Фибоначчи (одного из самых известных математиков средневековой Европы) определяются следующими формулами:*

$$F_0 = F_1 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{при} \quad n = 2, 3, \dots$$

*Напечатать все числа Фибоначчи, которые меньше некоторого заданного числа.*

Количество таких чисел нам неизвестно, поэтому воспользуемся циклом **while**. В теле цикла аккуратно производим присваивания, учитывая, что очередное число становится предыдущим при вычислении следующего.

```
Program Fib;  
var G,F,H,M:Integer;  
Begin  
  Write('Введите число-ограничитель '); Readln(M);  
  G:=1; F:=1; {G=F0, F=F1}  
  While F<M do  
    Begin  
      Write(F:5);  
      H:=G; {G=F[n-2], F=F[n-1]}  
      G:=F; F:=H+G {G=F[n-1], F=F[n]}  
    End;  
  Writeln  
End.
```



Иногда бывает, что вычислять члены последовательности надо именно по рекуррентной формуле, хотя в исходной формулировке об этом не сказано ни слова.

**Пример 9.13\*.** Пусть члены последовательности вычисляются по формуле:

$$A_m = (-1)^{m-1} \frac{x^{2m-1}}{(2m-1)!}$$

Надо для заданного  $X$  найти сумму всех членов, по модулю больших некоторого числа  $Eps$ .

Подставим  $m = 1, 2, 3$  и посмотрим, как же выглядит последовательность. Получим  $A_1 = x$ ,  $A_2 = -x^3/3!$ ,  $A_3 = x^5/5!$ . Искомая сумма выглядит так:

$$S = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^{m-1} \frac{x^{2m-1}}{(2m-1)!} + \dots$$

Если мы будем искать члены последовательности по формуле, нам придется каждый раз повторно проделывать много лишней работы. Так, для нахождения, например,  $A_3$  мы вычислим  $x^5$  (а это придется делать в цикле посредством умножения), вычислим  $5!$  (тоже в цикле), а затем будем таким же образом считать  $A_6$ , хотя понятно, что для вычисления  $x^7$  достаточно  $x^5$  умножить на  $x^2$ , а  $7!$  можно получить из  $5!$  домножением его на 6 и 7. Следовательно, оказывается удобным производить вычисления по рекуррентной формуле. Получим ее:

$$\frac{A_{m+1}}{A_m} = \frac{(-1)^m}{(-1)^{m-1}} * \frac{x^{2m+1}}{x^{2m-1}} * \frac{(2m-1)!}{(2m+1)!} = -x^2 * \frac{1}{(2m)(2m+1)}$$

Таким образом,  $A_{m+1} = -A_m * \frac{x^2}{(2m)(2m+1)}$ .

Вычисления нужно проводить по этой формуле не только потому, что программа получится более эффективной. Дело в том, что при расчетах по первой формуле для «далеких» членов последовательности в числителе и в знаменателе могут получаться очень большие числа (мы ведь уже пробовали считать факториал!). Поэтому, хотя сам член последовательности  $A_m$  небольшой,  $x^{2m}$  и  $(2m)!$  могут оказаться настолько большими, что не поместятся в отведенную для них память, и вычисления будут прерваны. При использовании рекуррентной формулы вычисления пройдут до конца.

Надо сказать, что члены этой последовательности, начиная с некоторого слагаемого, с увеличением  $m$  по абсолютной величине уменьшаются, поэтому задача поставлена корректно.

Искомая сумма сначала равна 0, первое слагаемое  $A = X$ , а его номер 1. Теперь осталось выписать цикл, в тело которого записать вычисление суммы, подготовленную выше формулу получения очередного слагаемого, а также не забыть про увеличение номера слагаемого. Можно заметить, что в формуле встречается удвоенный номер, — будем именно его использовать, поэтому примем изначально  $M = 2$ , и увеличивать его будем на 2, в формуле же  $2 * M$  заменим на  $M$ .

```
Program Sinus;
Var A,X,S,Eps  : Real;
    M:Integer;
Begin
    Write(' X= '); Readln(X);
    Write('Точность Eps='); Readln(Eps);
    S:=0; A:=X; M:=2;
    While Abs(A)>Eps do
    Begin S:=S+A;
        A:=-A*X*X/ (M* (M+1) );
        M:=M+2
    End;
    Writeln (S:20:10);
    Writeln(sin(x):20:10)
End.
```

Проверяя работу этой программы, имейте в виду, что  $X$  — вещественное, можно использовать также числа, меньшие 1. Программа будет правильно работать для не очень больших  $X$  (до 20), иначе получаются слишком большие числа, вычисления с которыми (мы уже об этом говорили) производятся неправильно.

В конце программы выведено не только полученное значение суммы, но и значение  $\sin(x)$ . Проверьте — они равны (или очень близки)! Это не случайность. Дело в том, что эта бесконечная последовательность — так называемое разложение функции  $\sin(x)$  в ряд Тейлора ( $x$  — радианы). Таким образом, мы научились вычислять значение функции  $\sin$ , не пользуясь ни таблицами, ни стандартной функцией языка. Впрочем, мы говорили, что наша программа работает только для  $|X| < 20$ , как же вычислить синус для всех остальных  $X$ ? Можно несколько увеличить возможности программы, воспользовавшись типом **Extended** вместо **Real**. Но следует учесть, что  $X = 20$  — это угол 20 радиан (а не градусов!!), значения синуса для таких больших углов можно вычислять, пользуясь свойством периодичности этой функции.

Мы не будем углубляться в высшую математику, в курсе которой подобные разложения изучаются, скажем только, что все элементарные функции представляются в виде подобных бесконечных сумм. Вычислив сумму некоторого количества первых элементов такой последовательности, можно получить приближенное значение функции. Чем большее количество элементов складывать, тем точнее будет значение. Слагаемые в этих суммах, начиная с некоторого номера, уменьшаются, поэтому вычисления можно производить с некоторой точностью (пока слагаемые не станут «очень маленькими», меньше заданной точности).

## Элементы последовательности вводятся с клавиатуры

Когда последовательность вводится с клавиатуры, может быть задано количество элементов в ней. В этом случае задача чаще всего решается с помощью цикла **For**. В цикле надо вводить очередной элемент последовательности и сразу же его обрабатывать, потому что на следующем шаге мы его уже «забудем».

Программа для решения большинства задач, где последовательность вводится извне, обычно разбивается на 3 части:

- 1) перед циклом — присваивание переменным начальных значений;
- 2) в теле цикла — ввод очередного члена последовательности и его обработка;
- 3) после цикла — выдача ответа.

**Пример 9.14.** *С клавиатуры вводится последовательность из  $N$  символов. Сколько среди них цифр?*

Как определить, является ли символ цифрой, мы уже знаем. Напомним, что для этого не нужно знать никаких кодов (тем более, они могут различаться в разных версиях). Важно знать, что в любой кодировке цифры-символы идут подряд, от '0' до '9'.

```

Program Cifra;
var I,N,K : Integer;
    C      : Char;
Begin
    K:=0;
    Write('Введите количество элементов
                                     последовательности ');
    Readln(N);
    for I:=1 to N do
    Begin Write('Введите ',I,' элемент '); Readln(C);
        If (C>='0') and (C<='9') Then K:=K+1;
    End;
    Write('Введено ',K,' цифр')
End.
```

Иногда бывает, что при некоторых условиях не надо вводить всю последовательность целиком: если решение задачи найдено, ввод можно прекратить. Вот пример такой задачи.

**Пример 9.15.** *С клавиатуры вводится последовательность из  $N$  целых чисел. Все ли числа в ней четные?*

```
var N,A,I : Integer; {количество чисел, само число}
    Nechet: Boolean; {сигнал, что появилось нечетное
                                                              число}
Begin
    Write('Количество элементов последовательности ');
                                                    Readln(N);
    Nechet:=False; {пока нечетных чисел не обнаружено}
    I:=1; {начнем с первого элемента}
    While (I<=N)and Not Nechet Do
{Будем продолжать цикл, если еще не просмотрены все
элементы последовательности и не встретили нечетное
число}
        Begin Write(' Введите ',I,' число '); Readln(A);
                If Odd(A) Then Nechet:=True;
                I:=I+1
        End;
    Write('В последовательности ');
    If Nechet then Writeln('не все числа четные')
        Else Writeln('все числа четные')
End.
```

Если все числа в последовательности четные, нам придется их все ввести (а как иначе проверить?). А вот как только встретилось нечетное число, работу можно закончить, остальные числа не вводить — ответ и так ясен.

**Замечание.** Эту задачу можно решить, используя цикл **For** и процедуру **Break** (заменяем один цикл на другой и добавляем **Break** после **Then** — там теперь будет составной оператор). Сделайте это самостоятельно.

Число элементов в последовательности может быть заранее неизвестно: задается, чем заканчивается последовательность. Вот пример такой задачи.

**Пример 9.16.** *С клавиатуры вводится слово (последовательность букв) с точкой в конце. Определить, сколько раз входит в слово заданная буква, выписать, на каких позициях в слове она стоит.*

Для того чтобы определить, сколько раз в последовательности встречается заданный символ, необходим счетчик, значение которого в начале равно

нулю, а как только вводится нужная буква — увеличивается на единицу. Но этого счетчика для решения задачи не хватит, придется завести еще один, который будет увеличиваться на единицу, когда введена любая буква, — он нужен для определения номера позиции.

Обратите внимание, буквы считываются оператором **Read** (а не **ReadLn**). В данной программе это принципиально, дает возможность правильно отобразить информацию на экране, не смешивая вводимое слово с выводимыми числами. При работе программы слово вводится сразу целиком, в одну строчку. Мы знаем, что, встретив оператор **Read**, программа останавливается и ждет ввода. В данном случае надо в ответ на первый оператор для ввода буквы ввести не первую букву слова, а все слово целиком (с точкой в конце). Оператор **Read** прочитает первую букву слова, программа ее обработает, напечатает, если надо, номер позиции. Когда цикл будет выполняться в следующий раз, оператор ввода не будет останавливать выполнение программы и ждать очередного ввода: будет прочитана следующая буква из уже напечатанного на экране набора (в этом и состоит отличие операторов **Read** и **ReadLn**). Таким образом, для человека за компьютером работа программы выглядит так, как будто слово вводится целиком, одним оператором. На самом деле оно целиком печатается на экране (и хранится в некоторой специальной памяти), а программа считывает буквы по одной по очереди в цикле.

Заметим, что если при работе с этой программой попытаться ввести слово по одной букве, после каждой нажимая клавишу ввода (как это было бы надо, если бы стоял оператор **ReadLn**), код клавиши ввода будет воспринят как код символа из последовательности и номера позиций будут определяться неправильно. Да и вводимые буквы будут напечатаны на экране вперемешку с числами выводимого решения.

```
Program Slovo;
var I,K : Integer; {номер позиции, количество искомых
                  символов}
    C,P : Char; {Очередная буква в слове и буква для поиска}
Begin
  Write('Введите букву, которую будем искать '); Readln(P);
  Write('Введите слово с точкой в конце ');
  I:=1; K:=0;
  Repeat
    Read(C);
    If C=P Then Begin Write(I:3); K:=K+1
                     End;
    I:=I+1
  Until C='.';
  If K=0 Then Writeln('Буква в последовательности не
                     встречается')
```

```
Else Begin Writeln(' - позиции, на которых стоит
                                         буква');
           Writeln('Буква встречается ',K, ' раз')
End.
End.
```

В подобных задачах, когда количество элементов не задается напрямую, последний элемент имеет особый статус: он не является членом последовательности, а служит для того, чтобы пометить ее конец. Важно так составить программу, чтобы нельзя было ни сосчитать, ни обработать этот последний элемент наряду с остальными «нормальными» членами последовательности. В предыдущей задаче мы об этом не задумывались: определяется наличие буквы в слове, лишняя проверка (когда проверяется, не равна ли точка искомой букве) на результат не влияет. Однако это не всегда так.

***Пример 9.17.** Непустая последовательность целых чисел заканчивается нулем. Найти их произведение.*

В этой задаче нам очень важно не включить 0 в число сомножителей, ведь в таком случае результат и считать не надо, всегда будет получаться 0. Задачу можно решить с помощью оператора **While** (при этом придется первое число прочитать отдельно, до цикла, чтобы в условии значение  $X$  было определено). В теле цикла сначала ставим вычисление произведения, а потом ввод. Таким образом, 0 введется после очередного перемножения и на этом цикл завершится.

```
Program Proizv;
var X,P:Real; {число из последовательности, произведение}
Begin
  Write('Введите 1-е число '); Readln(X);
  P:=1;
  While X<>0 do
  Begin P:=P*X;
        Write('Вводите число (признак конца - 0) ');
        Readln(X);
  End;
  Writeln('Произведение', P:9:2)
End.
```

Не нравится, что ввод числа пришлось записать два раза? Можно воспользоваться циклом **Repeat**, но тогда внутри цикла придется проверять, не ноль ли введен:

```
Program Proizv1;
var X,P:Real; {число из последовательности, произведение}
Begin
  P:=1;
```

```

Repeat
    Write('Вводите число (признак конца - 0) ');
    Readln(X);
    If X<>0 Then P:=P*X
Until X=0;
Writeln('Произведение', P:9:2)
End.

```

Как лучше? Ввести первое число надо всего один раз, а дополнительную проверку придется делать в цикле для каждого члена последовательности.

В подобных задачах элементами последовательности могут быть не только числа, но и символы.

**Пример 9.18.** *С клавиатуры вводится слово — последовательность букв, заканчивающаяся точкой. Одинаковы ли первая и последняя буквы этого слова?*

Обозначим  $B_1$  первую букву слова,  $B_p$  — последнюю. И нам понадобится еще одна переменная для ввода букв. Сравнить нужно всего две буквы, а переменных надо три! Зачем еще одна переменная? Дело в том, что последним введенным символом будет не буква, а точка (таково условие задачи, точка помечает конец слова). То есть на самом деле надо сравнивать первый символ введенной последовательности с предпоследним. А как догадаться, что очередной символ — предпоследний? Никак — ведь пока мы не ввели точку, мы ее не «видим». Придется все время перед вводом очередной буквы запоминать предыдущую. Запоминать будем в  $B_p$  (по окончании цикла в этой переменной будет храниться символ, стоящий перед точкой), а вводить очередную букву в  $B$ .

```

Program Bukv;
var B1, Bp, B : Char;
Begin
    Write('Введите слово с точкой в конце ');
    Read(B1); B:=B1;
    While (B<>'.') do
        Begin Bp:=B;
              Read(B)
        End;
    If B1='.'
        Then Writeln('Введено пустое слово')
        Else Begin Write('Первая и последняя буквы в слове ');
              If B1=Bp Then Writeln('равны')
              Else Writeln('не равны')
            End
    End.
End.

```

Наша программа правильно работает и в том случае, если сразу введена точка, — печатает соответствующее сообщение, и если перед точкой введена всего одна буква, — такое слово расценивается как слово, с совпадающими первой и последней буквами.

**Пример 9.19\*.** *Определить, является ли заданная последовательность символов, оканчивающаяся точкой, правильной записью целого десятичного числа (возможно, со знаком). Вывести введенное число или сообщение об ошибке.*

Число будем вводить «в строчку» с помощью процедуры **Read**. Сначала определим, не является ли первый символ знаком «+» или «-»: если это действительно так, запомним знак в переменной *Z* и прочитаем следующий символ.

Остальные символы будем вводить не до конца последовательности, а пока они «правильные», т. е. являются цифрами. Ведь как только попадет символ — не цифра, работу надо прекращать — понятно, что последовательность числом не является.

Введем счетчик *I* для подсчета цифр во введенном числе (позже поясним, зачем это нужно) и переменную *K* для накапливания ответа — числа.

В цикл мы попадаем, только если очередной член последовательности — цифра. Но эта цифра вводится как символ. Нам же нужна цифра-число (типа **Integer**). Такие цифры можно умножать и складывать, из них легко получить число (из цифр 3, 4 и 5 число получается так:  $((3 * 10) + 4) * 10 + 5$ ).

Как же из цифры-символа получить цифру-число? Можно, конечно, недолго думая, записать длинный оператор варианта (*C* — цифра-символ, *R* — соответствующая цифра-число):

```
Case C of
  '0':  R:=0;
  '1':  R:=1;
  ...
End;
```

и так далее до 9. Но ведь слишком длинно и однообразно получается, даже выписывать целиком не хочется. Попробуем немножко подумать и записать это короче.

Выведем формулу, связывающую код символа-цифры с соответствующей цифрой-числом. Пусть символ '0' имеет код **F**, тогда код '1' — **F+1** (ведь мы знаем, что цифры расположены подряд), код '2' — **F+2**, код '9' — **F+9**. То есть **Ord('1')=F+1**; **Ord('9')=F+9** — и так для всех цифр. Но ведь **F=Ord('0')**. Значит, если *C* — цифра-символ, то соответствующая цифра-число *R* получается по формуле:

```
R:=Ord(C)-Ord('0')
```



Цикл завершается, как только встретилась не цифра, причем точка тоже не является цифрой. Поэтому по окончании цикла надо проверить, если последний прочитанный символ — точка, значит, вся последовательность прочитана, других не цифр в ней нет. И необходима еще одна проверка: в последовательности может вообще не быть ничего (кроме завершающей точки), может быть только знак и точка. Такая последовательность тоже числом не является. Вот для этой проверки мы и считали количество введенных цифр — оно должно быть больше 0. В этом случае надо вывести получившееся число, не забыв учесть знак.

```

Program Celoe;
var Z,C : Char; {знак, обрабатываемый символ}
    K,I : Integer; {Получившееся число, количество
                    введенных цифр}
Begin
  Write('Число - ? ');
  Read(C);
  Z:='+'; {Определяем Z для случая, когда число
           вводится без знака}
  If (C='+') Or (C='-') Then Begin Z:=C; Read(C) End;
  K:=0; I:=0;
  While (C>='0') and (C<='9') do
  Begin K:=K*10+ord(C)-ord('0');
        I:=I+1;
        Read(C)
  End;
  If (I>0) and (C='.') Then Begin If Z='- ' Then K:=-K;
                                Writeln(K)
                                End
                                Else Writeln('Ошибка')
  End.

```

## Задачи 9.25–9.55. Работа с последовательностью

*В следующих задачах последовательность чисел задается формулой, где  $R$  — число,  $N$  — его номер. Тип чисел определите самостоятельно.*

9.25. Формула  $R_n = \sqrt{N}$ . Найти первый элемент последовательности, который превосходит заданное  $A$ . На каком месте он стоит?

9.26.  $R_n = \frac{1}{n^2}$ . Найти сумму всех членов последовательности, превосходящих заданное  $Eps$ .

- 9.27. Выписать  $N$  первых членов арифметической прогрессии (напомним,  $n$ -й член арифметической прогрессии с первым членом  $a_1$  и разностью  $d$  вычисляется по формуле:  $a_n = a_1 + (n - 1) * d$ ).
- 9.28. Выписать  $N$  первых членов геометрической прогрессии ( $n$ -й член геометрической прогрессии с первым членом  $b_1$  и знаменателем  $q$  (оба не равны 0) вычисляется по формуле:  $b_n = b_1 * q^{n-1}$ ).

**Числа Фибоначчи (определение см. в примере 9.12)**

- 9.29. Найти  $N$ -е число Фибоначчи.
- 9.30. Напечатать четные числа Фибоначчи из первых  $N$  чисел этой последовательности.
- 9.31. Вычислить сумму всех чисел Фибоначчи, которые меньше  $K$ .

**Вычисления по формулам**

- 9.32\*. Ниже представлены формулы, по которым вычисляются члены последовательности. Записанные суммы — разложение выписанных в левой части равенства функций в ряд Тейлора (см. пример 9.13). Подсчитать сумму в правой части равенства с точностью до некоторого  $Eps$ , убедиться, что она действительно равна с заданной точностью функции в левой части.

$$\text{а) } e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots, x \in (-\infty, +\infty)$$

$$\text{б) } \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots + (-1)^n \frac{x^{2n}}{(2n)!} + \dots, x \in (-\infty, +\infty)$$

$$\text{в) } \ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots + (-1)^{n+1} \frac{x^n}{n} - \dots, x \in (-1, 1]$$

$$\text{г) } \ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \dots - \frac{x^n}{n} - \dots, x \in [-1, 1)$$

$$\text{д) } (1+x)^m = 1 + mx + \frac{m(m-1)}{2!} x^2 + \dots + \frac{m(m-1)\dots(m-n+1)}{n!} x^n + \dots, x \in (-1, 1)$$

$$\text{е) } \operatorname{arctg}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \dots + (-1)^{n+1} \frac{x^{2n-1}}{2n-1} - \dots, x \in (-1, 1)$$

- 9.33\*. Для формул из задачи 9.32 составить  $K$  строк следующей таблицы:

- столбец 1 — значения  $Eps$  равны 0.1, 0.01, 0.001 и т. д.;
- столбец 2 — результат вычисления суммы последовательности с данной точностью;
- столбец 3 — количество слагаемых, которое потребовалось для получения результата.

***В задачах 9.34–9.55 последовательность вводится с клавиатуры. Условие каждой задачи может быть представлено в двух вариантах:***

- ***в последовательности  $N$  элементов ( $N$  вводится с клавиатуры);***
- ***количество элементов заранее неизвестно; последовательность завершается некоторым «особым» элементом, который ее членом не является, а используется только как признак конца.***

***В задачах 9.34–9.40 задается последовательность целых чисел.***

- 9.34. Найти среднее арифметическое четных чисел.
- 9.35. Каких чисел в последовательности больше — четных или нечетных?
- 9.36. Выписать числа, которые делятся на 3, но не делятся на 5.
- 9.37. Сколько чисел отличается от первого на 1?
- 9.38. Есть ли в последовательности нечетные числа?
- 9.39. Для всех членов последовательности, кроме первого и последнего, определить, сколько из них больше своих «соседей» (предыдущего и последующего чисел).
- 9.40. Является ли заданная последовательность знакопеременной?

***В задачах 9.41–9.46 задается последовательность вещественных чисел.***

- 9.41. Найти среднее арифметическое отрицательных членов последовательности.
- 9.42. Найти среднее геометрическое положительных членов последовательности.
- 9.43. Является ли заданная последовательность неубывающей?
- 9.44. Верно ли, что заданная последовательность убывающая?
- 9.45. Является ли заданная последовательность арифметической прогрессией?
- 9.46. Является ли заданная последовательность геометрической прогрессией?

***В задачах 9.47–9.55 задается последовательность символов. В качестве признака конца обычно используется точка. Последовательность символов может вводиться «в строчку» и трактоваться как «слово».***

- 9.47. Входит ли в последовательность заданный символ?
- 9.48. Сколько раз заданный символ входит в последовательность?
- 9.49. Сколько раз в слово входит его первая буква?
- 9.50. Является ли заданная последовательность символов латинским словом (начинается с большой или маленькой латинской буквы, все остальные символы — маленькие латинские буквы)?

- 9.51. Является ли заданная последовательность символов правильной записью двоичного числа (первым символом может быть знак «+» или «-»)?
- 9.52. Является ли данная последовательность символов правильной десятичной записью вещественного числа  $<1$  (вида 0.123)?
- 9.53. Состоит ли последовательность из одинаковых символов?
- 9.54. Вывести введенный текст, удалив из него все цифры.
- 9.55. Вывести введенный текст, удалив из него «лишние» пробелы, т. е. из нескольких подряд идущих пробелов оставить один.

## Вокруг максимума

**Поиск наибольшего (наименьшего) элемента в последовательности** — одна из стандартных, часто встречающихся задач. Остановимся на ней подробнее. Сначала сформулируем ее не для чисел, а возьмем пример «из жизни».

***Задача.** За оказанную услугу падишах разрешил Ходже Насреддину выбрать самый крупный алмаз из его казны. Он высыпал на стол  $N$  алмазов (на вид они все примерно одинаковые), дал Ходже чашечные весы (у этих весов две чашки, на каждую можно класть взвешиваемые предметы; более тяжелая чашка опускается вниз) и предупредил, что, если Ходжа определит самый тяжелый алмаз неправильно или будет слишком долго возиться, он отрубит ему голову. Какой алгоритм избрать Ходже?*

Как бы хотелось взвесить все камни по очереди на цифровых весах, записать вес каждого, выбрать наибольший... А как выбрать? Из 3, 5, даже 7 чисел мы с легкостью, с первого взгляда, выберем самое большое. А если чисел много, очень много (например, они записаны в несколько столбиков в книге размером с телефонный справочник)? Поэтому вернемся к Ходже и будем вместе с ним взвешивать по два алмаза (т. е. сравнивать по два числа).

Возьмем первый и второй алмазы, положим каждый на свою чашку весов. Взвесим. Более тяжелый оставим. А вместо более легкого возьмем следующий, пока еще не взвешенный алмаз. Таким образом, у нас на одной чашке весов всегда будет находиться самый тяжелый камень из всех уже взвешенных (текущий максимум, наибольшее число из рассмотренного куска последовательности). Сравнив вес этого камня с весом последнего, мы за  $N - 1$  взвешиваний определим самый тяжелый.

Обозначим номер самого тяжелого камня  $N_{\max}$ , а его вес  $V_{\max}$ . Вначале мы вынуждены предположить, что самым тяжелым является первый камень (максимальным в последовательности из одного числа является само это число), его мы положим на первую чашку весов. Номер очередного камня

будем записывать в переменную  $K$  (первоначально оно будет равно 2, ведь на другую чашку мы сначала положим второй камень), его вес обозначим  $V$ .  
Получается вот такая блок-схема (рис. 9.8).

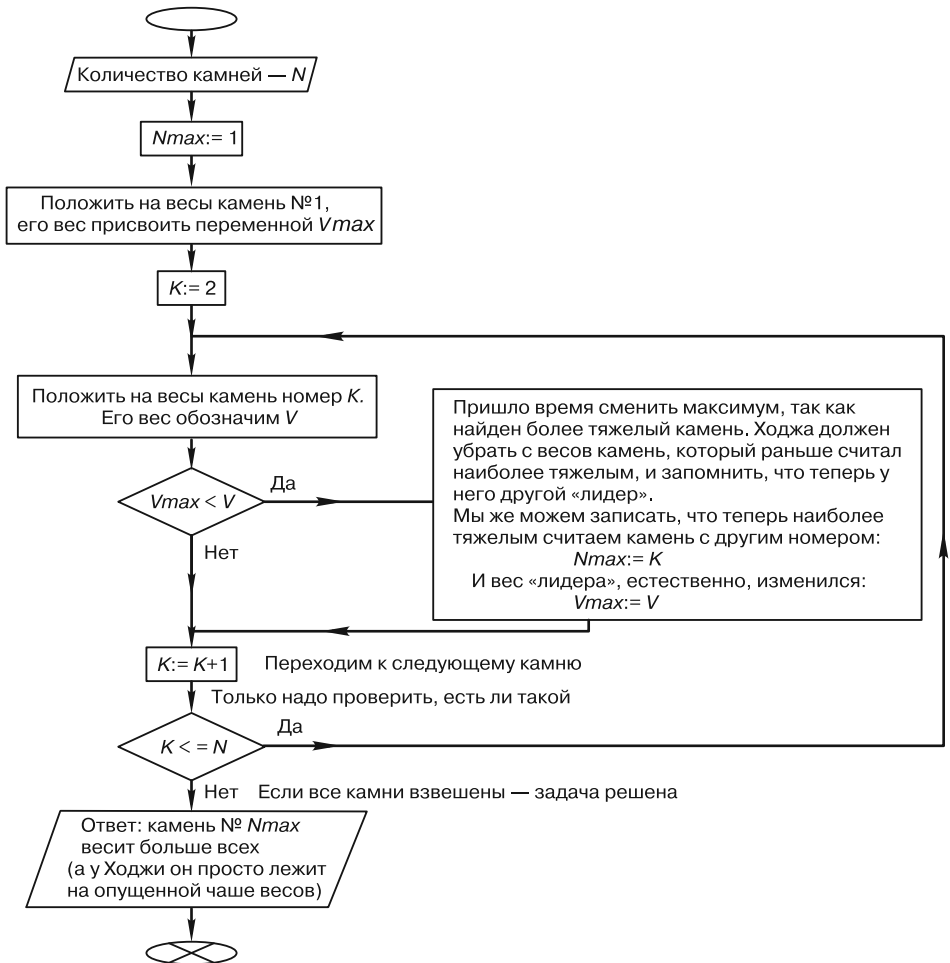


Рис. 9.8

**Задание.** Как будет работать алгоритм, если имеется несколько самых тяжелых камней одинакового веса? Найдется ли один из них? Какой: с наименьшим номером, с наибольшим, какой-то другой?

Вернемся от алмазов к точным формулировкам и решим эту задачу для последовательности чисел. Вот как ее можно сформулировать.

**Пример 9.20, а.** Среди  $N$  ( $N \geq 1$ ) вещественных чисел, вводимых с клавиатуры, найти наибольшее. На каком месте в последовательности стоит

*это наибольшее число? (Если такое число не одно, вывести номер одной из позиций.)*

Итак, мы все время ищем текущий максимум — наибольшее число из введенных. В последовательности из одного числа оно само наибольшим и является, поэтому первое число объявим максимальным — введем его сразу в переменную *VMax*. Это число, естественно, стоит на первом месте, поэтому переменной, предназначенной для номера позиции, присвоим 1. Цикл начнем с  $I = 2$ , ведь первое число мы уже ввели и обработали. В цикле будем вводить очередное число и, если оно больше *VMax*, будем менять текущий максимум и номер его позиции.

```
Program Maximum;
var K,N, Nmax : Integer; {счетчик, количество, место
                           максимума}
    V, VMax    : Real; {Очередное число, максимум}
Begin
Write('Количество элементов последовательности ');
Readln(N);
Write('Введите 1-е число '); Readln(VMax);
Nmax:=1;
for K:=2 to N do
Begin Write('Введите ',i,'-е число '); Readln(V);
    If V>VMax Then Begin VMax:=V;
                      Nmax:=K
                    End
End;
Writeln('Наибольшее число ', VMax:5:1);
Writeln('Его позиция ', Nmax)
End.
```

Если в последовательности несколько максимальных элементов, будет выведена позиция первого.

Обратите внимание: при работе с этой последовательностью (в которой может быть очень много чисел) мы не храним в памяти все числа, только два из них, с которыми работаем в каждый момент.

**Пример 9.20, б.** *А как посчитать, сколько их, этих максимальных элементов?*

Конечно, очень удобно бы было сначала найти максимум, а потом просмотреть последовательность еще раз и определить, сколько раз найденное число в ней встречается. Но последовательность мы нигде не храним, все надо решать за один просмотр. Введем дополнительный условный оператор, который будет проверять, не равно ли число текущему максимуму, — в этом случае их количество увеличивается на 1. А вот если текущий максимум

изменяется, все прежние подсчеты количества максимальных элементов надо забыть и начать счет снова с единицы.

```

Program SkMaxm;
var I,N, Sk : Integer; {счетчик, количество элементов,
                        сколько раз встречается Max}
    X, Max   : Real; {Очередное число, максимум}
Begin
    Write('Количество элементов последовательности ');
    Readln(N);
    Write(' Введите 1-е число '); Readln(Max);
    Sk:=1;
    for I:=2 to N do
    Begin Write('Введите ',i,'-е число '); Readln(X);
        If X=Max Then Sk:=Sk+1 {Количество максимальных
                                элементов увеличилось}
        Else
            If X>Max Then Begin Max:=X; {Новый максимальный
                                        элемент}
                            Sk:=1 {Подсчет надо начинать сначала}
                        End
        End;
    Writeln('Наибольшее число ', Max:5:1);
    Writeln('В последовательности таких чисел ', Sk)
End.

```

А как быть, если надо найти наибольший элемент не из всех элементов последовательности, а только среди некоторых, обладающих определенным свойством? Например:

**Пример 9.20, в.** *В последовательности из  $N$  целых чисел найти наибольшее нечетное число. Известно, что хотя бы одно нечетное число в последовательности есть.*

Понятно, что сравнивать нужно только нечетные числа. Если вводится четное число, надо просто пропускать его. Некоторая проблема состоит вот в чем: обратите внимание, в задаче поиска максимума мы вначале объявляли максимальным первое число. Здесь этого делать нельзя: ведь первое число может оказаться четным. Какое же число в начале объявить максимальным? Первое нечетное!

Таким образом, задача разбивается на две части.

1. Найти первое нечетное число в последовательности (в условии сказано, что оно обязательно встретится).
2. Обычным образом просматривать последовательность, сравнивая с текущим максимумом только нечетные числа.

```
Program Max2;
var J, I, N, Max, A : Integer;
Begin
    Write('Количество элементов последовательности');
                                         Readln(N);

    J:=1;
    Repeat
        Write('Введите ', J, ' элемент '); Readln(Max);
        J:=J+1
    Until Odd(Max);
    For I:= J to N do
    Begin Write('Введите ', I, ' элемент '); Readln(A);
        If Odd(A) Then If A>Max Then Max:=A
    End;
    Writeln(Max)
End.
```

Можно эту задачу решать и другим способом. Например, в качестве начального максимума взять самое маленькое нечетное число (отрицательное), но для этого его нужно знать или вычислить (через **MaxInt**). Мы же предлагаем более общий способ решения.

***Пример 9.20, г.** А теперь пусть нам надо найти не только самое большое число в последовательности, но и «предыдущее», то, которое будет стоять на втором месте, если числа в последовательности расположить по убыванию (оно может быть равно максимальному, а может быть меньше него). В последовательности не менее двух чисел.*

Так как надо найти два «максимума», вначале по аналогии со «стандартной» задачей по поиску максимума следует два первых числа последовательности присвоить переменным *MAX1* (здесь мы будем хранить самое большое число) и *MAXp* (здесь мы будем хранить второй «максимум»). Причем присваивание надо произвести так, чтобы *MAX1* был больше *MAXp*.

Далее, как и в «стандартной» задаче, вводим очередной элемент последовательности. Возможны 3 случая его расположения.

1. Очередной элемент больше *MAX1*. В этом случае надо поменять значение *MAX1* (оно станет равно введенному элементу) и *MAXp* (оно станет равно *MAX1*). Обратите внимание: эти присваивания надо делать в определенном порядке!
2. Очередной элемент находится между *MAXp* и *MAX1*. В этом случае меняется только *MAXp* (становится равно введенному элементу).
3. Очередной элемент меньше *MAXp* (в этом случае он, естественно, меньше и *MAX1*). Здесь ничего не меняется.



```
Program Max3;
var I, N : Integer;
    MAX1, MAXp, A: Real;
Begin
    Write('Количество элементов последовательности ');
    Readln(N);
    Write('Введите 1-й элемент '); Readln(MAX1);
    Write('Введите 2-й элемент '); Readln(A);
    If A<=MAX1 Then MAXp:=A
        Else Begin MAXp:=MAX1; MAX1:=A
            End;
    For I:= 3 to N do
    Begin Write('Введите ', I, '-й элемент ');
        Readln(A);
        If A>MAX1 Then Begin MAXp:=MAX1; MAX1:=A
            End
            Else If A>MAXp Then MAXp:=A
        End;
    End;
    Writeln(MAX1, ' ', MAXp)
End.
```

## Задачи 9.56–9.65. Поиск наибольших и наименьших значений

*В этих задачах последовательность чисел вводится с клавиатуры. Количество чисел либо задается до ввода последовательности, либо оно неизвестно, задается некоторый признак конца последовательности.*

- 9.56. Верно ли, что самое маленькое число в последовательности вещественных чисел введено раньше самого большого?
- 9.57. Найти разность между самым большим и самым маленьким числами в последовательности вещественных чисел.
- 9.58. Определить, сколько раз в последовательности символов встречается символ с самым большим и самым маленьким кодом.
- 9.59. Последовательность вещественных чисел. Найти самое большое число. На каком месте оно стоит? Если наибольших чисел в последовательности несколько, определить, на каком месте находятся первое и последнее вхождения.
- 9.60. Найти самое большое нечетное число в последовательности вещественных чисел (известно, что хотя бы одно нечетное число в последовательности есть).
- 9.61. Найти самое большое нечетное число в последовательности вещественных чисел, если же нечетных чисел нет, выдать сообщение об этом.

- 9.62. В последовательности не менее двух чисел. Найти 2 наименьших числа среди всех членов последовательности и определить, на каких местах они стоят.
- 9.63\*. В последовательности не менее трех чисел. Определить 2 наименьших не равных друг другу числа среди всех членов последовательности. Если это невозможно, напечатать соответствующее сообщение.
- 9.64\*. В последовательности не менее трех чисел. Определить 3 наибольших числа среди всех членов последовательности. Определить места, на которых они стоят.
- 9.65\*. Найти в последовательности целых чисел два самых маленьких числа, которые делятся на 3. Если таких чисел нет или такое число всего одно — выдать соответствующее сообщение.

## Вложенные циклы

В теле оператора цикла могут находиться какие угодно операторы, в том числе и операторы цикла. Например:

```
For I:=1 to 3 do
Begin Write(3);
    For J:=1 to 4 do
    Begin Write(4);
        For K:=1 to 5 do
            Write(5)
        End
    End
End
```

Обратите внимание на форму записи: тело каждого оператора цикла пишется несколько правее, соответствующие друг другу операторные скобки **Begin-End** стоят друг под другом. Благодаря такой форме записи мы можем судить об уровне вложенности оператора.

Следует учитывать, что чем больше уровень вложенности оператора, тем большее количество раз он будет выполняться. Так, в нашем примере **Write(3)** находится в теле всего одного цикла (первый уровень), число 3 напечатается 3 раза (столько, сколько раз выполняется первый цикл). Оператор **Write(4)** стоит на втором уровне, находится в теле двух циклов, выполнится  $3 \cdot 4 = 12$  раз, а оператор, стоящий на третьем уровне, **Write(5)** выполнится  $3 \cdot 4 \cdot 5 = 60$  раз!

***Задание.** Подумайте, что будет напечатано в результате выполнения данного фрагмента программы. Мы сказали, сколько раз напечатаются разные числа, осталось выяснить, в каком порядке они будут печататься.*

Обратите внимание, в нашем примере параметры операторов цикла все разные. Так обязательно должно быть, иначе будет изменение параметра цикла в теле цикла, что запрещено.

Часто бывает, что для решения задачи с вложенными циклами удобно сначала написать внутренний цикл, проверить его работу, а потом его «заключить» во внешний цикл. Рассмотрим пример.

**Пример 9.21.** Вывести все простые числа от 2 до некоторого заданного  $K > 2$ .

Для одного числа мы эту задачу уже решали (см. пример 9.11). Там мы определяли, является ли число  $N$  простым. Для решения нашей задачи достаточно в той программе заменить ввод числа  $N$  на цикл **For N:=2 to K do** и не забыть заключить тело этого цикла в операторные скобки.

```
Program Prost1;  
var I,N,K:Integer;  
    T      :Boolean;  
Begin  
    Write('число '); Readln(K);  
    For N:=2 to K do  
    Begin T:=True;  
        for I:=2 to Round(Sqrt(N)) do  
            If N mod I =0 Then T:=False;  
        If T Then Writeln (N:6)  
    End  
End.
```

Вложенные, а точнее, объемлющие циклы удобно использовать, когда хочется получить несколько решений своей задачи (для разных входных данных).

Например, несколько раньше мы рассмотрели пример 9.7, написали программу для возведения числа в целую неотрицательную степень. Можно сделать, чтобы эти действия выполнялись в цикле, чтобы можно было возвести в нужную степень не одно число, а любое их количество. Нужно только договориться об условии окончания цикла. Например, пусть для этого надо будет ввести в качестве основания или показателя степени 0.

```
Program Stepen1;  
var A,N,I,P:Integer; {Основание и показатель степени,  
                      параметр цикла, ответ}  
Begin  
    Repeat  
        Write('Введите основание степени '); Readln(A);  
        Write('Введите показатель степени '); Readln(N);  
        P:=1;  
        For I:=1 to N do  
            P:=P*A;  
        Writeln('Ответ: ',A,'^',N,'=',P)  
    Until (A=0) or (N=0)  
End.
```

**Замечание.** Конечно, в этой программе при  $A=0$  производится куча ненужных действий: число ноль  $N$  раз умножается само на себя. Подумайте, как сделать программу более рациональной.

**Пример 9.22.** Известно, что для целых чисел выполняются следующие соотношения:

$$\begin{aligned}1 &= 1^2 \\ 1 + 3 &= 2^2 \\ 1 + 3 + 5 &= 3^2 \quad \text{и т. д.}\end{aligned}$$

*Доказательство этих равенств к языку Паскаль никакого отношения не имеет, но все же советуем попробовать его проделать. Мы же на Паскале напишем программу, которая будет печатать  $K$  таких равенств.*

Для решения нам понадобятся:

- число, равное номеру строки;
- количество слагаемых в левой части, которое равно номеру строки;
- слагаемые — последовательные нечетные числа.

Теперь осталось все это написать, используя два цикла **For**. Во внутреннем цикле не надо печатать последнее слагаемое, допечатаем его отдельно, так как после него, в отличие от остальных, идет знак «=», а не «+».

```
var K, I, J : Integer;
Begin
  Write('Число строк '); Read(K);
  For I:=1 to K do
    Begin For J:=1 to I-1 do
      Write (2*J-1, '+');
      Writeln(2*I-1, '=', I*I)
    End
  End.
```

## Задачи 9.66–9.70. Вложенные циклы

- 9.66. Про каждое число из заданного диапазона  $[N, K]$  напечатать, простое оно или составное.
- 9.67. Выписать все составные числа из заданного диапазона  $[N, K]$ .
- 9.68. Натуральное число называется совершенным, если оно равно сумме всех своих делителей, кроме самого этого числа (например, совершенным является число  $6 = 1 + 2 + 3$ ). Напишите программу получения всех совершенных чисел из диапазона от 1 до  $N$ .
- 9.69. Из заданного диапазона  $[M, N]$  выписать все числа, у которых ровно 2 делителя (отличных от 1 и самого числа).

9.70. Исходя из уже известных нам соотношений (см. пример 9.22):

$$1 = 1^2;$$

$$1 + 3 = 2^2;$$

$$1 + 3 + 5 = 3^2 \text{ и т. д.}$$

можно сделать вывод, как определить целую часть квадратного корня некоторого целого числа. Для этого надо вычитать из числа все нечетные числа по порядку (начиная с 1), пока остаток не станет меньше следующего вычитаемого числа или равен нулю. Количество выполненных действий и будет искомой целой частью корня. Например:  $9 - 1 = 8$ ;  $8 - 3 = 5$ ;  $5 - 5 = 0$ . Выполнено 3 действия, квадратный корень числа 9 равен 3.

Написать программу, реализующую данный метод.

## Решение задач методом перебора

Циклы, особенно вложенные, часто используются для решения задач методом перебора. Это такой метод, при котором перебираются в какой-либо последовательности все возможные значения (виды, состояния) исследуемого объекта и выбираются те, которые отвечают условиям задачи.

***Пример 9.23.** Определить количество трехзначных натуральных чисел, сумма цифр которых равна K. Напечатать эти числа.*

Можно решать эту задачу, рассматривая все трехзначные числа от 100 до 999 и вычисляя сумму цифр каждого числа (мы это уже делали с помощью операции **div** и **mod**). Это и есть метод перебора: все возможные числа проверяются на соответствие условию.

Приведем здесь решение задачи другим способом. Методом перебора будем составлять всевозможные числа из трех цифр. Это даст нам возможность не вычислять каждый раз цифры числа, мы их сразу будем иметь в «готовом» виде. Остается только найти их сумму и, если она «хорошая», выписать число. В приведенной ниже программе мы немного обманываем пользователя: печатаем не число, а составляющие его цифры. На экране ответ выглядит так же, как если бы это было число. Впрочем, обман невелик, и, если нужно, получить число из составляющих его цифр особого труда не составит.

Итак, организуем перебор по трем составляющим число цифрам: первая может меняться от 1 до 9 (если она = 0, число уже не является трехзначным), а вторая и третья — от 0 до 9.

```
Program Perebor;  
var K, L, C1, C2, C3 : Integer;  
Begin  
  Write('Сумма '); Read(K);  
  L:=0;
```

```
For C1:=1 to 9 do
  For C2:=0 to 9 do
    For C3:=0 to 9 do
      If C1+C2+C3=K Then Begin
        Writeln (C1,C2,C3);
        L:=L+1
      End;
    Writeln('Всего чисел  ', L)
  End.
```

Какой из способов решения лучше? В первом случае операторы во внутреннем цикле выполняются 900 раз, во втором получаем  $10 * 10 * 9 = 900$ . Тоже 900! Но во втором случае в теле цикла меньше операторов (не надо вычислять цифры).

Можно сделать программу более эффективной, предусмотрев в некоторых случаях досрочный выход из цикла. Например, если в каком-то цикле параметр — одна из цифр — больше требуемой суммы, из такого цикла можно выходить. *Сделайте это самостоятельно.*

**Пример 9.24.** В некотором государстве валютой являются футики. Показать, что любую целочисленную денежную сумму больше 7 футиков можно выплатить только купюрами по 3 и 5 футиков. Написать программу, на вход которой подается некоторое количество  $K$  футиков, а она печатает всевозможные способы представления этой суммы в виде наборов 3-футиковых и 5-футиковых монет. То есть для любого  $K > 7$  подбирает  $I$  и  $J$ , такие что  $K = 3 * I + 5 * J$ .

Самое малое количество монет некоторого достоинства, которое можно взять, 0, самое большое количество 3-футиковых монет, которое может понадобиться для выдачи  $K$  футиков, равно  $K \text{ div } 3$ , а 5-футиковых —  $K \text{ div } 5$ . Вот и займемся простейшим перебором: будем брать разное количество монет, от минимального до максимального, и смотреть, не получается ли нужная сумма.

```
Program Money;
var K,I,J : Integer;
Begin
  Write('Количество футиков '); Read(K);
  For I:=0 to K div 3 do
    For J:=0 to K div 5 do
      If 3*I+5*J=K Then Writeln ('3*',I,'+5*',J,'=',K);
    End.
```

Часто бывает, что в таких задачах надо найти не все возможные решения, а лишь какое-то одно. Тогда, как только решение обнаружено, цикл надо прервать: методы досрочного прекращения циклов мы обсуждали выше.

## Задачи 9.71–9.74. Метод перебора

- 9.71. Дано натуральное число. Представить его в виде произведения двух сомножителей всеми возможными способами.
- 9.72. Вывести на экран в возрастающем порядке все трехзначные числа, в десятичной записи которых нет одинаковых цифр. Задачу решить двумя способами (с использованием операций деления и без них).
- 9.73. Можно ли данное число  $N$  представить в виде суммы квадратов двух натуральных чисел?
- 9.74. Вывести все числа  $n$ , меньшие  $N$ , которые можно представить в виде суммы квадратов трех натуральных чисел:  $n = a^2 + b^2 + c^2$ , причем  $a \leq b \leq c$ .

## Работа с таблицами

С таблицами часто работают с помощью вложенных циклов. При этом (так как таблица печатается по строкам) внутренний цикл работает с элементами строки, а это значит, что он должен проработать столько раз, сколько в таблице столбцов. Внешний цикл отвечает за работу со строками в целом. В нем количество повторений зависит от количества строк.

Для печати таблицы в программе обязательно должен присутствовать оператор перевода строки **Writeln**, причем он должен стоять во внешнем цикле, его надо выполнить столько раз, сколько в таблице строчек.

**Пример 9.25.** Напечатать «прямоугольник» из звездочек размером  $M \times N$  ( $M$  строк,  $N$  столбцов).

Для этого нам нужно «нарисовать» несколько строчек из звездочек. А как заставить компьютер сделать строчку длиной  $K$ ? Это цикл:  $K$  раз напечатать символ «\*», после этого надо перейти на следующую строку. Эти два действия надо повторить столько раз, сколько подобных строчек мы хотим изобразить. По предлагаемой блок-схеме (рис. 9.9) получится прямоугольник шириной  $K$  и высотой  $N$ .

Аккуратно запишем алгоритм на Паскале: во внутреннем цикле всего один оператор (печать звездочки), во внешнем — два оператора (только что описанный оператор цикла и оператор перехода на следующую строку), эти два оператора надо заключить в операторные скобки, чтобы они оба выполнялись в цикле.

```
For I:=1 to N do
Begin For J:=1 to K do
    Write('*');
Writeln
End
```

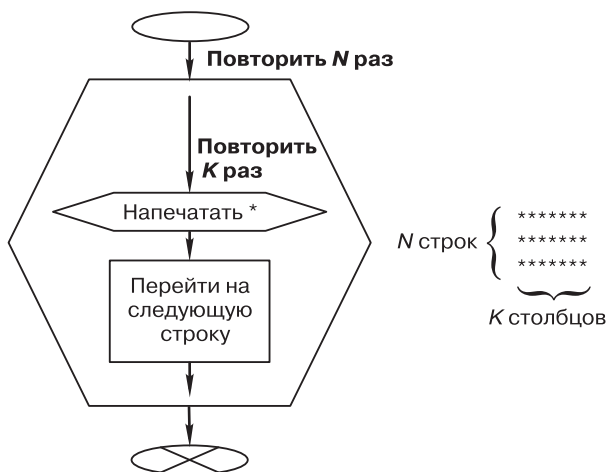


Рис. 9.9

**Пример 9.26.** Что будет напечатано в результате выполнения следующего фрагмента программы?

```

I:=1;
For M:=1 to N do
  Begin For J:=1 to K do
    Write(I, ' ');
    Writeln;
    I:=I+1
  End
End
  
```

Такая программа (блок-схему см. на рис. 9.10) напечатает числа: в первой строке —  $K$  единиц, во второй —  $K$  двоек, и так  $N$  строк.

По этой блок-схеме и соответствующей ей программе очень хорошо можно наблюдать, насколько важны в программировании мельчайшие детали.

Например, если блок  $I:=I+1$  переставить внутрь вложенного цикла (сразу после печати значения  $I$ ), результат будет совсем другой. Внутренний цикл напечатает  $K$  чисел: в 1-й строке от 1 до  $K$ , во 2-й — следующие  $K$  чисел от  $K+1$  до  $2 \cdot K$  и т. д. Таким образом, напечатаются числа от 1 до  $K \cdot N$  подряд по  $K$  в каждой строчке.

```

I:=1;
For M:=1 to N do
  Begin For J:=1 to K do
    Begin Write(I, ' ');
          I:=I+1
    End;
    Writeln;
  End
End
  
```



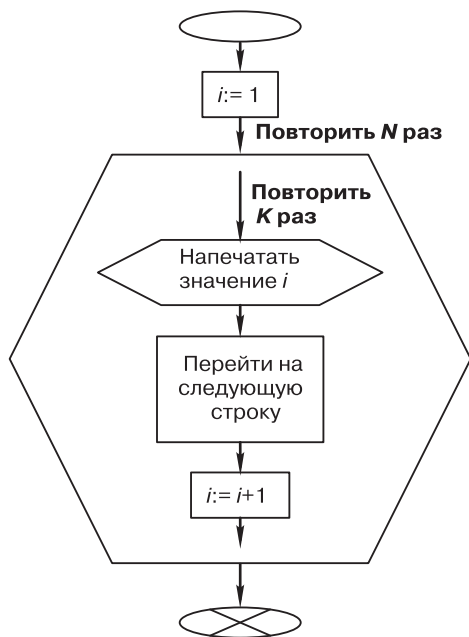


Рис. 9.10

А если еще и **I:=1** переставить внутрь большого цикла (перед «повторить K раз»), напечатается *N* одинаковых строчек с числами от 1 до *K*.

```

For M:=1 to N do
Begin I:=1;
  For J:=1 to K do
    Begin Write(I, ' ');
      I:=I+1
    End;
  Writeln;
End
  
```

Заметим, что первоначально предложенную программу можно было записать короче, без переменной *I* (можно заметить, что переменные *I* и *M* меняются одновременно, равны друг другу):

```

For M:=1 to N do
Begin For J:=1 to K do
  Write (M:4);
Writeln
End
  
```

Поэкспериментируем еще с этой программой. В операторе **Write (M: 4)** заменим переменную *M* на *J*. Какая таблица получится? В ней теперь не строки, а столбцы состоят из одинаковых чисел.

***Задание.** Выполните все предложенные здесь программы на компьютере, посмотрите, что они выводят. Модернизируйте их (например, запишите цикл «наоборот», от большего значения к меньшему), попробуйте догадаться, что будет выводить на экран такая программа. Проверьте себя с помощью компьютера.*

**Задачи 9.75–9.81. Работа с таблицами**

- 9.75. Напечатать таблицу — все символы с их кодами. В такой таблице 255 строк, она на экране не поместится, поэтому организовать вывод таблицы так, чтобы печаталось 20 строк таблицы, после чего вывод бы приостанавливался, пока человек не введет с клавиатуры некоторый символ для продолжения вывода или какой-то другой — для окончания работы.
- 9.76. Напечатать таблицу умножения (в том виде, в котором она изображается на обложке тетради):
- а) столбики располагаются последовательно, один под другим, печатаются на экране по очереди (организовать вывод по 20 строк с паузой — см. предыдущую задачу);
  - б\*) на экране располагается 2, 4 или 6 столбиков рядом друг с другом.
- 9.77. С помощью символьного вывода напечатать следующие «картинки» — рис. 9.11 (размеры — количество символов по горизонтали и по вертикали — вводятся с клавиатуры):

а) Рамка: ***** *           * *           * *****	б) Буква «Л»: * * * * * *       * *       * *       *	в) Буква «Х»: *       * *       * * *       * *       *	г) Знак «плюс»: * * ***** * *	д) Ромб: * *** ***** *** *
---	---	--	--	---

Рис. 9.11

9.78. «Нарисовать» (рис. 9.12) равнобедренные прямоугольные треугольники (размер катета вводится):

а)  ***** *** ** *	б)  * ** *** ****	в)  * ** *** ****	г)  ***** *** ** *
-----------------------------------	----------------------------------	----------------------------------	-----------------------------------

Рис. 9.12

9.79. Вывести в виде таблицы  $M * N$  все целые числа от 1 до  $M * N$  следующим образом:

а) По строкам поряд: 1 2 3 4 5 6 7 8 9 10 11 12	б) По столбцам поряд: 1 4 7 10 2 5 8 11 3 6 9 12	в) «Ходом быка» с начала: нечетные столбцы сверху вниз, четные — наоборот: 1 6 7 12 2 5 8 11 3 4 9 10	г) «Ходом быка» с конца: 10 9 4 3 11 8 5 2 12 7 6 1
---	--	---	---

9.80. Вывести на экран таблицу следующего вида (размеры задаются с клавиатуры):

а)  A B C D E F G H I J	б)  1 2 3 4 2 3 4 5 3 4 5 6	в)  ! ** ! ** ! ** ! ** ! ** ! ** ! ** ! ** ! ** ! ** ! ** ! **	г)  A A A A B B B C C D
д) 1 3 5 7 2 4 6 8 1 3 5 7 2 4 6 8	е) abcdefghij aabbccdde aaabbbcccd aaaabbbbcc	ж) 1 2 3 1 2 3 2 3 1 2 3 1 3 1 2 3 1 2 1 2 3 1 2 3	з*) 0 0 0 0 0 0 1 1 1 0 0 1 2 1 0 0 1 1 1 0 0 0 0 0 0

9.81. Дана логическая функция  $F(A, B, C) = A \text{ or } \text{not}(B \text{ and } C)$ . Напечатать таблицу ее значений в зависимости от значений переменных.

## Задачи 9.82–9.115. Оператор цикла.

### Разные задачи

- 9.82. Получить новое число путем добавления заданной цифры слева к десятичной записи заданного числа (например, из числа 123 и цифры 4 должно получиться число 4123).
- 9.83. Составить новое число из четных цифр заданного числа (если в числе нет четных цифр — ответ 0).
- 9.84. Получить новое число, вставив в десятичную запись заданного числа нули между всеми цифрами (например, из числа 123 надо получить 10203, из 40 — 400).
- 9.85. Найти число, полученное выписыванием в обратном порядке цифр заданного числа (например, из числа 123 надо получить 321).
- 9.86. Вывести число, которое получится путем удаления всех вхождений заданной цифры из десятичной записи числа (например, из числа 1232 и цифры 2 получится 13).
- 9.87. Удалить из десятичной записи числа цифру, стоящую на заданной позиции, если таковая цифра есть (например, если из числа 1234 удалить вторую цифру справа, получится 124).
- 9.88. Поменять местами наибольшую и наименьшую цифры в числе: а) считать, наибольшая и наименьшая цифра встречаются только по одному разу; б)\* если цифры могут встретиться в числе не один раз, выбрать такие пары цифр, чтобы число получилось максимальным.
- 9.89. Выписать разложение числа на простые множители.
- 9.90. Среди всех чисел из диапазона  $[M, N]$  найти число с наибольшим количеством делителей (если таких чисел несколько, найти любое из них).
- 9.91. Дано натуральное число  $N$ . Среди чисел от 1 до  $N$  найти все такие, запись которых совпадает с последними цифрами записи их квадрата (например,  $6^2 = 36$ ,  $25^2 = 625$ ).
- 9.92\*. Натуральное число из  $N$  цифр является числом Армстронга, если сумма его цифр, возведенных в  $n$ -ю степень, равна самому числу (например  $153 = 1^3 + 5^3 + 3^3$ ). Получить все числа Армстронга, состоящие из двух, трех и четырех цифр.

### Последовательности

- 9.93. Известно, что последовательность неубывающая. Определить количество различных элементов.
- 9.94. Известно, что последовательность неубывающая. Сколько раз встречается каждый элемент?

- 9.95\*. Вводится последовательность из  $N$  символов. Найти величину самой длинной последовательности из подряд стоящих одинаковых символов, а также напечатать, какой это символ (если таких последовательностей несколько, напечатать информацию о любой из них).
- 9.96\*. С клавиатуры вводится последовательность символов, заканчивающаяся точкой (в строчку). Напечатать ее еще раз так, чтобы подряд стоящие одинаковые буквы оказались на одной строке, а в случае, когда подряд стоят разные символы, строка бы переводилась.
- 9.97\*. Дана последовательность вещественных чисел. Выявить отрезки неубывания этой последовательности и вывести каждый из них на экран с новой строки.
- 9.98\*. Дана последовательность целых чисел. Определить количество элементов в самой длинной убывающей подпоследовательности и вывести первый и последний элемент такой подпоследовательности. Если подпоследовательностей окажется несколько, работать с первой из них.

### **Работа с последовательностью символов (словом, строкой, предложением, текстом)**

- 9.99. Вывести все слова из заданного текста в столбик. Считаем, что слова в тексте разделены пробелами: а) ровно одним пробелом; б) любым количеством пробелов ( $> 0$ ).
- 9.100. Вывести заданный текст по строкам, понимая под строкой либо очередные 10 символов, если в них нет запятой, либо часть текста до запятой включительно.
- 9.101. С клавиатуры вводятся строки, в каждой из которых записаны предложения (заканчиваются точкой). В каждом предложении не менее двух слов, слова разделяются ровно одним пробелом. Количество строк известно. Напечатать количество букв во втором слове в каждом предложении.
- 9.102. С клавиатуры вводится (в одну строку) предложение. В конце предложения может стоять точка, восклицательный или вопросительный знак. Слова разделяются некоторым количеством пробелов. Подсчитать количество слов в предложении.
- 9.103. С клавиатуры вводится (в одну строку) предложение. В конце предложения может стоять точка, восклицательный или вопросительный знак. Слова разделяются некоторым количеством пробелов. Напечатать то же самое предложение, но так, чтобы слова были разделены ровно одним пробелом.
- 9.104. С клавиатуры вводится предложение (в одну строку). Слова в нем могут разделяться пробелами (в любом количестве). В конце пред-

ложения стоит точка. Верно ли, что в предложении выполняется правило: у каждого слова (кроме первого) первая буква равна последней букве предыдущего слова?

- 9.105. С клавиатуры вводится предложение (в одну строку), заканчивающееся точкой. Слова разделены ровно одним пробелом и состоят из латинских букв. Напечатать слова, начинающиеся с большой буквы.
- 9.106. С клавиатуры вводится предложение (в одну строку), заканчивающееся точкой. Слова, если их несколько, разделены ровно одним пробелом. Есть ли в предложении слова, которые начинаются и заканчиваются на одну и ту же букву?
- 9.107\*. Строка (вводится посимвольно) состоит из чисел и знаков «+» и «-», заканчивается знаком «=». Вычислить значение соответствующего выражения. Пример:  $12 + 14 - 5 = 21$ .

### Системы счисления

- 9.108. Число в двоичной системе счисления вводится посимвольно. Перевести его в десятичную систему.
- 9.109. Перевести число из десятичной системы счисления в двоичную.
- 9.110. Перевести число из десятичной системы счисления в 16-ричную.
- 9.111\*. Перевести число из десятичной системы счисления в  $Q$ -ичную. ( $Q$  вводится с клавиатуры,  $1 < Q < 17$ ).
- 9.112\*. Число в 16-ричной системе счисления вводится посимвольно (используются цифры и большие латинские буквы, оканчивается точкой). Перевести его в двоичную систему.
- 9.113\*. Число в 16-ричной системе счисления вводится посимвольно (используются цифры и большие латинские буквы, оканчивается точкой). Перевести его в десятичную систему.
- 9.114\*. Вводится  $Q$  — основание системы счисления ( $1 < Q < 17$ ) и число в этой системе. Перевести число в десятичную систему счисления.
- 9.115\*. Пусть число вводится в виде строки (последовательности символов) следующим образом:
- первый символ: буква  $b$  или  $d$  — обозначение системы счисления (двоичная или десятичная);
  - $n$  ( $n > 0$ ) следующих символов — цифры в соответствующей системе счисления — целая часть числа;
  - точка;
  - $m$  ( $m > 0$ ) символов — дробная часть числа;
  - ввод заканчивается символом #.

Проверить правильность ввода. Если ошибок нет, вывести введенное число (в десятичном виде).

# Глава 10

## Массив

---

### Задание типов

Мы знаем в Паскале четыре стандартных типа данных **Integer**, **Real**, **Char** и **Boolean**. Оказывается, программист может создавать собственные типы данных. Конечно, он не может их делать какими угодно, ведь транслятор должен понимать, как с ними работать. Новые типы конструируются из уже имеющихся, понятных транслятору.

В каком месте программы задать новый тип (это действие называется «задание типа»)? Можно прямо в разделе описания переменных (**Var**) после имени переменной через двоеточие написать не один из четырех основных типов, а новый, сконструированный программистом.

Однако лучше, когда описание новых типов размещено в специальном разделе — разделе описания типов. Этот раздел помещается в разделе описаний (в начале программы), обычно он идет самым первым или после описания меток и констант. Ведь мы хотим задаваемым описанием типа пользоваться при описании переменных, значит, его надо поместить раньше раздела **Var**.

Выглядит описание типа так:

```
Type    <имя типа> = <задание типа>
```

Здесь слово **Type** — служебное слово языка Паскаль, название раздела описания типов данных. Имя типа придумывает программист (оно должно соответствовать общим для всех имен Паскаля правилам).

В одном разделе можно описывать несколько новых типов, описания отделяются друг от друга знаком «;», слово **Type** можно не повторять. Раздел описания типов от других разделов тоже отделяется знаком «;».

Один из самых простых способов задания типа — использование уже известного типа данных с новым, придуманным пользователем названием, т. е. новый тип данных является синонимом некоторого уже известного. Например:

```
Type Rost = Real;  
     Ves = Real;
```

На первый взгляд такое действие кажется совершенно ненужным: тип данных-то все равно один, однако программа становится понятнее. При вот таком описании переменных

```
Var R, P: Rost;  
    M, V: Ves;
```

уже без дополнительных комментариев понятно, что они означают; понятно, что в правильной программе не может быть выражения **R+V** (нельзя сантиметры складывать с килограммами). Некоторые трансляторы умеют «ловить» такую ошибку, ведь это «несоответствие типов». К сожалению, используемый многими транслятор Турбо Паскаля к их числу не относится.

Еще один способ задания типа — его конструирование. Конечно, создавать новые типы можно, только руководствуясь правилами языка, через уже имеющиеся в языке базовые типы.

Часто в программах используются типы, называемые ограниченными (другое название — тип-диапазон). Базовым для любого такого типа является уже имеющийся ординальный тип (т. е. тип **Real** не может фигурировать в качестве базового типа). Новый тип задается путем наложения ограничений на базовый. Из всего множества значений базового типа (всех целых чисел, всех символов) в качестве значений типа выбирается только некоторый отрезок, диапазон. Границы задаются с помощью двух констант базового типа. Первая константа задает минимальное, а вторая — максимальное значение допустимого диапазона. При этом первая константа должна быть не больше второй. Константы разделяются двумя точками.

Примеры:

```
Type      Month = 1..12;  
          Temperature = -273..1000;  
          Letter = 'a'..'z';
```

Базовым для типов **Month** (месяц) и **Temperature** (температура) является **Integer**, но переменные типа **Month** могут принимать значения только от 1 до 12 (включительно), а типа **Temperature** — от -273 до 1000. Для типа **Letter** базовым является **Char**, но его значениями могут быть не все символы, а только маленькие латинские буквы.

Над переменными ограниченного типа можно выполнять все те же действия, что и над переменными соответствующего базового типа. А вот значения они могут принимать только те, которые входят в описанный диапазон, в случае выхода за границы диапазона транслятор должен указывать ошибку (транслятор Турбо Паскаля с этой задачей успешно справляется).

Таким образом, программа не будет работать с заведомо неправильными данными (в нашем случае, например, с месяцем, номер которого равен 13, или с буквой «%»), а программисту будет подсказано место, где проявилась ошибка.



Ограниченный тип данных может существенно помочь программисту, однако его возможностей по написанию различных программ не увеличивает. А вот теперь мы познакомимся с массивом — типом данных, использование которого позволяет решать новый класс задач.

## Тип данных «Массив»

Мы уже решали задачи, в которых было большое количество входных данных, однако мы никогда все эти данные не держали в памяти целиком. Для решения задачи достаточно было помнить 1–2 элемента. Тем не менее существует достаточно большой класс задач, для решения которых надо держать в памяти все элементы последовательности. Одним из «хрестоматийных» примеров задач такого типа является следующий.

***Пример 10.1.** Имеются сведения о количестве осадков за каждый месяц в течение года. Найти среднее количество осадков и напечатать таблицу, в которой для каждого месяца будет выписано количество осадков в нем и отклонение этого количества от среднего.*

Для решения этой задачи нам придется «пробежаться» по всем числам (количествам осадков в каждом месяце) два раза: первый раз при поиске среднего, а второй раз — при расчете отклонения. Конечно, мы не будем вводить числа два раза, придется их все хранить в памяти. Как же обозначить, как назвать наши 12 переменных? Может, так: *A*, *B*, *C*, *D* и т. д.? Но тогда, во-первых, не для всех задач нам «хватит» букв. Ведь легко себе представить задачу, в которой нужно не 12 переменных, а гораздо больше (например, имеются сведения об осадках не за 12 месяцев, а за целый век, за 100 лет). А во-вторых, с такими обозначениями будет неудобно работать. Понятно, что в задаче надо будет воспользоваться оператором цикла (мы уже подсчитывали среднее арифметическое), как же мы его сможем написать с разными переменными на каждом шаге?

Для разрешения такой ситуации в программировании используют массивы (в научной литературе часто применяется термин «регулярный тип»). Массив в Паскале — упорядоченный набор однотипных переменных, причем количество этих переменных фиксировано, определяется при задании массива. Все переменные, входящие в массив, имеют одно и то же имя — имя массива, а различаются они по индексу — номеру (месту) в массиве.

## Описание массива

Таким образом, при задании массива необходимо указать его имя (общее имя всех входящих в него переменных), тип (опять же, одинаковый для всех компонент) и тип индексов (номеров) переменных.

Описание массива выглядит так:

```
Array [<тип индекса>] of <тип компонент>
```

Здесь **Array** и **of** — служебные слова («массив» и «из»), **<тип индекса>** — описание индексации компонент (элементов) массива и **<тип компонент>** — тип переменных, составляющих массив (выше говорилось, что он у них у всех один).

Тип индекса определяет, сколько в массиве компонент, этот тип должен давать возможность однозначно выбирать их. Следовательно, он не может быть **Real** (тип **Real** не является ординальным) и **Integer** (множество значений не ограничено). Язык Паскаль не накладывает никаких ограничений на количество компонент и размер массива, но понятно, что при работе на реальном компьютере такие ограничения (зависящие от реализации) всегда имеются.

Чаще всего в качестве типа индекса используется ограниченный тип на базе целого, причем нумерация компонент обычно начинается с 1. В этом случае возможно, например, такое описание:

```
Type Day=Array[1..365] of Real; — 365 вещественных компонент перенумерованы от 1 до 365.
```

Иногда удобно, используя целый тип в качестве базового, производить нумерацию компонент не с 1, а с любого другого целого числа, например, с 0 или даже с какого-либо отрицательного числа. Например, если в массиве хранится количество оценок каждой «номинации» (двоек, троек, пятерок и четверок), массив из 4 целых компонент может быть описан так:

```
Type Ocenka=Array[2..5] of Integer;
```

А если компоненты массива — показания некоторого прибора при разных значениях температуры, массив может быть таким:

```
Var RRR : Array[-273..2000] of Real;
```

Другие ординальные типы (**Char** и **Boolean**) тоже могут выступать в качестве типов индексов: как целиком, так и в качестве базовых для некоторого ограниченного типа. Например:

```
Var Pin : Array[Boolean] of Integer; — массив из двух целых чисел со значениями индексов False и True.
```

```
Var Class : Array['a'..'z'] of Real; — массив из 26 вещественных чисел со значениями индексов, равных маленьким буквам латинского алфавита.
```

Обратите внимание, в описании индексов массивов присутствуют только константы (переменные здесь использоваться не могут), так как еще до начала работы программы, на этапе описания переменных, должно быть известно количество компонент массива.

## Обращение к элементу массива

Для того чтобы работать с отдельным элементом массива, надо указать имя массива и индекс элемента в массиве. Например, если в программе описаны последние три массива, возможны такие переменные: **RRR[-273]**; **RRR[0]**; **RRR[1000]**; **Pin[True]**; **Class['a']**; **Class['v']**. При этом индекс не обязательно должен быть константой, в квадратных скобках может стоять и выражение. Важно, чтобы оно было указанного в описании типа и его значение не выходило за границы описанного отрезка: **RRR[2\*T-1]**.

Неправильный тип выражения в индексе (например, **Integer** вместо **Char**) «поймает» любой транслятор, а вот ошибка «выход за границы массива» (когда тип индекса правильный, но значение индексного выражения не попадает в указанный отрезок) автоматически замечается далеко не всегда. В частности, транслятор Турбо Паскаля нуждается в специальной настройке (в опциях компилятора должен быть включен флажок **Range Checking**), только в этом случае Паскаль-система обратит ваше внимание на такую ошибку. Советуем включать этот флажок при работе на компьютере и быть очень внимательным при написании программ. Работа с переменной, индекс которой не входит в обозначенный в описании диапазон, недопустима, это серьезная ошибка, даже если программа выдает какие-то результаты (правильность этих результатов сомнительна и нестабильна).

Элемент массива, к которому мы обращаемся, является обычной переменной. Естественно, ее тип — это тот тип, который задан в описании массива. С этим элементом допустимо производить все операции, которые возможны над данными этого типа: присваивание, сравнение, ввод, вывод и т. п. Например:

```
If RRR[-273]<RRR[J] Then Class['b']:=1.23
```

А какие операции допустимы в Паскале над переменными типа «массив», т. е. над переменными, описанными как массив, если их использовать без индекса? Увы, над массивами недопустимы никакие арифметические операции (оно и понятно — как сложить два массива: суммировать пары компонент или просто объединить все компоненты?), массивы «целиком» нельзя вводить и выводить с помощью процедур **Read** и **Write**, это приходится делать только поэлементно. Над массивами в целом определена единственная операция — присваивание, да и она допускается только в том случае, если массивы одного типа (имеют одно имя типа или описаны в одной строчке, через запятую). Так, при описании:

```
Var M1,M2 : Array[-5..5] of char
```

возможно присваивание **M1 := M2**.

Вернемся теперь к нашей задаче. Для ее решения нам понадобится массив из 12 (по числу месяцев) вещественных чисел. Мы уже говорили, что числа придется просматривать как минимум два раза, т. е. понадобится два оператора цикла.

В первом цикле мы введем элементы массива и здесь же посчитаем их сумму — годовое количество осадков. Для работы с элементами массива воспользуемся переменной *Osad[J]*. В цикле значение индекса *J* будет меняться — сначала мы будем работать с переменной *Osad[1]*, потом — *Osad[2]* и так далее, последней будет переменная *Osad[12]*.

Во втором цикле мы будем вычислять значения и печатать строки искомой таблицы.

```
Program Osadki;
Type Massiv=Array[1..12] of Real;
Var Osad : Massiv; {количество осадков в каждом месяце}
    Sred : Real;
    J    : Integer; {счетчик}
Begin
  Sred:=0;
  Writeln('Введите количество осадков в каждом месяце');
  For J:=1 to 12 do
  Begin Write(J, ' - '); Readln(Osad[J]); {Вводим значения}
    Sred:=Sred+Osad[J] {Ищем сумму}
  End;
  Sred:=Sred/12; {среднегодовое количество осадков}
  Writeln('Среднее количество осадков =', Sred:8:2);
  Writeln('N кол-во осадков отклонение'); {Заголовок
                                           таблицы}
  For J:=1 to 12 do
    Writeln(J:3, Osad[J]:10:2, Osad[J]-Sred:16:2);
End.
```

В этой задаче мы заполняли массив числами, которые вводили с клавиатуры.

**Пример 10.2.** *С клавиатуры вводится последовательность из  $N$  целых чисел. Напечатать сначала отрицательные числа, а затем положительные.*

Для решения этой задачи понадобится целых 3 оператора цикла! Сначала мы введем все компоненты массива, за второй просмотр выпишем отрицательные числа, а за третий — положительные.

```
Const N=10;
Var MInt : Array [1..N] of Integer;
    J    : Integer; {счетчик}
Begin Writeln('Введите ', N, ' целых чисел');
  For J:=1 to N do
```

```

Begin Write(J:2, ' - ');
      Readln(MInt[J])
End;
Writeln('Отрицательные числа');
For J:=1 to N do
  If MInt[J]<0 Then Write(MInt[J]:5);
Writeln;
Writeln('Положительные числа');
For J:=1 to N do
  If MInt[J]>0 Then Write(MInt[J]:5);
Writeln;
End.

```

Обратите внимание, как в этой задаче описан массив: сначала описана константа  $N$ , а затем массив из  $N$  элементов. Далее константа  $N$  используется во всех операторах цикла. Использование константы удобно тем, что при работе с такой программой можно очень быстро поменять длину массива — надо всего-навсего заменить число в верхней строчке, при этом для массива другой длины программа тоже будет работать правильно, ведь количество элементов массива в виде числа нигде не используется, только в виде константы  $N$ .

Как можно видеть из предыдущих задач, с массивами очень удобно работать, используя оператор цикла **For**, который позволяет просматривать все элементы массива подряд. Однако это не означает, что при работе с массивами используют только этот цикл. В следующей задаче удобны циклы с условием (**While** или **Repeat-Until**).

***Пример 10.3.** Задан массив вещественных чисел. Напечатать сначала числа, стоящие на нечетных местах, а затем числа, стоящие на четных местах.*

В этой задаче также придется использовать три цикла: для ввода и для вывода каждой последовательности. Задачи очень похожи, запишем для этой только вывод последовательностей:

```

Writeln('Числа, стоящие на нечетных местах');
I:=1;
While I<=N do
Begin Write(Mas[i]:5);
      I:=I+2
End;
Writeln;
Writeln('Числа, стоящие на четных местах');
I:=2;
While I<=N do
Begin Write(Mas[I]:5);
      I:=I+2
End;
Writeln;

```

**Пример 10.4, а.** «Перевернуть» массив, т. е. напечатать его компоненты в обратном порядке.

Для описания алгоритма решения этой задачи тип элементов массива значения не имеет, поэтому решим задачу для массива символов: будем переворачивать слова — так будет интереснее смотреть на результаты.

Но сразу возникает такая проблема: слова бывают разной длины. Понятно, что мы должны перед вводом слова спросить пользователя, сколько в нем букв, или же как-то пометить конец слова, например поставить в конце точку. Таким образом, работать нам надо будет именно с тем количеством букв, которое будет введено, причем оно может быть каждый раз разным.

Но ведь описать массив придется в разделе описаний, до ввода слова! Придется сделать это «с запасом». Опишем массив из 20 символов (более длинные слова рассматривать не будем). Таким образом, когда мы введем слово из  $N$  букв, в массиве в первых  $N$  позициях будут буквы нашего слова, а в оставшихся  $20 - N$  позициях — «мусор», там будут находиться какие-то случайные значения.

Такой прием — описание массива большого размера и работа с его частью — применяется достаточно часто, и программисту нужно быть внимательным, чтобы использовать в программе только те переменные, которые были в ней определены, не залезть в «мусор». Дело в том, что такую ошибку автоматически отследить не может даже Паскаль-система со специально включенными опциями: ведь никакого выхода за границы массива здесь нет, есть выход за используемую часть массива, что заметить может уже только человек.

```
Var Slovo : Array [1..20] of Char;  
  I,N : Integer; {счетчик и количество букв}  
Begin Writeln('Введите слово с точкой в конце');  
  I:=0;  
  Repeat  
    I:=I+1;  
    Read(Slovo[I]);  
  Until Slovo[I]='.';  
  N:=I-1; {Букв в слове на 1 меньше, ведь была введена  
точка}  
  Writeln;  
  Writeln('Перевернутое слово');  
  For I:=N downto 1 do Write(Slovo[I]);  
  Writeln  
End.
```

Заметим, что работать со словами (массивами символов) можно несколько по-другому, этим мы займемся чуть позже.

**Пример 10.4, б.** *Задан некоторый массив. Переписать его компоненты в другой массив так, чтобы они там оказались в обратном порядке.*

Пусть заданный массив называется *Slovo*, а создаваемый — *P*. Они одного типа, их можно описать вместе (через запятую). Первой компоненте массива *P* надо присвоить значение *N*-й компоненты массива *Slovo*, 2-й — значение  $(N - 1)$ -й компоненты, 3-й —  $(N - 2)$ -й. Таким образом получаем, что *J*-й компоненте массива *P* надо присвоить значение  $(N + 1 - J)$ -й компоненты массива *Slovo*. Оператор цикла будет выглядеть так:

```
For J:=1 to N do P[J]:=Slovo[N+1-J]
```

**Пример 10.5.** *Проверить, является ли заданное слово палиндромом. (Слово так называется, если оно одинаково читается слева направо и справа налево.)*

Будем брать буквы, стоящие на равном удалении от начала и конца слова, и проверять, совпадают ли они. Цикл достаточно сделать до середины слова, а соответствие индексов для равноудаленных от концов слова (симметричных) букв мы уже исследовали в предыдущей задаче.

Для проверки удобно ввести логическую переменную. Назовем ее *Palindr*. Значением ее будет **True**, если слово палиндромом является, и **False** в противном случае. Решение задачи может быть следующим.

```
Palindr:=True;
For I:=1 to N div 2 do
  If Slovo[I]<>Slovo[N+1-I] Then Begin Palindr:=False;
                                   Break
                                End;
Writeln (Palindr)
```

Обнаружив, что слово не является палиндромом, мы прекращаем проверку (выходим из цикла **For** с помощью оператора **Break**). Эти же действия можно было реализовать с помощью цикла **Repeat**, реализовав выход из него при значении **Palindr=False** и при прохождении всего массива. *Сделайте это самостоятельно.*

**Пример 10.6.** *Заполнить массив последовательными числами Фибоначчи.*

Мы уже решали подобную задачу (см. пример 9.12). Там надо было не заполнить массив числами, а просто вывести их на экран. В данном случае задача решается даже легче: нам не надо переопределять переменные (предыдущие числа Фибоначчи), они у нас хранятся в массиве.

```
Program Fibonacci;
Const N=10;
var I:Integer;
    Fib :Array[1..N] of Integer;
```

```
Begin
    Fib[1]:=1; Fib[2]:=1; {F0 и F1}
    For I:=3 to N do
        Fib[i]:=Fib[i-1]+Fib[i-2];
    For I:=1 to N do Write(Fib[i]:5);
    Writeln
End.
```

**Пример 10.7.** *Описан массив из  $N$  символов. С клавиатуры вводится слово (его конец отмечается звездочкой). Занести буквы слова в массив. Если количество букв больше, чем длина массива, занести в массив только первые  $N$  букв слова. Если количество букв меньше, чем длина массива, заполнить оставшуюся часть массива символами «звездочка».*

Буквы слова считываем в компоненты массива. Это нельзя делать с помощью цикла **For**, так как мы не знаем, сколько букв в слове (знаем, что оно заканчивается звездочкой). Воспользуемся циклом **Repeat** (не забудьте, что в этом цикле значение переменной, которая используется для задания индекса в массиве, автоматически не увеличивается, надо писать соответствующий оператор).

Когда должен закончиться такой цикл?

Во-первых, его надо прекратить, если закончится слово (введен символ «звездочка»), — это случай, когда длина слова меньше длины массива. Заметим, что звездочка попадет в массив, но в данном случае это не страшно — по условию задачи оставшиеся места в массиве и надо заполнить звездочками.

Во-вторых, цикл надо прекратить, если закончатся места в массиве, — это случай, когда длина слова больше или равна длине массива.

Для прекращения цикла достаточно, чтобы выполнилось хотя бы одно из условий, значит, в логическом выражении должна быть связка **Or**.

Если имеет место 2-й случай, буквы слова, которые в массив не помещаются, можно не считывать, это в задаче не требуется.

Если в слове букв меньше, чем мест в массиве, надо оставшиеся места заполнить звездочками. Это легко сделать в цикле **For**, только организовать его надо таким образом, чтобы массив заполнялся не с первой позиции, а с той, которая оказалась незанятой буквами слова.

Конечно, в конце надо напечатать получившийся массив — весь, с начала до конца, проверить, правильно ли он заполняется при вводе слов разной длины.

Заметим, что здесь можно по-разному манипулировать с переменными, которыми обозначается место в массиве.

В начале программы можно этой переменной присвоить 0, а потом увеличивать ее на 1 перед тем, как записать в массив очередную букву, можно



в начале присвоить переменной *I*, а увеличивать ее после записи очередной буквы. В зависимости от выбранного варианта надо рассмотреть, какое значение будет у переменной по окончании цикла, что и с чем надо сравнивать, чтобы проверить, не закончился ли массив, не записана ли звездочка.

Обязательно протестируйте свою программу для случаев, когда в массиве букв столько же, сколько позиций в массиве; на одну меньше; на одну больше.

```
Const N=7;
Var A : Array [1..N] of Char;
    C : Char;
    I, J : Integer;
Begin
    I:=1;
    Write('Вводите слово, в конце поставьте звездочку  ');
    Repeat
        Read(A[I]);
        I:=I+1;
    Until (I=N+1) Or (A[i-1]='*');
    For J:=I to N do A[J]:='*';
    Writeln('Получился массив');
    For J:=1 to N do Write(A[j]:2);
    Writeln
End.
```

**Пример 10.8.** *Задан массив SIM из N символов. Переписать из него большие латинские буквы в массив BIG, а маленькие — в LIT. Распечатать полученные массивы.*

Эта задача очень похожа на задачу из примера 10.2 — эффект работы программы будет такой же, в обоих случаях напечатаются сначала элементы массива, обладающие одним свойством, потом — обладающие другим свойством. Однако решение будет разное, так как здесь от нас требуется не просто напечатать буквы, а сначала разместить их в разных массивах.

Понятно, что в этой задаче нам надо описать 3 массива (они указаны в условии). Количество элементов в массиве *SIM* задано. А как описывать массивы *BIG* и *LIT*, сколько элементов в них должно быть? Понятно, что элементов в них будет не больше, чем в первоначальном массиве (ведь в нем кроме латинских букв могут быть и другие символы). Однако придется нам описывать оба этих массива как массивы из *N* элементов. Ведь может возникнуть такая ситуация, что в первоначальном массиве будут только большие (или наоборот, только маленькие) латинские буквы.

Конечно, при таком описании у нас обязательно часть одного из массивов (или части обоих) не будет заполнена буквами из заданного массива.

Что же будет находиться в этих «хвостиках»? Значения этих переменных (элементов массивов с соответствующими индексами) будут не определены, значит, пользоваться этими значениями нельзя (например, нельзя печатать их на экране), и могут быть эти значения какими угодно, причем разными на разных компьютерах, при разных сеансах работы и т. п.

Как же нам распечатать получившиеся массивы, если нельзя печатать значения переменных, которые не определены? Надо запоминать, сколько в массиве «правильных» значений, т. е. тех значений, которые были переписаны из заданного массива, и печатать только их. Запоминать количество записанных в массив букв несложно, нам все равно придется это количество подсчитывать, ведь записывать в массив надо на определенное место, и номер этого места (индекс в массиве) не равен номеру буквы в первоначальном массиве — обратите на это внимание!

```
Const N=10;
Var SIM, LIT, BIG : Array [1..N] of Char;
    Ns, Nl, Nb, J : Integer;
Begin Writeln('Вводите заданный массив');
    For Ns:=1 to N do
        Begin Write(Ns:2, ' - ');
            Readln(SIM[Ns])
        End;
    Nb:=0; Nl:=0;
    For Ns:=1 to N do
        If (SIM[Ns]>='A') and (SIM[Ns]<='Z')
        Then Begin Nb:=Nb+1;
                BIG[Nb]:=SIM[Ns]
            End
        Else
            If (SIM[Ns]>='a') and (SIM[Ns]<='z')
            Then Begin Nl:=Nl+1;
                    LIT[Nl]:=SIM[Ns]
                End;
    Writeln('LIT');
    For J:=1 to Nl do
        Write(LIT[J]:3);
    Writeln;
    Writeln('BIG');
    For J:=1 to Nb do
        Write(BIG[J]:3);
    Writeln;
End.
```

**Пример 10.9\*.** *Определить, есть ли в массиве одинаковые элементы.*

Во-первых, введем логическую переменную *Odinak*, значением которой будет **True**, если одинаковые элементы есть, и **False** — в противном случае.

Так как вначале мы еще никаких одинаковых элементов в массиве не нашли, **Odinak:=False**. Надо сравнить первый элемент массива со всеми остальными, т. е. с 2, 3, ..., *N*. Если равных не нашлось, надо искать, нет ли в массиве элементов, равных второму. Второй элемент с первым уже сравнивать не надо, это уже сделано, и, главное, не надо его сравнивать со вторым, т. е. с самим собой, иначе у нас в любом массиве найдутся равные элементы!

Таким образом получаем, что мы должны сравнивать каждый элемент со всеми последующими и проделывать это сравнение надо для всех элементов с первого до (*n*–1)-го. Почему не до *n*-го? А его не с чем сравнивать, у него следующих нет!

```
Const N=10;
Var Mas : Array [1..N] of Integer;
    I, J : Integer;
    Odinak: Boolean;
Begin Writeln('Введите ', N, ' целых чисел');
  For I:=1 to N do
    Begin Write(I:2, ' - ');
      Readln(Mas[I])
    End;
    Odinak:=False;
    For I:=1 to N-1 do
      For J:=I+1 to N do
        If Mas[I]=Mas[J] Then Odinak:=True;
      Writeln(Odinak)
    End.
```

Решение можно немного усовершенствовать. Как только мы обнаружили совпадающие элементы, из цикла можно выйти (вставив в программу оператор **Break**).

А вот задача гораздо сложнее.

**Пример 10.10\*.** *Вывести на экран все элементы массива, которые входят в него больше одного раза.*

С первого взгляда задачи кажутся похожими: здесь тоже надо искать совпадающие элементы. Действительно, первые три строчки в этой программе такие же, как и в предыдущей. А вот дальше начинаются существенные

различия. В предыдущем примере нам достаточно было просто обнаружить одинаковые элементы, здесь же один из них надо напечатать. Однако если, встретив совпадающие элементы, мы будем сразу же один из них печатать, один и тот же элемент может напечататься несколько раз. Например, при обработке массива 1 2 1 1 сначала обнаружится совпадение первого и третьего элементов (напечатается единица), потом обнаружится совпадение первого и четвертого элементов (напечатается еще одна единица). Значит, обнаружив совпадение и напечатав элемент, из цикла надо выходить. Достаточно ли этого для того, чтобы не было лишней печати? Нет! В нашем примере лишняя единица напечатается, когда будет обнаружено совпадение третьего и четвертого элементов (это уже другой цикл, он выполняться должен обязательно).

Как же быть? Придется каждый раз перед тем, как печатать очередной элемент из пары совпадающих, проверять, а не было ли уже в массиве такого раньше. Заведем для этой проверки логическую переменную *Was* (она будет равна **True**, если в начале массива элемент уже встречался). Вначале *Was* равна **False**, а как только найдем элемент, присвоим ей **True**. Вот теперь понятно, надо ли печатать элемент: это зависит от значения *Was*. Напечатав элемент, покинем цикл, чтобы не выполнять лишних сравнений.

```
Const N=10;
Var Mas:Array[1..N] of Integer;
    I,J,k:Integer;
    Was:Boolean;
Begin {Ввод массива Mas};
  For I:=1 to N-1 do
    For J:=I+1 to N do
      If Mas[i]=Mas[j] Then
        Begin Was:=False;
          For K:=1 to I-1 do
            If Mas[k]=Mas[i] Then Begin Was:=True;
                                  Break
                                End;
          If Not Was Then Begin Write(Mas[i]);
                              Break
                            End
        End;
      End;
    End;
  Writeln
End.
```

Вот так сложно (тройная вложенность цикла) решается простая с виду задача.

## Задачи 10.1–10.21. Массив. Заполнение, печать

- 10.1. Написать программу, в которой описывается массив из  $N$  элементов, этим элементам присваиваются заданные значения, а потом весь массив печатается. Присвоить элементам массива следующие значения:
- 10.1.1. Целые числа подряд, начиная с 1 (1, 2, 3, 4, 5, ...).
  - 10.1.2. Целые числа подряд, начиная с заданного (вводится с клавиатуры).
  - 10.1.3. Квадраты целых чисел по возрастанию.
  - 10.1.4. Целые числа с  $N$  до 1 по убыванию.
  - 10.1.5. Четные числа, начиная с 2 по возрастанию.
  - 10.1.6. Нечетные числа, начиная с заданного введенного нечетного числа по убыванию.
  - 10.1.7. Числа, кратные 3, начиная с 3 по возрастанию.
  - 10.1.8. Квадраты четных чисел по возрастанию.
  - 10.1.9. Степени числа 2, начиная с первой: 2, 4, 8, 16, 32, ...
  - 10.1.10. Числа 0 и 1 по очереди: 0, 1, 0, 1, 0, 1, ...
  - 10.1.11. Числа Фибоначчи.
  - 10.1.12. Элементы арифметической прогрессии (первый член и разность задаются с клавиатуры).
  - 10.1.13. Элементы геометрической прогрессии (первый член и знаменатель задаются с клавиатуры).
  - 10.1.14. Пары равных чисел: 1, 1, 2, 2, 3, 3, 4, 4, и т. д. (учтите,  $N$  может быть нечетным).
  - 10.1.15. Маленькие латинские буквы: a, b, c, d, ... (Замечание: коды букв использовать нельзя!)
  - 10.1.16. Пары латинских букв: большая и маленькая: A, a, B, b, C, c, D, d, ...
  - 10.1.17. Последовательности из  $K$  одинаковых чисел, например для  $K = 4$ : 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, ...
  - 10.1.18. Последовательности из  $K$  одинаковых маленьких латинских букв, например для  $K = 3$ : a, a, a, b, b, b, c, c, c, d, d, d, ...
  - 10.1.19. Первое число (оно вводится с клавиатуры) повторяется 1 раз, второе — 2 раза, третье — 3 раза и т. д. Например, если первое число 1: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ...
  - 10.1.20. Буква «a» повторяется 1 раз, «b» — 2 раза, «c» — 3 раза и т. д. Пример: a, b, b, c, c, c, d, d, d, d, ...
- 10.2. В следующих задачах у некоторых элементов массива надо изменить значения. После этого полученный массив напечатать.
- 10.2.1. Все четные числа заменить нулями.
  - 10.2.2. Обнулить все числа, перед которыми стоит четное число.

- 10.2.3. Все числа, не делящиеся на 3, заменить на ближайшие к ним, делящиеся на 3.
- 10.2.4. Все отрицательные числа заменить на их модули.
- 10.2.5. Каждое число заменить его квадратом.
- 10.2.6. Каждый третий элемент массива заменить на сумму двух предыдущих.
- 10.2.7. Заменить на противоположные по знаку все элементы массива, которые стоят после первого нуля.
- 10.2.8. Заменить на 0 все отрицательные числа массива, стоящие до первого положительного (если положительных нет, нулями заменятся все элементы массива).
- 10.3. В следующих задачах надо ввести какой-нибудь массив, переставить в нем указанным образом элементы, а затем его распечатать.
  - 10.3.1. Поменять местами первый и последний элементы массива.
  - 10.3.2. Поменять местами самый первый элемент массива и самый большой (если таких несколько, выбрать любой).
  - 10.3.3. Поменять местами максимальный и минимальный элементы массива (если таких несколько, выбрать любые).
  - 10.3.4. Поменять местами соседние пары элементов (первый со вторым, третий с четвертым и т. п.).
  - 10.3.5. Поменять между собой элементы массива, равноотстоящие от концов (1 с последним, второй с предпоследним и т. п.).

## Работа с одним массивом

- 10.4. Имеются сведения о росте каждого ученика в классе. Найти значение самого большого и самого маленького роста и для каждого ученика напечатать: его рост, насколько он выше самого маленького ученика и ниже самого большого.
- 10.5. С клавиатуры вводится последовательность целых чисел. Напечатать сначала те, которые делятся на 3, а потом те, которые делятся на 5 (некоторые числа могут быть напечатаны 2 раза).
- 10.6. С клавиатуры вводится последовательность символов. Напечатать сначала маленькие латинские буквы, входящие в нее, а потом большие.
- 10.7. Из массива выписать числа, у которых равные соседи, а потом числа, у которых соседи неравные (первый и последний элементы массива не рассматривать).
- 10.8. Задан символьный массив. Заменить в нем все маленькие латинские буквы на соответствующие большие, а все остальные символы — на символ '0'.
- 10.9. Подсчитать, сколько раз входит в массив его последний элемент.
- 10.10. Посчитать, сколько в массиве элементов, у которых равные соседи.

## Работа с несколькими массивами

- 10.11. Из одного массива целых чисел переписать содержимое в другой следующим образом: чтобы в начале массива оказались все нечетные числа в обратном порядке, а потом четные в прямом.
- 10.12. Символы из одного массива переписать в два: в первый переписать цифры, а во второй маленькие латинские буквы. Порядок символов оставить тот же.
- 10.13. Компоненты из одного массива переписать в два: в первый — компоненты, стоящие на нечетных местах, а во второй — компоненты, стоящие на четных местах.
- 10.14. Из массива вещественных чисел переписать в другой неотрицательные числа в обратном порядке.
- 10.15. Из одного массива переписать информацию в другой следующим образом: сначала компоненты, стоящие на четных местах, а затем компоненты, стоящие на нечетных местах.
- 10.16. Из двух массивов переписать информацию в один: сначала все содержимое первого массива, а затем все содержимое второго.
- 10.17. Из двух символьных массивов переписать в третий цифры (сначала все из первого, а потом все из второго).
- 10.18. Из заданного массива создать второй так, чтобы в нем вся информация была продублирована: два раза подряд записан первый элемент, потом два раза второй и т. п.
- 10.19. Из заданного массива переписать информацию в другой два раза следующим образом: сначала все элементы в прямом порядке, а затем в обратном.
- 10.20\*. Сравнить два массива (два массива будем считать равными, если в них одинаковое количество компонент и каждая компонента первого массива равна соответствующей компоненте второго массива).
- 10.21\*. Выписать элементы, которые встречаются в массиве ровно два раза.

## А нужен ли массив?

В предыдущем разделе мы рассмотрели довольно много задач с последовательностями (чисел или символов), которые решаются без использования массивов. В этой главе мы узнали о существовании этой новой структуры данных (массив) и теперь можем ее использовать.

Может быть, стоит организовывать работу с массивом во всех задачах, где есть много однотипных данных? Нет, если задачу можно решить без использования массива, ее лучше именно так и решать: это экономит и память, и время решения. Например, такая задача: имеется последовательность из 100 целых чисел. Сколько первых чисел надо сложить, чтобы сум-

ма стала больше 1000? Если при решении этой задачи использовать массив, надо ввести все 100 чисел, а потом просматривать их, искать сумму. А вдруг сумма первых двух уже больше 1000? Тогда остальные ни просматривать, ни вводить не надо!

Конечно, если задача «учебная», тренировочная, и в ней прямо задано, что надо произвести какие-то действия над массивом, ее именно с массивом и надо решать, не задумываясь. Если же в задаче задана некоторая последовательность, стоит задуматься, необходимо ли хранение всей последовательности в памяти. Если необходимо — надо использовать массив, если же для решения задачи достаточно хранить в памяти несколько членов последовательности, можно обойтись без массива.

**Пример 10.11.** С клавиатуры вводится последовательность  $N$  целых чисел. Найти сумму самой длинной возрастающей подпоследовательности.

Будем считать длину возрастающей последовательности ( $L$ ) и сумму чисел в ней ( $S$ ). Обозначим максимальную длину возрастающей последовательности  $L_{\max}$ , а сумму ее элементов —  $S_{\max}$ . Вначале длина (и максимальная, и текущая) равны 1, а сумма элементов — первому элементу. Сравнивая соседние элементы, узнаем, осталась ли последовательность возрастающей. Если это так, увеличиваем длину последовательности, считаем сумму и сразу же проверяем: если длина превысила максимум, максимум надо изменить. Если же следующий элемент таков, что последовательность перестала быть возрастающей, счет надо начинать сначала (этот элемент является началом новой возрастающей последовательности). Снова длина равна 1, а сумма — элементу.

```
Const N=10;
Var A:array[1..N] of integer;
    i,S,L,Lmax,Smax: integer;
Begin  For i:=1 to N do Readln(A[i]);
      L:=1; Lmax:=1; S:=A[1]; Smax:=S;
      For i:=1 to N-1 do
        If A[i+1]>A[i] Then Begin L:=L+1;
                                S:=S+A[i+1];
                                If L>Lmax Then
                                    Begin Lmax:=L;
                                           Smax:=S;
                                           End
                                End
        Else Begin S:=A[i+1];
                  L:=1;
                End;

      Writeln(Smax)
End.
```



Задача дает правильный ответ, но эффективно ли такое решение? Нужен ли здесь массив? В нашей программе всего два оператора цикла: для ввода массива и для его исследования. Эти действия можно совместить, так как при просмотре массива нам нужно знать значения только двух соседних элементов. И запоминать эти значения можно не в массиве, а в двух переменных. Назовем их *Sled* и *Pred* и перепишем программу, чтобы решить задачу без использования массива. Единственное существенное изменение: переменные будут меняться. То, что была на очередном шаге «следующей», на следующем станет «предыдущей».

```
const N=10;
var Sled,Pred: integer;
    j,S,L,Lmax,Smax: integer;
Begin Write('Введите первый элемент '); Readln(Sled);
    L:=1; Lmax:=1; Smax:=S; S:=Sled;
    For i:=2 to N-1 do
        Begin Pred:= Sled;
            Write('Введите ', i, ' элемент '); Readln(Sled);
            If Sled>Pred Then Begin L:=L+1;
                                S:=S+Sled;
                                If L>Lmax Then Begin Lmax:=L;
                                                    Smax:=S;
                                                    End;
                                Else Begin S:=Sled;
                                            L:=1;
                                        End
                            End;
            Writeln(Smax);
        End.
```

**Пример 10.12.** С клавиатуры вводится последовательность *N* целых чисел. Выявить отрезки неубывания в ней и вывести каждый из них на экран с новой строки.

Эта задача «хитро» сформулирована, но на самом деле она совсем несложная. Будем печатать очередное число. Если следующее число больше или равно ему, значит, отрезок неубывания продолжается и следующее число надо печатать на той же строке. В противном случае после печати очередного числа надо перевести строку. Остается только не забыть напечатать последнее число (мы не обработали его в цикле, потому что его ни с чем сравнивать не надо, для него нет следующего элемента).

```
const N=10;
var A:array[1..N] of integer;
    i:integer;
Begin
    For i:=1 to N do Readln(A[i]);
```

```
For i:=1 to N-1 do
Begin Write(A[i]:5);
      If (A[i+1]<A[i]) then Writeln;
End;
Writeln(A[n]:5);
End.
```

Можно ли эту задачу решить без использования массива? Ситуация та же, что и в предыдущем случае: в каждый момент времени нам достаточно только двух элементов массива. Чтобы вводимые и выводимые числа не «перемешались», надо использовать оператор **Read** вместо **ReadLn**. *Попробуйте изменить программу самостоятельно!*

Рассмотрим наиболее часто встречающиеся простые задачи, в которых работа ведется с последовательностью (символов или чисел), с точки зрения необходимости использования массивов при их решении.

Массив не нужен	Без массива не обойтись
Найти минимум, максимум, среднее арифметическое, сумму	Для каждого элемента указать, насколько он отличается от среднего арифметического, минимума, максимума
Определить, сколько элементов в последовательности отличаются от первого	Определить, сколько элементов в последовательности отличаются от последнего
Напечатать элементы последовательности в том порядке, в котором они были введены	Напечатать элементы последовательности в обратном порядке. Напечатать сначала отрицательные числа последовательности, а затем положительные
Входит ли в последовательность заданный элемент?	Есть ли в последовательности одинаковые элементы?
Обладают ли все элементы последовательности указанным свойством (например, являются ли все символы цифрами)?	Перестановки элементов: сдвиг, перестановка по возрастанию и т. п.

## Перестановка элементов массива

Только что было сказано, что все задачи, связанные с перестановками элементов, как правило, решаются с использованием массивов.

***Пример 10.13.** Циклически сдвинуть все элементы массива на одну позицию вправо.*

Сдвиг элементов массива на одну позицию вправо: первый перемещается на второе место, второй — на третье и т. д., т. е. каждый элемент перемещается на следующее место. Если при этом еще и последний перемещается на первое место, сдвиг называется циклическим.

Пусть числа в массиве будут вещественные.

Сдвигать элементы придется, начиная с конца. Ведь если мы сначала первый элемент передвинем на второе место, второй элемент «потеряется». Надо сдвигать предпоследний на последнее место и так далее. А вот чтобы последний не забыть, запомним его во вспомогательной переменной (назовем ее  $P$ ) и после сдвига поместим его на первое место.

Получившийся массив распечатаем.

```
Const N=10;
Var Mas : Array [1..N] of Real;
    I, J : Integer;
    P : Real;
Begin Writeln('Введите ', N, ' чисел');
  For I:=1 to N do
    Begin Write(I:2, ' - '); Readln(Mas[I])
    End;
  P:=Mas[N];
  For I:=N downto 2 do
    Mas[I]:=Mas[I-1];
  Mas[1]:=P;
  For I:=1 to N do
    Write(Mas[I]:5:1);
  Writeln
End.
```

**Пример 10.14.** Поменять местами минимальный и максимальный элементы массива.

Максимальный и минимальный элементы в последовательности мы искать умеем, но здесь надо найти не только сами элементы, но и определить места, на которых они стоят (чтобы потом переставить). Но если мы будем знать место, на котором стоит элемент (а место — это индекс элемента), то мы можем определить и сам элемент. Таким образом, запоминать будем места, на которых стоят найденные максимум и минимум. Обозначим их  $NMax$  и  $NMin$ . В качестве их начального значения возьмем 1 (т. е. предположим, что минимум и максимум стоят на первой позиции в массиве). Далее будем просматривать все элементы (со второго по последний). Если элемент больше текущего максимума, максимум надо изменить (менять будем место, оно станет равно индексу  $I$  рассматриваемого элемента). Аналогично вычисляем место минимума.

Эти действия можно было проделать не в отдельном цикле, а совместить с вводом.

По окончании этого цикла мы знаем наибольший и наименьший элементы массива. Если таких элементов несколько, то возьмем один из них (*определите самостоятельно какой*). Напечатаем их для проверки.

Теперь осталось поменять их местами. Чтобы поменять местами два элемента, надо один где-то запомнить (мы это сделаем в переменной  $P$ , запомним в ней максимум), а затем на его место записать минимальный элемент, а на место минимального —  $P$ .

Осталось только распечатать массив, чтобы убедиться, что все сделано правильно.

```
Const N=5;
Var    Mas : Array [1..N] of Real;
      I,J : Integer;
      P   : Real;
      NMin, NMax:Integer;
Begin Writeln('Введите ', N, ' чисел');
  For I:=1 to N do
    Begin Write(I:2, ' - '); Readln(Mas[I])
    End;
  NMin:=1; NMax:=1;
  For I:=2 to N do
    If Mas[I]>Mas[NMax] Then NMax:=I
    Else If Mas[I]<Mas[NMin] Then NMin:=I;
  Writeln(Mas[NMin]:10:2,Mas[NMax]:10:2);
  P:=Mas[NMax];
  Mas[NMax]:=Mas[NMin];
  Mas[NMin]:=P;
  For I:=1 to N do
    Write(Mas[I]:5:1);
  Writeln
End.
```

Иногда для того, чтобы переставить элементы так, как это требуется в задаче, приходится описывать и использовать вспомогательный массив.

**Пример 10.15.** *Переставить элементы в массиве целых чисел так, чтобы сначала оказались все отрицательные числа, потом все положительные, потом все нули.*

Эту задачу можно решить несколькими способами. Например, можно каким-то образом менять некоторые элементы местами друг с другом, чтобы в конце концов они оказались в нужном порядке. Мы же здесь выберем другой способ: воспользуемся дополнительным массивом. Запишем в него сначала все отрицательные числа (для этого придется просмотреть весь массив от начала до конца), потом, просматривая заданный массив от начала до конца еще раз, запишем в дополнительный массив все положительные числа. А вот чтобы записать нули, просматривать ничего не надо — нулями

заполняются все оставшиеся в массиве места. Только надо вспомнить, что нам в качестве ответа нужен не введенный для удобства работы дополнительный массив, а первоначальный. Поэтому придется все значения из вспомогательного массива переписать в основной, а заодно и недостающие нули добавить (если нулей в массиве нет, этот цикл просто не будет работать).

При переписывании чисел из основного массива в дополнительный следует помнить, что они могут занимать разные позиции в массивах. Например, если введен такой массив: 1, 2, -1, -2, ..., то на первое место в дополнительном массиве попадет третье число из заданного. Поэтому для работы с дополнительным массивом введем специальный счетчик  $J$  — номер позиции, куда будем писать очередной элемент. Вначале  $J$  положим равным 0. Обнаружив очередное число, которое надо переписать в массив, будем  $J$  увеличивать. Таким образом,  $J$  в каждый момент равно количеству чисел в дополнительном массиве.

После двукратного просмотра заданного массива, когда в дополнительный массив переписаны все положительные и отрицательные числа, в нем занято  $J$  позиций. Оставшиеся  $N - J$  позиций надо заполнить нулями (понятно, что просматривать массив для этого уже не надо).

После этих действий «ответ» полностью сформирован в дополнительном массиве, осталось переписать все содержимое дополнительного массива в основной. Так как массивы описаны одинаково, это можно сделать одним оператором присваивания, без использования цикла.

Остается только сказать, что и здесь дополнительный массив придется описывать такого же размера, как и основной.

```
J:=0;
For I:=1 to N do {Переписываем отрицательные числа
                  в дополнительный массив}
  If Mas[I]<0 Then Begin J:=J+1;
                      Dop[J]:=Mas[I]
                  End;
For I:=1 to N do {Переписываем положительные числа
                  в дополнительный массив}
  If Mas[I]>0 Then Begin J:=J+1;
                      Dop[J]:=Mas[I]
                  End;
For I:=J+1 to N do {Добавляем в конец массива
                  нужное количество нулей}
  Dop[I]:=0;
Mas:=Dop; {Переписываем содержимое
          дополнительного массива в основной}
```

## Задачи 10.22–10.29. Перестановка элементов

- 10.22. Циклически сдвинуть элементы массива на одну позицию влево.
- 10.23. Найти в массиве самый маленький элемент и, если он не стоит на первом месте, переставить его туда следующими способами: а) поменяв местами с первым элементом; б) освободив для него первое место, сдвинув нужные элементы вправо, не меняя их порядка.
- 10.24. Удалить из массива минимальный элемент (при удалении одного из элементов оставшиеся надо сдвигать влево). На освободившееся место продублировать последний элемент.
- 10.25. Удалить из массива все элементы, стоящие на нечетных местах. Оставшиеся элементы сдвинуть к началу массива, а пустые места заполнить нулями.
- 10.26. Имеется некоторый массив. С клавиатуры вводится элемент и номер позиции, на которую этот элемент надо вставить в массив. Выполнить задачу, если это возможно (в массиве есть такая позиция). Для вставки элементы массива надо сдвинуть вправо, последний элемент, который теперь в массив не помещается, — пропадает.
- 10.27. Имеется некоторый массив. С клавиатуры вводится новый элемент (его надо вставить в массив) и элемент массива, после которого надо произвести вставку нового. Выполнить задачу, если это возможно (в массиве есть введенный элемент). Если заданных элементов несколько: а) вставить после первого; б) вставить после каждого. При этом для освобождения места элементы массива сдвигаются вправо, те, которые не помещаются, — пропадают.
- 10.28. Имеется некоторый массив. С клавиатуры вводится номер элемента, который надо удалить. Выполнить задачу, если это возможно (в массиве есть элемент с таким номером). При удалении элементы массива надо сдвинуть, никаких пустых мест между элементами быть не должно. На пустые места поставить нули.
- 10.29. Имеется некоторый массив целых чисел. С клавиатуры вводится значение элемента, который надо удалить. Выполнить задачу, если это возможно (в массиве есть такой элемент). Если таких элементов несколько: а) удалить первое вхождение элемента; б) удалить все такие элементы. На пустые места поставить нули.

## Сортировка

Конечно, сортировки (перестановки элементов массива в определенном порядке, например по возрастанию) являются частным случаем перестановок, однако задача это настолько важная, что рассмотрим ее отдельно. Существу-

ет много различных методов сортировки массива, одни лучше (быстрее) работают, если заданный массив сильно перемешан, другие — если он «почти отсортирован». Отметим сразу, что методы, которые мы рассматриваем здесь, не являются наиболее эффективными, мы их выбрали за то, что они достаточно легко реализуются, за «понятность» алгоритмов.

Научимся переставлять компоненты массива так, чтобы в конце концов они оказались стоящими по неубыванию.

**Пример 10.16.** Сортировка выбором.

Мы умеем искать в массиве из  $N$  элементов максимум, умеем ставить его на нужное место. В отсортированном массиве наибольший элемент должен стоять на последнем месте. Таким образом, если мы найдем, на каком месте в массиве стоит наибольший элемент (и сам элемент, конечно, тоже), и поменяем найденный максимум местами с последним элементом, один из элементов массива уже займет свое место, останется отсортировать массив из  $N - 1$  элементов. Прodelав с ним аналогичные действия (теперь ищем максимум в массиве из  $N - 1$  элементов и переставляем его на  $(N - 1)$ -е место), получим, что уже два последних элемента стоят на нужных местах. Таким образом, с каждым шагом уменьшая количество компонент в массиве, мы «дойдем» до массива из двух элементов, в котором наибольший надо будет поставить на второе место. На этом сортировка закончится, все элементы окажутся на своих местах.

```
Const N=20;
Type Massiv=Array [1..N] of Real;
Var   A   : Massiv;
      Max : Real;
      Mmax,J,K : Integer;
Begin Writeln('Введите ', N, ' чисел');
  For J:=1 to N do
    Begin Write(J:2, ' - '); Readln(A[J])
    End;
  For K:=N downto 2 do {Так будет меняться размер
                        нашего массива}
    Begin Mmax:=1;
      For J:=2 to K do
        If A[J]>A[Mmax] Then Mmax:=J; {Определяем
                                     место максимума}
      Max:=A[Mmax]; {Перестановка}
      A[Mmax]:=A[K];
      A[K]:=Max
    End;
  For J:=1 to N do
    Write(A[J]:5:1);
  Writeln
End.
```

**Пример.** Пусть задан массив из 5 элементов: 2, 5, 3, 3, 4. На первом шаге мы поменяем местами числа 5 и 4, получим массив 2, 4, 3, 3, 5. Теперь рассматриваем массив из четырех элементов (без последнего числа 5, которое уже стоит на своем месте). В нем максимальный элемент 4, меняем его местами с числом 3, получаем: 2, 3, 3, 4, 5. Теперь рассматриваем массив из трех элементов, находим в нем максимальный элемент 3 (причем находим мы ту тройку, которая стоит на втором месте). Как это ни смешно, меняем его местами с числом 3, стоящим на последнем месте, получаем, естественно, то же, что и было. И последний шаг: в массиве из двух элементов находим максимальный — это число 3, оно и так стоит на последнем месте, но в нашем алгоритме никаких проверок не предусмотрено, так что меняем его местами (получается, что само с собой).

Почему у нас 2 последних шага получились ненужными? Да потому, что уже после третьего шага массив получился отсортированный, мы же (точнее, наш алгоритм) этого не заметили и продолжали действовать по намеченному плану. Как быть? Проверять после каждого шага, не получился ли уже отсортированный массив? Можно, но при этом придется делать уж слишком много проверок. Вспомним, мы говорили, что предлагаемые здесь методы сортировки не являются достаточно эффективными, мы выбрали их за то, что они достаточно легко реализуются, за «понятность» алгоритмов.

Впрочем, некоторые улучшения внести в алгоритм можно. Можно проверять, не стоит ли максимум на последнем месте, и хотя бы в таком случае не менять его местами «с самим собой».

*Сделайте это самостоятельно.*

**Пример 10.17.** Сортировка обменом (метод «пузырька»).

Будем рассматривать пары соседних элементов (сначала 1-й и 2-й, потом 2-й и 3-й и так далее, до  $(N-1)$ -го и  $N$ -го). Если обнаруживаем, что элементы стоят в «неправильном» порядке (первый больше второго), меняем их местами.

За один такой просмотр массива у нас некоторые элементы поменяются местами, но на своем месте могут оказаться пока не все. Сколько же раз надо таким образом просматривать массив?

Можно заметить, что самый большой элемент в результате наших манипуляций, где бы он ни стоял, обязательно окажется на своем «законном» месте — в конце массива (он как бы всплывает на поверхность, как пузырек). Таким образом, после первого просмотра самый большой элемент окажется на своем (последнем) месте, во время второго — элемент, чуть поменьше максимального (или равный ему), окажется на своем (предпоследнем) месте. Можно гарантировать, что за  $N - 1$  просмотров все элементы встанут на свои места.



```

Const N=20;
Type Massiv=Array [1..N] of Real;
Var    A   : Massiv;
       P   : Real;
       I,J : Integer;
Begin Writeln('Введите ', N, ' чисел');
  For J:=1 to N do
    Begin Write(J:2, ' - '); Readln(A[J])
    End;
  For I:=1 to N-1 do
    Begin For J:=1 to N-1 do
      {Внимание! Цикл до N-1, внутри есть индекс J+1}
      If A[J]>A[J+1] Then Begin P:=A[J];
                           A[J]:=A[J+1];
                           A[J+1]:=P;
                        End
    End;
  For J:=1 to N do
    Write(A[J]:6:1);
  Writeln
End.

```

**Пример.** Возьмем тот же массив, который мы брали, чтобы проследить работу метода сортировки выбором: 2, 5, 3, 3, 4.

Первый проход: сравниваются 2 и 5, они стоят в «правильном» порядке, менять ничего не надо, дальше сравниваются 5 и 3, они стоят «неправильно», надо их поменять местами, получаем 2, 3, 5, 3, 4. Опять сравниваются 5 и 3 (только пятерка теперь стоит на третьем месте, а тройка и вовсе другая, та, которая стоит на четвертом месте), они опять стоят «неправильно», меняем местами: 2, 3, 3, 5, 4. Осталось сравнить 5 и 4 и тоже поменять их местами: 2, 3, 3, 4, 5. Как видим, максимальный элемент — 5 — теперь стоит на «своем» последнем месте. Некоторые другие элементы тоже поменяли свои места. Таков результат нашего первого прохода: 2, 3, 3, 4, 5.

Второй проход. Сравниваем пары рядом стоящих чисел и убеждаемся, что все они стоят в «правильном» порядке. Никакие перестановки не нужны, массив не изменяется. Однако по нашему алгоритму мы должны сделать еще 2 прохода массива — совершенно лишние в данном случае.

Как прекратить просмотры вовремя, чтобы не делать лишних действий? Удобнее всего отслеживать, не встали ли все элементы на свои места. Это делается просто: все элементы стоят на своих местах, если во время просмотра массива не было сделано ни одной перестановки элементов. Значит, надо завести логическую переменную (назовем ее *Perest*). Она будет равняться **True**, если были сделаны перестановки, и **False** в противном случае. Так как количество просмотров теперь заранее неизвестно, вместо цикла **For** будем использовать **Repeat**. Приведем здесь только измененный фрагмент программы:

```
Repeat {повторяем просмотр массива}
  Perest:=False; {перестановок пока не было}
  For J:=1 to N-1 do
    If A[J]>A[J+1] Then
      Begin P:=A[J];
        A[J]:=A[J+1];
        A[J+1]:=P;
      Perest:=True {произошла перестановка}
    End
  Until Not Perest; {цикл до тех пор,
    пока перестановок не будет}
```

Как можно видеть, в обоих методах производится несколько проходов массива, во время каждого прохода какие-то элементы меняются местами. Чтобы более детально разобраться, как методы работают, полезно распечатывать состояние массива после каждого прохода. *Попробуйте сделать это.* Печать массива надо вставить во внешний цикл, перед его окончанием.

Также можно заметить, что в первом алгоритме массив всегда просматривается  $(N-1)$  раз, даже если он изначально правильно отсортирован, во втором методе (с переменной *Perest*) количество просмотров зависит от первоначального состояния массива.

**Задание.** Оба метода можно улучшить.

В первом методе, если наибольший элемент и так стоит на последнем месте, его менять местами (с ним самим!) не надо.

Во втором методе можно учесть, что за каждый проход на последнее место становится «правильный» элемент, «хвост» массива больше просматривать не надо, можно изменить заголовок цикла **For**.

Существует еще немало разных способов сортировки массива. Они различаются по разным параметрам: времени работы, количеству сравнений, количеству перемещений элементов. Конечно, эти параметры зависят от первоначального вида массива.

**Задание.** Дополните программу, чтобы вычислялось количество сравнений и количество перестановок элементов массива. Посчитайте, чему равны эти значения для обоих методов, когда массивы одинаковой длины, но разного вида («правильно» отсортированный массив; массив, отсортированный в обратном порядке; массив, где числа стоят в случайном порядке, и т. п.).

**Замечание.** При перестановках элементов массива, особенно при сортировках, часто встречаются ошибки, вызывающие появление копий каких-то элементов массива взамен других (которые совсем исчезают) либо появление каких-то посторонних элементов, которых раньше в массиве не было.

Поэтому результаты, которые выдает такая программа, надо проверять особенно тщательно: следить не только за тем, чтобы элементы стояли в правильном порядке, но и за тем, чтобы это были именно те самые элементы, которые были введены вначале.

***Пример 10.18.** Есть 2 массива (возможно, разной длины), отсортированные по неубыванию элементов. Соединить их содержимое в третий массив таким образом, чтобы этот массив также оказался отсортированным.*

Этот алгоритм называется **сортировка-слияние**. В реальной жизни такая задача встречается достаточно часто. Представьте себе: составлены списки двух классов, фамилии учеников в них записаны по алфавиту. А теперь надо объединить эти два класса в один, соблюдая при этом алфавитный порядок следования фамилий.

Конечно, можно просто записать в результирующий массив сначала все элементы первого массива, после них — все элементы второго массива, а потом получившийся массив отсортировать одним из вышеприведенных методов. Это очень неэффективный способ, так как мы здесь не используем тот факт, что заданные массивы уже были отсортированы.

Как же соединить два массива в один, не нарушая следования элементов в порядке неубывания? Надо сравнивать пары элементов из заданных массивов (начиная, естественно, с первых) и меньший из этой пары переносить в результирующий массив. На следующем этапе будет сравниваться опять пара элементов из разных массивов: один из них — это тот, который «остался» от предыдущего сравнения, а другой — следующий по порядку из другого массива. При этом надо очень аккуратно следить за индексами: вначале все они будут равны единице, а вот потом в каждом массиве они будут свои, будут меняться по своим правилам. В результирующем массиве к индексу будет добавляться единица после каждого сравнения пары элементов, так как после этого сравнения один из элементов занимает в нем свое место, и индекс надо «сдвинуть». А вот в заданных массивах после каждого сравнения индекс будет увеличиваться только в одном из них — в том, из которого элемент переносится в результирующий массив: так как элемент обработан, он больше не нужен, индекс можно «сдвинуть».

Как долго надо сравнивать пары? Пока в обоих массивах есть элементы. Как только один из массивов закончится, пары для сравнения уже не будет. Возможно ли, что оба массива закончатся одновременно? Никогда! Мы сравниваем пару элементов, один из пары забираем, второй остается. Обязательно ли раньше закончится тот массив, в котором было меньше элементов? Нет, возможно, элементов в нем меньше, но среди них есть элементы, которые больше элементов другого массива, т. е. их надо поставить в конец результирующего массива, а его начало заполнить элементами из «длинного» массива.

Таким образом, по окончании цикла, в котором сравниваются пары элементов, мы имеем следующую ситуацию: один из заданных массивов обработан, все элементы из него перенесены в результирующий массив, а в другом массиве остались элементы. Причем все они больше элементов, которые уже стоят в результирующем массиве, их надо просто переписать в его конец.

Чтобы это сделать, надо сначала разобраться, какой из массивов закончился. Это легко сделать, сравнив один из индексов с размерностью массива. После этого остается необработанный «хвостик» переписать в результирующий массив. Здесь тоже важно не запутаться с индексами: в каждом из массивов они «смотрят» как раз на нужные позиции, надо не забыть после копирования каждого элемента увеличивать их на единицу.

Обозначим данные массивы  $A1$  и  $A2$ , пусть их размерность, соответственно,  $N1$  и  $N2$ . Результирующий массив обозначим  $S$ , а его размерность будет  $N1 + N2$ . Индексы в массивах будем обозначать идентификатором, состоящим из двух символов: первый символ —  $i$ , а второй символ 1 или 2 для заданных массивов и  $S$  для результирующего. Фрагмент блок-схемы изображен на рис. 10.1.



Оставшиеся компоненты надо занести в результирующий массив, начиная с позиции  $is$

Рис. 10.1

```
{Сортировка-слияние}
Const N1=5; N2=7;
Var A1 : Array [1..N1] of Real;
    A2 : Array [1..N2] of Real;
    S  : Array [1..N1+N2] of Real;
    i1, i2, is, j : Integer;
Begin
    {Ввод первого массива}
    Writeln('Введите ', N1, ' чисел по неубыванию');
    For i1:=1 to N1 do
    Begin Write(i1, '- '); Readln(A1[i1])
    End;
    Writeln('Введите ', N2, ' чисел по неубыванию');
    For i2:=1 to N2 do
    Begin Write(i2, '- '); Readln(A2[i2])
    End;
    Writeln('Первый массив');
    For i1:=1 to N1 do Write(A1[i1]:8:1); Writeln;
    Writeln('Второй массив');
    For i2:=1 to N2 do Write(A2[i2]:8:1); Writeln;
    i1:=1; i2:=1; is:=1;
    While (i1<=N1) and (i2<=N2) do
    Begin If A1[i1]<A2[i2] Then Begin
            S[is]:= A1[i1];
            i1:=i1+1
        End
        Else Begin
            S[is]:= A2[i2];
            i2:=i2+1
        End;
        is:=is+1
    End;
    If i1>N1 Then For j:=is to N1+N2 do Begin
            S[j]:=A2[i2];
            i2:=i2+1
        End
        Else For j:=is to N1+N2 do Begin
            S[j]:=A1[i1];
            i1:=i1+1
        End;
    Writeln('Результат');
    For is:=1 to N1+N2 do Write(S[is]:8:1); Writeln
End.
```

Отметим, что описанные здесь действия можно производить и по-другому, например при записи «хвостика» в результирующий массив не проверять, какой массив закончился, а написать два цикла:

```
For j:=i1 to N1 do Begin S[is]:=A1[j];  
                        is:=is+1  
                        End;  
For j:=i2 to N2 do Begin S[is]:=A2[j];  
                        is:=is+1  
                        End
```

Дело в том, что если закончился первый массив, то  $i1 = N1 + 1$ , т. е. начальное значение переменной цикла больше конечного. Цикл в этом случае работать не будет. Таким образом, в зависимости от того, какой массив закончился, будет работать один из циклов и можно обойтись без проверки условия.

Кроме самого алгоритма сортировки-слияния массивов нам пришлось в программе написать операторы для ввода двух массивов и для вывода результата. В этих операторах совершенно необязательно использовать именно те индексы, о которых мы договорились, это сделано для общности. Обратите внимание, что для правильной работы алгоритма требуется, чтобы заданные массивы были отсортированы по неубыванию. В программе это условие не проверяется, но, если ввести неверные данные, алгоритм будет давать неверный результат.

## Задачи 10.30–10.35. Сортировка

- 10.30. Отсортировать массив по невозрастанию методом пузырька.
- 10.31. Отсортировать массив по невозрастанию выбором, устанавливая максимум на первое место.
- 10.32. Отсортировать массив по неубыванию (невозрастанию) выбором, устанавливая минимум на нужное место.
- 10.33. Произвести сравнение двух методов сортировки. Один и тот же массив отсортировать двумя методами и посмотреть, в каком случае получается больше сравнений и перестановок элементов. Проверку надо производить на разных массивах (элементы стоят по убыванию, по возрастанию, в случайном порядке).
- 10.34. Заданы 2 массива. Первый отсортирован по неубыванию, второй — по невозрастанию. Соединить их в один упорядоченный массив (для наиболее эффективного решения надо использовать метод сортировки-слияния).

10.35. Оцените эффективность метода «сортировка-слияние» следующим образом. Задайте 2 упорядоченных массива, соедините их в третий массив (простым добавлением второго массива после первого), отсортируйте, посчитайте количество сравнений и перестановок. Соедините те же массивы методом «сортировка-слияние», подсчитайте количество сравнений и перестановок.

## Поиск в массиве

*Пример 10.19. Входит ли в массив заданный символ? Если входит, то на каком месте стоит? (В случае, если символ входит в массив несколько раз, в качестве ответа выдать номер одной из позиций.)*

Эту задачу можно решить несколькими способами.

Самый простой: в цикле **For** просматриваем все элементы массива, если нужный найден, присваиваем переменной *Mesto* (требуемый номер позиции) индекс этого элемента. А чему же будет равно *Mesto*, если элемент в массив не входит? Удобно, чтобы эта переменная в таком случае равнялась нулю, причем это присваивание надо сделать до поиска. В таком случае, когда мы выйдем из цикла, переменная *Mesto* будет равна 0 (если элемент не входит) или номеру его позиции (если он входит).

*Посмотрите, номер какой позиции будет нам выдан в случае, если заданный элемент встречается в массиве несколько раз.*

Это решение не очень хорошо тем, что даже в том случае, когда элемент уже найден, мы продолжаем просматривать массив. А ведь этого делать не надо — ответ готов. Поэтому дополним нашу программу оператором **Break** — будем выходить из цикла, как только найдем заданный элемент (посмотрите, изменился ли выдаваемый ответ в случае, когда элемент входит в массив несколько раз).

```
Program Poisk_v_mas;  
Const N=5;  
var I, Mesto : Integer;  
      X : Char;  
      A : Array[1..N] of Char;  
Begin  
  Writeln('Введите массив ', N, ' символов');  
  For I:=1 to N do  
    Begin Write(I, ' - '); Readln(A[i]); ;  
  End;  
  Write('Что будем искать? '); Readln(X);  
  Mesto:=0;
```

```
For I:=1 to N do
    If X=A[i] Then Begin Mesto:=I; Break End;
If Mesto>0 Then
    Writeln('Есть!!! Стоит на месте номер ', Mesto)
    Else Writeln('Не найдено!!!')
End.
```

Другой способ — без оператора **Break**. В этом случае, чтобы иметь возможность своевременно выйти из цикла, надо использовать цикл **While** или **Repeat**. В заголовке цикла будет проверка двух условий: не является ли просматриваемый элемент искомым и не закончился ли массив, а в теле цикла — только один оператор, увеличение индекса массива.

При написании программы этим способом надо аккуратно выписывать условие цикла, следить, чтобы не произошел выход за границу массива.

При решении задачи первым способом мы после цикла проверяли значение переменной *Mesto*, чтобы понять, входит ли искомым элемент в массив. Здесь никаких дополнительных переменных нет, придется еще раз проверить, не является ли текущий элемент искомым.

```
Program Poisk_v_mas;
Const N=5;
var I: Integer;
    X : Char;
    A : Array[1..N] of Char;
Begin
    Writeln('Введите массив ', N, ' символов');
    For I:=1 to N do
        Begin Write(I, ' - '); Readln(A[i]); ;
        End;
    Write('Что будем искать? '); Readln(X);
    I:=1;
    While (I<N) and (A[i]<>X) do {*****}
        I:=I+1;
    If A[i]=X Then Writeln('Есть!!! Стоит на месте номер ', I)
        Else Writeln('Не найдено!!!')
End.
```

Давайте внимательнее посмотрим на значение переменной *I* — индекс массива. Вначале (перед циклом)  $I = 1$ , поэтому сравнения в заголовке цикла могут быть успешно выполнены (чего нельзя было бы сделать в случае  $I = 0$ , ведь тогда бы пришлось работать с элементом  $A[0]$ , которого нет).



Цикл заканчивается, когда  $I < N$ , т. е. самый последний элемент массива в цикле не обрабатывается. Можно ли в заголовке цикла поставить  **$I \leq N$**  (строка программы, помеченная звездочками)? Пусть заданный элемент в массив не входит, тогда тело цикла выполнится для  $I = 1, 2, \dots, N$ , и  $I$  станет равно  $N + 1$ . Как будут осуществляться проверки в заголовке цикла? Проверка  $I \leq N$  даст **False**, но, несмотря на это, вторая проверка тоже должна быть произведена. А проверять придется значение  $A[N + 1]$ , которое не существует.

Таким образом, в цикле приходится не проверять последний элемент. Как же быть, а вдруг он-то как раз нам и нужен? Эта проверка будет произведена после цикла, ведь, если среди первых  $N - 1$  элементов нужный не найден, мы выходим из цикла и при этом  $I = N$ , так что после цикла будем проверять как раз последний элемент.

Существуют и другие способы решения.

Давайте подумаем, сколько элементов массива нам придется просматривать при поиске? В самом удачном случае, когда искомый элемент стоит на первом месте, — только один, а в самом неудачном, когда искомый элемент стоит на последнем месте или вообще не входит в массив, нам придется просмотреть все  $N$  элементов массива.

***Пример 10.20.** Пусть имеется массив целых чисел, элементы которого упорядочены по неубыванию. Наиболее эффективным образом реализовать поиск заданного элемента (выдать ответ, входит ли элемент в массив).*

Конечно, можно написать точно такую же программу, как мы писали в предыдущем примере, но она не будет эффективной, ведь мы не используем условие, что массив упорядочен. Как можно улучшить программу, зная о таком свойстве массива? Можно прекратить просматривать массив, если просматриваемое число больше искомого. Здесь мы используем упорядоченность, но тоже в наилучшем случае элемент найдется сразу, а в наихудшем (если ищется число, большее всех, находящихся в массиве) придется просмотреть весь массив.

Рассмотрим алгоритм, который помогает существенно уменьшить количество просматриваемых элементов.

Во-первых, сразу определимся, каков диапазон значений находящихся в массиве элементов, входит ли искомый элемент в этот диапазон. Таким образом, если наш элемент меньше самого первого или больше самого последнего, задача решена.

Проверим, не являются ли первый или последний элементы искомыми. Если нет, значит, искомый элемент может находиться в массиве где-то между ними. Проверим, не в середине ли массива. Как найти эту самую серединку? Надо взять среднее арифметическое индексов (в начале это 1 и  $N$ ).

При этом надо помнить, что индексы в массиве могут быть только целыми, так что надо производить целочисленное деление. Если средний элемент равен искомому, задача решена. А если нет, мы можем существенно уменьшить область поиска. Так как массив упорядочен, мы теперь точно знаем, в какой части может находиться искомый элемент: «левее» среднего (если он меньше) или «правее» (если он больше). Таким образом, продолжаем поиск уже не в целом массиве, а в его «кусочке». Ищем таким же образом — проверяем средний элемент. Нашли — радуемся и заканчиваем работу, не нашли — опять делим «кусочек» на части и снова ищем уже в уменьшенном «кусочке».

И сколько же раз так уменьшать «кусочки»? Понятно, что даже если элемент не найдется, до бесконечности это делать нельзя. Когда закончить? Ну, во-первых, если вдруг будет обнаружено равенство «среднего» элемента и искомого, работа заканчивается. Также работу надо закончить, когда «кусочек» станет настолько мал, что его уже не надо делить на части. У «кусочка» какой «длины» уже нет серединки? Если массив из трех элементов, средний элемент в нем есть, надо его просмотреть, а вот если в массиве всего два элемента, никакой «серединки» у него нет, проверять нечего — элемент в массив не входит.

Этот алгоритм основывается на методе половинного деления. Этот метод мы уже использовали для работы с последовательностью и с функцией (см. примеры 9.4, 9.5).

**Пример.** Пусть массив из 10 элементов такой:

Индексы	1	2	3	4	5	6	7	8	9	10
Элементы	1	2	100	101	110	200	300	301	450	1000

Будем искать число 200. Оно больше 1 и меньше 1000, не равняется этим числам, следовательно, продолжаем поиск.

**1 шаг.** Средний элемент в массиве, у которого первый индекс 1, а последний 10, находится на месте  $(1 + 10) \div 2 = 5$ , т. е. это число 110.

**2 шаг.**  $110 < 200$ , поэтому поиск надо продолжать в «правой» части массива, т. е. начальный индекс будет теперь равен 5, а конечный — 10. Ищем «серединку» этой части:  $(5 + 10) \div 2 = 7$ , это 300.

**3 шаг.**  $200 < 300$ , надо искать в «левой» части (но не всего массива, а данного «кусочка»). Начальный индекс останется 5, а конечный станет 7. «Серединка» этого «кусочка» — на шестом месте, число найдено.

Посмотрим, как бы работал наш алгоритм, если бы надо было найти число 150. Были бы проделаны все действия, которые нужны для числа 200, но число не было бы найдено, поиск бы продолжился.



```

Begin KS:=(K1+K2) div 2;
      {поиск "серединки"}
If A[KS]=X
Then Poisk:=True
Else If A[KS]<X
      Then K1:=KS
           {поиск "слева"}
      Else K2:=KS
           {поиск "справа"}
End
End;
If Poisk Then Writeln('Есть!!!')
Else Writeln('Не найдено!!!')
End
Else
Writeln('Вы ввели неправильную последовательность')
End.

```

## Вспомогательный массив

В предыдущих примерах был задан массив, мы его каким-то образом обрабатывали. Бывают ситуации, когда программист сам заводит дополнительный (вспомогательный) массив для хранения в нем какой-то информации, которая нужна для работы программы.

**Пример 10.21.** Циклически сдвинуть элементы массива из  $N$  элементов на  $K$  позиций влево ( $K$  вводится с клавиатуры).

Мы делали циклический сдвиг массива на одну позицию (см. пример 10.13), для этого нам пришлось запомнить один элемент, чтобы после сдвига оставшихся поставить его на нужное место. Как поступить при сдвиге на  $K$  позиций?

Конечно, можно  $K$  раз произвести сдвиг на одну позицию. Но если  $K$  достаточно большое, это может быть слишком неэффективное решение. Если же производить сдвиг сразу на  $K$  позиций, придется  $K$  элементов где-то запомнить. Где? Надо создать вспомогательный массив для их хранения. Понятно, что в этом массиве должны поместиться  $K$  элементов, однако описать его как **Var Vsp : Array [1..K] of Real** нельзя, ведь при описании массива количество элементов в нем должно быть известно, а значение  $K$  станет известно только при работе программы, когда его введут с клавиатуры. Придется вспомогательный массив объявить, как и основной, из  $N$  компонент.

Сначала запомним первые  $K$  чисел во вспомогательном массиве, потом оставшиеся передвинем. При этом важно не запутаться в индексах: сдвиг начинается с  $(K + 1)$ -го элемента, он ставится на 1-е место,  $(K + 2)$ -й

элемент — на 2-е место, и так далее, пока  $N$ -й не встанет на  $(N - K)$ -е место. Этот цикл можно записать разными способами (в предлагаемом ниже варианте  $I$  считается от  $K + 1$  до  $N$ , можно  $I$  считать от 1 до  $N - K$ ; индексы элементов в цикле при этом будут другие).

После сдвига надо те элементы, которые ждут своей очереди во вспомогательном массиве, поставить на место. Их место — в «хвосте» массива, с позиции  $(N - K + 1)$  до  $N$ .

```
Const N=10;
Var    Mas, Vsp : Array [1..N] of Real;
        I, K : Integer;
Begin Writeln('Введите ', N, ' чисел');
  For I:=1 to N do
    Begin Write(I:2, ' - '); Readln(Mas[I])
    End;
  Write('На сколько позиций сдвинуть (меньше ', N, ')? ');
  Readln(K);
  For I:=1 to K do
    Vsp[I]:=Mas[I]; {запоминаем первые K элементов}
  For I:=K+1 to N do
    Mas[I-K]:=Mas[I]; {сдвиг оставшихся элементов}
  For I:=N-K+1 to N do
    Mas[I]:=Vsp[I-N+K]; {Ставим на место группу
                           из K элементов}
  For I:=1 to N do {Распечатаем массив}
    Write(Mas[I]:5:1);
  Writeln
End.
```

**Задание.** В этом примере мы попросили ввести с клавиатуры число для сдвига  $K < N$  и для этого случая написали программу. Если  $N = K$ , ничего сдвигать не надо, все компоненты массива останутся на месте. А если  $K > N$ , на сколько надо сдвигать массив? Определите это и допишите программу, чтобы она работала для любого  $K$ .

**Пример 10.22.** С клавиатуры вводится слово из латинских букв с точкой в конце. Сколько в нем гласных букв?

Интересно, что для решения этой задачи само слово в массиве хранить не нужно, но нужен массив для хранения гласных букв (ведь компьютер не знает, какие буквы являются гласными, надо ему это «объяснить», поместить их в массив). Поэтому в начале программы определим этот массив, просто «вручную». В Турбо Паскале есть возможность определить такой массив в разделе констант, можете посмотреть в справочнике, как это делается. Мы обойдемся операторами присваивания.

В программе будем вводить все буквы слова, пока не встретим точку (цикл **Repeat**). Каждую букву будем проверять, не является ли она гласной, т. е. не входит ли в массив гласных букв. Здесь мы можем использовать цикл **For**, причем не надо из него выходить, как только нашли букву — в цикле всего 6 элементов, не страшно сделать несколько лишних проверок. Заметим, что мы будем делать и совсем уж «ненужную» проверку: точку мы тоже проверим, не является ли она гласной буквой (мы ведь посмотрим, не точку ли нам ввели, только в конце цикла). Но, повторяем, так как массив небольшой, это не страшно.

```
Var Glas : Array[1..6] of 'a'..'z';
B      : Char;
i, KG  : Integer;
Begin Glas[1]:='a'; Glas[2]:='e'; Glas[3]:='i';
      Glas[4]:='o'; Glas[5]:='u'; Glas[6]:='y';
      KG:=0;
Write('Введите слово с точкой на конце ');
Repeat
  Read(B);
  For i:=1 to 6 do
    If Glas[i]=B Then KG:=KG+1
  Until B='.';
Writeln('В слове гласных букв ',KG )
End.
```

**Замечание.** Мы здесь описали массив не из символов (**Char**), а только из латинских букв. К сожалению, практического смысла это не имеет, так как если по ошибке присвоить компоненте массива не латинскую букву, а какой-то другой символ, транслятор ошибки не обнаружит (а должен бы!). Почему мы сразу, при описании массива не сказали, что он не просто из букв, а именно из гласных букв? Потому что «просто буквы» — латинский алфавит — закодированы подряд, и когда мы указываем диапазон 'a'..'z', он включает весь алфавит. Гласные буквы не стоят подряд, мы не имеем возможности как-то коротко их описать, можем только перечислить, что и приходится делать в программе.

Посмотрите еще раз внимательно на формулировку нашей задачи: нужно подсчитать, сколько в слове гласных букв, всяких. Эта задача очень отличается от задачи «Подсчитать, сколько в слове различных гласных букв». Так, в первом случае в слове «аа» — две гласные буквы, а во втором — одна. Несмотря на похожесть формулировки, задача про различные гласные буквы решается совсем по-другому, там одним массивом гласных букв не обойтись. Мы ее решим чуть позже, но советуем сейчас подумать, как бы вы ее решили — существует много способов.

## Метод подсчета

Этот метод применим только в тех случаях, когда элементы, с которыми надо работать, принадлежат к типу-диапазону (границы диапазона точно известны).

**Пример 10.23\*.** *Сортировка без сравнений. (Этот метод часто называют «сортировка подсчетом».)*

Пусть в массиве находятся маленькие латинские буквы. Сразу заметим, что в массиве должны находиться только они, наличие в нем других элементов приведет к ошибке «выход за границы массива». Учтем это, но в приведенной ниже программе правильность ввода отслеживать не будем. *Добавьте это в программу самостоятельно.*

Для решения мы воспользуемся следующей хитростью. Мы знаем, что в массиве может быть только 26 различных элементов: буквы от «а» до «z». Подсчитаем, сколько раз в массиве встречается каждая буква, у нас получится 26 чисел. Будем их хранить в массиве *Dop*, причем индексацию в массиве будем производить латинскими буквами: в переменной *Dop*['a'] подсчитаем количество вхождений буквы «а», в *Dop*['b'] — количество вхождений буквы «b» и так далее.

Чтобы подсчитать количество вхождений букв, сначала все 26 компонент массива *Dop* заполним нулями. Затем надо просматривать заданный массив *A* и добавлять 1 к нужной компоненте массива *Dop*. *A*[*J*] — это какая-то латинская буква, надо добавить 1 к компоненте массива *Dop* с этим индексом. Вот и получаются у нас одни квадратные скобки внутри других. И посмотрите, какой здесь интересный оператор цикла получился: просматриваем массив букв *A*, а заполняем массив чисел *Dop*. После этого цикла массив *Dop* заполнен, про каждую букву известно, сколько раз она встретилась в заданном массиве.

```
Const N=20;
Type Massiv=Array [1..N] of Char;
Var   A   : Massiv;
      Dop  : Array['a'..'z'] of Integer;
      I,J  : Integer;
      H    : Char;
Begin Writeln('Введите ', N, ' букв');
  For J:=1 to N do
    Begin Write(J:2, ' - '); Readln(A[J])
    End;
  For H:='a' to 'z' do
    Dop[H]:=0;
  For J:=1 to N do {Заполнение дополнительного массива}
    Dop[A[J]]:=Dop[A[J]]+1;
```

Оказывается, теперь мы можем напечатать нашу строку в отсортированном виде. Сначала в отсортированном массиве должны стоять буквы «а» (если они есть), их количество мы подсчитали в переменной *Dop*['a']. Если мы именно такое количество букв «а» напечатаем, это и будет начало отсортированного заданного массива. Если далее то же самое проделать с буквами «b», «с» и остальными, получим первоначальный массив в отсортированном виде. Причем, если какая-то буква в массив не входила, значение соответствующей компоненты массива *Dop* будет равняться нулю и цикл **For** работать не будет, соответствующая буква не напечатается.

```
For H:='a' to 'z' do {Печать текста в отсортированном
                                                                виде}
    For J:=1 to Dop[H] do
        Write(H:2);
    Writeln
```

В приведенном выше решении мы просто распечатали заданный массив в отсортированном виде, сам массив не меняли. Понятно, это сделать достаточно просто: вместо печати букв, обозначенных переменной *H*, организовать их запись в массив.

```
I:=0;
For H:='a' to 'z' do {Переписывание массива}
    For J:=1 to Dop[H] do
        Begin I:=I+1;
              A[I]:=H
        End;
    For I:=1 to N do {Печать получившегося массива}
        Write(A[I]:2);
    Writeln
End.
```

Интересно, что в этой задаче первоначально заданный текст даже не обязательно хранить в массиве (у нас он называется *A*). В цикле, который у нас называется «Заполнение дополнительного массива», можно считывать значение очередной буквы (не запоминая ее) и изменять нужную компоненту массива *Dop*.

С помощью приведенного выше метода можно решать еще несколько типов задач: определить, какие буквы входят, а какие не входят в массив, посмотреть, есть ли повторяющиеся буквы, подсчитать, сколько раз входит в массив каждая буква, какая буква встречается чаще всего, распределить буквы по частоте встречаемости в массиве. Все это легко сделать, имея массив *Dop*, в котором записано, сколько раз встречается каждый элемент введенного массива.



При решении некоторых из перечисленных задач, пользуясь этим методом, можно обойтись без дополнительного массива (подсчитывать, сколько раз буква встречается в массиве, но не хранить это значение, а сразу использовать). Например, если надо напечатать, сколько раз каждая из букв входит в массив, можно в нашей программе вместо элементов массива *Dop* использовать целочисленную переменную. Значение этой переменной (сколько раз входит в текст каждая буква) надо в цикле (от 'a' до 'z') вычислять и печатать. Правда, избавившись от дополнительного массива, мы здесь вынуждены 26 раз (столько букв) просматривать текст, так что вряд ли такое решение можно считать более эффективным.

```
For H:='a' to 'z' do
Begin S:=0;
    For J:=1 to N do
        If A[J]=H Then S:=S+1;
    Writeln (H:2, S:5)
End;
```

Можно сравнить этот способ со «стандартным» алгоритмом, когда мы поочередно просматриваем все элементы массива и выясняем, сколько раз каждый элемент в него входит. Конечно, если массив короткий, просмотров будет значительно меньше, но возникает следующая проблема: если просматриваемая буква встречается не первый раз, ее печатать уже не надо. Значит, надо «возвращаться назад», для каждой буквы проверять, не обработана ли она уже. Поэтому при работе с длинной строкой метод себя оправдывает.

Помните, мы «стандартным» методом печатали в примере 10.10 элементы, которые встречаются в массиве больше одного раза? Если известен диапазон изменения элементов массива (например, пусть это будут латинские буквы), задача также решается приведенным выше методом. Сравните, насколько «элегантнее» это решение! Правда, за «красоту» приходится расплачиваться созданием дополнительного массива. После ввода строки и вычисления значений элементов дополнительного массива надо просмотреть его и вывести нужные буквы.

```
For H:='a' to 'z' do
    If Dop[H]>1 Then Write(H:2);
```

Напомним еще раз, что метод подходит только для случаев, когда элементы, которые могут входить в массив, поддаются перечислению и диапазон разброса значений невелик по сравнению с длиной массива. Скажем, для вещественных чисел метод не подходит совершенно, для целых чисел подходит только в том случае, если значения ограничены.

Как можно видеть, мы начали с сортировки, но выяснили, что метод можно использовать для решения разнообразных задач. Рассмотрим еще несколько задач, при решении которых можно использовать этот метод.

**Пример 10.24\*.** *С клавиатуры вводится последовательность из  $N$  двузначных натуральных чисел. Сколько различных элементов содержится в ней?*

Задачу можно решить различными способами, но в данном случае применим только что описанный метод. Его можно использовать здесь, так как элементы последовательности принадлежат к типу диапазон: от 10 до 99. Кстати, чтобы застраховаться от ошибок ввода, мы и опишем число для хранения элемента последовательности не как целое, а как число из этого диапазона. Тогда при соответствующей настройке транслятора программа будет прерываться при неправильном вводе. Воспользуемся дополнительным массивом (назовем его *C*) из 90 элементов (столько двузначных натуральных чисел). Нумеровать его компоненты будем не с 1, а с 10 до 99, то есть каждый индекс будет «отвечать» за свое число. В этой задаче нам подсчитывать количество чисел не нужно, нужно ответить на вопрос типа «да-нет», поэтому массив *C* у нас будет логический. В начале заполним его значениями **False**. Далее вводим по очереди элементы последовательности и компоненте с индексом, равным введенному числу, присваиваем **True**. Конечно, при такой работе вполне вероятна ситуация, что некоторым компонентам мы присвоим **True** несколько раз, однако на результат это никак не повлияет.

По окончании ввода последовательности в дополнительном массиве сохранятся значения **False** — у тех компонент, которые в исходном массиве не встречаются, и **True** — у тех компонент, которые в массиве нашлись. Осталось подсчитать количество **True** в дополнительном массиве.

```
Const N=15;
Var   M: 10..99;
      C : Array[10..99] of Boolean;
      I, S : Integer;
Begin
  For i:=10 to 99 do C[i]:=False;
  Writeln('Введите массив - двузначные натуральные числа' );
  For I:=1 to N do
    Begin Write(i, ' - '); Readln(M);
           C[M]:=True
    End;
  S:=0;
  For i:=10 to 99 do
    If C[i] Then S:=S+1;
  Writeln ('Ответ ', S)
End.
```

**Пример 10.25\*.** С клавиатуры вводится последовательность символов, заканчивающаяся точкой. Составить из всех входящих в нее цифр наибольшее возможное число.

Здесь нам понадобится дополнительный массив, индексами которого будут уже не буквы, а цифры-символы. Конечно, сначала его обнулим. А затем будем вводить последовательность символов (пока не встретится точка), распознавать цифры и подсчитывать их: добавлять единицу к значению соответствующей компоненты дополнительного массива. Если в предыдущих задачах вводимая последовательность была «однородной», состояла либо из букв, либо из определенных чисел, здесь в последовательности разные элементы, но будем «отбирать» только цифры и работать только с ними.

Вспомним, что мы отличаем цифры от других символов благодаря тому, что они пронумерованы подряд, поэтому любая цифра находится в диапазоне от '0' до '9'.

По окончании ввода последовательности (а мы ее нигде не храним, не запоминаем) в компонентах массива *Dop* хранится, сколько раз встретились во введенной строке каждая цифра.

Остается только напечатать самое большое число. Оно будет начинаться с девяток (если они есть), потом пойдут восьмерки, семерки и т. д. Получилось решение нашей первоначальной задачи — сортировка! Нам надо напечатать последовательность введенных цифр, отсортированную по невозрастанию.

```

Var B : Char;
    Cif : Array['0'..'9'] of Integer;
    I : Integer;
Begin
    For B:='0' to '9' do Cif[B]:=0;
    Repeat Read(B);
        If (B>='0') and (B<='9') Then Cif[B]:=Cif[B]+1
    Until B='.';
    For B:='9' downto '0' do
        For I:=1 to Cif[B] do Write(B);
    Writeln
End.
```

**Пример 10.26\*.** С клавиатуры вводится последовательность маленьких латинских букв (непустая), заканчивающаяся точкой. Какая буква встречается в последовательности чаще всего? (Если таких букв несколько, вывести любую из них.)

Мы уже знаем, что надо подсчитать, сколько раз каждая буква встречается в последовательности. Будем эти данные хранить в массиве *Buk*. В качестве индексов удобно использовать буквы латинского алфавита. Заметим,

что если бы в задаче речь шла о буквах русского алфавита, использовать их в качестве индексов не получилось бы: про латинские буквы известно, что они стоят подряд, про русские буквы такого заранее сказать нельзя, их порядок зависит от используемой программы-русификатора. Для работы с русскими буквами пришлось бы их пронумеровать и пользоваться этими номерами.

Итак, получаем дополнительный массив из 26 чисел; каждое число означает, сколько раз данная буква содержится в последовательности. Теперь надо в этом массиве найти самое большое число и напечатать букву, которая ему соответствует (а это индекс компоненты массива). Воспользуемся обычным алгоритмом поиска максимума, причем здесь его можно даже немного упростить: вначале присвоим переменной, в которой хотим хранить максимум, какое-нибудь отрицательное число (а не значение первой компоненты массива). Здесь это можно сделать, так как мы точно знаем, что в нашем массиве нет отрицательных чисел и, более того, есть хотя бы одно положительное число (последовательность по условию непустая). Просматриваем массив, находим максимальное значение, не забываем запомнить букву, которая ему соответствует. По окончании этого цикла ответ готов, осталось только его распечатать.

```
Var   Buk : Array['a'..'z'] of Integer;
      B,C : Char;
      R: Integer;
Begin
  For C:='a' to 'z' do Buk[C]:=0;
  Writeln('введите текст с точкой в конце ');
  Repeat
    Read(B);
    If (B>='a') and (B<='z') Then Buk[B]:=Buk[B]+1
  Until (B='.');
  {*****}
  R:=-1;
  For C:='a' to 'z' do
    If Buk[C]>R Then Begin R:=Buk[C]; B:=C
                    End;
  Writeln(B, ' - ', R)
End.
```

Необходимо отметить, что эту задачу можно решать и другими способами. Например, не заводить дополнительный массив, но зато хранить в памяти введенную последовательность. Тогда можно подсчитывать, сколько раз входит во введенную строку каждая буква, и запоминать максимальное значение.

Какой способ лучше? Мы уже рассуждали на эту тему. Если входная последовательность большая — то первый, если маленькая — второй.

*Пример 10.27\*. С клавиатуры вводится текст — последовательность символов, заканчивающаяся точкой. Выписать все латинские буквы, для которых соблюдается следующее условие: в текст входит и маленькая буква, и соответствующая ей большая, причем большая входит большее число раз.*

Эту задачу удобно решать методом подсчета, так как в этом случае не надо хранить вводимую последовательность (а она ведь может быть большой длины и, главное, неизвестно какой, так что непонятно, как описывать массив).

Так же как и в предыдущих случаях, будем подсчитывать число вхождений букв в последовательность. Для этого заведем два массива: *Bl* и *Ll* — для подсчета количества вхождений больших и маленьких букв соответственно (*Big Letters* и *Little Letters*). Сначала оба массива обнулим.

Начинаем обработку последовательности — делаем это в цикле, пока не встретим точку. Считываем очередной символ, проверяем, не является ли он большой латинской буквой (а все большие латинские буквы, как нам известно, располагаются в диапазоне от 'A' до 'Z'). Если встретилась большая буква — добавляем единицу к соответствующей компоненте массива *Bl*. Индексом этой компоненты будет служить обрабатываемая буква. Если символ большой буквой не является, аналогично проверяем, не является ли он маленькой буквой, и прodelываем те же действия.

По окончании этого цикла в массивах *Bl* и *Ll* числа стоят на позициях, соответствующих буквам, которые встретились в последовательности. Остается проверить условие.

Для каждой маленькой буквы будем проверять, нашлась ли она в последовательности. Для этого воспользуемся циклом **For** по всем маленьким буквам алфавита от 'a' до 'z'. Если буква в последовательности была, надо проверять второе условие, смотреть, сколько раз встретилась соответствующая большая буква.

Но как же, зная маленькую букву, найти парную ей большую? Мы знаем, что и те, и другие буквы стоят по алфавиту, значит, разница между кодами большой буквы и соответствующей ей маленькой — постоянное число, его можно посчитать, например, как  $\text{Ord}('A') - \text{Ord}('a')$ . Значит, чтобы, из кода маленькой буквы получить код парной ей большой, надо к коду маленькой прибавить это число. А получить из кода буквы мы можем с помощью функции **Chr**. Таким образом, если в переменной *m* (типа **Char**) хранится маленькая латинская буква, парную ей большую можно получить по такой формуле:  $\text{Chr}(\text{Ord}(m) + \text{Ord}('A') - \text{Ord}('a'))$ .

Вычислив букву, мы знаем, компоненту с каким индексом надо исследовать в дополнительном массиве больших букв. Если проверка прошла успешно, букву надо напечатать.

```
Var  Bl: Array['A'..'Z'] of Integer;  
      Ll: Array['a'..'z'] of Integer;  
      B,C: Char;  
      I,S: Integer;  
Begin  
  For C:='A' to 'Z' do Bl[C]:=0;  
  For C:='a' to 'z' do Ll[C]:=0;  
  Writeln('Введите текст с точкой в конце ');  
  Repeat  
    Read(B);  
    If (B>='A') and (B<='Z') Then Bl[B]:=Bl[B]+1  
    Else  
      If (B>='a') and (B<='z') Then Ll[B]:=Ll[B]+1  
  Until (B='.');  
  For C:='a' to 'z' do  
    If Ll[C]>0 Then {Маленькая буква есть}  
      If Bl[Chr(Ord(C)+Ord('A')-Ord('a'))]>Ll[C]  
        Then Write(C:3);  
  Writeln;  
End.
```

**Пример 10.28\*.** Напечатать все буквы, входящие в последовательность, по убыванию частоты их встречаемости.

Понятно, что первая часть у этой программы будет такая же, как и у 10.26 (до строки из звездочек).

Как же нам напечатать буквы в порядке частоты встречаемости? Может быть, для этого надо отсортировать массив *Вук*? Это несложно, но для решения задачи недостаточно. Отсортировав этот массив, мы получим в порядке убывания числа, буквы же у нас ни в каком массиве не содержатся, их мы не сможем напечатать так, как это задано в условии. Значит, действуя по этому алгоритму, придется завести массив букв и, переставляя числа в массиве *Вук*, переставлять и буквы (с такими же индексами) в массиве букв. Таким образом, сортируя массив чисел, мы будем менять расположение букв и в конце концов получим требуемый массив, который и сможем распечатать. Довольно сложно получается: надо еще один массив заводить, перестановки сразу в двух массивах производить, да еще так, чтобы элементы переставились «одинаково».

Можно попробовать более легкий способ. Мы ведь умеем искать самую «частую», «максимальную» букву. Надо ее найти, напечатать и «вычеркнуть». Тогда в оставшемся массиве можно будет снова найти «максимальную» букву, это будет уже вторая по частоте встречаемости буква, напечатать ее и тоже «вычеркнуть». И так 26 раз (столько букв в латинском алфавите). Вот только как же вычеркнуть букву? Оказывается, в данном случае это несложно. Мы ведь имеем дело не с буквами, а с числами, с количеством букв. Можно это число у «ненужной» буквы заменить, например, на отрицательное. Это и будет нам «сигналом», что буква вычеркнута.

Таким образом, чтобы напечатать буквы в порядке убывания частоты встречаемости, надо действия, которые мы делали в предыдущем примере, записать в цикле (26 раз, по числу букв, чтобы все напечатать). После поиска «максимальной» буквы печатаем ее и «вычеркиваем» — заменяем соответствующее ей значение в массиве *Buk* на  $-10$  (или любое другое отрицательное число).

При поиске, какую букву надо печатать, мы никак не указываем, что отрицательные числа не надо рассматривать. Почему? Потому что мы точно знаем: в начале наш массив содержал 26 неотрицательных чисел. Мы ищем 26 «максимумов» — все они будут неотрицательными, отрицательные числа не будут рассматриваться, потому что они обязательно меньше максимальных.

```
For I:=1 to 25 do
Begin R:=-1;
    For C:='a' to 'z' do
        If Buk[C]>R Then Begin R:=Buk[C]; B:=C
                        End;
    Writeln(B, ' - ', R); Buk[B]:=-10;
End
```

## Задачи 10.36–10.45. Метод подсчета

*Договоримся, что здесь под строкой понимается не переменная типа String, а некоторая (возможно, очень длинная) последовательность символов.*

- 10.36. Задана строка из маленьких латинских букв. Определить, какая буква в ней встречается чаще всего.
- 10.37. Выписать все маленькие латинские буквы, которые входят в строку, и указать, сколько раз каждая в ней встречается.

- 10.38. С клавиатуры вводится строка. Выписать входящие в нее буквы в порядке увеличения частоты встречаемости.
- 10.39. Вводится строка символов, заканчивающаяся точкой. Выписать маленькие латинские буквы, которые не встречаются в строке.
- 10.40. Из скольких разных символов состоит введенная строка (в строке могут быть любые символы)?
- 10.41. Из строки удалить все повторные вхождения в нее больших латинских букв.
- 10.42. В строке оставить только символы, которые входят в нее по 2 раза.
- 10.43. Даны 2 слова одинаковой длины. Проверить, можно ли получить второе слово из первого перестановкой букв.
- 10.44. С клавиатуры вводится последовательность цифр, заканчивающаяся точкой. Какой наибольшее целое число можно составить из всех этих цифр? А наименьшее?
- 10.45. С клавиатуры вводится последовательность символов (строка), заканчивающаяся точкой. Можно ли из всех входящих в нее цифр составить палиндром (напечатать слова «да» или «нет»)? Если палиндром составить можно, напечатать его. Если таких палиндромов несколько, напечатать наименьший (наибольший).

## Строки

Задачи, связанные с обработкой массивов символов, с одной стороны, достаточно часто встречаются, с другой стороны, несколько специфичны, поэтому с массивами символов можно работать особым образом. Называются такие символьные массивы строками.

Мы неоднократно использовали строковые константы. Это последовательности символов, заключенные в апострофы. Строковые константы можно выводить на экран с помощью стандартной процедуры вывода целиком, а не поэлементно.

В Паскале существует возможность использования не только строковых констант, но и переменных (т. е. переменных, значения которых будут строкового типа). В роли строковых типов выступают одномерные массивы, компоненты которых — символы, а тип индекса непременно задается диапазоном от 1 до какого-то целого положительного числа (но не более 255). В стандарте Паскаля эти массивы непременно должны быть упакованными (компактно размещенными в памяти), для этого в описание добавляется слово **Packed**. В Турбо Паскале это слово добавлять не обязательно, там все массивы упаковываются автоматически. Например:

```
Var Slovo : Packed Array[1..10] of Char
```



Во многих версиях языка (в том числе в Турбо Паскале) есть даже специальный тип **String**. Описание **String[N]**, где **N** — целое положительное число, соответствует описанию **Packed Array[1..N] of Char**. Описание **String** без квадратных скобок тождественно описанию **String[255]** (поэтому для экономии памяти, если длина строки известна, лучше ее указывать явно).

Как бы ни была описана строка, на нее переносятся все свойства массива. К отдельным компонентам строки (символам) можно обращаться, указывая имя строки и позицию нужного символа. Строковым переменным можно присваивать значения других строковых переменных или строковых констант.

К тому же описание символьного массива как строки дает дополнительные возможности.

1. В отличие от обычных массивов строки можно сравнивать, используя операции отношения ( $<$ ,  $=$ ,  $<$ ,  $>$  и др.).
2. Во многих версиях языках над строками (описанными как **String**) и символами определена операция «сцепление», которая обозначается знаком «+» и официально называется «конкатенация». В разговорной речи ее также называют «сложение». Результат сцепления двух строк — строка, полученная приписыванием всех символов второй строки справа к первой строке. Например, в результате выполнения операторов

```
V:= 'вино';  
G:= 'град';  
S:=V+G
```

значением строковой переменной **S** будет строка **'виноград'**.

3. Строки можно выводить целиком, используя процедуры **Write** и **Writeln**.
4. Строку, описанную с помощью **String**, можно вводить целиком процедурами **Read** и **Readln**.

Остановимся на вводе чуть подробнее. Пусть в программе есть оператор **Readln(S)** или **Read(S)**, где **S** — строка, описанная как **STRING[N]**. С клавиатуры вводятся некоторые символы. Ввод, естественно, заканчивается нажатием клавиши **Enter** (перевод строки). Если количество символов в набранной строке меньше или равно **N**, все они станут значением строки **S**. Если набрано символов больше **N**, процедура ввода присвоит переменной **S** первые **N** из набранных символов. При этом, если чтение производилось с помощью **Readln**, оставшийся кусок набранной строки «потеряется», дальнейший ввод будет происходить с новой строки. Если же использовалась процедура **Read**, дальнейший ввод будет происходить, начиная с первого непрочитанного символа. Напомним также, что никакие символы: про-

белы, точки, запятые, любые другие знаки — разделителями для строк не являются, две строки при вводе «принудительно» можно разделить либо клавишей **Enter**, либо указывая длину строки в описании.

Пример:

```
Var S2 : String[2];   S3 : String[3];   S4 : String[4];
```

Пусть на клавиатуре набрано:

```
1234 <Enter>
```

После работы оператора **Read (S2, S3, S4)** переменные примут следующие значения:  $S2 = '12'$ ,  $S3 = '34'$ ,  $S4 = ''$ . Для переменной  $S3$  «останется» всего 2 символа, а вот переменной  $S4$  символов вообще «не хватит», она примет значение «пусто». Никакого дополнительного ввода программа ждать не будет.

А для работы фрагмента **Readln (S2) ; Readln (S3, S4)** потребуется ввести две строки. Пусть введено:

```
1234 <Enter>  
ABCDEF <Enter>
```

Переменные станут равны  $S2 = '12'$ ,  $S3 = 'ABC'$ ,  $S4 = 'DEF'$ .

Конечно, если формат ввода (вид вводимой строки) четко задан в условии задачи, программист обязан при написании программы его учитывать, если же формат ввода строки определяется программистом, удобнее ввод каждой строки производить с новой строчки, пользуясь процедурой **Readln**.

Строковый тип часто используют для обработки слов. Понятно, что слова могут быть разной длины. При этом поступают так, как мы это уже делали в предыдущих задачах: описывают строку некоторой достаточно большой длины, а используют только ее часть, нужную для хранения слова. Таким образом, когда процедурой ввода читается сразу целиком строка (некоторой неизвестной длины  $N$ ), она помещается в начало массива, с 1-й позиции по  $N$ -ю. Оставшиеся позиции с  $(N+1)$ -й по последнюю в строке будут заполнены «мусором». Очень важно в программе работать именно с введенными символами, не залезть в «мусор». Для этого можно, например, пометить конец слова специальным символом (часто используется точка) или заранее заполнить массив, например, пробелами — тогда в нем не будет «мусора», все элементы будут определены программистом.

Во многих версиях языка (в том числе в Турбо Паскале) длину введенной строки можно узнать с помощью специальной встроенной функции **Length (<строковая переменная>)**. Существуют и другие встроенные функции для работы со строками, их перечень можно посмотреть в описании версии языка, которой вы пользуетесь. В Турбо Паскале наиболее часто используются следующие процедуры и функции.

<code>Delete (S, Index, Count)</code>	Процедура	Удаляет из строки <i>S</i> <i>Count</i> символов, начиная с символа под номером <i>Index</i> . При этом символы, расположенные правее удаляемых, автоматически сдвигаются влево
<code>Insert (Z, S, Index)</code>	Процедура	Помещает в строку <i>S</i> подстроку <i>Z</i> , начиная с символа под номером <i>Index</i> . При этом символы автоматически сдвигаются вправо
<code>N:=Length (S)</code>	Функция	Определяет длину строки
<code>P:= Pos (Z, S)</code>	Функция	Отыскивает в строке <i>S</i> первое вхождение подстроки <i>Z</i> и возвращает номер позиции, с которой эта подстрока начинается. Если подстрока не найдена, возвращается ноль
<code>Z:=Copy (S, Index, Count)</code>	Функция	Копирует из строки <i>S</i> <i>Count</i> символов, начиная с символа под номером <i>Index</i>
<code>Val (S, R, P)</code>	Процедура	Конвертирует строку <i>S</i> в число <i>R</i> (целое или вещественное). Если преобразование выполнено успешно, <i>P</i> = 0, иначе <i>P</i> показывает позицию ошибочного символа
<code>Str (R, S)</code>	Процедура	Конвертирует число <i>R</i> в строку <i>S</i> (результат — строка из символов-цифр числа)

**Пример 10.29.** С клавиатуры вводятся слова (каждое на новой строке) до тех пор, пока не будет введено слово «Конец». Напечатать те слова, у которых первая и последняя буквы совпадают.

Вводимое слово будем записывать в переменную типа **String**. Опишем ее как строку из 30 символов (слова большей длины вводить не будем). Все слово целиком считывается одной процедурой **ReadLn** (напомним еще раз: это возможно только для массивов-строк, все остальные массивы надо вводить по одной компоненте с помощью оператора цикла). Вводить строки надо очень аккуратно. Компьютер, конечно, термин «слово» не понимает, он вводит строку (ввод строки заканчивается нажатием клавиши **Enter**). Поэтому, если вы в одной строке напишете несколько слов (через запятую или пробел), компьютером вся строка будет понята как единое слово и записана в переменную *S*. Если вы попытаетесь ввести строку длиннее объявленных 30 символов, в переменную *S* попадут только первые 30, остальные «потеряются».

Чтобы эта программа закончила работу, надо ввести слово «Конец» именно так, как оно написано в программе (в нашем случае русскими буква-

ми, первая буква большая, остальные маленькие, без пробелов и каких-либо других знаков). Имейте в виду, что при работе с символами различаются большие и маленькие буквы, русские и латинские (даже если они выглядят совершенно одинаково), пробел является самостоятельным символом.

В нашей задаче внутри оператора цикла будет ввод слова и сравнение его первой и последней букв. Первая буква в слове стоит, естественно, на первом месте, а номер места последней равен длине слова.

```
Const Kon='Конец';  
Var S : String[30];  
Begin Repeat  
    Write('Слово  '); Readln (S);  
    If S[1]=S[Length(S)] Then Writeln(S)  
Until S=Kon  
End.
```

**Пример 10.30.** С клавиатуры вводится слово. Если в нем присутствуют какие-то другие символы кроме латинских букв (больших и маленьких), напечатать сообщение об ошибке. Иначе преобразовать слово так, чтобы оно начиналось с большой буквы (поменять первую букву на соответствующую большую, если она маленькая), а все остальные буквы были маленькие.

Можно использовать уже известное нам свойство кодировки символов: латинские буквы расположены подряд; сначала идут все большие (от «А» до «Z»), а потом (возможно, с некоторым промежутком) — все маленькие. Таким образом, получается, что «расстояние» (разность кодов) между большой и соответствующей ей маленькой буквами всегда одинаковое. Вычислим эту разность в начале программы и поместим в переменную *R* — она нам пригодится. Напомним, что с русскими буквами такое «не пройдет», так как русские буквы часто бывают расположены не подряд.

Для того чтобы пометить, есть ли в слове ошибка, введем логическую переменную *Err*, которой придадим значение **True**, если обнаружим ошибку (символ, не являющийся латинской буквой).

Сначала проверим первую букву слова. Если это маленькая латинская буква (т. е. лежит в диапазоне от 'a' до 'z'), ее надо заменить на соответствующую большую: к коду буквы прибавить *R* и поставить в слово на первое место букву с получившимся кодом. Если первый символ слова буквой не является, надо сообщить об ошибке и закончить работу.

В случае «хорошего» первого символа надо просмотреть все остальные символы слова. Будем их рассматривать в цикле как элементы массива с номерами от 2 до последнего (равного длине слова). Если очередной символ — большая латинская буква, заменяем ее на маленькую (на букву, код которой на *R* меньше), если встретился «плохой» символ, отмечаем ошибку и заканчиваем работу.

Ну и по окончании цикла печатаем либо слово, либо сообщение об ошибке.

Обращаем ваше внимание, что здесь мы пользуемся стандартной функцией **Chr**, которая по коду выдает символ. Аргумент этой функции (код символа) может быть только в пределах от 0 до 255. У нас аргумент «выскочить» за эти пределы не может, так как мы проверяем, является ли он нужной буквой.

```

Var S   : String[30];
    R, K : Integer;
    Err  : Boolean;
Begin R:=ord('A')-ord('a');
    Err:=False;
    Write('Слово '); Readln (S);
    If (S[1]>='a') and (S[1]<='z')
        Then S[1]:=chr(ord(S[1])+R)
    Else
    If (S[1]<'A')or (S[1]>'Z') Then Err:=True
    Else For K:=2 To Length(S) do
        Begin If (S[K]>='A')and(S[K]<='Z')
            Then S[K]:=chr(ord(S[K])-R)
            Else
            If (S[K]<'a') or (S[K]>'z')
                Then Begin Err:=True;
                        Break
                    End
                End;
        End;
    If Err
        Then Writeln ('Ошибка!!! В слове посторонние
                                символы.')
        Else Writeln(S)
    End.

```

В написанных выше программах мы пользовались встроенной функцией Турбо Паскаля **Length**, определяющей длину строки. Важно знать, что она правильно работает, если длина строки не изменяется, если же со строкой работать как с массивом, добавляя символы поэлементно, функция даст неправильное значение. В таком случае надо либо не использовать **Length**, обходиться «своими силами», либо для удаления и вставки символов пользоваться специальными встроенными процедурами и функциями (см. таблицу выше). Эти процедуры и функции, меняя слово, меняют и сведения о его длине. Заметим, что информация о длине строки хранится в служебном байте строки с номером ноль, и его значение можно менять и вручную.

Если не пользоваться функцией **Length**, то один из способов отделить в строке-массиве значащую его часть от «мусора» мы знаем: надо все вре-

мя помнить (хранить в какой-либо переменной) количество значащих символов в массиве. Значение этой переменной надо увеличивать при вставке букв в слово и уменьшать при их удалении. Этот способ в данном случае нехорош тем, что не позволяет выводить массив целиком при помощи процедуры **Write**. Она не умеет выводить не весь массив, а только его часть, без «мусора».

Можно использовать такой прием: сделать, чтобы в массиве после букв слова был не «мусор», а некоторые специальные символы, которые не будут мешать читать слово при выводе. Такими символами могут быть, например, пробелы.

Используя этот прием, решим следующую задачу.

**Пример 10.31\*.** *С клавиатуры вводится слово (непустая последовательность латинских букв с точкой в конце). Если в слове есть удвоенные буквы (пары стоящих рядом одинаковых букв), удалить одну из каждой пары.*

Используем при решении в качестве специального дополнительного символа не пробел, а символ подчеркивания « », чтобы его было видно в тексте и при печати.

Один из способов решения этой задачи следующий.

```
Const Pusto='_____'; {20 символов ' '}
```

```
Var S : Packed Array [1..20] of Char;
```

```
    K,L,J : Integer;
```

```
Begin S:=Pusto;{Вначале все слово заполняем специальными
```

```
                                                    символами}
```

```
    Write('Введите слово с точкой в конце ');
```

```
    L:=0;
```

```
    Repeat
```

```
        L:=L+1;
```

```
        Read (S[L]);
```

```
    Until S[L]='.';
```

```
    S[L]:='_';{вместо точки в конец слова надо
```

```
                                                    поставить " "}
```

```
    L:=L-1;{длина слова}
```

```
    K:=1;
```

```
    Repeat
```

```
        If S[K]=S[K+1] Then
```

```
            Begin For J:=K+1 to L do
```

```
                S[J]:=S[J+1]; {сдвиг}
```

```
                L:=L-1; {длину уменьшаем}
```

```
            End;
```

```
        K:=K+1;
```

```
    Until S[K]='_';
```

```
    Writeln(S)
```

```
End.
```

Вначале мы весь массив заполняем специальными символами, а потом в начало массива ставим буквы слова, которые вводятся с клавиатуры (это можно было сделать и наоборот: ввести слово, а потом дополнить массив символами «\_»).

В процессе ввода мы посчитали длину слова (переменная *L*) и в дальнейшем будем пользоваться этим значением. Этого можно было и не делать: конец слова в массиве определить легко, ведь после него стоит «\_». Использование переменной *L* в нашей задаче дает возможность написать программу с внутренним циклом **For**. Без нее пришлось бы **For** заменить на цикл с условием (например, **While**).

Обнаружив две рядом стоящие одинаковые буквы, мы удаляем вторую из них, сдвигая «хвост» слова на одну позицию влево. При этом на месте последней буквы (мы ее передвинули) появится символ «\_».

Наша программа удаляет вторую букву из пары совпадающих. Если в слове три буквы подряд одинаковы, удалена будет только одна (вторая), если подряд стоят 4 одинаковые буквы, после удаления останутся две (удалится по одной из каждой пары).

***Задание.** Исправьте приведенную программу, используя стандартную процедуру *Delete*, которая автоматически изменит длину строки после удаления символа.*

Интересно, что эта программа одной небольшой заменой переделывается в программу для решения следующей задачи: если в слове есть несколько подряд стоящих одинаковых букв, удалите их, оставив только одну. *Найдите, что нужно изменить.*

***Пример 10.32.** С клавиатуры вводится предложение, состоящее из трех слов. Все слова разделены ровно одним пробелом. В конце предложения никаких «особых» знаков нет. Одинаковы ли первое и последнее слова в предложении? (Например: «Думай, голова, думай».)*

Заведем две строковые переменные *S1* и *S3* — для первого и третьего слов соответственно. Опишем их как **String[50]** (не будем использовать слова слишком большой длины).

Для решения этой задачи нам надо прочитать и сохранить в памяти первое слово (используем для этого строковую переменную *S1*). Прочитать его оператором **Read(S1)** нельзя, потому что таким образом прочитается все предложение (или первые его 50 символов, если оно большое). Мы уже говорили, что для компьютера пробел является таким же символом, как буква, точка и т. п., ему надо «объяснить», что необходимо прочитать только буквы, стоящие до первого пробела. Для этого будем считывать информацию по одной букве и добавлять ее к строке операцией конкатенации, формируя слово. Строковую переменную надо перед этими действиями «обнулить», присвоить ей значение «пустая строка». «Пустая строка» записывается так: ставятся подряд два апострофа (между апострофами ничего нет, даже про-

бела). Вот к этой пустой строке и будут поочередно приписываться буквы нашего слова.

Как только в переменную *C* прочитается пробел, цикл прекратится (при этом заметим, что пробел тоже присоединится к слову в качестве последней «буквы»).

Второе слово нам надо пропустить, поэтому читаем буквы до пробела включительно в переменную *C*, но записывать их никуда не будем.

А вот третье слово предложения прочитаем сразу целиком оператором **ReadLn (S)**. Его по-другому прочитать и нельзя: если считывать буквы по одной, пользуясь оператором цикла, непонятно, как этот цикл завершать, ведь неизвестна ни длина слова, ни чем оно заканчивается.

К прочитанному слову мы добавим пробел, чтобы его можно было целиком сравнивать с первым словом (ведь к первому слову мы пробел при чтении добавили).

```
Var S1,S3 : String [50];
    C : Char;
Begin Writeln('Введите предложение');
      S1:=''; {Два апострофа подряд – пустая строка}
      Repeat {Читаем первое слово}
        Read (C);
        S1:=S1+C
      Until C=' ';
      Repeat {Пропускаем второе слово}
        Read (C);
      Until C=' ';
      Readln(S3); S3:=S3+' ';
      Write('Слова ');
      If S1<>S3 Then Write(' не ');
      Writeln('равны')
End.
```

Обратите внимание: мы в этой задаче хранили в памяти (в переменных *S1* и *S3*) только информацию, нужную для решения задачи. Второе слово из предложения мы не запоминали, но прочитать нам его пришлось, иначе мы не смогли бы «добраться» до третьего слова.

Эту задачу можно было бы решать и по-другому, например ввести целиком предложение и выделять из него отдельные слова.

Вообще говоря, большинство задач со строками допускают несколько путей решения: можно использовать встроенные строковые функции или обходиться без них, часто можно при решении вообще не использовать строковый тип. Изучая тему «Циклы», мы решали задачи, где с клавиатуры вводилась последовательность букв и ее надо было каким-то образом обработать. Часто такие задачи решают с помощью строк. Посмотрим, как это делается.



**Пример 10.33.** (Другой способ решения примера 9.18.) С клавиатуры вводится слово (последовательность букв), заканчивающееся точкой. Одинаковы ли первая и последняя буквы этого слова?

Со строковыми переменными и функциями все получается гораздо проще: надо ввести слово-строку, проверить его первый символ (вдруг это точка), а потом сравнить первый и предпоследний символы. Какой способ лучше? Конечно, первый длиннее, в нем проще запутаться, с другой стороны, для работы первой программы нужно меньше памяти (мы помним всего две буквы, а не всю строку). К тому же, если подходить формально, во втором случае длина слова ограничена предельно допустимой длиной строки (255 символов), в первом случае никаких ограничений нет.

```
Program Bukv1;  
var S : String;  
Begin  
  Readln(S);  
  If S[1]='.' Then Writeln('Введено пустое слово')  
    Else Begin Write('Первая и последняя буквы  
                  в слове ');  
              If S[1]=S[length(S)-1] Then Writeln  
                ('равны')  
              Else Writeln  
                ('не равны')  
            End  
End.  
End.
```

Решим еще несколько задач двумя способами: с использованием строк и без них.

**Пример 10.34.** С клавиатуры вводится предложение, заканчивающееся точкой, восклицательным или вопросительным знаком. Напечатать предложение без лишних пробелов (т. е. не печатать все пробелы в начале предложения, а между словами оставить только по одному пробелу).

Способ решения без использования строковой переменной. Вводить символы будем по одному, сразу обрабатывать и, если надо, печатать.

Сначала учтем, что предложение может начинаться с некоторого количества пробелов, которые надо пропустить, т. е. ввести, но не печатать. Последующие символы надо печатать, если они не являются пробелами. Если же встретился пробел, надо напечатать только один из группы подряд стоящих. Как это сделать? Придется запоминать, был ли предыдущий символ пробелом. Для этого можно ввести логическую переменную.

```
var p, pred : char;  
    Prob     : boolean;
```

```

begin
  writeln('Введите предложение ');
  repeat
    read(p)
  Until p<>' '; {Избавились от пробелов в начале}
  Prob:=False; {Найден символ - не пробел}
  Write(p);
  While (p<>'.' ) and (p<>'!') and (p<>'?') do
  Begin Pred:=p;
    Read(p);
    If p<>' ' then Begin Write(p); Prob:=False End
                      else If Not Prob Then Begin Write(p);
                                                                Prob:=True
                                                                End
  End;
end.

```

Решая задачу с использованием строковой переменной, вводим в нее все предложение целиком. Затем будем искать пару стоящих рядом пробелов и удалять один из них. Когда таких пар не останется, лишним в предложении может быть только один пробел в самом начале. Надо проверить, есть ли он там, и, если есть, удалить.

```

var S:String;
begin
  write('Введите предложение '); Readln(S);
  While pos(' ', S)>0 do {ищем 2 пробела подряд}
    delete(S, pos(' ', S), 1);
  If pos(' ', S)=1 then delete(S,1,1); {Если надо, удаляем
                                         начальный пробел}
  Writeln(S)
end.

```

Отметим, что здесь мы не просто печатаем предложение, а приводим введенную строку к другому виду.

**Пример 10.35.** *С клавиатуры вводится предложение на английском языке (слова разделяются одним или несколькими пробелами, в конце предложения точка). Напечатать слова, начинающиеся с большой буквы.*

Можно эту задачу решить, например, так. Если первая буква в слове (в том числе и во всем предложении) большая, значит, надо это слово печатать, т. е. выводить символы, пока не встретится пробел или точка. Если первая буква в слове (т. е. после пробела) маленькая (или это вообще не буква, любой другой символ), это слово печатать не надо, т. е. пропускаем все символы до очередного пробела или до точки. Оба этих условия оформляем в виде циклов, и их мы тоже вставляем в цикл, заканчивающийся, когда встретится точка.

Обратите внимание, при таком алгоритме мы пропускаем «лишние» пробелы между словами так же, как и в предыдущем примере: каждый из них просматриваем в «одношаговом» цикле, считываем, но не печатаем.

```

Program Zagl_Buk;
Var C : Char;
Begin Writeln('Вводите предложение ');
      P:=True;
      Repeat
        Read(C);
        If (C>='A') and (C<='Z')
          Then Begin Repeat Write(C);
                     Read(C)
                     Until (C=' ') or (C='.');
                     Write(' ')
          End
        Else Repeat Read(C)
                  Until (C=' ') or (C='.');
      Until C='.'
End.

```

Эту задачу можно решить и по-другому. Например, можно запоминать предыдущий символ, а также ввести «флажок» — булевскую переменную и при различных их сочетаниях печатать или не печатать очередную букву.

А теперь со строками. Предложение сразу вводим в строку *S*. Будем выделять из него слова (а разделителем слов является пробел) и смотреть, подходит ли нам очередное слово. Для того чтобы работать со следующим словом предложения, выделенное слово удаляем и опять рассматриваем предложение. Так действуем до тех пор, пока в предложении не останется единственное, последнее, слово (об этом будет свидетельствовать то, что в предложении не будет пробелов). Это слово тоже надо проверить. Заметим, что выделяемые нами «слова» не всегда являются действительно словами. Например, если во введенном предложении слова разделены несколькими пробелами, в качестве очередного «слова» мы выделим пробел. В данной задаче это не страшно: пробел не начинается с большой буквы, мы его не напечатаем, однако бывают задачи, где к выделению слов из предложения надо подходить более аккуратно (кстати, при первом способе решения эта проблема легко обходится).

```

var S, SCopy : String;
    N       : Integer;
begin
  write('Введите предложение '); Readln(S);
  Repeat
    N:=Pos(' ', S);

```

```
If N>0 Then Begin SCopy:=Copy(S,1,N);    {Первое слово}
                S:=Copy(S,N+1, Length(S))  {Оставшаяся
                                                часть}
                End
            Else SCopy:=S; {В предложении осталось всего
                                                одно слово}
            If (SCopy[1]>='A') and (SCopy[1]<='Z')
                Then Write(SCopy);
Until N=0;
Writeln
End.
```

Рассмотрим в качестве примера разные способы решения вот такой задачи.

**Пример 10.36.** В базе данных некоторого склада имеются сведения о 50 товарах. Информация о каждом продукте представлена в отдельной строке в следующем формате: если продукт в данный момент на складе отсутствует, то записано только его название (символы без пробелов), если же он есть на складе, далее для пищевых продуктов следует вес в килограммах (целое число) и прочие сведения, а для непищевых после названия записана фирма-производитель (буквами), и, может быть, другие сведения. Все сведения в каждой строке отделены друг от друга ровно одним пробелом, заканчиваются все записи точкой. Найти общий вес пищевых продуктов, хранящихся на складе.

При решении этой задачи надо правильно выбрать нужные строки, касающиеся пищевых продуктов: это строки, где после нескольких символов идет пробел, затем — целое число, затем опять пробел и еще какая-то информация. Строки другого формата (в которых меньше двух пробелов, на втором месте стоит не число и т. п.) нам не нужны.

Один из способов решения: ввести строку, а далее разбираться, записаны ли в ней нужные сведения, выделять из нее число (вес), прибавлять его к общему весу продуктов.

Для строки введем строковую переменную *Stroka*, а для суммарного веса — *S*. Для ввода 50 строк воспользуемся оператором цикла.

В прочитанной строке сначала найдем пробел, воспользуемся встроенной функцией **Pos**, результат ее работы (позицию пробела) запишем в переменную *P*. Если  $P = 0$ , пробелов в строке нет. Значит, в ней находится только название товара, а самого товара на складе нет. С такой строкой дальше работать не будем.

Если же один пробел найден, удалим из строки ее содержимое до этого пробела (название товара) и поищем в ней второй пробел. Если его обнаружить не удалось, дальше со строкой работать также не надо. Если же пробел

нашелся, выделим часть строки до пробела (воспользуемся **Copy**), в ней должно содержаться число. Однако это число представлено в «строковом» виде, в виде набора символов-цифр. Для превращения его в «обычное» число воспользуемся процедурой **Val**. Если создать число из строки возможно (все символы «правильные»), получившееся число будет помещено в переменную *Int*, а значение *P* будет равно 0, иначе  $P > 0$  (в этом случае в этой переменной хранится номер позиции, на которой стоит «неправильный» символ, а значение *Int* не определено).

Остается лишь добавить получившееся число к общей сумме.

```

Var Stroka : String [100];
    S, I, P, Int : Integer;
Begin S:=0;
    For I:=1 to 50 do
        Begin Readln(Stroka);
            P:=Pos(' ', Stroka);
            If P>0 Then Begin
                Delete(Stroka, 1, P);
                P:=Pos(' ', Stroka);
                If P>0 Then Begin
                    Stroka:=Copy(Stroka, 1, P-1);
                    Val(Stroka, Int, P)
                    If P=0 Then S:=S+Int
                End
            End
        End;
        Writeln ('Сумма ', S)
    End.

```

А теперь второй способ — посмотрим, как решить эту задачу не только без использования стандартных строковых функций, но даже без строк.

Будем считывать строки посимвольно. Сначала «пропустим» первое слово, прочитаем символы до первого пробела (либо, если пробела в строке нет — до точки). Если строка до конца не считана, займемся ее дальнейшей обработкой: будем считывать символы до очередного пробела, проверять, являются ли они цифрой. Символ-цифру будем добавлять к числу, если же встретилась не цифра, работу надо прекратить. Получившееся число надо добавить к общему весу только в том случае, если мы вышли из цикла с  $H = 0$  (прочитан пробел после числа).

Обработка нужной части строки закончена, остальная информация нам не нужна, ее надо пропустить (процедура **ReadLn**).

```

Var H: Char;
    S, I, P, Int : Integer;
Begin S:=0;
    For I:=1 to 50 do

```

```
Begin
  Repeat
    Read(H);
  Until (H=' ') or (H='.' );
  If H<>'.' Then
    Begin P:=0; Int:=0;
      Repeat
        Read(H);
        If (H>='0') and (H<='9')
          Then Int:=Int*10+ord(H)-ord('0')
          Else P:=1
        Until P=1;
        If H=' ' Then S:=S+Int
      End;
    Readln
  End;
  Writeln(S)
End.
```

Трудно сказать, какой из способов лучше, тем более здесь программы по объему практически одинаковы. Заметим, что предложенные программы не во всех случаях работают совершенно одинаково. Стоит чуть уточнить формулировку задачи, и выбор способа решения будет уже очень важен. Например, если в формулировку добавить, что название продукта может быть более 255 символов, решать задачу первым способом нельзя (ведь длина строки ограничена), если же изменить формат вводимой строки, не требовать, чтобы она обязательно заканчивалась определенным символом, не пройдет решение вторым способом.

## Задачи 10.46–10.58. Символьные массивы, строки

- 10.46. Определить, не пользуясь строковыми функциями, входит ли данная подстрока в заданную строку.
- 10.47. Подсчитать, сколько раз входит данная подстрока в заданную строку. Решить эту задачу двумя способами: с использованием строковых функций и без них.
- 10.48. С клавиатуры вводится текст (последовательность символов). Проверить, является ли он правильным идентификатором.
- 10.49. Предложение называется палиндромом, если оно одинаково читается слева направо и справа налево. При этом пробелы между словами не учитываются, большие и маленькие буквы не различаются. (Пример: «А роза упала на лапу Азора».) Написать программу, которая проверяет, является ли введенное предложение палиндромом.

- 10.50. Вводится слово (последовательность маленьких латинских букв). Переставить в нем буквы так, чтобы сначала шли все гласные буквы, а затем все остальные.
- 10.51\*. Выписать из предложения все слова, в которых какая-либо буква встречается 2 раза (не обязательно подряд).
- 10.52. С клавиатуры вводится предложение (в одну строку), состоящее не менее чем из 4 слов. Слова разделены ровно одним пробелом. Является ли третье слово зеркальным (состоит из тех же букв в обратном порядке) для первого?
- 10.53. С клавиатуры вводится предложение (в одну строку), заканчивающееся точкой. Слова состоят из латинских букв и, если их несколько, разделены ровно одним пробелом. Напечатать слова, которые начинаются и заканчиваются на одну и ту же букву. Совет: каждое слово предложения надо сохранять в переменной строкового типа. Когда слово целиком будет введено, проверить, надо ли его печатать.
- 10.54. С клавиатуры вводятся строки, в каждой из которых написаны фамилия и имя (разделенные ровно одним пробелом). Количество строк известно. Напечатать фамилии всех людей с именем Иван.
- 10.55. С клавиатуры вводятся слова (последовательности латинских букв), каждое с новой строки, до тех пор, пока не будет введен символ «!». В скольких из них встречается буква «а»?
- 10.56. С клавиатуры вводятся 10 слов, каждое с новой строки (слова состоят из латинских букв). Найти то, которое стоит первым по алфавиту (наименьшее в смысле лексикографического порядка).
- 10.57. С клавиатуры вводится 20 слов, каждое с новой строки (в словах могут встречаться русские и латинские буквы — маленькие и большие, знак дефис). Напечатать сначала те из них, которые состоят только из больших латинских букв, а потом те, которые состоят только из маленьких.
- 10.58. Удалить из слова (состоящего из маленьких латинских букв) все гласные буквы.

## Матрицы

Вспомним описание массива: **Array [тип индекса] of [тип компонент]**. Мы говорили, что тип компонент может быть любым, но рассмотрели примеры массивов только с компонентами четырех стандартных типов. Оказывается, тип компонент может быть и нестандартным, в том числе массивом. То есть возможна работа с массивом массивов.

Это не так страшно, как может показаться на первый взгляд, в жизни мы с массивами массивов встречаемся и прекрасно в них разбираемся. Мы находим свое место в театре: а ведь зал — это массив рядов, а каждый

ряд — массив кресел; сначала нужно найти нужный ряд, а потом место в нем. На железнодорожном вокзале мы имеем дело с массивом поездов (поезд — массив из вагонов, вагон — массив из мест): находим нужный поезд и потом свое место в нем.

Вообще массивом массивов является любая таблица: ведь она состоит из нескольких строк, а каждая строка — уже из отдельных компонент. В математике такая таблица (при условии, что все компоненты одного типа) называется двумерным массивом или матрицей.

Пусть мы выставили героям мультфильма оценки по разным параметрам их внешнего вида:

	Герой фильма	1. Шерсть (волосы)	2. Хвост	3. Усы
1	Дядя Федор	4	2	2
2	Почтальон Печкин	3	2	3
3	Кот Матроскин	5	5	5
4	Пес Шарик	5	4	4

Часть таблицы, обведенная жирной линией, состоит из оценок, компоненты в ней все одного типа. Ее можно представить в виде такой матрицы.

```
Type Ocenka=2..5; {Ограниченный тип, оценки - целые  
                                     числа от 2 до 5}  
Stroka = Array [1..3] of Ocenka; {В каждой строке  
                                     3 оценки}  
Var Tabl = Array[1..4] of Stroka; {Таблица состоит  
                                     из 4 строк}
```

Имея такое описание, мы можем объявить, например, переменную **Var S: Stroka;** и работать таким образом:

*Tabl*[3] — это третья строка таблицы (оценки Матроскина), можно написать **S:=Tabl [3]**, так как это массивы одного типа. Напомним, что никакие другие действия над массивами, кроме присваивания, недопустимы.

*S*[1] — при сделанном выше присваивании это первая оценка Матроскина («шерстистость»). Эта же оценка является значением переменной *Tabl*[3][1]. Понимать эту запись следует так: в таблице взять 3-ю строку, а в ней 1-ю компоненту. Чаще то же самое записывают следующим образом: *Tabl*[3, 1]. Порядок индексов здесь очень важен!

*Tabl*[*M*, *K*] — это оценка, которая стоит в *M*-й строке *K*-м столбце.

Напишем несколько фрагментов программ для данной таблицы (с приведенным выше описанием переменных). Для работы с таблицей нам понадобятся счетчики *J*, *I* — целые.



Сначала напечатаем таблицу (имена персонажей, названия столбцов пока выводить на экран не будем — пройдет немного времени, мы и это сделаем).

```
For I:=1 to 4 do {цикл по количеству персонажей -
                                                    по строкам}
Begin  For J:=1 to 3 do {печать одной строки}
        Write (Tabl[I,J]:4);
        Writeln {после очередной строки переход на следующую
                                                    строку}
End
```

Подсчитаем, сколько во всей таблице двоек:

```
K:=0;
For I:=1 to 4 do {цикл по количеству персонажей -
                                                    по строкам}
    For J:=1 to 3 do {цикл по оценкам - по содержимому
                                                    строки}
        If Tabl[I,J]=2 Then K:=K+1;
Writeln('количество двоек в таблице ', K)
```

Найдем среднюю оценку внешности каждого героя:

```
For I:=1 to 4 do {цикл по количеству персонажей}
Begin Sum:=0;
    For J:=1 to 3 do {цикл по оценкам}
        Sum:=Sum+Tabl[I,J]; {найдена сумма оценок I-го героя}
    Writeln('средняя оценка ', I, 'героя ', Sum/4:4:1)
End
```

Выпишем номера персонажей, которые по каждому параметру имеют наивысшую оценку — «5» (у нас для первого параметра «шерсть» это Матроскин и Шарик, для второго «хвост» — Матроскин, и для третьего тоже Матроскин).

```
For I:=1 to 3 do {цикл по параметрам, по столбцам }
Begin Writeln ('Параметр номер ', I);
    For J:=1 to 4 do {цикл по количеству персонажей
                                                    (строка) }
        If Tabl[J,I]=5 Then Write(J:4)
End
```

Обратите внимание, в первом фрагменте у нас компонента таблицы *Tabl[I, J]*, а во втором индексы стоят наоборот: *Tabl[J, I]*. Это потому, что в первом фрагменте у нас внешний цикл (параметр цикла *I*) — по строкам, во внутреннем цикле параметр *J*, этот цикл — по элементам строки, т. е. по столбцам таблицы. *I* означает номер строки, *J* — номер столбца. Во втором фрагменте все наоборот: внешний цикл по столбцам, а внутренний по строкам.

Рассмотрев немного примеров, вернемся к теории.

Итак, в Паскале возможны многомерные массивы (в принципе размерность массива никакими правилами не ограничена, но человеку трудно представить размерность больше 3, поэтому чаще всего работают с 1-, 2- и иногда с 3-мерными массивами). Описать многомерный массив можно двумя способами (покажем на примере двумерного).

1. Сначала описать тип «массив из некоторых простых компонент», а затем тип (или переменную) «массив из описанных массивов»:

```
Type ElMas=Array [<тип индекса1>] of <простой тип>;  
      DMas=Array [<тип индекса2>] of ElMas;
```

В этом случае могут быть описаны переменные **Var S:ElMas; Komp:Dmas**. К переменной типа *ElMas* надо обращаться с одним индексом *S[<индексное выражение>]*, к переменной типа *Dmas* можно обращаться с двумя индексами *Komp[J1, J2]* — тогда это компонента массива простого типа, и с одним индексом *Komp[J]* — тогда это «внутренний» массив типа *Elmas*.

2. Описать многомерный массив в одном описании

```
Type DMas=Array [<тип индекса1>] of Array [<тип индекса2>]  
                                         of <простой тип>;
```

или

```
Type DMas=Array [<тип индекса1>, <тип индекса2>]  
               of <простой тип>;
```

К описанной таким образом переменной можно обращаться только с двумя индексами, чтобы работать с компонентой простого типа.

Второй способ описания многомерных массивов на письме выглядит несколько короче, в памяти компьютера это описание практически ничем не отличается от первого. Однако, описывая массив вторым способом, мы лишаемся возможности работать с «внутренним» массивом, со строками таблицы целиком, поэтому первый способ предпочтительнее.

## Решение задач с матрицами

При решении задач следует помнить, что исходные данные для программы — это не таблицы, не массивы, а последовательность некоторых значений (чисел, символов). Элементы этой последовательности могут быть прочитаны один за другим, например, с клавиатуры с помощью процедуры ввода **Read**. Каким именно переменным (с индексами, без индексов, как будут расставлены индексы в переменных) будут присвоены эти значения, полностью зависит от того, как составлена программа. Скажем, матрицу можно вводить по строкам (сначала все элементы первой строки, с первого по последний, потом все элементы второй и т. д.), можно по столбцам, а можно и каким-то другим известным только программисту способом.

Большинство задач с матрицами, встречающихся в школьном курсе, обычно принадлежат к одной из следующих групп.

1. Надо произвести какие-то действия со всей матрицей. Например, найти сумму, произведение элементов всей матрицы, найти минимальный, максимальный элемент, определить его позицию.

Решение таких задач мало чем отличается от решения задач с одномерными массивами. Вначале надо произвести некоторые предварительные действия (присваивание переменным начальных значений), потом с помощью двух операторов цикла, вложенных друг в друга, просмотреть все элементы матрицы. По окончании цикла выдать ответ.

2. Некоторые действия надо произвести над строками матрицы (найти сумму, произведение элементов каждой строки, максимальный элемент в строке, напечатать строки, обладающие определенными свойствами).

При решении задач такого типа тоже будет использовано два оператора цикла. Один — внешний — будет отвечать за работу с каждой строкой, а внутри него надо организовать действия с одной строкой совершенно так же, как это делается с одномерным массивом (которым строка и является). Конечно, будет использован оператор цикла для перебора элементов строки. При этом надо внимательно следить, чтобы действия, которые производятся над строкой, не «выскочили» из внешнего цикла (применить операторные скобки **Begin–End**).

Также полезно помнить, что, если матрица описана так, что в описании использован тип «строка», появляются дополнительные возможности по работе со строками целиком.

3. То же самое, что и в п. 2, только действия надо произвести над столбцами матрицы. Задача решается аналогично, только никаких дополнительных возможностей по работе со столбцами в целом нет, приходится все элементы рассматривать по очереди, по одному.

Конечно, все многообразие задач с матрицами этим не ограничивается. Чтобы сначала разобраться со стандартными задачами, сделаем следующее.

***Пример 10.37.** Найти минимальный элемент во всей матрице, в каждой строке и в каждом столбце. Для каждого ответа выписать сам элемент и его позицию (номер строки и столбца).*

```
Program Minimum;  
Const M=3; N=4;  
Type Stroka=Array [1..N] of Real;  
Var A : Array[1..M] of Stroka;  
    I,J : Integer;  
    Min : Real;  
    Str, Stlb : Integer; {Позиция минимального элемента:  
                        номер строки и столбца}
```

```
Begin
  Writeln('Вводите элементы матрицы ');
{Задача типа 1: работа со всей матрицей}
  For I:=1 to M do
    For J:=1 to N do
      Begin Write('Элемент [' , I , ' , ' , J , ']=');
        Readln(A[I,J])
      End;
  {Напечатаем получившуюся матрицу}
  {Задача типа 2 - печать строк}
  For I:=1 to M do
    Begin For J:=1 to N do
      Write(A[I,J]:6:1);
    Writeln
  End;
  {Найти минимальный элемент во всей матрице - задача
                                                    типа 1}
  Min:=A[1,1]; Str:=1; Stlb:=1;
  For I:=1 to M do
    For J:=1 to N do
      If A[I,J]<Min Then Begin Min:=A[I,J];
        Str:=I; Stlb:=J
      End;
  Writeln('Минимальный элемент всей матрицы ' , Min:5:1);
  Writeln('стоит в ' ,Str, ' строке, ' , Stlb, ' столбце');
  {Минимальный элемент в каждой строке - задача типа 2}
  For I:=1 to M do
    Begin Write('строка ' , I);
      Min:=A[I,1]; Stlb:=1;
      For J:=2 to N do
        If A[I,J]<Min Then Begin Min:=A[I,J];
          Stlb:=J
        End;
      Writeln(' минимум', Min:4:1, ' стоит в ' ,Stlb,
        'столбце')
    End;
  {Минимальный элемент в каждом столбце - задача типа 3}
  For I:=1 to N do
    Begin Write('столбец ' , I);
      Min:=A[1,I]; Str:=1;
      For J:=2 to M do
        If A[J,I]<Min Then Begin Min:=A[J,I];
          Str:=J
        End;
      Writeln(' минимум', Min:4:1, ' стоит в ' ,Str,
        'строке')
    End;
End;
End.
```

При решении этой задачи нам не пришлось воспользоваться тем, что мы описали отдельный тип «строка матрицы». Это может пригодиться, например, в такой задаче.

**Пример 10.38.** *Поменять местами строки матрицы так, чтобы минимальный элемент матрицы оказался в первой строке.*

Мы нашли позицию минимального элемента, он стоит в строке с номером *Str*. Надо эту строку поменять местами с первой. Для этой операции нам понадобится дополнительный массив, в котором придется сохранить одну из строк. Опишем его так:

```
Var Dop : Stroka;
```

Этот массив и строки матрицы — одного типа, значит, присваивание массивов можно производить не поэлементно, а целиком:

```
Dop:=A[1];
A[1]:=A[Str];
A[Str]:=Dop
```

Поработаем теперь с символьной матрицей.

**Пример 10.39.** *Описать и ввести представленную матрицу. Напечатать в строчку слова, которые получаются, если читать их в столбцах (сверху вниз).*

Л	О	Г
Е	Д	А
С	А	Д

```
Var MS:Array[1..3, 1..3] of Char;
    K,L : Integer;
Begin Writeln('Вводите слова по буквам');
    For K:=1 to 3 do
        Begin Writeln(K, '-е слово ');
            For L:=1 to 3 do
                Begin Write(L, '-я буква ');
                    Readln(MS[K,L])
                End
            End;
        For L:=1 to 3 do
            Begin For K:=1 to 3 do
                Write(MS[K,L]);
            Writeln
            End
        End.
End.
```

В этом решении мы объявили матрицу из символов (значит, пользоваться целиком строками нельзя, такой тип в задаче не объявлен). Слова

вводим по одной букве. В данном случае каждый элемент матрицы вводится на отдельной строке со своей собственной подсказкой (мы так уже делали в предыдущей задаче). Способ достаточно долгий, зато гарантирует правильный ввод: видно, какую именно компоненту матрицы надо ввести в данный момент.

Можно выбрать способ ввода короче: печатать при вводе сразу целую строку матрицы. Вводятся компоненты все равно будут по одной (в цикле), но человек будет на клавиатуре набирать слово целиком и только после этого нажимать клавишу ввода. Обратите внимание, нажатие клавиши ввода тоже отражено в программе: после ввода очередного слова стоит оператор **ReadLn**, который «проглатывает» нажатие клавиши ввода, иначе она тоже будет понята как очередной символ.

Подобным образом можно вводить не только символы, но и числа, только их в строке надо разделять пробелами, и **ReadLn** при таком вводе не обязателен.

```
Writeln('Вводите слова');
For K:=1 to 3 do
Begin Writeln(K, '-е слово  ');
      For L:=1 to 3 do
        Read(MS[K,L]);
      Readln
End;
```

А нельзя ли работать со словами нашей матрицы как со строками (тип **String**)? Можно. При этом к элементам матрицы допустимо обращаться двумя способами: по-старому, указывая два индекса через запятую в квадратных скобках, и по-новому, указывая отдельно номер строки и номер буквы в ней. К тому же появляется возможность работать сразу с целой строкой. Описание и ввод будут выглядеть так:

```
Type Str=String[3];
Var MS:Array[1..3] of Str;
    K,L : Integer;
...
Writeln('Вводите слова');
For K:=1 to 3 do
  Begin Writeln(K, '-е слово  ');
        Readln(MS[K]);
      End
```

Интересно, что вторая часть программы (вывод столбцов матрицы в строчку, кстати, в математике эта процедура называется транспонированием матрицы) совершенно не зависит от способа ввода символов. Как бы ни были введены строки, выводить элементы матрицы надо по одному, вывести сразу целый столбец не удастся, такого типа данных у нас нет.

И вот еще на что стоит обратить внимание в этой задаче. И при вводе (давайте рассматривать первый вариант), и при выводе используется одна и та же переменная  $MS[K, L]$ , но при вводе элементы читаются по строкам, а выводятся — по столбцам. Мы здесь еще раз хотим подчеркнуть, что это никак не зависит от имени переменной, которая использована для индексации, а только от операторов программы, от того, в каком порядке и с какими переменными использованы циклы.

**Пример 10.40.** *С клавиатуры вводится целое  $N > 1$ . Заполнить и напечатать единичную матрицу размера  $N * N$ . (Пример единичной матрицы  $3 * 3$  приведен ниже.)*

1	0	0
0	1	0
0	0	1

Единичной матрицей называется квадратная матрица, у которой на главной диагонали стоят единицы, а все остальные элементы — нули. Главная диагональ — это диагональ, которая идет от левого верхнего угла к правому нижнему (другая диагональ называется побочной). Заметим, что обычно понятие главной диагонали применимо к квадратным матрицам, как в нашем случае.

Отметим чисто учебное назначение данной задачи: чтобы напечатать заданную таблицу чисел, использовать матрицу, тип «массив», не нужно. Мы покажем, как решать такую задачу с матрицами, чтобы научиться заполнять их числами нужным образом.

Задачу можно решить несколькими способами. Выберем здесь такой: сначала заполним нулями всю матрицу (точнее весь «квадратик») — для этого понадобятся два оператора цикла, вложенных друг в друга. Затем на диагонали проставим единицы, это делается с помощью одного оператора цикла. Остается только напечатать ответ.

```

Var Matr:Array[1..10, 1..10] of 0..1;
  I, J : Integer;
Begin Write('Размер матрицы '); Readln(N);
  For I:=1 to N do {Заполнение нулями всей матрицы}
    For J:=1 to N do
      M[I, J]:=0;
  For I:=1 to N do {Заполнение главной диагонали
                                                              единицами}
    M[I, I]:=1;
  For I:=1 to N do {Печать матрицы}
  Begin For J:=1 to N do
    Write(M[I, J]:2);
    Writeln
  End
End.
```

Решение очень простое, но мы здесь делаем лишние действия: на диагонали ставим сначала нули, а потом вместо них — единицы. Можно этого не делать, ставить нужные числа сразу. Например, можно использовать условный оператор, в котором проверять, не равны ли индексы у рассматриваемого элемента, и в этом случае ставить не ноль, а единицу.

Можно действовать еще сложнее: каждую строку заполнять в 3 этапа: сначала нули, которые стоят до единицы, потом сама единица, потом нули, которые стоят после единицы. Количество нулей на каждом этапе зависит от номера строки и легко вычисляется.

## Задачи 10.59–10.71. Работа с матрицей

- 10.59. Задана символьная матрица. Подсчитать, сколько раз в нее входит символ «!». Выписать, сколько раз этот символ входит в каждую строку и в каждый столбец.
- 10.60. Выписать номера столбцов целочисленной матрицы, сумма чисел в которых больше 100.
- 10.61. Выписать номера столбцов и строк целочисленной квадратной матрицы, в которых число 1 встречается ровно 1 раз.
- 10.62. Есть ли в матрице вещественных чисел строка, в которой все числа отрицательные?
- 10.63. В целочисленной квадратной матрице найти сумму элементов, стоящих над главной диагональю.
- 10.64. В матрице вещественных чисел определить столбец, сумма элементов в котором наибольшая, и вывести на экран все элементы этого столбца.
- 10.65. В целочисленной квадратной матрице переставить элементы в столбцах так, чтобы минимум каждого столбца оказался на главной диагонали.
- 10.66. Есть ли в символьной матрице строка или столбец, состоящие из одинаковых символов?
- 10.67. Задана целочисленная матрица, состоящая из 0 и 1. Посчитать, сколько у каждого элемента соседей (единиц вокруг него).
- 10.68. Задана целочисленная квадратная матрица. Является ли она магическим квадратом (суммы во всех столбцах и строках равны)?
- 10.69. Сравнить две символьные матрицы (одинаковы они или нет).
- 10.70\*. Есть ли в данной матрице одинаковые строки или столбцы?
- 10.71\*. В целочисленной матрице поменять местами строки так, чтобы они оказались отсортированными по возрастанию сумм их элементов.



# Глава 11

## Процедуры и функции

---

Мы познакомились практически со всеми операторами языка Паскаль. Как видите, их очень немного, и выполняют они самые простые, элементарные действия. Хотелось бы иметь возможность увеличить «способности» компьютера, «научить» его выполнять и некоторые более сложные действия. В программировании расширение возможностей языка достигается за счет использования аппарата подпрограмм.

Часто приходится сталкиваться с ситуацией, когда при решении основной задачи требуется несколько раз решить одну и ту же логически завершённую подзадачу (возможно, с разными входными значениями). Решение такой подзадачи описывается в виде подпрограммы. Использование подпрограммы в основной программе (решающей основную задачу) называется **обращением к подпрограмме** или **вызовом подпрограммы**.

В описании подпрограммы перечисляются действия, которые должны быть сделаны (например, вычисления, ввод, вывод), указывается, какие из объектов могут быть изменены в момент обращения, как задать их значения. Такие объекты описываются в виде **формальных параметров**. В нужный момент подпрограмма вызывается, указываются входные значения — **фактические параметры**, с которыми будут выполняться действия, и происходит выполнение всего набора описанных в ней действий. Благодаря этому текст самой программы разбивается на смысловые части, становится понятнее, а иногда и существенно короче.

В языке Паскаль существуют две разновидности подпрограмм: **процедуры** и **функции**. Они бывают **стандартными** и **нестандартными**. Стандартные процедуры и функции входят в состав языка, ими можно пользоваться в любой программе без всякой «предварительной подготовки». Нестандартные процедуры и функции программист должен описать сам.

Прежде чем учиться писать процедуры и функции, давайте вспомним, что мы о них уже знаем. Обобщим свой опыт использования стандартных процедур и функций, ведь мы ими неоднократно пользовались, писали в своих программах обращения к ним. Итак, что же мы знаем об использовании подпрограмм в Паскале?

Наиболее часто используемые нами стандартные процедуры — это процедуры ввода **Read (Readln)** и вывода **Write (Writeln)**, знаем мы также процедуру выхода из цикла **Break**. Вспомним, какими стандартными

функциями мы пользовались: **Abs**, **Sqr**, **Sqrt**, **Sin**, **Cos**, **Ord**, **Chr**, **Odd** и т. д. Что у них общего?

И у процедуры, и у функции должно быть уникальное, только ей присущее имя, по которому мы к ней обращаемся (вызываем). В обращении к подпрограмме после имени в скобках могут стоять переменные, константы, выражения. Как было сказано выше, они являются фактическими параметрами — так задаются конкретные значения, с которыми будет работать подпрограмма в данный момент. Например, в операторе **Y:=Sqr(3)** функция **Sqr** будет работать с параметром 3, вычислит значение  $3^2$ , которое и будет присвоено **Y**.

Бывают процедуры и функции без параметров (например, **WriteLn**, **Break**), тогда скобки не ставятся. В процедурах ввода и вывода количество параметров — любое (по этому признаку они уникальны!), во всех других известных нам пока процедурах и функциях — строго определенное (часто функция зависит от одного параметра). При обращении к подпрограммам надо следить за соответствием типов параметров: например, параметр процедуры ввода **Read** не может быть массивом, у функции **Odd** параметр может быть только один, причем он должен быть обязательно целого типа и т. п.

Какое различие между процедурами и функциями мы могли заметить? Функция обычно вычисляет какое-то значение (часто говорят — «возвращает значение»); можно говорить о значении функции. Процедура выполняет какое-то действие (например, выводит что-то на экран), она может менять значения переменных, но вот у самой процедуры никакого значения нет.

С процедурами и функциями мы работаем по-разному. Для вызова процедуры мы в нужном месте программы просто выписываем ее имя (с параметрами, если они нужны). Получившаяся конструкция называется «оператор процедуры». (Вспомним, мы говорили, что название «оператор **Read**» — не вполне правильное, теперь будем знать, что правильно надо говорить «оператор процедуры **Read**».)

Обращение к функции не выполняется как самостоятельный оператор, для вызова функции ее имя используется при построении выражений, как будто это имя переменной (различие в том, что после имени функции могут стоять в скобках параметры). Полученные выражения используются в разных операторах. Имя функции может встретиться в правой части оператора присваивания: **A:=Abs(X)**, в условном операторе: **If Odd(K) Then...**, в процедуре вывода: **Write(Ord(C))** — всего не перечислишь.

В Паскале кроме стандартных подпрограмм программисту можно использовать и свои собственные. Правила обращения к таким подпрограммам такие же, как и к стандартным. Разница состоит в том, что эти «личные» подпрограммы, в отличие от стандартных, должны быть предварительно описаны. Программист сам придумывает имя своей подпрограмме, определяет, какие у нее будут параметры, описывает действия, которые должны производиться при вызове данной подпрограммы.

## Описание процедур и функций

Описываются процедуры и функции в разделе описаний, обычно после описания типов и констант. Дело в том, что в Паскале действует правило: в каждом описании можно пользоваться тем, что описано раньше, а в процедурах и функциях иногда приходится пользоваться описанными ранее типами и константами.

Описание процедуры/функции состоит из заголовка и тела. Заголовок — это:

- служебное слово **Procedure** или **Function**;
- имя процедуры/функции (подчиняется правилам написания идентификаторов Паскаля);
- параметры (если они есть) — в скобках, с указанием типа;
- для функции указывается тип ее результата (значения) после двоеточия. Заметим здесь, что тип результата функции должен быть простым типом (в частности, не может быть массивом, файлом, записью).

Приведем примеры заголовков:

```
Function Oh(a:integer) : LongInt;  
Procedure Ah(a1:integer; a2: Char);
```

Параметры, указываемые в заголовке описания подпрограммы, называются формальными. Синтаксически формальные параметры являются идентификаторами, они не имеют никаких значений. Значения им передадут фактические параметры, подставленные на их место при обращении к подпрограмме. Дело в том, что описание подпрограммы носит чисто декларативный характер — действия в нем только описываются, выполняться они будут, когда подпрограмма будет вызвана из основной программы. Чтобы пояснить, с какими параметрами что надо делать (например, первый надо на единицу увеличить, а второй возвести в квадрат), параметры и обозначаются формально некоторыми идентификаторами.

Тело подпрограммы в общем-то по синтаксису ничем не отличается от обычной программы. Только в конце после **End** точка не ставится (обычно ставится точка с запятой, так как дальше идут другие описания или основная программа, а все эти части программы на Паскале отделяются друг от друга точкой с запятой).

В теле процедуры/функции может быть, как и в теле программы, раздел описаний: описания переменных и констант, описания других процедур и функций. И, как и в программе, после раздела описаний идет раздел операторов, заключенный в операторные скобки **Begin–End**. Важная особенность для функции: в ее разделе операторов должен присутствовать хотя бы один выполняемый оператор присваивания, где в левой части стоит имя

функции, т. е. имени функции должно быть что-то присвоено (так как функция всегда имеет какое-то значение).

Никаких ограничений на количество описаний, операторов, на вид операторов в процедурах и функциях не накладывается, они могут быть любые. В частности, внутри одной функции или процедуры может быть обращение к другим процедурам/функциям. Важно, чтобы они были описаны к моменту обращения.

Обычно стараются, чтобы описываемые процедуры/функции были «специализированы», выполняли какое-либо одно определенное действие. То есть, например, никакими правилами языка Паскаль не запрещается, чтобы в теле функции присутствовали операторы ввода/вывода, однако обычно в функции выполняют только вычисления, а ввод данных и вывод результата производят в основной программе или в другой процедуре/функции.

## Обращение к подпрограмме. Фактические параметры

Рассмотрим, что происходит при обращении к подпрограмме. Разберем следующий пример.

*Пример 11.1. Как будет работать следующая программа? Что будет выведено на экран?*

```
Function Kub(X:integer):integer; {Описание функции}
Begin {Тело функции}
    Kub:=X*X*X
End;
Procedure PrintKub (N,K:Integer); {Описание процедуры}
{Тело процедуры}
Var I:Integer; {Описание переменной внутри процедуры}
Begin
    For I:=N to K do
        Writeln(I:3, Kub(I):10);
    Writeln;
End;
Begin {Основная программа}
    Writeln(Kub (-3)); {Обращение к функции
                        с параметром -3}
    PrintKub(1,2); {Обращение к процедуре
                   с параметрами 1 и 2}
    PrintKub(5,10); {Обращение к процедуре
                    с параметрами 5 и 10}
End.
```

Сначала описана функция *Kub*. У нее один параметр — целое число. В описании сказано, что надо это число возвести в куб и результат присвоить имени функции, т. е. можно сказать, что функция возвращает куб своего аргумента.

Далее следует описание процедуры *PrintKub*. У нее два параметра — целые числа. В теле этой процедуры есть свое собственное описание — описана целая переменная *I*.

Что должна делать эта процедура? В ее теле есть цикл. Сколько раз он будет работать? Это определяется параметрами. Количество шагов цикла будет рассчитываться, когда при обращении к процедуре будут указаны соответствующие фактические параметры. В теле цикла есть обращение к только что описанной функции *Kub*. То есть в описании процедуры сказано, что следует напечатать кубы нескольких чисел. Все вычисления и сама печать будут производиться, только когда процедура будет вызвана из основной программы.

В основной программе мы обращаемся к функции с параметром  $-3$ . Число  $-3$  является фактическим параметром, именно с ним будет работать функция, именно оно в нашем случае будет возводиться в куб. В результате такого обращения к функции будут выполнены вычисления, описанные в теле функции:  $(-3)*(-3)*(-3)$ . Получившееся значение  $(-27)$  будет присвоено имени функции *Kub*. В основной программе это число напечатается.

Далее идет обращение к процедуре *PrintKub* с фактическими параметрами 1 и 2. Формальный параметр *N* получил значение 1, а *K* — значение 2 (можно сказать, что произошли присваивания **N:=1**; **K:=2**). Теперь будут выполняться действия, описанные в процедуре: сначала произойдет обращение к функции *Kub* с параметром 1 (*X* примет значение 1), вычислится  $1*1*1$ , полученное значение напечатается. Потом *X* получит значение 2, выполнятся действия  $2*2*2$ , результат присвоится имени функции *Kub* и напечатается.

Здесь мы можем видеть, что очень важно соблюдать правильный порядок параметров. Обращение к процедуре с теми же параметрами, но в другом порядке: **PrintKub(2,1)** — приведет к печати лишь пустой строки. Цикл в теле процедуры в этом случае работать не будет, так как можно считать, что выполнятся действия **N:=2**; **K:=1**.

Сразу заметим, что при вызове подпрограммы важен не только порядок параметров, но и их количество, и соответствие типов. В самом деле, понятно, что в операторе процедуры **PrintKub(1)** допущена ошибка: неизвестно, чему равен второй параметр. Неправильными будут и обращения к нашим подпрограммам вида: **Y:=Kub(' '); PrintKub(1.2, 5.7)**. С такими параметрами подпрограммы не смогут работать: в первом случае потому, что символы нельзя перемножать, а во втором потому, что

в заголовке цикла **For** нельзя использовать вещественные числа. Ошибки такого типа отслеживает транслятор (для того чтобы он мог это делать, мы и описываем формальные параметры в заголовке процедуры, указываем их тип).

Вернемся к нашей программе. В ней мы видим еще одно обращение к процедуре. Теперь  $N$  становится равным 5, а  $K$  — 10, и обращение к функции *Kib* с разными параметрами происходит уже 6 раз.

Мы здесь сказали далеко не все, что следовало бы знать о параметрах для того, чтобы правильно описывать процедуры и функции. В дальнейшем по мере решения задач будем уточнять и пополнять наши знания.

## Принцип локализации

Мы говорили, что в теле подпрограммы могут быть описания констант, типов, переменных, других подпрограмм. Все эти описания являются **локальными**. Это означает, что они действуют только внутри самой подпрограммы. Вне подпрограммы описанные в ней переменные, типы данных и константы не существуют. Точнее, недоступны.

Описания, приведенные в основной программе, являются **глобальными**, т. е. их действие распространяется на весь текст, записанный ниже.

*Пример 11.2.* Как используются переменные в следующей программе? Что будет напечатано?

```
Var A : Integer; {Глобальная переменная}
Procedure P1(N:Integer); {Формальный параметр}
  Var I:Integer; {Локальная переменная}
  Begin For I:=1 to N Do WriteLn(A)
  {В теле этой процедуры можно использовать:
                                глобальную переменную A,
                                локальную переменную I,
                                формальный параметр N}
End;
{Основная программа} Begin {Здесь можно использовать
                             только глобальную переменную A }
  A:=9;
  P1(5); {Обращение к процедуре
        с фактическим параметром 5}
End.
```

После обращения к процедуре формальный параметр  $N$  заменяется на фактическое значение 5, и в результате работы процедуры (и всей программы) 5 раз напечатается число 9 (значение глобальной переменной).

А как же будет работать программа, если в ней имеются одинаково названные глобальные и локальные переменные?

**Пример 11.3.** Как меняется значение переменной *A* в этой программе? Что будет напечатано?

```
Var A:Integer; {описание глобальной переменной}  
Procedure Oy; {описание процедуры без параметров}  
  Var A:Integer; {описание локальной переменной}  
  Begin  
    A:=2; Write('*', A) {работа с локальной переменной}  
  End;  
Begin {Основная программа}  
  A:=5;  
  Oy; {обращение к процедуре}  
  Writeln('#',A) {работа с глобальной переменной}  
End.
```

Программа начинает свою работу с того, что глобальной переменной присваивается значение 5. Затем происходит обращение к процедуре, и в ней переменной *A* присваивается 2. Но это локальная переменная, отличная от глобальной (они в памяти компьютера хранятся в разных местах), поэтому присваивание двойки значение глобальной переменной не меняет. Затем печатается значение 2. После работы процедуры печатается значение глобальной переменной, т. е. 5. В результате программа напечатает: \*2#5.

Таким образом, если имеются глобальные и локальные переменные с одинаковыми именами, в подпрограмме работа ведется с локальной переменной, а в основной программе — с глобальной.

Этот принцип, называемый **принципом локализации**, относится не только к описаниям переменных, а ко всем описаниям, имеющимся в программе. Он позволяет при написании очередной подпрограммы не задумываться о том, можно ли в ней использовать объекты с уже задействованными в вызывающем блоке именами. Таким образом, в одной программе в разных процедурах/функциях могут быть описаны объекты (типы данных, константы, переменные) с одинаковыми именами, но «мешать» они друг другу не будут, в каждой процедуре/функции будут действовать ее локальные описания.

Из принципа локализации также следует, что любая процедура/функция может пользоваться глобальными объектами (типами данных, константами, переменными, другими подпрограммами), описания которых размещены выше ее, в основном разделе описаний.

## Задачи 11.1–11.3. Вызов процедуры и функции

11.1. Пусть описана функция  $KVT(x)$ , которая вычисляет значение квадратного трехчлена  $x^2 + 2x - 4$ :

```
Function KVT (X:Real) :Real;  
Begin  
    KVT:=x*x+2*x-4  
End;
```

Могут ли в основной программе встретиться следующие операторы? Если оператор правильный, посчитайте, что будет результатом его работы, если неправильный, объясните почему.

- a) `Writeln(KVT(3));`
- b) `KVT:=2;`
- c) `KVT(X):=2;`
- d) `X:=KVT(2);`
- e) `If KVT(-10)>0 Then V:=0 else V:=-5;`
- f) `For I:=-1 to 1 do Write(KVT(I));`
- g) `For I:=1 to KVT(2) do Write(I);`

Какой оператор надо написать, чтобы вывести наименьшее значение трехчлена?

11.2. Описанная ниже функция по заданной скорости и времени вычисляет пройденный путь:

```
Function Track (V,T:real) :Real;  
Begin  
    Track:=V*T  
End;
```

Какой оператор надо написать в основной программе, чтобы

- a) присвоить переменной  $S$  пройденное за 2 ч расстояние при скорости 30 км/ч;
- b) напечатать, какое транспортное средство преодолело большее расстояние: катер шел 5 ч со скоростью 12,3 км/ч, велосипедист ехал 3 ч со скоростью 22,8 км/ч?

Используя описанную функцию, напишите программу для решения следующей задачи: турист часть пути проехал на автомобиле, часть на поезде и остаток прошел пешком. Скорости на всех отрезках пути и затраченное время вводятся с клавиатуры. Выяснить, какой путь преодолел турист.



11.3. Описанная ниже процедура печатает площадь полной поверхности и объем цилиндра, если ей задать высоту и радиус этого цилиндра:

```
Procedure Cll (R,H:Real);  
Begin Writeln('Площадь поверхности =', 2*Pi*R*(H+R):8:2);  
      Writeln('Объем цилиндра =', Pi*R*R*H)  
End;
```

Напишите программу, в которой используется данная процедура. В программе вводятся радиус и высота цилиндра и печатаются площадь поверхности и объем данного цилиндра, а также цилиндра с радиусом на 1 меньше введенного.

## Работа с процедурами

### Процедуры без параметров

Когда мы выполняем подряд несколько программ в Паскаль-системе, результаты их выполнения остаются на экране. Часто это мешает, хочется, чтобы на экране находился только результат решения последней задачи.

*Пример 11.4. Написать процедуру очистки экрана.*

```
Procedure Clear;  
  Var I:Integer;  
Begin  
  For I:=1 to 25 Do  
    Writeln  
End;
```

Теперь, если в любую программу в раздел описаний поместить описание этой процедуры, можно вызывать процедуру *Clear* для очистки экрана. Например, это можно делать в самом начале программы. Отметим еще раз, что использованная в описании процедуры переменная *I* является локальной, существует только в процедуре. В программе, других процедурах можно использовать переменную с таким же именем (для этого ее надо описать), они не будут мешать друг другу.

Цикл в нашей процедуре работает 25 раз потому, что на экране в Паскаль-среде 25 строк и печать 25 пустых строк как раз и приводит к полной очистке экрана. В ABC-Паскале процедура тоже будет работать, с ее помощью можно разделять порции выдаваемой информации (можно делать поменьше строчек).

Пусть у нас в программе печатается несколько результатов, и мы хотим на печати отделить эти результаты друг от друга, например, строкой из звездочек.

*Пример 11.5, а. Написать процедуру, печатающую на экране строку из звездочек.*

Программа очень похожа на предыдущую: также понадобится локальная переменная для счетчика цикла, в теле процедуры — цикл, печатающий 80 (столько символов помещается на экране в Турбо Паскале) звездочек.

```
Procedure StrZv;  
  Var I:Integer;  
  Begin  
    Writeln;  
    For I:=1 to 80 do  
      Write('*');  
    Writeln  
  End;
```

Выше у нас было описано несколько разных процедур и функций, которые что-либо подсчитывали, выдавали какие-то результаты. Покажем, как можно красиво распечатать результаты, используя описанные ранее процедуры и функции. В описательной части программы эти процедуры и функции должны быть описаны, мы здесь описания приводить не будем, так как описания одинаковых процедур в разных программах ничем не отличаются (это и позволяет их широко использовать, написав один раз).

```
Program Nice;  
<Описания PrintKub, Cll, Clear, StrZv>  
  Begin  
    Clear;  
    PrintKub(1,2);  
    PrintKub(5,10);  
    StrZv;  
    Writeln('Путь=',Track(15,4));  
    StrZv;  
    Cll (2.5,10)  
  End.
```

Теперь результаты вычислений отделены друг от друга строчками звездочек.

## Процедуры с входными параметрами

Продолжим писать процедуры для украшения вывода. Усложним написанную выше процедуру: хотим печатать не 80 звездочек, а заданное количество определенных символов.

*Пример 11.5, б. Написать процедуру, печатающую на экране строку из одинаковых символов. Количество символов и символ задаются.*

При вызове этой процедуры пользователь будет определять, сколько и каких символов он хочет напечатать. Для этого в процедуре должно быть два соответствующих параметра. Напомним, параметры пишутся в скобках в заголовке процедуры, после параметра указывается его тип.

```
Procedure StrSim(N:Integer; Sim:Char);
Var i:Integer;
Begin Writeln;
      For I:=1 to N do
          Write(Sim);
      Writeln
End;
```

Как видим, тело процедуры практически не изменилось: лишь константы мы заменили на переменные-параметры (80 на  $N$ , '\*' на  $Sim$ ). Однако в результате этих незначительных изменений у пользователя существенно расширились возможности. Покажем несколько примеров обращения к этой процедуре:

- `StrSim(80, '*')` — результат работы такой же, как и у предыдущей процедуры (из примера 11.5, *a*), напечатается строка из 80 звездочек;
- `StrSim(40, '_')` — будет «нарисована» горизонтальная полоска с левого края до середины экрана.

В задачах предыдущего раздела нам часто приходилось выводить на экран массив. Для этого удобно иметь в своей «библиотеке» специальную процедуру.

**Пример 11.6.** Вывести на экран первые  $N$  элементов целочисленного массива. (Если задать  $N$  равным длине массива, на экран будет выводиться весь массив.)

```
Type Mas=Array [1..100] of Integer;
Procedure PrintMas (Var A:Mas; N:Integer);
Var J:Integer;
Begin Writeln('Массив');
For J:=1 to N do Write(A[J]:5);
Writeln
End;
```

В этой процедуре два параметра: массив и количество элементов, которое надо вывести (отметим, важно не выйти за границы массива, надо, чтобы выполнялось условие  $N \leq 100$ ).

Мы знаем, что в скобках после параметра надо указать его тип, точнее, название типа. В нашем случае для параметра  $N$  мы указываем название стандартного типа **Integer**, для параметра  $A$ , массива, стандартного имени типа не существует, поэтому перед описанием процедуры надо описать

тип массива, дать ему имя, чтобы использовать его в заголовке процедуры. Совмещать эти два действия, описывать массив или какой-то другой тип прямо в заголовке процедуры нельзя.

Отметим, что при описании типа *Mas* указано, что это массив целых чисел, поэтому для печати массива символов придется писать другую процедуру (она может выглядеть точно так же, только в заголовке будет другой тип массива). В нашем описании перед именем массива стоит слово **Var** — что это означает, скажем позже.

Часто в задачах бывает, что количество элементов массива, с которыми ведется работа, определено, не изменяется (например, надо работать со всеми элементами). В этом случае в процедуре можно использовать глобальную константу. Важно, чтобы эта константа была описана до описания процедуры.

```
Const N=50;
Type Mas=Array [1..N] of Integer;
Procedure PrintMas (Var A:Mas);
Var J:Integer;
Begin Writeln('Массив');
      For J:=1 to N do Write(A[J]:5);
      Writeln
End;
```

Такая процедура умеет печатать только весь массив целиком.

Мы уже говорили, что в процедурах и функциях можно использовать другие подпрограммы, важно, чтобы вызываемая процедура/функция была уже описана. Например, если перед нашей процедурой печати массива описать процедуру очистки экрана, ее вызов можно вставить в процедуру *PrintMas*, тогда массив будет печататься на чистом экране. А с помощью процедуры *StrSim* можно при печати нескольких массивов отделять один от другого.

Не всегда результат вычислений подпрограммы надо печатать, иногда его надо сохранить в переменной. В этом случае удобно пользоваться функциями.

***Пример 11.7.** Написать функцию, вычисляющую  $N!$  (для  $N \geq 1$ ). Использовать ее для вычисления значения выражения  $(X!-Y!)/(X-Y)!$  при  $X > Y$ .*

```
Var X,Y: Integer;
Function  Factor(N:Integer):LongInt;
Var F:LongInt; I:Integer;
Begin F:=N;
      For I:=N-1 downto 2 do
        F:=F*I;
      Factor:=F
End;
```

```
{Основная программа}  
Begin Write ('X='); Readln(X);  
      Write ('Y='); Readln(Y);  
      Writeln( (Factor(X)-Factor(Y)) /  
              Factor(X-Y) :15:2)  
End.
```

Разберем это решение. В начале программы идет описание переменных  $X$  и  $Y$ , которые понадобятся для вычисления значения выражения. Эти переменные должны иметь целые значения.

**Описание функции.** Функция называется *Factor*, у нее один аргумент — целое число. Это и указано в скобках. Значение аргумента мы будем передавать функции из основной программы. После этого указывается тип результата функции (этим заголовок описания функции отличается от заголовка процедуры). Результат у нас целый, но мы помним, что наибольшее целое число стандартного типа **Integer** в Турбо Паскале — 32767 и уже 7! будет больше этого значения. Поэтому опишем нашу функцию как **LongInt** — этот тип позволяет работать с целыми значениями гораздо большего размера.

В теле функции описаны две локальные переменные:  $F$  и  $I$ . Напомним, что эти переменные определены только в этой функции. В основной программе, в других функциях могут быть переменные с такими же именами, но они не будут оказывать никакого влияния на значения этих переменных.

Факториал мы считать умеем. Произведение чисел будем накапливать в переменной  $F$ , не забыв присвоить ей начальное значение.

По окончании вычислений результат присвоим имени функции. Напомним, что в теле функции обязательно должен быть хотя бы один выполняемый оператор присваивания, в котором имя функции приобретает какое-то значение. Это и будет ее результат, «ответ», то, что функция вычисляет и передает основной программе. Если такого оператора не будет, значение функции будет не определено. При обращении к такой функции описанные вычисления будут производиться, однако имя функции окажется без значения. Мы уже говорили, что работа с переменными без значения является ошибкой. Говорили и о том, что часто такая ошибка автоматически не обнаруживается, некоторые трансляторы переменным без значения (и функциям) присваивают какие-то значения самостоятельно. Таким образом, в программе будет использовано не то значение функции, которое было вычислено, а какое-то «с потолка». Такую ошибку обычно бывает нелегко найти: функция написана правильно, а программа выдает неправильный результат. Внимательно следите за соблюдением этого правила!

А теперь выясним еще один важный вопрос. Сначала результат функции мы получили в переменной  $F$ , а затем ее значение присвоили имени функ-

ции *Factor*. А нельзя ли было обойтись без дополнительной переменной, вычислять значение функции сразу в переменной *Factor*? В данном случае нельзя! Дело в том, что тогда бы в теле функции присутствовал оператор **Factor := Factor\*I**, в котором имя функции указано не только в левой части оператора присваивания, но еще и в правой.

Это неправильная запись, транслятор в данном месте программы обнаружит ошибку. Давайте разберемся, какую.

Упоминание имени функции в правой части оператора присваивания, в условном операторе (**If Factor...**), в операторе вывода — везде, кроме левой части оператора присваивания, означает обращение к этой функции. А мы знаем, что при обращении к функции надо вслед за ее именем в скобках указывать фактические параметры (если они есть). В нашем случае у функции есть один параметр, вот транслятор и потребует, чтобы после имени функции в правой части оператора присваивания стояли скобки и параметр в них. Будет ли правильным наше описание функции, если мы «пойдем на поводу» у транслятора, добавим скобки и аргумент? Синтаксически — да, такая работа с функциями (и с процедурами) в Паскале, в принципе, разрешена. Она называется **рекурсией**. Если в функции используется такой оператор, получается, что функция описывается рекурсивно, сама через себя. Функцию факториал и в самом деле можно описать рекурсивно, но надо уметь это делать. Рекурсией мы займемся несколько позже (в том числе и рекурсивную функцию вычисления факториала напишем), пока же рекурсией пользоваться не будем, поэтому надо внимательно следить, чтобы в описании функции ее имя использовалось правильно.

**Обращение к функции из основной программы.** В основной программе все просто: вводим с клавиатуры значения переменных и обращаемся к функции. Для этого в скобках пишем нужный аргумент (это может быть переменная, константа, выражение), а имя функции используем в каком-либо операторе. В нашем случае это оператор-процедура вывода **Writeln**.

Обращение к функции может быть в основной программе или в другой функции (но, еще раз отметим, не в описании самой функции!). Кроме процедуры вывода, как в нашем случае, обращение к функции может встретиться и в других операторах, например:

- в операторах присваивания:

```
X:=Factor(Y-5); Y:=2*Factor(X) div Factor(Y);
```

- в условных операторах, операторах варианта:

```
If Factor(X) > Factor(Y-1) Then...;
```

- в операторах цикла:

```
For I:=1 to Factor(N) do...; While (Factor(K)<A) do...
```

Рассматривая работу программы, в которой есть описания процедур и функций, следует помнить, что действия выполняются в той последовательности, в которой они указаны в разделе операторов основной программы. Описание функции находится в начале программы, в разделе описаний, но действия, указанные в этом описании, будут выполняться только тогда, когда функция будет вызвана из основной программы. В нашем случае факториал числа будет вычисляться только после того, как будут введены оба значения  $X$  и  $Y$  и понадобится вывести ответ.

**Пример 11.8.** *С клавиатуры вводятся значения длин катетов прямоугольного треугольника (в см). Выяснить, на сколько сантиметров изменится гипотенуза треугольника, если каждый катет увеличить на 1 см.*

```
Program Treugolnik;
Function Hip(a,b:Real):Real;
Begin Hip:=SQRT(SQR(a)+SQR(b));
End;
Var a,b,c1,c2 :Real;
Begin
  Write('Введите значения длин катетов треугольника ');
  Readln(a,b);
  c1:=Hip(a,b);
  c2:=Hip(a+1, b+1);
  Writeln('гипотенуза увеличится на',c2-c1:6:2,' см');
End.
```

Раздел операторов в теле функции (вычисление гипотенузы) в этой задаче получился совсем маленький: одна формула, но если бы его не было, нам бы пришлось в основной программе писать эту формулу два раза; первый раз для катетов  $a$  и  $b$ , а второй раз — для катетов  $(a+1)$  и  $(b+1)$ . Программа получилась бы менее наглядной.

Здесь и в описании функции, и в основной программе использованы одни и те же переменные  $a$  и  $b$ . Смысл этих переменных разный. В подпрограмме — это формальные параметры, они не имеют никаких числовых значений и нужны, чтобы показать, какие объекты входят в формулу, как они связаны, как вычисляется значение функции. В основной программе — это уже фактические параметры, значения этих переменных вводятся извне и подставляются в формулу. Описание  $a$  и  $b$  в заголовке функции распространяется только на функцию, в основной программе оно не действует, именно поэтому мы описываем переменные  $a$  и  $b$  в основной программе (при этом, еще раз подчеркнем, совершенно не обязательно, чтобы эти имена в основной программе совпадали с именами в функции).

## Задачи 11.4–11.12. Процедуры с входными параметрами и функции

- 11.4. Заданы целые числа  $M$  и  $N$  и символ  $C$ . Написать процедуру, которая рисует на экране рамку размером  $M * N$  из символов  $C$  (внутри рамки пусто). В основной программе применить процедуру 3 раза, построив несколько разных рамок.
- 11.5. Написать процедуру для украшения вывода. Процедура получает на вход вещественное число и печатает его в рамке (рамочка создается из некоторых символов) посередине экрана.
- а) Формат печати числа указывается в теле процедуры;
  - б) процедура вычисляет формат печати числа — в дробной части печатается столько же цифр, сколько в целой.
- В основной программе проверить работу процедуры при печати различных чисел.
- 11.6. Заданы два катета прямоугольного треугольника.
- а) Найти периметр. Написать соответствующую функцию. В основной программе вычислить (показать на примере), во сколько раз увеличится периметр, если каждый из катетов увеличится в 2 раза.
  - б) Написать процедуру, которая печатает длины всех сторон такого треугольника и его периметр. В основной программе ввести некоторые длины катетов, напечатать ответы для треугольника с введенными катетами, а также для треугольника, у которого один катет на 1 см больше введенного, а другой — на 2 см меньше.
- 11.7. Заданы три числа — ребра прямоугольного параллелепипеда. Написать функцию, которая вычисляет его объем. В основной программе ввести некоторые значения длин ребер, вычислить объем прямоугольного параллелепипеда с заданными длинами ребер, а также с длинами ребер на 1 меньше.
- 11.8. Заданы три числа — ребра прямоугольного параллелепипеда. Написать процедуру, которая печатает площадь полной поверхности и объем параллелепипеда. В основной программе напечатать результаты для двух параллелепипедов, размеры которых вводятся с клавиатуры.
- 11.9. Написать функцию, которая вычисляет сумму цифр натурального числа. В основной программе ввести 3 числа и вывести то число, у которого сумма цифр максимальная.
- 11.10. Написать функцию, которая вычисляет количество вхождений заданного символа в массив. В основной программе ответить на вопрос, какой символ в массиве встречается чаще: первый или последний.



- 11.11. Написать функцию, которая вычисляет сумму  $N$  первых элементов целочисленного массива. Посчитать, какая сумма больше: элементов, стоящих в первой половине массива, или элементов всего массива. Напечатать ту часть массива, сумма которой больше (использовать написанную выше процедуру).
- 11.12\*. Написать функцию, вычисляющую среднее арифметическое элементов массива, которые стоят начиная с позиции  $K1$  до  $K2$  (включительно). Выяснить, какие  $K1$  и  $K2$  надо взять, чтобы среднее арифметическое получилось наибольшим. Напечатать элементы массива, участвовавшие в подсчете этого результата.

## Параметры-переменные и параметры-значения

До сих пор мы писали функции, которые что-то вычисляли (и больше ничего не делали), и процедуры, которые не вычисляли никаких значений, а только выполняли некоторые действия. На самом деле и в процедуре, и в функции можно произвести вычисления и результат присвоить каким-то параметрам. Перед этими параметрами в описании подпрограммы в заголовке должно стоять слово **Var**.

Про параметр со словом **Var** говорят, что это параметр-переменная. И в самом деле, при вызове подпрограммы соответствующий фактический параметр должен быть только переменной (вспомним, что служебное слово **Var** и произошло от слова *Variable* — переменная). Ведь этому параметру может быть что-то присвоено, а присваивать значение можно только переменной (не константе, не выражению).

Параметры без слова **Var** в описании называются параметры-значения (иногда говорят, что параметры вызываются по значению). При вызове подпрограммы на место этих параметров подставляются фактические значения (в том числе константы).

При написании процедур и функций будем пользоваться следующими советами.

1. Сначала надо выяснить, что в задаче задано, а что следует найти, вычислить. Похожие действия вы делаете при записи условий задач по физике, геометрии, используя заголовки «дано» и «найти». Здесь же полезно написать не только название переменной, но и ее тип.
2. Если «найти» надо какой-то единичный объект (значение которого выражается числом, символом), удобнее описывать функцию (тип результата и будет ее типом). Если же «найти» надо несколько значений (в том числе массив — в массиве ведь много значений) или никаких результатов выдавать вообще не надо, удобнее оформить действия в виде процедуры.
3. В заголовке процедуры должны быть указаны все параметры (и из «дано», и из «найти»). При описании функции параметр «найти» в скобках не указывается, это значение функции, ее результат.

4. Все параметры, которые попали в «найти», обязательно надо объявить как параметры-переменные (со словом **Var**), иначе в теле процедуры будут происходить нужные вычисления, но полученные значения параметров не будут передаваться в основную программу. При вызове процедуры на месте этих параметров могут стоять только переменные.
5. Некоторые параметры из «дано» иногда тоже описывают со словом **Var**. Обычно так поступают с массивами, это экономит время и память. Дело в том, что при обращении к процедуре параметр-значение копируется, а копировать целый массив без необходимости ни к чему. Тем более, при вызове подпрограммы на месте параметра-массива мы ничего, кроме имени, поставить и не можем. При этом надо внимательно следить, чтобы массив не был случайно изменен, ведь как только к параметру приписали слово **Var**, он теряет свою «защиту от изменений».
6. Параметры-значения описываются без слова **Var** (вызываются по значению). Работа в этом случае производится с копией значения (формальному параметру, описанному в процедуре, присваивается указанное при вызове фактическое значение). С такими параметрами удобно работать, так как при вызове подпрограммы на их место можно подставить не только переменную, но и константу или выражение. Даже если внутри подпрограммы значение такого параметра меняется, при выходе из нее оно восстанавливается (так как меняется значение не самого параметра, а его копии).

Исходя из этих советов, в предыдущей задаче в заголовке функции перед именем массива мы и поставили **Var**.

***Пример 11.9.** С клавиатуры вводятся значения длин катетов прямоугольного треугольника (в см). Выяснить, на сколько сантиметров изменятся гипотенуза и периметр треугольника, если каждый катет увеличить на 1 см.*

Этот пример похож на пример 11.8, но там надо было вычислять только гипотенузу, а здесь еще и периметр. Итак:

Дано: катеты **a** и **b** типа **Real**

Найти: гипотенузу **c**: **Real**;  
периметр **p**: **Real**

```
Program Treugolnik;
Procedure Treug(a,b:Real; Var c,p:Real);
Begin c:=SQRT(SQR(a)+SQR(b));
      p:=a+b+c
End;
Var a,b,c1,c2,p1,p2 :Real;
```

Begin

```
Write('Введите значения длин катетов треугольника ');
Readln(a,b);
Treug(a,b,c1,p1);
Treug(a+1,b+1,c2,p2);
Writeln('гипотенуза увеличится на',c2-c1:6:2,' см');
Writeln('периметр увеличится на',p2-p1:6:2,' см')
```

End.

В теле процедуры мы присваиваем двум переменным значения, вычисляемые по формулам. Обратите внимание, имени процедуры ничего присваивать нельзя (в отличие от имени функции, которому обязательно нужно присвоить результат).

В основной программе мы два раза обращаемся к процедуре *Treug*. В первом случае вычисления будут производиться для введенных значений длин катетов  $a$  и  $b$ , результаты запишутся в переменные  $c1$  и  $p1$ . Во втором случае значениями длин катетов будут  $a + 1$  и  $b + 1$ , а результаты запишутся в переменные  $c2$  и  $p2$ .

**Пример 11.10.** Написать процедуру для ввода  $N$  первых компонент целочисленного массива с клавиатуры.

Такой процедурой очень удобно пользоваться, если для решения задачи требуется ввести несколько массивов. Если  $N$  равно длине массива, будет введен весь массив.

Дано: ничего. В этой задаче никаких исходных данных нет, все вводится с клавиатуры в процессе работы.

Найти (ввести): количество компонент массива

**N : Integer;**  
массив **A : Mas**

В процедуре будет два формальных параметра, оба с **Var**, так как они находятся в разделе «найти». Оба фактических параметра в результате действия процедуры должны поменяться, эти изменения должны быть сохранены и переданы в основную программу.

```
Type Mas=array[1..50] of integer;
Procedure VvodMas (Var A:Mas; Var N:Integer);
Var i:Integer;
Begin
  Write('кол-во вводимых компонент массива ');
  Readln(N);
  for i:=1 to N do
    Begin
      write('A[' ,i, ']='); readln(A[i]);
    End;
End;
```

Имея такое описание, надо в основную программу вставлять вызов этой процедуры, указывая в скобках имя массива, который нужно ввести, и переменную, в которую надо занести количество компонент ввода: **VvodMas (X,Dx) ; VvodMas (Chisla, Kvo)**. Важно, что количество вводимых компонент не должно превышать 50 (так у нас описан тип *Mas*).

## Примеры использования процедур и функций

*Пример 11.11. С клавиатуры вводятся координаты вершин двух треугольников. Определить, какой из них имеет большую площадь, и выписать его координаты.*

Будем хранить координаты вершин треугольника в матрице размером  $3 * 2$ . В первом столбце будут абсциссы (координата  $X$ ), во втором — ординаты (координата  $Y$ ). Каждая строчка будет соответствовать очередной вершине треугольника. Таким образом,  $T[1,1]$  — координата  $X$  первой вершины, а  $T[3,2]$  — координата  $Y$  третьей вершины. Длины сторон треугольника также будем хранить в массиве (из трех элементов).

Понадобятся следующие типы данных:

```
Type Treug=Array[1..3,1..2] of Real;
      DlinS=Array[1..3] of Real;
```

Теперь займемся вводом координат вершин треугольника.

Дано: в этой задаче никаких исходных данных нет

Найти (ввести): массив типа **Treug**

Так как в «найти» — массив, функцию написать нельзя, будем писать процедуру с одним параметром, причем он будет с **Var**.

```
Procedure VvodKoord(Var T:Treug);
Var V:Integer;
Begin
  For V:=1 to 3 do
  Begin
    Writeln ('Введите координаты ',V, ' вершины ');
    Write('X=');   Readln(T[V,1]);
    Write('Y=');   Readln(T[V,2])
  End
End;
```

Пусть известны координаты концов отрезка, вычислим его длину.

Дано: **x1, y1** — координаты первого конца;  
**x2, y2** — координаты второго.  
 Координаты — вещественные числа

Найти: длину отрезка.  
 Результат имеет тип **Real**

В «найти» — одно значение, можно написать функцию. У нее будет 4 параметра (то, что дано). А тип результата — **Real**. В теле функции всего один оператор присваивания. В левой его части имя функции, а в правой — арифметическое выражение, формула для вычисления длины отрезка по его координатам.

```
Function DlinOtr (X1,Y1,X2,Y2:Real):Real;
Begin DlinOtr:= Sqrt (Sqr (X2-X1)+Sqr (Y2-Y1))
End;
```

По этой формуле можно вычислить длины всех сторон треугольника.

Дано: массив координат вершин  
треугольника, тип **Treug**

Найти: массив длин всех сторон,  
тип **Dlins**

Надо писать процедуру, у нее будет два параметра. Параметр «найти» надо обязательно писать с **Var**, параметр «дано» — в данном случае массив, его с **Var** писать не обязательно, но мы говорили, что часто это делают для экономии памяти и времени.

У первой стороны концы отрезков имеют в нашем массиве индексы 1 и 2, у второй — 2 и 3; их можно вычислить в одном цикле. Длину третьей стороны, концы которой имеют индексы 3 и 1, придется вычислять отдельно. При вычислении воспользуемся только что описанной функцией *DlinOtr*.

```
Procedure DStoron (Var T:Treug;Var R:Dlins);
Var I:Integer;
Begin For I:=1 to 2 do
      R[I]:=DlinOtr (T[I,1],T[I,2],T[I+1,1],
                    T[I+1,2]);
      R[3]:=DlinOtr (T[3,1],T[3,2],T[1,1],T[1,2])
End;
```

Зная длины всех сторон, можно найти площадь треугольника по формуле Герона: вычислить квадратный корень из произведения полупериметра на разности полупериметра со всеми сторонами.

Дано: **a,b,c:Real** — длины сторон  
треугольника (вещественные числа)

Найти: площадь треугольника.  
Результат имеет тип **Real**

Найти надо одно значение, значит, можно написать функцию. Для решения задачи нам понадобится вспомогательная переменная — полупериметр. В теле функции она будет описана как локальная переменная *p*.

```
Function SGeron(a,b,c:Real):Real;
Var p:Real;
Begin p:=(a+b+c)/2;
      SGeron:=Sqrt(p*(p-a)*(p-b)*(p-c));
End;
```

Также надо уметь печатать координаты вершин треугольника.

<u>Дано:</u> массив координат вершин треугольника, тип <b>Treug</b>	<u>Найти:</u> ничего
--	----------------------

В процедуре будет один параметр, писать **Var** не обязательно, так как это параметр «дано», но мы помним, что массивы обычно вызывают по имени (с **Var**).

```
Procedure PrintTr(Var T:Treug);
Var V:Integer;
Begin Writeln('Координаты вершин треугольника ');
      For V:=1 to 3 do
        Writeln(V:5, '    X=', T[V,1]:6:1, '    Y=', T[V,2]:6:1);
End;
```

В основной программе надо описать переменные.

```
Var T1,T2 : Treug; {координаты вершин треугольников}
    R1,R2 : DlinS; {длины сторон треугольников}
    S1,S2 : Real; {площади}
```

Вот такой длинный получился раздел описаний. В нем мы «научили» компьютер вводить и выводить координаты вершин треугольника, вычислять длины сторон, если известны координаты вершин. Теперь этими новыми «умениями» надо правильно воспользоваться.

```
Begin Writeln('Ввод координат первого треугольника');
      VvodCoord(T1);
      Writeln('Ввод координат второго треугольника');
      VvodCoord(T2);
      DStoron(T1,R1); {вычисление длин сторон треугольников}
      DStoron(T2,R2);
      Writeln('Треугольник с большей площадью');
      If SGeron(R1[1],R1[2],R1[3]) > SGeron(R2[1],R2[2],R2[3])
        Then PrintTr(T1)
        Else PrintTr(T2)
End.
```

Использование процедур/функций никоим образом не увеличивает возможности компьютера (в теле каждой процедуры/функции мы можем написать только уже известные операторы), но делает программу понятнее, нагляднее. Применение процедур/функций также дает возможность разбить большую задачу на несколько более мелких подзадач, каждая из которых решается отдельно. Ну и, конечно, облегчает программирование решения одинаковых задач с разными исходными данными в рамках одной общей программы.

## **Задачи 11.13–11.21. Процедуры и функции с входными и выходными параметрами**

11.13. Написать следующие процедуры (функции):

- заданы 3 числа, надо поставить их в порядке возрастания;
- заданы 3 числа в порядке возрастания, надо выяснить, можно ли построить треугольник с такими длинами сторон;
- заданы длины сторон треугольника, найти его площадь и периметр.

Пользуясь этими процедурами и функциями, написать программу: переставить введенные 3 числа в порядке возрастания, проверить, могут ли они быть длинами сторон треугольника. Если треугольник с такими длинами сторон построить можно, напечатать его площадь и периметр. Если треугольник построить нельзя, изменить числа: самое большое число заменить на среднее (среднее в наборе теперь будет два раза) и для получившегося треугольника напечатать площадь и периметр.

11.14. Написать процедуру для безошибочного ввода целого числа из заданного диапазона. При ошибочном вводе (вводятся символы, отличные от цифр, получившееся число выходит за границы диапазона) предлагается ввести число еще раз. В основной программе с помощью этой процедуры ввести размерность и компоненты целочисленной матрицы.

11.15. Реализовать работу с векторами. Вектор (двумерный или трехмерный) задается в виде набора координат. Найти длину вектора, определить взаимное расположение векторов (равны, параллельны и т. п.), найти сумму, произведение на число. Используя эти процедуры, можно решать задачи с векторами, например из школьного учебника.

11.16. Написать процедуры и функции для работы с целым числом:

- а) найти сумму цифр числа;
- б) найти самую большую цифру;
- в) выписать цифры числа в обратном порядке;

- г) выписать четные цифры числа;
- д) добавить в число после каждой цифры 0;
- е) составить число из цифр данного числа, повторив каждую цифру два раза;
- ж) составить новое число, добавив к каждой цифре данного числа 1 (9 заменить на 0);
- з) найти цифру числа, которая стоит на заданном месте;
- и) заменить одну цифру на другую.

11.17. Написать процедуры и функции для работы с массивом (размерность массива — константа):

- а) найти сумму элементов массива;
- б) найти самое маленькое число в массиве;
- в) выписать элементы массива в обратном порядке;
- г) выписать нечетные элементы массива;
- д) все элементы массива сделать равными заданному числу;
- е) распечатать элементы массива с индексами от  $k_1$  до  $k_2$ ;
- ж) напечатать элементы массива, принадлежащие промежутку  $[a, b]$ ;
- з) найти сумму элементов массива, принадлежащих промежутку  $[a, b]$ ;
- и) заменить первое вхождение заданного элемента на другое число;
- к) заменить все вхождения заданного элемента на другое число;
- л) определить, сколько раз в массив входит заданный элемент;
- м) найти максимальное нечетное число в массиве;
- н) поменять местами элементы массива, стоящие на заданных местах.

11.18. Написать самостоятельно строковые процедуры и функции: поиска, вставки, удаления, замены. Проверить их правильность, заменив в программах, обрабатывающих строки (гл. 10), встроенные процедуры и функции на собственные.

11.19\*. В гл. 1 рассказывалось о работе с длинными числами (пример 1.5) — когда число представляется в виде массива цифр. Таким образом можно представлять как очень большие целые числа, так и очень точные дробные. Реализовать процедуры и функции для работы с такими числами: сравнения, сложения, вычитания, умножения, деления.

11.20\*. Реализовать работу с длинными числами в заданной системе счисления.

11.21\*. Написать процедуры (функции) для перевода чисел из одной системы счисления в другую, для вычисления суммы и разности чисел, представленных в разных системах счисления.



# Глава 12

## Рекурсия

---

Мы знаем, что при описании процедуры или функции в ее теле может быть обращение к еще каким-то процедурам или функциям. В частности, функция или процедура может обратиться сама к себе. Такое описание называется рекурсивным. Рекурсия — один из распространенных приемов программирования.

### Работа рекурсивных процедур и функций

Рассмотрим пример рекурсивной функции (оператор, благодаря которому она является рекурсивной, специально помечен):

```
Function Rec(K:Integer):Integer;  
Begin  
    If K<=0 Then Rec:=0  
        Else Rec:=K+Rec(K-1) {Это рекурсивное  
                                обращение}  
End;
```

Посмотрим, как будет работать такая функция. Напомним, чтобы увидеть работу функции на компьютере, надо написать программу, в которой будет вызов функции от какого-то аргумента и печать результата. Для всех неположительных  $K$  функция будет давать результат 0, при вычислении никаких рекурсивных вызовов не происходит. Для  $K = 1$  получим  $\text{Rec}(1) = 1 + \text{Rec}(0) = 1 + 0 = 1$  — один раз функция сама себя вызывает. При  $K = 2$  будет 2 рекурсивных вызова:  $\text{Rec}(2) = 2 + \text{Rec}(1) = 2 + 1 = 3$ . Проследим работу функции для достаточно большого числа:  $\text{Rec}(10) = 10 + \text{Rec}(9) = 10 + 9 + \text{Rec}(8) = 10 + 9 + 8 + \text{Rec}(7) = \dots = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 55$ .

Пример рекурсивной процедуры:

```
Procedure PRec(K:Integer);  
Begin  
    If K<=0 Then Writeln  
        Else Begin Write(K:3);  
                    PRec(K-1)  
                {Здесь процедура рекурсивно обращается сама к себе}  
            End  
End;
```

Как работает эта процедура? Если аргумент процедуры неположителен, она только переводит строку. При вызове PRec(2) напечатается число 2 и вызовется процедура с параметром 1, в результате чего напечатается 1, вызовется процедура с параметром 0 и переведется строка.

Заметим, что предложенные здесь процедура и функция (а также и многие, рассматриваемые ниже) носят чисто демонстрационный характер, не выполняют никаких «особенных» действий. Так, наша функция вычисляет сумму всех чисел от своего аргумента до 1, а процедура эти числа печатает.

Рассмотренный выше вид рекурсии, когда процедура (функция) вызывает сама себя, называется *прямой рекурсией*. Еще бывает *косвенная рекурсия*, в ней участвуют несколько процедур (функций), которые обращаются одна к другой «по кругу». Например (выпишем здесь полную программу):

```
Program KosvRec;
Procedure P1(k: integer); forward; {пояснение
                                   этого слова см. ниже}
Procedure P2(k: integer); forward;
Procedure P1(K:Integer); {Первая процедура}
Begin
    If K<=0 Then Writeln
                Else P2(K div 2) {вызов второй процедуры}
End;
Procedure P2(K:Integer); {Вторая процедура}
Begin
    If K<=0 Then Writeln
                Else Begin Write(K:3);
                           P1(K-1) {вызов первой процедуры}
                        End
End;
Begin {Основная программа}
    P1(10) {вызов процедуры}
End.
```

Здесь процедура P1 вызывает P2, а P2, в свою очередь, вызывает P1. В Паскале такая конструкция допустима, но, чтобы он ее понял, необходимо некоторое «предисловие». Дело в том, что в Паскале объекты обычно должны быть описаны до использования. Здесь же мы используем процедуру P2 до описания (причем перестановка процедур ничего не изменит). Использование объекта до описания допускается только в рекурсивных конструкциях, причем в данном случае транслятор надо предупредить о том, что описание последует позже: именно поэтому перед описаниями процедур стоят строчки со словом **forward**.

Разберемся, как работает приведенная конструкция. Если в основной программе вызвать P1(10), будет произведен вызов P2(5), напечатается число 5, вызовется P1(4), затем P2(2), напечатается 2, вызовется P1(1) и в конце P2(0). Таким образом, будет напечатано 5 2.

**Пример 12.1.** Задана рекурсивная функция:

```
Function Rec (k:Integer):Integer;
Begin
    If k>5 Then Rec:=0
        Else Rec:=3+Rec(k+2)
End;
```

Вычислить значение  $F(5)$  и  $F(3)$ .

При  $k = 5$  получим **Rec := 3 + Rec (5+2) = 3 + Rec (7) = 3 + 0 = 3** — происходит один рекурсивный вызов.

При  $k = 3$  функция вызывает сама себя уже 2 раза: сначала вычисляя значение от 5, а потом от 7. Получается **Rec := 3 + Rec (3+2) = 3 + 3 + Rec (7) = 6**.

**Пример 12.2.** Что будет напечатано на экране при выполнении вызова  $F(20)$ ?

```
Procedure F(n: integer); forward;
Procedure G(n: integer); forward;
Procedure F(n: integer);
Begin
    if n > 0 Then
        G(n - 2);
End;
Procedure G(n: integer);
Begin
    Writeln(n);
    if n > 1 Then
        F(n - 3);
End;
```

Изобразим схематически, с какими аргументами вызывают друг друга процедуры:

$F(20) \rightarrow G(18) \rightarrow F(15) \rightarrow G(13) \rightarrow F(10) \rightarrow G(8) \rightarrow F(5) \rightarrow G(3) \rightarrow F(0)$

На этом работа прекратится, так как процедура  $F$  работает только при положительном аргументе. При этом будут напечатаны аргументы функции  $G$  (в ее теле стоит оператор печати): 18 13 8 3 (числа будут напечатаны в столбик).

### Задания

1. Что будет напечатано при вызове  $G(16)$ ?
2. Приведите пример обращения к процедуре, чтобы было напечатано 6 чисел.

3. Существует ли такое  $K$ , что при вызове  $F(K)$  ряд напечатанных чисел будет заканчиваться числом 0? Можно ли подобрать такое число для вызова  $G(K)$ ? Если числа подобрать можно, приведите соответствующий пример, если нет — докажите.

**Пример 12.3.** Сколько звездочек будет напечатано на экране при выполнении вызова  $F(30)$ ?

```
Procedure F(n: integer); forward;
Procedure G(n: integer); forward;
Procedure F(n: integer);
Begin
    Write ('**');
    if n > 0 then
        G(n + 1);
End;
Procedure G(n: integer);
Begin
    Write('*');
    if n > 5 then
        F(n div 2);
End;
```

Схема вызовов получается следующая:

$F(30) \rightarrow G(31) \rightarrow F(15) \rightarrow G(16) \rightarrow F(8) \rightarrow G(9) \rightarrow F(4) \rightarrow G(5)$

На этом работа заканчивается, так как для продолжения вызовов аргумент функции  $G$  должен быть больше 5. Теперь посчитаем звездочки. Функция  $F$  при каждом вызове печатает 2 звездочки, вызывается 4 раза,  $G$  печатает одну звездочку, вызывается тоже 4 раза.  $8 + 4 = 12$ .

**Пример 12.4.** Сколько раз будет напечатано на экране число 16 при выполнении оператора **Write(R(10))**? Чему будет равно значение функции?

```
Function R(n: integer) :Integer;
Begin
    Writeln(n);
    if (n<=15) Then
        R:=R(n+2)+R(n+4)
    Else
        R:=100;
End;
```

Здесь последовательность рекурсивных вызовов в строчку описать не удастся, так как каждая функция делает 2 вызова. Будем строить схему, которая называется «дерево». Такое название схема получила из-за похожести

на рисунок обычного (например, садового) дерева. У нее тоже будет корень и веточки, только расти дерево будет сверху вниз — так удобнее рисовать, ведь мы начнем с корня, в котором расположим первый вызов функции. В прямоугольниках будем писать, как будет выглядеть вызов функции. Если при данном аргументе происходят рекурсивные вызовы (если они есть, то их всегда два), расположим их ниже нарисованного прямоугольника и соединим с ним стрелками (см. рис. 12.1):

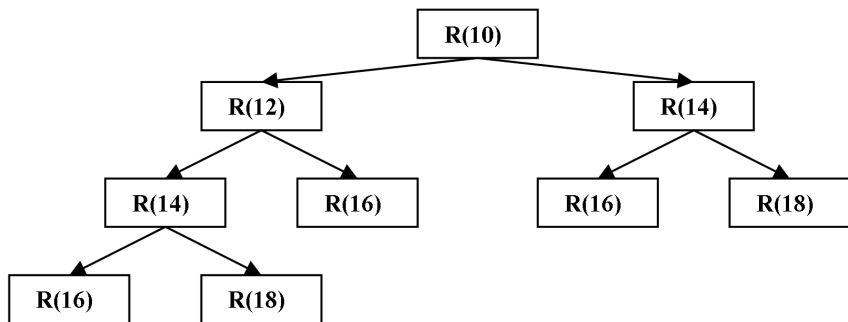


Рис. 12.1

Видим, что число 16 появляется 3 раза.

Чтобы понять, как подсчитывается значение функции, поучимся сначала его считать на более легких примерах:  $R(14) = R(16) + R(18) = 100 + 100$ ;  $R(12) = R(14) + R(16) = 200 + 100 = 300$ . Можно заметить, что в ответе будет столько слагаемых (каждое из которых равно 100), сколько конечных «листочков» у соответствующего рекурсивного дерева. В нашем случае листочков будет 5, ответ 500. Впрочем, его можно получить и основываясь на вышеприведенных подсчетах:  $R(10) = R(12) + R(14) = 200 + 300$ .

### Задания

1. Нарисуйте дерево для вызова  $R(5)$ . Какое число появляется в дереве чаще других? Сколько будет сделано рекурсивных вызовов? Чему будет равно значение функции?
2. Можно ли подобрать такой аргумент, при котором результат функции будет равен 1000?
3. Измените числа в приведенном выше описании функции. Нарисуйте рекурсивное дерево и подсчитайте результат для разных аргументов для своей функции.

Заметим, что проверить правильность всех этих заданий очень просто: надо выполнить соответствующую программу на компьютере. Конечно, дерево компьютер на экране не нарисует, но числа, которые должны в нем присутствовать, напечатает и результат функции покажет.

## Задачи 12.1–12.5. Работа рекурсивных процедур и функций

12.1. Что напечатается при вызовах  $G(3)$ ,  $F(3)$ ,  $G(1)$ ,  $F(1)$  ?

```

Procedure F(n: integer); forward;
Procedure G(n: integer); forward;

Procedure F(n: integer);
  Begin Write(n, ' ');
    if n < 15 then
      G(2*n);
  End;
Procedure G(n: integer);
  Begin Write(n, ' ');
    if n <= 20 then
      F(n + 2);
  End;

```

12.2. Для каждого из вызовов  $T(10)$ ,  $R(10)$ ,  $R(8)$ ,  $T(6)$  ответить на следующие вопросы. Какое самое большое число напечатается? Сколько напечатается чисел? Чему будет равен результат работы функции?

```

Function R(n: integer):integer; forward;
Function T(n: integer): integer; forward;
Function R(n: integer):Integer;
begin
  writeln(n);
  if n <= 20 then
    R:=T(n + 5)+R(2*n)
  else R:=1
end;
Function T (n: integer):Integer;
begin
  Writeln(n);
  if n < 30 then
    T:=R(n + 2)
  else T:=2
end;

```

12.3. Нарисуйте рекурсивное дерево и подсчитайте сумму всех чисел, которые будут напечатаны в процессе работы программы:

```

Function R(n: integer) :Integer;
begin
  writeln(n);

```

```

    if (n<=10) and (n>3) then
        R:=R(n mod 4) + R(n+5) + R(n+2)
    else
        R:=1;
    end;

Begin
    Write(R(8))
End.

```

- 12.4. С рекурсией приходится сталкиваться и в повседневной жизни, например, при расчетах дохода по банковским вкладам используется так называемая формула сложного процента. Пусть вклад  $R$  рублей положен в банк и каждый месяц к нему добавляется  $P$  процентов. Причем надо учесть, что каждый месяц вклад увеличивается, соответственно проценты каждый раз будут рассчитываться от большего числа. Составьте формулу сложного процента. Пусть вклад 100 рублей положен под 2% в месяц. Сколько денег будет через год? Сколько денег (как минимум) надо положить, чтобы доход за год превысил 100 рублей?
- 12.5. Бактерии размножаются делением. Пусть известно, что «новорожденная» бактерия некоторого вида через час разделится на 2 (которые, в свою очередь, будут развиваться по тем же правилам). Напишите рекурсивную формулу для расчета количества бактерий в зависимости от времени. Пусть изначально было 5 бактерий. Сколько их будет через 6 часов? Через сколько часов будет не менее 500 бактерий?

## Рекурсивные алгоритмы

Любой алгоритм, реализованный с помощью операторов цикла, можно реализовать с помощью рекурсии, и наоборот, рекурсивный алгоритм можно написать с использованием операторов цикла.

**Пример 12.5.** Написать вычисление факториала положительного числа с помощью рекурсии.

Посмотрите пример 9.6 в главе «Операторы цикла». Там мы воспользовались нерекурсивной формулой вычисления факториала:

$$N! = 1 * 2 * 3 * 4 * \dots * (N - 1) * N \quad \text{для всех } N \geq 1.$$

Но ведь есть и другая формула, рекурсивная:

$$N! = \begin{cases} 1 & \text{при } N = 1; \\ N * (N - 1)! & \text{при } N > 1. \end{cases}$$

Напишем функцию в точном соответствии с этой формулой, она получится рекурсивной:

```
Function FactorR (N:Integer) : Integer;  
Begin  
  If N=1 Then FactorR:=1  
    Else FactorR:= N*FactorR(N-1)  
End;
```

Какой алгоритм лучше? Рекурсивный, пожалуй, понятнее, логичнее, он в точности повторяет формулу. Однако он менее эффективен, чем алгоритм с циклом: будет медленнее работать и потребует больше памяти. Дело в том, что при работе циклического алгоритма в памяти хранятся только описанные в начале переменные, объем требуемой памяти не зависит от аргумента функции. При работе рекурсивного алгоритма при каждом рекурсивном обращении создаются новые переменные, как это происходит всегда при вызове процедуры (функции). Значения переменных хранятся в памяти, в стеке. Стек — это специальным образом организованное хранилище данных, элементы из него извлекаются в порядке, обратном тому, в котором поступали. Такая структура данных еще называется FILO — First In Last Out (первый поступивший элемент будет извлечен последним). При работе нашей функции сначала числа  $N, N-1, N-2, \dots, 3, 2, 1$  записываются в стек, а потом будут из него извлекаться (в обратном порядке) и перемножаться:  $1*2*\dots*(N-1)*N$ . Понятно, что чем больше значение аргумента функции  $N$ , тем больше памяти и времени потребуется.

**Пример 12.6.** Вычислить число Фибоначчи с помощью рекурсии.

Вспомним, что алгоритм с использованием оператора цикла рассмотрен в примере 9.12. Но ведь числа Фибоначчи задаются рекурсивной формулой:

$$F_n = \begin{cases} 1 & \text{при } n = 0, 1; \\ F_{n-2} + F_{n-1} & \text{при } n > 1. \end{cases}$$

Запишем на Паскале:

```
Function FibR(N:Integer):Integer;  
Begin  
  If N<=1 Then FibR:=1  
    Else FibR:=FibR(N-2)+FibR(N-1)  
End;
```



Как работает эта функция? Чтобы вычислить, например, пятое число последовательности, она выполняет оператор присваивания  $\text{FibR} := \text{FibR}(4) + \text{FibR}(3)$ . А в этом операторе два рекурсивных вызова:  $\text{FibR} := \text{FibR}(3) + \text{FibR}(2)$  и  $\text{FibR} := \text{FibR}(2) + \text{FibR}(1)$ , каждый из которых инициирует следующие рекурсивные вызовы. Дерево вызовов будет выглядеть, как на рис. 12.2.

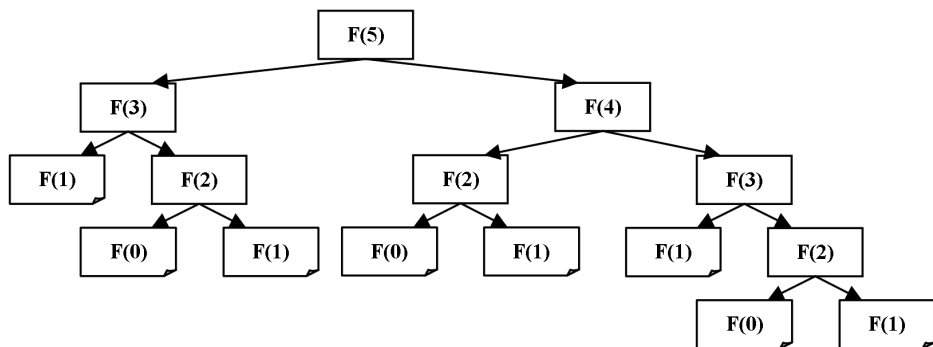


Рис. 12.2

Здесь мы «концевые» вызовы, которые не ведут к дальнейшей рекурсии, поместили в несколько другие прямоугольники — с загнутым уголком. Значения функции в них равны 1. Значения во всех остальных прямоугольниках получаются суммированием всех единиц, висящих на исходящих из соответствующего прямоугольника веточках.

Причем, обратите внимание, для вычисления **Fib(4)** надо знать значение **Fib(3)**, которое вычисляется во втором слагаемом. Однако функция эти вычисления производит еще раз. Если вы возьмете число побольше и распишете, как здесь работает рекурсия, то увидите, как лавинообразно возрастает количество рекурсивных вызовов, сколько раз вычисления дублируются. Поэтому рекурсивное описание функции вычисления чисел Фибоначчи обычно приводят как пример нерационального. То, что рекурсивная функция работает гораздо медленнее, можно заметить даже на наших быстрых компьютерах. Если вы подсчитаете достаточно большое число Фибоначчи (например, 40-е), то увидите, что нерекурсивная функция выдает ответ мгновенно, а при работе рекурсивной приходится некоторое время подождать. (Заметим, для того чтобы вычислить такое большое число, придется значение функции определить как **LongInt**.)

Зачем же тогда писать рекурсивные алгоритмы, если они менее эффективны? Дело в том, что существуют задачи, рекурсивное исполнение которых много проще и понятнее, чем алгоритмы с циклом.

*Пример 12.7, а. Посмотрите, что делает следующая программа:*

```
Program Print;  
Procedure Pr; {Описание процедуры}  
Var a:char;  
Begin  
    Read(a);  
    If a='*' Then Writeln  
        Else Begin Write(a);  
                Pr {Рекурсивный вызов}  
            End  
End;  
Begin {Основная программа}  
    Pr  
End.
```

Для работы программы надо ввести последовательность символов (обратите внимание, ввод производится оператором **Read**, а не **Readln**, значит, символы можно вводить в строчку). Процедура считывает один символ. Если это звездочка, процедура переводит строку и заканчивает работу. Если же был введен другой символ, работает ветвь **Else**: введенный символ печатается, а затем происходит рекурсивный вызов, т. е. выполнение процедуры начинается сначала, вводится очередной символ. Таким образом, чтобы процедура завершилась, входная последовательность символов должна заканчиваться звездочкой. В результате работы процедура печатает введенную последовательность (без звездочки), например, если будет введено МАРШ\*, выведется МАРШ.

*Пример 12.7, б. Изменим предложенную программу: в ветке **Else** поменяем местами операторы. Что теперь будет выводиться на экран?*

Выпишем условный оператор со сделанными изменениями:

```
If a= '*' Then Writeln  
    Else Begin Pr; {Рекурсивный вызов}  
            Write(a)  
        End
```

Как и в предыдущем случае, при вводе звездочки процедура переводит строку и заканчивает работу. Если введен другой символ, происходит рекурсивный вызов. Процедура должна в переменную **a** прочитать очередной символ. Но ведь в этой переменной хранится предыдущий символ, не потеряется ли он? Оказывается, нет, переменная называется так же, но место в памяти отводится другое, мы ведь не просто переходим к первому опера-

тору процедуры, а вызываем ее заново, значит происходит резервирование памяти (**var a : Integer**). Мы уже говорили об этом при обсуждении примера 12.5: значения сохраняются в стеке. Итак, если на вводе МАРШ\*, в стек сначала попадает буква М, потом А и так далее. Когда будет прочитана звездочка, выполнение последней вызванной процедуры закончится, будет выполняться следующий за вызовом оператор **Write(a)**. Значение **a** при этом будет то, которое прочиталось при предыдущем вызове, оно будет взято из стека, т. е. напечатается Ш. При этом окончится работа очередной процедуры и опять будет выполняться **Write(a)** со следующим значением **a** из стека. Это будет буква Р. Далее аналогично будут извлекаться из стека буквы и печататься. Таким образом, программа напечатает ШПРАМ — введенные буквы в обратном порядке.

Получается, мы написали программу, которая печатает введенную последовательность в обратном порядке. Как решить такую задачу без рекурсии? Придется куда-то записывать последовательность, а потом ее распечатывать. Причем все непросто: длина последовательности заранее не оговаривается, а размеры массива должны быть известны до работы программы. Почему таких проблем не возникает при написании рекурсивного алгоритма? Дело в том, что большую часть работы в этом случае компьютер берет на себя: он организует хранилище информации (стек), складывает и извлекает оттуда символы.

Еще один пример, где рекурсивный алгоритм гораздо проще нерекурсивного — задача о Ханойской башне (популярная головоломка конца XIX века, несмотря на название, к Ханюю никакого отношения не имеет).

***Пример 12.8.** Имеется три стержня, на одном из них нанизаны колечки разного размера: большие внизу, маленькие сверху (как в детской пирамидке). Надо перенести за наименьшее число ходов всю пирамидку на другой стержень. За один ход разрешается перекладывать только одно кольцо (естественно, надо брать с любого стержня самое верхнее кольцо), причем при любом перекладывании верхнее кольцо должно быть меньше нижнего. Написать программу, на вход которой подается количество колец и номера стержней (того, на котором стоит пирамидка, и того, на который ее надо переставить), а выводится инструкция человеку для правильной перестановки колец.*

Составим рекурсивный алгоритм. Рассмотрим сначала простейший случай: пусть пирамидка из двух колец на стержне 1, надо перенести ее на стержень 2. Верхнее кольцо переносим на стержень 3, нижнее — на 2. Остается маленькое кольцо поставить на место.

Предположим, мы умеем правильно переносить с одного стержня на другой пирамидку из  $K - 1$  колец. Как перенести пирамидку из  $K$  колец?

Пусть на стержне 1 пирамидка из  $K$  колец и ее надо перенести на стержень 2. Переносим пирамидку из  $K - 1$  колец на вспомогательный стержень 3 (мы ведь договорились, что умеем это делать), нижнее (самое большое) кольцо освободилось, переносим его на стержень 2, и теперь пирамидку из  $K - 1$  колец тоже переносим на 2. Давайте оформим это в виде программы, причем в процессе решения будем подсчитывать количество операций. Инструкцию будем выписывать в виде номера действия, номера стержня, с которого надо взять верхнее кольцо, и номера стержня, на который надо положить. Какое колечко взять со стержня, указывать не надо, так как можно брать только самое верхнее.

Возникает задача, совершенно не связанная с рекурсией: как узнать номер дополнительного стержня? При рассуждениях выше мы считали, что пирамидку со стержня 1 надо переставить на стержень 2, а стержень 3 — вспомогательный. В ходе решения задачи при перестановке колец стержни будут все время меняться: если нам изначально надо пирамидку из  $K$  колец переложить с 1-го на 2-й, то, значит, пирамидку из  $K - 1$  колец надо переложить сначала с 1-го на 3-й, а потом с 3-го на 2-й. Например, в процессе перестановки часть пирамидки может оказаться на стержне № 3, а переложить ее надо на № 2. В этом случае дополнительным является № 1. Но существует еще 5 различных вариантов начального и конечного размещений пирамидки. Чтобы их все не расписывать по отдельности, придется придумать формулу, по которой можно получить номер дополнительного стержня, зная номера начального и конечного. Если стержням присвоить номера 1, 2 и 3, то сумма номеров будет 6, таким образом, чтобы узнать номер дополнительного, надо от 6 отнять номера двух других.

```
Program Hanoi_B;  
{глобальные переменные и константы}  
const Z=' Переложите колечко с '  
{Запомним этот текст, чтобы не повторять его}  
Var Kp:Integer;{количество перекладываний}  
Procedure Hanoi(K,N1,N2:Integer);  
{Описание рекурсивной процедуры}  
Var Nd:Integer;  
{Локальная переменная, номер дополнительного стержня}  
Begin Nd:=6-N1-N2;  
  If K=1 Then Writeln(Z,N1, ' на ', N2) {база рекурсии}  
  Else {база рекурсии, простейший случай}  
    If K=2 Then Begin Kp:=Kp+1; Writeln(Kp,Z, N1, ' на ',Nd);  
                  Kp:=Kp+1; Writeln(Kp,Z, N1, ' на ',N2);  
                  Kp:=Kp+1; Writeln(Kp,Z, Nd, ' на ',N2)  
    End
```

```

        Else Begin {рекурсивный шаг}
            Hanoy(K-1,n1,nd);
            Kp:=Kp+1; {Kp - номер операции}
            Writeln(Kp, Z,N1,' на ',N2);
            Hanoy(K-1,Nd,N2);

        End

End;
{Основная программа}
Var K,N1,N2: Integer;
Begin
    Write('Сколько колечек в башне '); Readln(K);
    Write('Номер стержня, на котором колечки ');
                                                Readln(N1);
    Write('Номер стержня, на который надо
                                                переставить '); Readln(N2);
    Kp:=0; {Глобальная переменная для подсчета
                                                количества операций}

    Hanoy(K,N1,N2)
End.

```

**Задание.** Выведите формулу для подсчета числа перекладываний в зависимости от количества колечек.

Следуя инструкции, которую печатает написанная нами программа, вы сможете переставить с одного стержня на другой пирамидку из любого количества колечек. Впрочем, оговоримся, слово «любого» здесь не совсем уместно. Выше мы уже отмечали, что рекурсивные программы используют больше памяти, чем нерекурсивные, и количество необходимой памяти обычно зависит от начальных данных. При некоторых достаточно больших числах (величина зависит от версии транслятора, его настроек) может не хватить памяти в стеке для хранения данных. В этом случае во время выполнения программы возникает ошибка **STACK OVERFLOW ERROR** — переполнение стека. Следует иметь в виду, что ошибка вызвана не тем, что алгоритм в принципе неправильный, а тем, что он (как и любой алгоритм) работает не для всех начальных данных. Для рекурсивных алгоритмов ограничения на входные данные обычно более строгие, чем для нерекурсивных.

Мы написали несколько программ с рекурсией, посмотрим, что у них общего, каковы особенности конструирования рекурсивных алгоритмов.

Когда процедура (функция) сама себя вызывает, происходят действия, аналогичные действиям, описываемым с помощью операторов цикла: повтор определенных операций. Мы помним, что неправильно написанный

оператор цикла может выполняться вечно, никогда не завершиться. С рекурсивным алгоритмом может случиться то же самое. За счет чего завершается рекурсия? Во-первых, при вызове должен изменяться аргумент, во-вторых, в процедуре (функции) должна быть хотя бы одна нерекурсивная ветвь (в описанных выше процедурах и функциях это действия при  $K \leq 0$ ).

Можно сказать, что любое рекурсивное определение состоит из двух частей:

- база рекурсии — описание действий в некотором простейшем случае, это конечные действия, они не содержат рекурсивных вызовов;
- шаг рекурсии — описание действий для перехода от одного значения аргумента к другому.

Вторая часть содержит рекурсивное обращение, именно она делает определение или описание функции рекурсивным, первая часть не менее важна, так как она останавливает рекурсию.

В простейших рекурсивных процедурах и функциях эти две части рекурсивного определения обычно оформляются в виде условного оператора. Для написания процедуры надо подумать, как должно выглядеть рекурсивное определение: в каком (самом простом) случае можно сразу дать ответ, как, зная результат для одного аргумента, вычислить ответ для следующего.

### *Пример 12.9. Найти сумму цифр натурального числа.*

Вспомним, при нерекурсивном решении мы пользуемся циклом: отделяем от числа последнюю цифру (при помощи нахождения остатка от деления на 10 — операция `mod`), а само число все время уменьшаем, удаляя из него последнюю, обработанную цифру (при помощи операции целочисленного деления на 10 — `div`). Сформулируем алгоритм в терминах, удобных для рекурсии.

База рекурсии: проще всего работать с однозначным числом — сумма его цифр равна самому числу.

Шаг рекурсии — это правило, как от одного числа перейти к другому, с меньшим количеством цифр, как при этом посчитать результат. Если мы «вычеркнем» из числа последнюю цифру, получим число с меньшим количеством цифр. Сумма цифр первоначального числа будет равна сумме цифр получившегося плюс «вычеркнутая» цифра.

Осталось записать алгоритм на языке программирования:

```
Function Sumc (N:LongInt):Integer;  
Begin If N<10 Then Sumc:=N {база}  
      Else Sumc:=N mod 10 + Sumc(N div 10)  
        {рекурсивный шаг}  
End;
```

**Пример 12.10.** Найти наибольший общий делитель двух натуральных чисел.

Обычно для решения этой задачи мы раскладываем числа на множители, из общих множителей составляем наибольший общий делитель. Однако такое решение потребует хранения множителей в памяти, т. е. придется заводить массивы. Воспользуемся алгоритмом Евклида, который позволяет не только не запоминать множители, но даже не производить деление. Если числа равны, то наибольший общий делитель равен одному из них (это будет база рекурсии). В противном случае (рекурсивный переход) надо искать наибольший общий делитель других двух чисел: меньшего из заданных и их разности.

Доказательство правильности алгоритма предлагаем провести самостоятельно в качестве математического упражнения. Отметим лишь, что алгоритм обязательно завершится: так как мы производим вычитание, числа, над которыми производятся действия, и их разность все время будут уменьшаться, в конце концов разность уменьшится до нуля.

Посмотрите, как легко реализуется этот алгоритм с помощью рекурсивной функции:

```
Function NOD (a, b: integer): integer;  
begin  
    if a = b then NOD := a    {база}  
        else if a > b then NOD := NOD(a - b, b)  
            else NOD := NOD(a, b - a)  
end;
```

### Задание

1. Выпишите, с какими аргументами будут происходить рекурсивные вызовы NOD, для NOD(70, 110), для NOD(11, 9). Чтобы проверить себя, выполните программу с вызовом этой функции, добавив в тело функции строку с печатью аргументов.
2. Напишите аналогичную функцию без использования рекурсии.

**Пример 12.11.** Строки (цепочки цифр) создаются по следующему правилу. Первая строка состоит из одной цифры 0. Каждая из последующих цепочек создается такими действиями: в очередную строку сначала записывается цифра, на 1 большая последней цифры в предыдущей строке, к ней слева дважды подряд приписывается предыдущая строка.

*Вот первые 4 строки, созданные по этому правилу:*

*(1) 0*

*(2) 001*

*(3) 0010012*

*(4) 001001200100123*

*Требуется ответить на следующие вопросы:*

*а) запишите 7-ю строку последовательности;*

*б) подсчитайте, сколько символов будет в 10-й строке;*

*в) определите, какой символ стоит в 9-й строке на 15-м с конца месте.*

Поступим хитро: ответы на эти вопросы оставим для самостоятельного решения, а для проверки правильности напомним программу, которая будет печатать строку с заданным номером. Но сразу отметим, что задача имеет смысл только для 10 строк, на большее количество не хватит цифр.

Результат работы нашей функции будет типа `STRING` — строка, аргумент функции — целое число, заданный номер строки.

База рекурсии: задаем правило для формирования первой строки, это строка из одного символа `'0'`. Шаг рекурсии: создаем строку с номером  $N$  из строки с номером  $N - 1$ . То есть для работы нам понадобится предыдущая строка. Ее надо получить рекурсивно, с помощью описываемой функции (с аргументом  $N - 1$ ).

Теперь будем строить новую строку по заданному правилу. Здесь нам очень поможет то, что в Паскале для строк определена операция «+», и работает она совсем не так, как для чисел (ведь для строк никакую сумму в арифметическом смысле и нельзя найти!). Эта операция просто присоединяет строки одну к другой, образуя одну «длинную» строку. Здесь нам надо соединить 3 элемента: два раза взять предыдущую строку и прибавить еще один символ (посмотрите в условии правило, по которому он определяется). Чтобы этот символ определить, воспользуемся встроенной функцией «длина строки» `LENGTH`. Также вспомним, что со строкой в Паскале можно работать как с массивом (поскольку она и является массивом символов), т. е. последний символ строки `S` определяется выражением `S[L]`, где `L` — длина этой строки. Чтобы получить символ-цифру на единичку больше, надо единичку прибавлять не к самому символу (он у нас типа `char`, над ним арифметические операции не разрешены), а к его коду, а из кода опять получать символ. Понадобятся встроенные функции `ORD` — определяет код символа и `CHR` — по коду определяет символ. Таким образом, сама функция достаточно простая, алгоритм — просто переписывание условия задачи на Паскале. Все сложности заключены в работе со строками, к рекурсии никакого отношения не имеют.



Получается следующая программа:

```

Program RecStroka;
Function RST(N:Integer):String;{описание рекурсивной
функции}
Var L:Integer; P:Char;
    S:String;{локальные переменные функции}
Begin
    If N=1 Then RST:='0'{база рекурсии}
    Else Begin S:=RST(N-1); {рекурсивный шаг}
        L:=Length(S);
        P:=Chr(1+ORD(S[L]));
        RST:=S+S+P
    End;
End;
{Основная программа}
Var N:Integer;
Begin Write('N='); Readln(n);
    Writeln(RST(n))
End.

```

## Задачи 12.6–12.19. Написать рекурсивную процедуру или функцию

- 12.6. Напишите рекурсивную функцию  $DD(N)$ , которая задается следующей формулой:

$$DD(N) = \begin{cases} 0 & \text{при } N = 0, \\ 1 & \text{при } N = 1 \text{ и } N = 2, \\ N + DD(N - 3) & \text{при } N \geq 3. \end{cases}$$

- 12.7. Напишите рекурсивную функцию, зависящую от натурального числа  $N$ , которая вычисляет произведение всех четных чисел от 1 до  $N$  включительно.
- 12.8. Напишите рекурсивную процедуру, зависящую от натурального числа  $N$ , которая печатает все четные числа от 2 до  $N$  включительно.
- 12.9. Напишите рекурсивную процедуру, на вход которой подается последовательность символов, заканчивающаяся числом 0, а она печатает те числа этой последовательности, которые не делятся на 3 (в любом порядке).
- 12.10. Напишите рекурсивную функцию для вычисления произведения цифр натурального числа.

- 12.11. Напишите рекурсивную процедуру, которая печатает (в любом порядке) цифры натурального числа: а) все; б) только четные; в\*) сначала четные, а затем нечетные.
- 12.12. Напишите рекурсивную функцию, которая определяет количество цифр в натуральном числе.
- 12.13. Напишите рекурсивную процедуру, которая печатает самую старшую (левую) цифру натурального числа.
- 12.14. Напишите программу, которая, обращаясь к рекурсивной функции, вычисляет  $N$ -й член арифметической прогрессии (первый член, разность и  $N$  задается с клавиатуры).
- 12.15. Напишите рекурсивную функцию для возведения натурального числа в натуральную степень.
- 12.16. Напишите программу, которая, обращаясь к рекурсивной функции, помогает решить задачу 12.4 (про сложный процент).
- 12.17. Строки (цепочки букв) создаются по следующему правилу. Первая строка состоит из одной буквы (задается с клавиатуры). Каждая из последующих цепочек создается такими действиями: в начало и конец очередной строки приписывается буква, следующая по алфавиту после последней буквы в предыдущей строке. Вот первые 3 строки, созданные по этому правилу (в качестве первой буквы введена буква О):
- (1) О
  - (2) POP
  - (3) RPOPR

Напишите программу, которая будет печатать строку с заданным номером. В программе нельзя использовать операторы цикла и перехода.

- 12.18\*. Задано натуральное  $N$ . Напечатать все возможные варианты расположения в ряд чисел от 1 до  $N$ . Пример для  $N = 3$ :

1 2 3    1 3 2    2 1 3    2 3 1    3 1 2    3 2 1

- 12.19\*. Быстрая сортировка. Выбирается «опорный» элемент (например, средний элемент в массиве). Остальные элементы переставляются так, чтобы те, которые меньше опорного, оказались слева от него, а остальные — справа. Далее к левой и правой частям массива (если их длина более 1) рекурсивно применяется тот же метод.

# Глава 13

## Работа с файлами

---

Файл — упорядоченный набор данных одного типа, хранящийся в памяти компьютера или на внешних носителях под определенным именем. Понятие «файл» никоим образом не относится только к языку Паскаль и даже к программированию. Этот термин известен вам из общего курса информатики. С файлами вы сталкиваетесь, когда хотите записать песню на плеер, переписать подготовленный в Word доклад на флешку, послать другу фотографию по электронной почте. Программы на Паскале мы тоже храним в файлах.

В языке Паскаль есть средства для работы с файлами. Иногда их очень удобно использовать. В этом случае, например, результаты, полученные одной программой, можно записать в файл и потом использовать как данные для другой программы. Или вместо того, чтобы каждый раз производить ввод большого количества данных с клавиатуры, вводить их из заранее подготовленного файла.

Сразу оговоримся, что именно в работе с файлами различия в разных версиях Паскаля проявляются более всего. Мы коснемся только некоторых способов работы, которые без всяких изменений подойдут для языка Турбо Паскаль и с незначительными изменениями будут применимы для других версий.

Здесь мы познакомимся с работой с **файлами последовательного доступа**. Доступ к информации в таком файле осуществляется строго по очереди. Начать записывать что-то в файл можно только с самого начала, после первой компоненты записывается вторая, потом третья и т. д. Никакие пропуски, пустые места невозможны. Чтение из файла тоже происходит только с самого начала, с первой компоненты. Прочитав ее, можно читать вторую, затем третью. «Перепрыгнуть» через несколько компонент или вернуться на несколько компонент назад нельзя (можно начать с начала).

По своему внутреннему содержанию файлы делятся на **типизированные** и **текстовые**. В типизированном файле хранятся данные одного определенного типа (например, целые числа или массивы вещественных чисел). В текстовом файле хранится «текст», подробнее мы об этом поговорим отдельно.

## Описание файла

Файловый тип всегда надо описывать. Описание типизированного файла выглядит так:

```
File of <тип компонент>
```

А описание текстового файла еще проще: одно слово **Text** (это стандартное имя типа).

Компоненты в типизированном файле могут быть любого типа, кроме файлового. Можно задать файл целых чисел, файл символов, файл массивов и т. п. Про компоненты текстового файла поговорим позже.

При работе с файлами всегда важно помнить, какого типа компоненты в нем хранятся. Записать в файл или прочитать из файла можно только цельную компоненту соответствующего типа (например, из файла массивов можно прочитать только целый массив, а никак не отдельный его элемент).

## Стандартные процедуры и функции для работы с файлами

### Процедура Assign

При работе программы, в которой описана файловая переменная, в оперативной памяти создается файл, с которым ведется работа. После завершения программы этот файл (и вся хранящаяся в нем информация) «пропадает», становится недоступным, как и значения всех других описанных в программе переменных.

Если мы хотим, чтобы файл сохранился, должна быть установлена связь между файловой переменной и реальным файлом (например, на диске). Для этой цели используется процедура **Assign**.

Процедура имеет два параметра:

- первый — файловая переменная (предварительно описанная так, как сказано выше);
- второй — имя реального файла. Имя файла может быть задано в виде строковой константы или в виде переменной строкового типа.

Имя реального файла может содержать не только непосредственно имя, но еще и префикс — путь к файлу. Он записывается по правилам операционной системы. Если префикс не указывается, файл должен находиться в рабочей директории Паскаль-системы (обычно это та директория, в которой находится и сама программа). Посмотреть, какая директория для Паскаль-системы в настоящий момент является рабочей, можно, нажав клавишу **F3** (под окошком с именами файлов выписывается полный путь к ним).

Имя файла может также содержать расширение (до трех букв после точки). Напомним, если файл содержит Паскаль-программу, он должен обязательно иметь расширение **pas** (иначе Паскаль-система не будет его компилировать). Если же мы создаем файл с какой-то информацией, и с ним будет работать только наша программа, то расширение файла мы можем придумать любое.

Паскаль-система при нажатии клавиши **F3** по умолчанию показывает только файлы с расширением **pas**. Чтобы увидеть файлы с другими расширениями, надо в окошке **Name** вместо **\*.pas** набрать **\*.\*** (для просмотра списка всех файлов этой директории) или, например, **\*.txt** (для просмотра списка файлов с расширением **txt**).

Примеры обращения к процедуре **Assign**:

```
Assign(F, S) (здесь F — файловая переменная, S — строковая переменная,  
значением которой является имя файла);  
Assign(T, 'letters.txt');  
Assign(Rt, 'C:\data\Chisla.num').
```

Процедура **Assign** — единственная из изучаемых нами процедур для работы с файлами, где используется имя реального файла. Во всех остальных процедурах используются только файловые переменные. Если работа идет с реальным файлом, процедура **Assign** должна быть применена к файловой переменной до того, как с ней начнут работать другие процедуры. Обычно она пишется в начале программы.

Мы рассмотрим два режима работы с файлами: режим записи и режим чтения. Файл может находиться в режиме записи — в этом случае в него можно только заносить информацию (читать из него нельзя), или в режиме чтения — в этом случае, наоборот, можно только считывать находящиеся в файле компоненты, а записывать новые нельзя.

## Процедура ReWrite

Для перевода файла в режим записи (открытия файла для записи) используется процедура **ReWrite**. У нее один параметр — файловая переменная. Процедура **ReWrite** создает файл с этим именем (если его не было) или стирает из файла все, что там было (если такой файл существовал). После ее работы в файл можно будет записывать компоненты, начиная с первой по счету. Пример: **ReWrite(F)**.

## Процедура ReSet

Для перевода файла в режим чтения (открытия файла для чтения) используется процедура **ReSet**. У нее такой же параметр — файловая переменная. Эта процедура сможет успешно работать, если файл, соответствующий ука-

занной в ней файловой переменной, существует — создан в данной программе или имеется на диске (и связан с файловой переменной процедурой **Assign**). Если соответствующего файла нет, будет зафиксирована ошибка и работа программы прервется. После работы процедуры **ReSet** из файла можно будет читать его компоненты, начиная с первой. Пример: **ReSet (F)**.

## Процедура Close

По окончании работы с файлом его надо закрыть. Для этого служит процедура **Close** (у нее один параметр — файловая переменная). Пример: **Close (F)**.

Использовать процедуру **Close** особенно важно по окончании записи информации в файл. Дело в том, что на самом деле переписывание информации в файл происходит не по одной компоненте, а достаточно большими порциями (некоторое количество компонент сначала накапливается в специальном буфере). Процедура **Close** не только закрывает файл, но и переписывает в файл информацию из буфера (если она там есть). Если по окончании записи в файл не воспользоваться **Close**, часть компонент может быть не записана в файл.

## Функция Eof

И еще одна очень нужная функция для работы с файлами: **Eof**. У нее один аргумент — файловая переменная. Это логическая функция (т. е. она принимает значения **True** или **False**). Ее имя — аббревиатура от слов *End Of File*. Можно считать, что в конце каждого файла стоит специальный значок — маркер конца. Эта функция имеет значение **False**, если в данный момент может быть прочитана компонента файла, и принимает значение **True**, если «видит» маркер конца; в этом случае производить чтение из файла нельзя.

## Запись информации в файл. Чтение из файла

Сначала — о работе с типизированными файлами. Для записи данных в файл и чтения данных из файла используются процедуры **Write** и **Read**, с которыми мы уже знакомы.

## Запись в файл

Чтобы записать что-то в файл (предварительно открытый процедурой **ReWrite**), надо написать **Write(F, <список вывода>)**, где **F** — файловая переменная, **<список вывода>** — выражения, переменные или константы, записанные через запятую. Как можно видеть, для вывода информации на экран мы писали почти то же самое, только не указывали

имя файла. Дело в том, что для компьютера все внешние устройства (экран, клавиатура, принтер и т. п.) являются файлами.

Разница в использовании процедуры **Write** для вывода информации на экран и для записи в типизированный файл состоит в том, что при печати на экран в списке вывода могут быть переменные и значения разных типов из некоторого разрешенного набора (вспомним, напечатать значения типов **Integer**, **Real**, **String** можно, а вот целочисленный массив сразу напечатать нельзя, его можно выводить только по одной компоненте). При записи в типизированный файл в списке вывода могут быть значения только одного типа — того, который использован при описании файла. Зато это может быть любой тип, кроме файлового, в том числе, например, и массив. Процедурой **Writeln** при записи в типизированный файл пользоваться нельзя.

***Пример 13.1, а.** Записать в файл  $N$  целых чисел.*

Оформим решение задачи в виде процедуры (если процедурами пользоваться не хочется, можно текст из тела процедуры перенести в соответствующее место программы — туда, где происходит обращение к процедуре).

```
Type FF=File of Integer;
Procedure ZapF(Var F:FF);
Var N,A,I :Integer;
Begin ReWrite(F);
      Write ('Сколько чисел  '); Readln(N);
      For I:=1 to N do
      Begin   Write(i, '-е число '); Readln(A);
              Write(F,A)
            End;
      Close(F)
End;
Var F:FF;
Begin Assign(F, 'FInt.int');
      Zapf(F);
End.
```

Сначала надо описать новый тип — файл целых чисел. В описании процедуры будет (в разделе «дано») формальный параметр — файл, его надо записать в скобках. Параметры-файлы в заголовках процедур всегда пишутся с **Var**!

В процедуре мы первым делом откроем файл для записи. В процедуре для работы с файлом мы пользуемся только файловой переменной, имя реального файла, его место на диске пока неизвестно, оно будет указано в программе. В цикле запишем нужное количество чисел и закроем файл.

В основной программе воспользуемся процедурой **Assign** для задания имени файла, в который хотим произвести запись, и вызовем описанную выше процедуру записи.

Наша программа ничего не выводит на экран, так что по окончании ее работы на экране ничего нового не появится. Как же проверить, работала ли процедура? Результатом ее работы является появление в рабочей директории файла **FInt.int**. Посмотрите (с помощью Паскаль-системы, проводника и т. п.), появился ли у вас такой файл.

А как же посмотреть, правильные ли числа записаны в файл, как узнать содержимое такого файла? Попробуем открыть его из Паскаль-редактора так же, как открываем программу на Паскале. И что обнаруживаем? Файл есть, но в нем записаны совершенно непонятные символы.

В чем дело? Производя запись в файл, мы сообщили транслятору, что будем вводить целые числа. Компьютер их соответствующим образом кодировал и помещал в файл. Когда мы открываем файл из Паскаль-системы, компьютер считает, что в файле хранится программа, т. е. символы, вот он и переводит записанные в файле коды в символы. Если бы мы создавали файл символов (например, **File of Char**), кодировка при записи и чтении была бы одинаковой, и мы бы прекрасно могли видеть в Паскаль-редакторе содержимое файла. Таким образом, в Паскаль-системе можно просматривать содержимое файлов, при записи которых использовалась «символьная» кодировка (к ним относятся файлы символов, файлы строк и в том числе и текст программы.). Содержимое файлов с другой кодировкой, например файлов чисел, в Паскаль-редакторе читается неправильно.

Так как же все-таки посмотреть содержимое файла, если в нем находятся не символы, а что-то другое? Для этого надо воспользоваться процедурой чтения компонент из файла и напечатать их на экране.

## Чтение из файла

Чтобы прочитать что-то из файла (предварительно открытого процедурой **ReSet**), надо написать в программе **Read(F, <список ввода>)**, где **F** — файловая переменная, а в списке ввода находятся переменные, которым должны быть присвоены значения, прочитанные из файла. Переменные должны быть того же типа, что и файл.

Как видим, чтение из файла похоже на ввод информации с клавиатуры. Про используемые здесь типы данных можно сказать то же, что было сказано про процедуру **Write**. С клавиатуры можно вводить данные разных типов, при условии, что это **Real**, **Integer**, **Char** или **String**; из типизированного файла можно читать данные только одного типа (который объявлен при описании файла), но это может быть любой тип.



Существенное различие между чтением с клавиатуры и чтением из файла состоит в том, что при необходимости ввода значения с клавиатуры программа останавливается и ждет, когда что-либо будет введено. При чтении из файла никакой остановки и ожидания не происходит. Если в файле не содержится необходимого количества компонент (в программе должна выполняться процедура чтения, а компоненты в файле закончились, читать нечего), фиксируется ошибка, работа программы прерывается.

Чтобы такого не происходило, при чтении информации из файла обычно пользуются функцией **Eof**. Перед чтением очередной компоненты проверяют, есть ли она, не закончился ли файл.

Добавим к написанной выше программе процедуру чтения чисел из файла (прочитанные числа выведем на экран).

***Пример 13.1, б.** Записать в файл  $N$  целых чисел. Прочитать записанные числа и вывести их на экран.*

Описание типа и функции записи в файл здесь повторно приводить не будем.

```
Procedure ReadF(Var F:FF);
Var A :Integer;
Begin Reset(F);
    Writeln('Содержимое файла');
    While Not Eof(F) Do
    Begin Read(F, A);
        Write(A:5)
    End;
    Writeln; Writeln('файл окончен');
    Close(F)
End;
Var F:FF;
Begin Assign(F, 'FInt.int');
    Zapf(F);
    ReadF(F);
End.
```

В процедуре чтения один параметр — файловая переменная, которая всегда записывается с **Var**.

Файл сначала открывается для чтения. Хотя в данном случае мы и знаем, сколько в файле компонент (мы сами вводим число  $N$ ), покажем, как производится чтение из файла, если количество компонент неизвестно.

Запись на Паскале **While Not Eof(F) Do** в переводе с английского означает «пока не достигнут конец файла, выполняй» — именно так ее и понимает компьютер.

Если файл **F** пуст, цикл не работает ни разу. Если же в файле есть хотя бы одна компонента, значение функции **Eof (F)=False**, тогда значение **Not Eof (F)=True**, действия цикла выполняются, одна компонента считывается, ее значение выводится на экран. Если прочитанная компонента была последней, значение **Eof (F)** становится **True** и цикл прекращается, иначе он выполняется еще раз.

По окончании работы с файлом его надо закрыть.

В основной программе надо по очереди обратиться к нужным процедурам. Отметим, что главного эффекта нашей программы на экране не видно. Точно такие же результаты на экране можно получить и при записи чисел в массив. Важным здесь является то, что массив по окончании работы программы пропадает, нигде в памяти не сохраняется; чтобы поработать с ним еще раз в другой программе, его надо вводить заново. Файл, соответствующий описанной файловой переменной, останется в памяти не только по окончании работы программы, на его сохранность не повлияет даже выключение компьютера. Он теперь будет в памяти до тех пор, пока мы сами его не удалим. Его содержимое можно прочитать другой программой в любой момент. Например, если из программы 13.1, б убрать обращение к процедуре **ZapF**, запись нового содержимого в файл производиться не будет, на экране распечатается содержимое файла **FInt**, записанное ранее. Вот в этом случае нам и сослужит хорошую службу то, что мы производили чтение из файла с помощью цикла **While** и функции **Eof**, а не с помощью цикла **For I:=1 to N do**. Ведь здесь мы значение переменной **N** уже не знаем.

## Примеры работы с файлами

Решение задач с файлами очень похоже на решение задач с последовательностями, вводимыми с клавиатуры (особенно если количество элементов в последовательности неизвестно, а ее конец помечен каким-либо особым элементом).

Обычно, работая с такой последовательностью, мы обрабатывали ее элементы в процессе чтения и сразу же «забывали». Компоненты файлов можно считывать и использовать по несколько раз. В этом есть их сходство с массивами. Различие состоит в том, что в массиве нам все элементы одинаково доступны, после 5-го можно посмотреть 1-й, потом 10-й и т. п. В файле же в каждый момент времени доступен только один элемент. После открытия файла можно прочитать только 1-ю компоненту, сразу после нее прочитать, например, 5-ю нельзя, надо сначала прочитать 2-ю, 3-ю и 4-ю. А если необходимо вернуться назад (прочитать еще раз уже прочитанную компоненту), надо еще раз открыть файл для чтения (**Reset**) и «добраться» до нужной компоненты.

При работе с типизированными файлами надо внимательно относиться к типу обрабатываемых компонент — он может быть только таким, который указан при описании файла. Записать или попытаться прочитать из файла компоненты разных типов нельзя.

При чтении из файла надо следить за тем, чтобы очередная компонента существовала. Для этого надо либо знать общее количество компонент в файле и количество уже прочитанных, либо пользоваться функцией **Eof**.

***Пример 13.2.** В файле **Letters.txt** записаны символы. Создать файл **Code.num**, в который записать коды этих символов.*

Для того чтобы можно было решить эту задачу, файл **Letters.txt** должен существовать. Как его создать? Можно воспользоваться процедурой из примера 13.1, а, подставив нужное имя файла. Только надо не забыть изменить тип файла и тип переменных. А проще, так как это файл символов, создать его в текстовом редакторе, например в Блокноте (только не в Word!), или в редакторе Паскаль-системы. Символы в файл надо записывать с самого начала без переводов строк и каких-либо разделителей (пробелов, запятых), так как они сами также являются символами и, если вы их напишете в файл, будут обработаны, как и все остальные символы.

В программе надо описать два файла (файловые переменные): заданный (файл символов) и искомый (файл целых чисел). Оба файла переменным ставим в соответствие названия файлов.

Заданный файл открываем для чтения, искомый — для записи. Чтение и запись производим в одном цикле — для всех компонент символьного файла.

```
Program CodLetters;  
Var Fsym : File of Char;  
    Fcod : File Of Integer;  
    Sym  : Char;  
Begin Assign(FSym, 'Letters.txt');  
    Assign(Fcod, 'Code.num');  
    ReSet(FSym); ReWrite(Fcod);  
    While Not Eof(FSym) Do  
        Begin Read(Fsym, Sym);  
            Write(Fcod, Ord(Sym))  
        End  
End.  
End.
```

В этой задаче нет никакого вывода на экран, поэтому результатов ее работы мы на экране не увидим. При правильной работе программы должен образоваться файл **Code.num**. Как посмотреть его содержимое? Мы уже говорили, что, так как это файл чисел, а не символов, из-за несовпадения кодировок просмотреть его с помощью текстового редактора (Блокнота или

Паскаль-редактора) нельзя. Можно воспользоваться программой из примера 13.1, б.

***Пример 13.3.** В файле символов **Letters.txt** переставить символы таким образом, чтобы в нем сначала были все цифры, а потом все остальное.*

В этой задаче файл символьный, значит, и вводить, и просмотреть его можно прямо из Паскаль-редактора. На экран в процессе выполнения программы ничего не выводится.

Файл — не массив, менять в нем символы местами нельзя. Создадим дополнительный файл, в который перепишем символы в нужном порядке, а потом из него всю информацию целиком перепишем в исходный файл.

```
Type FSym=File of Char;
Var F,Dop : FSym;
    H      : Char;
Procedure CopyF(Var FFrom, FTo :FSym);
{Копирование информации из файла FFrom в файл FTo}
Var H:Char;
Begin
    Reset(FFrom); ReWrite(FTo);
    While Not Eof(FFrom) Do
    Begin Read(FFrom,H);
        Write(FTo,H)
    End;
    Close(FFrom); Close(FTo)
End;
Begin Assign(F, 'Letters.txt');
    Assign(Dop, 'dop.txt');
    ReSet(F); ReWrite(Dop);
    While Not Eof(F) Do {Переписывание цифр
                        в дополнительный файл}
        Begin Read(F,H);
            If (H>='0') and (H<='9') Then Write(Dop,H)
        End;
        Close(F);
        ReSet(F);
    {Переписывание в дополнительный файл оставшихся
    символов}
    While Not Eof(F) Do
    Begin Read(F,H);
        If (H<'0') Or (H>'9') Then Write(Dop,H)
    End;
    Close(F); Close(Dop);
    CopyF(Dop, F)
End.
```

Не очень хорошо, что после выполнения программы в памяти остается ненужный файл **Dop. txt**. В Турбо Паскале его можно стереть с помощью специальной процедуры **Erase (F)**. Мы уже говорили, что в разных версиях Паскаля существуют разные возможности для работы с файлами, и учесть здесь их все в наши задачи не входит.

Эту задачу можно было решить и по-другому: просматривать исходный файл один раз, а не два, но зато завести два дополнительных файла. Во время просмотра в первый складывать цифры, а во второй — все остальное. Ну а потом переписать информацию в нужном порядке в исходный файл. *Попробуйте решить задачу этим способом самостоятельно.*

Мы раньше решали задачи такого же типа с массивами, здесь дело обстоит даже проще, не надо следить, чтобы запись новой компоненты производилась на новое место, — в файлах это происходит автоматически. Правда, надо вовремя и правильно открывать и закрывать файлы.

Решая эту задачу, мы узнали важную вещь. У нас нет специальных функций для дописывания информации в файл (будь то добавление элементов в начало, в конец или в середину файла), для перестановки элементов в нем, но эти действия можно выполнить, используя дополнительный файл.

Также в этой задаче мы научились переписывать полностью содержимое одного файла в другой, т. е. копировать файл. Это действие достаточно часто встречается, поэтому мы его оформили в виде процедуры.

**Пример 13.4\*.** *«Перевернуть» файл, т. е. переставить его компоненты так, чтобы они оказались в обратном порядке.*

Как мы уже знаем, переставлять внутри файла элементы нельзя, надо их в нужном порядке записать в дополнительный файл, а потом скопировать обратно.

Как же организовать переписывание? Сначала надо переписать последний элемент, затем предпоследний, затем пред-предпоследний и т. п.

Но ведь двигаться по файлу назад нельзя! Каждый элемент придется «доставать», продвигаясь к нему от начала файла.

Итак, сначала перепишем в дополнительный файл самый последний элемент. Для этого откроем файл для чтения и будем читать из него все элементы с первого по последний в переменную  $H$ . Когда цикл (и файл) закончится, в этой переменной будет находиться последний элемент файла, его и надо записать в дополнительный файл. В этом же цикле подсчитаем количество элементов в файле (переменная  $N$ ).

Теперь надо переписать в дополнительный файл предпоследний элемент. Но мы уже знаем его номер, это  $N - 1$ . Значит, чтобы прочитать из файла именно этот элемент, надо открыть файл, в цикле прочитать  $N - 1$  элементов в одну и ту же переменную, и по окончании цикла в ней будет как раз  $(N - 1)$ -й элемент.

Такие же действия надо проделать для  $(N-2)$ -го,  $(N-3)$ -го и всех остальных элементов до 1-го. Причем для работы с каждым элементом файл надо заново открывать, чтобы читать элементы с начала. Для выполнения всех этих действий организуем внешний цикл от  $N-1$  до 1 (это номера элементов, которые будут считываться из основного файла и переписываться в дополнительный). Таким образом, процедура **ReSet** для основного файла здесь выполнится столько раз, сколько в этом файле элементов.

Остается только скопировать информацию из дополнительного файла обратно в исходный. Эта процедура была описана в предыдущем примере, здесь ее (как и описание файлового типа) приводить не будем.

```
Var F,Dop : FSym;
    H      : Char;
    N,K,I  : Integer;
Begin Assign(F, 'Letters.txt');
    Assign(Dop, 'Dop.txt');
    ReSet(F); ReWrite(Dop);
    N:=0;
    While Not Eof(F) Do
        Begin Read(F,H);
            N:=N+1
        End; {В N теперь количество элементов в файле}
    Write(Dop,H); {Запись в Dop последнего элемента}
    Write(H);
    Close(F);
    For K:=N-1 downto 1 do
        Begin ReSet(F);
            For I:=1 to K do
                Read(F,H); {'промотаем' файл до нужного
                               элемента}
                Write(Dop,H); {Запись в Dop K-го элемента}
                Write(H);
                Close(F)
            End;
        Close(Dop);
        CopyF(Dop, F)
    End.
```

## Задачи 13.1–13.17. Типизированные файлы

*Задачи этого типа те же, что и задачи с массивами. Различия только в том, что обрабатываемая (и получаемая) информация находится не в массиве, а в файле, следовательно, подготовить файлы для работы и просмотреть полученные результаты можно не только во время выполнения программы, но и в любое другое время.*

- 13.1. В файле содержатся сведения о росте каждого ученика в классе. Найти значение самого большого и самого маленького роста и для каждого ученика напечатать: его рост, насколько он выше самого маленького ученика и ниже самого высокого.
- 13.2. В файле находится последовательность целых чисел. Напечатать сначала те, которые делятся на 3, а потом те, которые делятся на 5 (некоторые числа могут быть напечатаны 2 раза).
- 13.3. В файле находится последовательность символов. Напечатать сначала входящие в нее маленькие латинские буквы, а потом большие.
- 13.4. В файле находится последовательность целых чисел. Сначала выписать числа, у которых равные соседи, а потом числа, у которых соседи неравные (первое и последнее числа имеют только одного соседа, поэтому их не рассматривать).
- 13.5. Циклически сдвинуть элементы файла на одну позицию влево.
- 13.6. Задан символьный файл. Заменить в нем все маленькие латинские буквы на соответствующие большие, а все остальные символы — на '0'.
- 13.7. Символы из одного файла переписать в два: в первый переписать цифры, а во второй маленькие латинские буквы. Порядок символов оставить тот же.
- 13.8. Компоненты из одного файла переписать в два: в первый — компоненты, стоящие на нечетных местах, а во второй — компоненты, стоящие на четных местах.
- 13.9. Из одного файла переписать информацию в другой следующим образом: сначала компоненты, стоящие на четных местах, а затем компоненты, стоящие на нечетных местах.
- 13.10\*. Из файла вещественных чисел переписать в другой неотрицательные числа в обратном порядке.
- 13.11\*. Из одного файла целых чисел переписать содержимое в другой следующим образом: чтобы в начале файла оказались все нечетные числа в обратном порядке, а потом четные в прямом.
- 13.12. Из двух файлов переписать информацию в один: сначала все содержимое первого файла, а затем все содержимое второго.
- 13.13. Из двух символьных файлов переписать в третий цифры (сначала все из первого, а потом все из второго).
- 13.14. Из заданного файла создать второй так, что в нем вся информация будет продублирована: два раза подряд записан первый элемент, потом два раза — второй и т. д.
- 13.15. Из заданного файла переписать информацию в другой два раза следующим образом: сначала все элементы в прямом порядке, а затем в обратном.

- 13.16. Сравнить два файла. Два файла будем считать равными, если в них одинаковое количество компонент и каждая компонента первого файла равна соответствующей компоненте второго файла.
- 13.17\*. Отсортировать информацию, находящуюся в файле.

## Текстовые файлы

Особое место в Паскале занимают так называемые текстовые файлы. Хранятся в текстовом файле символы, но от обычного символьного файла (описанного **File of Char**) он отличается тем, что символы могут быть разбиты на порции, строки. Отличается такой файл и от файла строк (**File of String**), в частности, тем, что строки (**String**) в Паскале не могут быть длиннее 255 символов, а в текстовом файле никаких ограничений на длину строки не накладывается.

Главной же особенностью текстовых файлов является то, что в них можно записывать (и соответственно считывать из них) компоненты разных типов (помните, в типизированные файлы можно писать только компоненты того типа, который задан в описании файловой переменной). При этом при записи в файл данные автоматически преобразуются в символьные, а при чтении прочитанные символы автоматически конвертируются в нужный тип (тип переменной, указанный в списке ввода).

Именно это свойство текстовых файлов дает нам возможность выводить на экран и вводить с клавиатуры и числа, и символы. Ведь экран и клавиатура для компьютера «выглядят» как текстовые файлы, процедуры **Write** и **Read** работают с ними по умолчанию, если в качестве первого параметра не указан какой-либо другой файл.

Таким образом, правила работы с любым текстовым файлом такие же, как и правила работы с экраном и клавиатурой: в него можно записывать компоненты типов **Integer**, **Real**, **Boolean**, **Char**, **String**; и можно считывать компоненты всех перечисленных типов, кроме **Boolean**. При записи в текстовый файл информация кодируется, представляется в символьном виде, поэтому содержимое такого файла можно просматривать (и подготавливать) с помощью текстового редактора (в том числе редактора Паскаль-системы или Блокнота).

Напомним, что для описания этих файлов есть стандартный тип **Text** (ни в коем случае нельзя писать **File of Text**, надо только одно слово **Text**). Например, **M: Text**;

Записывая информацию в текстовый файл, можно пользоваться не только процедурой **Write**, но и **Writeln**. Эта процедура после записи в файл компонент списка вывода запишет туда еще и особый, невидимый на экране символ — признак конца строки. При чтении из такого файла можно пользоваться не только процедурой **Read**, но и **Readln**. Процедура **Readln** считывает из файла информацию, начиная от текущей компоненты до при-



знака конца строки (включительно) — таким образом можно считывать строки (например, в переменную типа **String**), признак конца строки при этом в переменную не заносится.

При посимвольном чтении из файла для распознавания особого символа — признака конца строки — служит логическая функция **EOLN(F)** (от англ. *End Of Line*). Она принимает значение **True**, если очередной текущей компонентой файла является признак конца строки.

Например, поместить в текстовый файл **F** строчку **123** можно несколькими способами:

- воспользовавшись текстовым редактором;
- из программы с помощью процедуры **Writeln(F,X)**, где **X** — переменная типа **String** со значением **'123'**;
- тремя процедурами **Write(F,x1)**; **Write(F,x2)**; **Writeln(F,x3)**, где все переменные символьного типа и равны **'1'**, **'2'** и **'3'** соответственно;
- одной процедурой записи в файл целого числа **123** (тип **Integer**):  
**k:=123; Writeln(F,k)**.

Результат во всех случаях будет одинаковый, символьного типа — при записи, если надо, произойдет конвертация.

Результат чтения из текстового файла определяется не только его содержанием, но и типом переменной, в которую оно производится. При чтении строки **'123'** из текстового файла процедурой **Readln(F,X)** переменная **X** станет равна:

- строке из трех символов **'123'**, если **X** типа **String**;
- целому числу **123**, если **X** типа **Integer**;
- символу **'1'**, если **X** типа **Char**.

Во втором случае происходит автоматическое преобразование строкового типа в целый. При этом заметим, что, если такое преобразование невозможно (например, надо в целую переменную прочитать строку **'ab'**), будет зафиксирована ошибка и программа будет прервана.

Как нам может помочь использование такого файла? Во-первых, можно в любой программе заменить вывод на экран выводом в текстовый файл (открывать и закрывать этот файл надо, как и обычные файлы). Тогда на результаты работы своей программы можно любоваться сколь угодно долго, причем в любой день после ее выполнения, а можно их распечатать и повесить на стенку. Более того, эти результаты могут служить входными данными для другой программы (если в ней везде вместо ввода с клавиатуры написать ввод из текстового файла). В этом случае, так как в файл могут быть записаны данные разных типов, их надо очень внимательно считывать в переменные соответствующего типа (чтобы не возникло ошибки и остановки программы).

Текстовые файлы часто используют как вспомогательный инструмент для отладки и проверки программ. Ведь часто при проверке программы приходится много данных вводить с клавиатуры (например, если программа работает с массивом). Можно создать несколько текстовых файлов с разными массивами и вводить данные из них, не тратя время на ввод с клавиатуры. Удобство такой работы состоит еще в том, что текстовый файл можно просматривать и редактировать в Паскаль-редакторе или в Блокноте.

В рамках данного учебника мы не можем изучить все возможности работы с файлами, особенно с текстовыми, однако в качестве примера их использования напомним забавную программку.

***Пример 13.5.** «Сочинение рассказа». Пусть имеются файлы слов. Написать программу, которая будет составлять из этих слов «рассказ».*

Текстовым файлом очень удобно пользоваться как файлом строк, т. е. записывать в него строки длиной не более 255 символов (например, слова) и считывать их процедурой **Readln (F, S)**, где **S** имеет тип **String**.

Предварительно в редакторе создадим три файла: прилагательных, существительных и глаголов. Чтобы слова хорошо сочетались между собой, надо, чтобы все существительные («кот», «мальчик») и прилагательные («умный», «лохматый») были в единственном числе, мужского рода, а все глаголы в третьем лице («бежит», «умывается»).

Напишем программу, которая будет выбирать из каждого файла по одному слову и строить из них предложения: «Умный мальчик бежит», «Лохматый кот умывается». «Рассказ» получится особенно забавным, если слова из файлов выбирать не по очереди, а случайным образом. В Турбо Паскале есть встроенная функция, которая нам в этом поможет. Функция **Random** (подробно описана в гл. 15) с одним параметром (целым числом больше единицы) генерирует случайные числа в диапазоне от 0 до значения параметра минус единица. Так, с помощью **Random (5)** можно получить числа от 0 до 4 в случайном порядке, например 4, 1, 3, 1, 2, 0 и т. д. Чтобы этот порядок был действительно случайным, в начале программы вызывается процедура без параметров **Randomize**.

Сначала в программе опишем необходимые переменные, функции и процедуры.

```
Program WWW; {сочинение рассказа}
Var Fp,Fs,Fg:Text; {файлы прилагательных, существительных
                    и глаголов}
    i,Np,Ns,Ng: Integer; {счетчик количества слов в файлах}
    S          : String[70];
```

Нам понадобится функция для подсчета количества слов в файле. У нее будет один параметр — файловая переменная (которая всегда в заголовке процедуры записывается с **Var**). Опишем локальную переменную **N** —

счетчик количества слов (строк, так как каждое слово у нас находится в отдельной строке).

Откроем файл для чтения. Пока он не закончится, будем считывать из него строки, все строки будем считывать в одну и ту же переменную, потому что хранить их все в памяти нам не надо. После каждого считывания будем увеличивать счетчик на 1.

В конце подпрограммы не забудем найденное значение счетчика присвоить имени функции.

```
Function Kolvo(Var F:Text):Integer; {количество слов  
                                     в файле}  
Var N:Integer;  
Begin N:=0; Reset(F);  
      While not(Eof(F)) do  
        Begin Readln(F); N:=N+1  
      End;  
      Kolvo:=N  
End;
```

Функция **Random** даст нам случайное число. Нам остается извлечь из файла строку с заданным номером. Напишем процедуру, которая будет печатать случайную строку из заданного файла. У этой процедуры два параметра: файловая переменная (файл, из которого надо прочитать слово) и номер этого слова в файле. Найденное слово процедура сразу печатает, больше оно не нужно, поэтому его мы в параметры не выносим.

С файлом каждый раз придется работать с начала, поэтому открываем его для чтения. Далее воспользуемся циклом **For**, чтобы прочитать из файла заданное количество строк. Все их читаем в одну и ту же переменную, поэтому по окончании цикла в переменной *S* будет строка, прочитанная последней, т. е. строка с заданным номером.

При чтении из файла мы здесь пользуемся циклом **For**, а не циклом **While** с функцией **Eof**, как это мы делали раньше. Почему? Дело в том, что перед обращением к функции мы посчитаем, сколько в файле строк, и позаботимся, чтобы номер считываемой строки всегда был не больше количества строк в файле. Таким образом, строка с заданным номером всегда в файле найдется. Остается только ее напечатать.

```
Procedure PrintS(Var F:Text; K:Integer);  
                                     {печать K-й строки}  
Var I : Integer; S : String;  
Begin Reset(F);  
      For I:=1 to K do Readln(F,S);  
      Write(S);  
End;
```

В основной программе вначале надо связать файловые переменные с именами реальных файлов и подсчитать количество слов в каждом файле.

Напишем теперь рассказ из 10 предложений. Каждое предложение состоит из трех слов: прилагательного, существительного и глагола. Будем по очереди их печатать. Воспользуемся для этого описанной выше процедурой печати слова. Ее параметрами будут соответствующая файловая переменная и номер строки в файле. Номер строки будем получать случайно с помощью функции **Random**. Если ее параметром сделать количество строк в файле, значением функции будет число от 0 до этого количества (не включая последнее). Нулевую строку мы считать не будем (так как ее нет), а вот последнюю иногда хотелось бы прочитать. Если результат функции **Random** увеличить на 1, получившиеся числа как раз и будут лежать в нужном диапазоне (от 1 до количества слов).

```
Begin      Randomize;
  Assign (Fp, 'ppp.txt'); Np:=Kolvo (Fp);
  Assign (Fs, 'sss.txt'); Ns:=Kolvo (Fs);
  Assign (Fg, 'ggg.txt'); Ng:=Kolvo (Fg);
  For I:=1 to 10 do
    Begin PrintS (Fp, Random (Np)+1); Write (' ');
          PrintS (Fs, Random (Ns)+1); Write (' ');
          PrintS (Fg, Random (Ng)+1); Writeln ('. ');
    End;
  Close (Fp); Close (Fg); Close (Fs)
End.
```

В следующей задаче поработаем с текстовым файлом, в строках которого содержится информация разного вида.

**Пример 13.6.** *Есть файл — база данных товаров, имеющихсЯ в продуктовом магазине. В каждой строке файла записано: название продукта (буквами без пробелов), производитель продукта (буквами без пробелов), цена продукта (в копейках, целое число  $>0$ ), количество дней, оставшихся до окончания срока годности (целое число  $\geq 0$ ). Все данные разделены ровно одним пробелом.*

*Создать файл со списком товаров для распродажи. В список вносятся товары, до окончания срока реализации которых осталось менее 3 дней. В каждой его строке должно содержаться название товара и новая цена. Цена устанавливается в зависимости от срока годности. Если товар должен быть реализован сегодня (до окончания срока годности осталось 0 дней), цена снижается на 50%, если завтра — на 25%, если в запасе еще 2 дня — на 10%.*

Вот пример текстового файла для такой программы (файл может быть подготовлен в Паскаль-редакторе):

```
Масло Вологда 3250 1
Масло Архангельск 3500 5
Молоко Лианозово 4000 20
Молоко Черкизово 2100 2
Колбаса Черкизово 31900 100
Колбаса Щелково 22000 0
```

Для решения этой задачи понадобятся две файловые переменные для текстовых файлов. Файл с данными надо подготовить очень тщательно, без ошибок, программа будет работать только с файлом, соответствующим условию задачи.

В этой программе самое сложное — ввод данных, так как в одной строке собраны данные разного типа (и все они хранятся в едином символьном представлении). Надо их аккуратно «расшифровать», прочитать в переменные нужного типа.

Можно прочитать строку целиком (оператором **Readln(F, S)**, где **S** — типа **String**), а потом разбивать эту строку на отдельные части. А можно считывать только необходимую информацию, сразу внося ее в разные переменные. Воспользуемся вторым способом.

Сначала надо считать название продукта в строковую переменную *Prod*. Сделать это оператором **Read(Prod)** или **Readln(Prod)** нельзя, так как при этом читаются не только нужные нам буквы (стоящие до пробела), а целая строка. Чтобы «поймать» пробел, придется считывать информацию из строки посимвольно. Будем заносить очередную букву в символьную переменную *C* и добавлять ее к строке (именно так работает «+», если применять его к строкам).

Следующее слово в строке — название производителя продукции — нам не нужно, его надо пропустить. Для этого будем считывать его буквы, пока не встретим пробел. Осталось прочитать два целых числа, считываем их в соответствующие переменные типа **Integer**, причем последнюю переменную прочитываем оператором **Readln**, чтобы следующее чтение производить с новой строки.

При вычислении новой цены продукта следует учесть, что работа ведется с целыми числами (расплачиваться половиной копейки никто не умеет), поэтому надо пользоваться целочисленным делением.

Полученную информацию надо записать в конечный файл. Запись производим оператором **WriteLn**, чтобы информация о каждом продукте содержалась в отдельной строке. В списке для записи у нас содержится информация разного типа: строкового и целого — текстовый файл это позволяет.

```
Program TextFile;{Работа с текстовым файлом}
Var FN,FK: Text;{Начальный и конечный файлы}
    Prod : String[30]; {Название продукта}
    Cena, Srok,i : Integer; {Цена, срок, счетчик}
    c: char;
Begin
    Assign (FN, 'F12_6n.txt');
    Assign (FK, 'F12_6k.txt');
    Reset(FN); Rewrite(Fk);
    I:=0;{Количество продуктов для распродажи}
    While not Eof(FN) do
    Begin Read(FN,c);{Считывается первая буква названия}
        Prod:=c;
        While c<>' ' do {Считываются все буквы названия -
                                до пробела}
            Begin Read(FN,c);
                Prod:=Prod+c {Буквы добавляются к названию}
            End;
        Repeat {Считываются буквы производителя -
                                до пробела}
            Read(FN,c);
        Until c=' ';{Считанная информация не запоминается}
        Read (FN,Cena);
        Readln(FN,Srok);
        {Срок в строке на последнем месте, считываем
                                с переводом строки}

        If Srok<3 then
            Begin I:=I+1;
                Case Srok of
                    0: Cena:=Cena div 2;
                    1: Cena:=(Cena div 4)*3;
                    2: Cena:=(Cena div 10)*9
                End;
                Writeln(Fk,Prod,' -- ',Cena div 100,' руб. ',
                    Cena mod 100,' коп.');
            End
        End;

        Close(FN); Close(FK);
        If I=0 Then Writeln('Товаров для распродажи нет')
            Else Writeln('В списке для распродажи
                                товаров ',I)
    End.

    При обработке приведенного выше входного файла получится вот такой
    выходной файл:
```

Масло -- 24 руб. 36 коп.

Молоко -- 18 руб. 90 коп.

Колбаса -- 110 руб. 0 коп.

## Задачи 13.18–13.30. Текстовые файлы

*Пусть для работы подготовлен текстовый файл, в который записаны слова (некоторые последовательности русских или латинских букв без пробелов). Каждое слово записано на отдельной строке, причем длина его не более 80 символов.*

- 13.18. Подсчитать, сколько слов в данном файле начинаются с заданной буквы.
- 13.19\*. Посчитать, сколько слов в данном файле (слова состоят из маленьких латинских букв) содержат хотя одну гласную букву. (Совет: составить массив из гласных букв: а, е, і, о, u.)
- 13.20. Найти в файле самое длинное слово, если таких несколько, выписать одно из них.
- 13.21. Найти в файле самое длинное слово, если таких несколько, выписать все.
- 13.22. Удалить из файла самое длинное слово.
- 13.23. Удалить из файла все слова, стоящие на нечетных местах.
- 13.24. Имеется некоторый файл слов. С клавиатуры вводится новое слово и номер строки, на которую его надо вставить в файл. Выполнить задачу, если это возможно (в файле есть такая позиция). Для вставки надо «раздвинуть» слова в файле.
- 13.25. Имеется некоторый файл слов. С клавиатуры вводится новое слово (его надо вставить в файл) и слово из файла, после которого надо произвести вставку нового. Выполнить задачу, если это возможно (в файле есть слово, после которого надо произвести вставку). Если заданных элементов (слов, после которых надо произвести вставку) несколько: а) вставить после первого; б) вставить после каждого.
- 13.26. Имеется некоторый файл слов. С клавиатуры вводится номер элемента, который надо удалить. Выполнить задачу, если это возможно (в файле есть элемент с таким номером). При удалении слова в файле надо сдвинуть, никаких пустых строк между элементами быть не должно.
- 13.27. Имеется некоторый файл слов. С клавиатуры вводится слово, которое надо удалить. Выполнить задачу, если это возможно (в файле есть такое слово). Если таких слов несколько: а) удалить первое вхождение; б) удалить все.

*В следующей группе задач в текстовом файле содержится некоторая входная информация, которую предстоит обработать указанным в задаче образом.*

- 13.28. Поработаем с входным файлом — базой данных из примера 13.6.
- 13.28.1. Сделать базу данных для следующего дня. Удалить из нее товары, у которых срок реализации равен 0, а у всех остальных уменьшить его на 1.
  - 13.28.2. Напечатать всех производителей товаров, которые продаются в магазине, причем, если какой-то производитель производит несколько товаров, его надо печатать только один раз.
  - 13.28.3. Найти среднюю цену для каждого наименования товаров, продающихся в магазине.
- 13.29. В файле в каждой строке записана фамилия человека и через пробел дата его рождения в формате dd.mm.yyyy (две цифры отводится на день, две цифры на месяц и четыре на год, данные разделяются между собой точками). С клавиатуры вводится сегодняшняя дата.
- 13.29.1. Выписать фамилии людей, у которых сегодня день рождения, и посчитать, сколько лет им исполняется.
  - 13.29.2. Выписать фамилии людей, у которых день рождения в этом месяце, причем пометить, был ли он уже, есть сегодня или еще будет.
- 13.30. В файле записаны сведения об учениках в классе. В каждой строке содержится фамилия (буквы без пробелов), буква «м» или «д» (мальчик или девочка) и рост в сантиметрах (целое число). Все данные разделены ровно одним пробелом.
- 13.30.1. Найти средний рост всех учеников, отдельно всех девочек и отдельно всех мальчиков.
  - 13.30.2. Напечатать фамилии самых высоких мальчика и девочки.
  - 13.30.3. Создать два новых файла, в которые записать отдельно мальчиков и девочек, причем пол учеников в этих файлах указывать не надо.



## Глава 14

# Комбинированный тип (запись)

---

Мы уже знакомы с типом данных «массив», знаем, что значением переменной этого типа является некоторый набор однотипных элементов. А как быть, если элементы разного типа, но их удобно было бы объединить, дав им общее название? Такие задачи встречаются довольно часто: при описании свойств, характеристик некоторого объекта часто оказывается, что эти характеристики разного типа.

Для описания таких объектов удобен комбинированный тип, или тип «запись». Запись состоит из полей, которые могут быть разных типов, в них помещаются разные характеристики объекта. Для работы с отдельной компонентой типа «запись» надо знать имя всей записи и имя поля.

При описании типа «запись» пишется служебное слово **Record**, после которого перечисляются имена полей (их придумывает программист), для каждого поля указывается его тип, поля отделяются друг от друга точкой с запятой и завершается описание этого сложного типа служебным словом **End**. Порядок полей в записи значения не имеет.

Поясним сказанное на примере. Пусть нам нужно описать объект «Книга», имеющий название, автора, количество страниц, цену. Надо отметить, есть ли в книге иллюстрации. Название и автор — данные строкового типа, количество страниц — целое, а цена (в рублях) — вещественное число, наличие/отсутствие иллюстраций — логического типа. Получаем следующее описание:

```
Var Kniga : Record  Nazv : String;
                   Avtor: String[30];
                   Str  : Integer;
                   Cena : Real;
                   Il   : Boolean
End;
```

Чтобы присвоить этой переменной какое-то значение, надо, например, с помощью операторов присваивания задать значение каждого поля. Для этого пишется название всей записи и, через точку, название поля.

```
Kniga.Nazv := 'Евгений Онегин';
Kniga.Avtor:= 'Пушкин';
Kniga.Str  := 178;
Kniga.Cena:= 59.75;
Kniga.Il   := False;
```

Над записями в целом определена только одна операция — присваивание. Если бы мы выше описали как запись не только переменную *Kniga*, но еще и *Kniga1* (их надо бы было написать через запятую), то оператор **Kniga1:=Kniga** был бы совершенно правильным и, записав после этого **Kniga1.Str:= 262; Kniga1.Cena:= 392.50; Kniga1.I1:= True**, получаем переменную, описывающую книгу с тем же названием и автором, но иллюстрированную (поэтому она содержит больше страниц и дороже стоит).

Переменные типа запись нельзя сравнивать друг с другом, выводить на экран, вводить с клавиатуры. Эти операции можно проделывать только с отдельными полями записи (если их тип это позволяет). С переменной вида **<имя записи>. <имя поля>** можно выполнять все действия, которые разрешены над переменными данного типа (типа поля). Например, в нашем случае можно сравнить количество страниц в книгах (скажем, в условном операторе):

```
If Kniga.Str= Kniga1.Str Then
```

или выписать на экране среднюю цену этих двух книг:

```
Writeln((Kniga.Cena+Kniga1.Cena)/2) .
```

## Работа с типом «запись»

Для демонстрации разных возможностей работы с новым типом создадим базу данных и поработаем с ней. Напишем две программы. Первая будет создавать базу данных и записывать ее в файл. Вторая — отвечать на разные вопросы, касающиеся объектов, находящихся в базе. В качестве объектов возьмем героев популярного мультфильма про семейку Симпсонов.

Нас будут интересовать следующие свойства: имя (строковый тип), возраст (пусть он будет целого типа), является ли герой школьником (логический тип), а также коэффициент интеллекта героя — IQ. Пусть тест IQ они проходили по 4 раза, соответственно имеется 4 числа — баллы, набранные каждым за эти тесты, эти числа будем хранить в массиве из четырех компонент. Получается следующий комбинированный тип:

```
Type Simpson=Record Name:String[10];  
                    Vozr:Integer;  
                    School:Boolean;  
                    IQ:Array[1..4] of Integer  
End;
```

Для работы с этим типом данных можно создать массив таких записей, тогда у нас появилось бы описание **Var MasS=Array[1..N] of Simpson**; но мы не будем работать с массивом, мы поместим наши записи

в файл, поэтому раздел описания переменных будет выглядеть следующим образом:

```
Var F : File of Simpson;  
    Simp : Simpson;  
    C : Char;  
    I, J, N : Integer;
```

Здесь описан файл из записей указанного выше типа, переменная — запись этого типа, а также некоторые другие переменные, которые понадобятся в программе.

***Пример 14.1.** Создать файл записей с характеристиками героев мультфильма «Симпсоны».*

В начале программы поставим в соответствие файловой переменной название файла на диске. Отметим, что название файла, как и расширение, можно выбрать любое; данный файл нельзя будет прочитать с помощью текстового редактора (в том числе и Паскаль-редактора), так как он состоит из объектов сложной структуры — записей. Также введем с клавиатуры количество записей в нашей базе данных.

```
Begin Assign(F, 'Simpson.dat');  
    Rewrite(F);  
    Write ('Количество '); Readln(N);
```

Теперь надо написать цикл, в котором по очереди вводить характеристики всех героев. Порядок ввода не важен. Данные строкового типа (имя) и целого (возраст) вводим с клавиатуры с помощью процедуры **Readln**, данные логического типа (является ли школьником) непосредственно с клавиатуры вводить нельзя. Для определения значения логического поля *School* введем с клавиатуры букву «д» или «н» — значение символьного типа и, воспользовавшись им, определим значение логической переменной.

```
For I:=1 to N do  
Begin Write(i, '   Имя '); Readln (Simp.Name);  
    Write('Возраст '); Readln(Simp.Vozr);  
    Write('Ходит в школу (д/н) '); Readln(C);  
    Simp.School:=C='д';
```

Поле *IQ* у нас является массивом, чтобы ввести его значения (их 4), надо воспользоваться оператором цикла. Индекс элемента массива пишем в квадратных скобках около названия поля *IQ*. Написанный таким образом, этот индекс относится именно к полю. Заметим, что если бы мы хотели хранить записи не в файле, а в массиве, пришлось бы создавать массив записей, тогда одна запись обозначалась бы как **Simp[i]**, в обозначениях отдельных полей точка бы ставилась после индекса всей записи: **Simp[i].Vozr**, а значения поля *IQ* записывались бы вот таким образом: **Simp[i].IQ[j]**.

В нашем же случае (записи храним в файле, поэтому в каждый момент времени работаем только с одной единственной записью) этот цикл будет выглядеть так:

```
For J:=1 to 4 do
  Begin Write(j, 'оценка '); Readln(Simp.IQ[j])
  End;
  Write(F, Simp)
End;
Close(F)
End.
```

На этом мы заканчиваем нашу программу и закрываем файл. На экране никаких сообщений не появляется (мы туда ничего не выводили), зато в памяти появляется файл с именем **Simpson.dat**.

Напомним, что увидеть его содержимое с помощью текстового редактора нельзя, надо написать программу, которая будет считывать из файла записи и печатать значения полей на экране. Вот сейчас этой программой и займемся: она напечатает нам значения введенных записей в виде таблицы (наша база данных), а также ответит на некоторые вопросы, касающиеся героев. Можно ли не делить программу на две части, сделать и ввод, и вывод в одной программе? Конечно, можно. Но тогда совершенно не нужно хранить информацию в файле, каждый раз ее придется вводить заново, что (при большом ее объеме), конечно же, неудобно. К тому же, при таком разделении программ можно создать несколько файлов с разными данными (но одинаковой структуры) и, изменяя имя файла, работать с разными базами данных.

***Пример 14.2.** Напечатать на экране таблицу базы данных о семейке Симпсонов.*

Итак, приступим. Раздел описаний в нашей второй программе будет практически таким же, как и в первой: надо описать точно такую же запись, к тому же понадобятся некоторые дополнительные переменные. Мы здесь этот раздел опустим.

В начале программы поставим в соответствие файловой переменной название нужного файла и приступим к печати нашей базы данных в виде таблицы. Для этого сначала откроем файл для чтения и напечатаем строку заголовка таблицы.

```
Begin Assign(F, 'Simpson.dat');
  ReSet(F);
  Writeln('    Имя    Возраст    Школьник    Оценки');
```

Таблицу будем печатать в цикле, пока не распечатаем всю целиком (заметим, так можно поступать, только если информации в файле мало, если она вся поместится на экране). В теле цикла считываем из файла очередную

запись. Обратите внимание: считываем из файла не отдельные поля, а именно целую запись: ведь у нас файл записей, мы туда именно такие компоненты (цельные записи) и записывали. Значение полей простых типов выводим на экран сразу, для печати элементов массива используем еще один оператор цикла.

```
While Not Eof(F) do
Begin Read(F,Simp);
      Write(Simp.Name:10,Simp.Vozr:10,Simp.School:10);
      For j:=1 to 4 do Write(Simp.IQ[j]:5);
      Writeln
End;  Writeln;
```

Если в предыдущей программе ввести данные, соответствующие пяти главным героям мультфильма, таблица получится следующая (да простят нас истинные знатоки сериала, если мы здесь в чем-то ошиблись):

Имя	Возраст	Школьник	Оценки			
Homer	40	False	0	2	1	40
Marge	38	False	30	50	60	40
Bart	10	True	10	6	20	40
Lisa	8	True	60	90	85	40
Maggie	2	False	2	4	2	10

Покажем некоторые приемы по работе с нашей базой данных. В приведенных ниже задачах будем выписывать только фрагменты программ (они могут функционировать как часть программы из предыдущей задачи), придется только описать некоторые дополнительные переменные.

***Пример 14.3.** Найти самого младшего героя, напечатать его имя и возраст.*

Все сведения у нас хранятся в файле. Что бы мы ни хотели с ними делать, файл всегда надо начинать считывать с начала, т. е. открывать для чтения процедурой **Reset**. (Заметим, что если бы база данных хранилась в массиве, никакие предварительные действия с массивом не потребовались бы, все компоненты массива одинаково доступны в любой момент.) После открытия файла считываем его первую запись, запоминаем имя и возраст (этот возраст пока и будем считать минимальным). Далее просматриваем все записи (пока не кончится файл), сравниваем возраст каждого вновь считанного героя с текущим минимальным. Если надо, меняем значения минимального возраста и имя самого младшего персонажа — работа ничем не отличается от поиска минимума в последовательности (да у нас ведь и есть последовательность, только она не вводится с клавиатуры, а считывается из файла).

```
Reset(F);
Read(F, Simp);
Ss:=Simp.Vozr; Sn:=Simp.Name;
While Not Eof(F) do
Begin Read(F, Simp);
      If Ss>Simp.Vozr Then Begin Ss:=Simp.Vozr;
                              Sn:=Simp.Name
                              End
End;
Writeln('Самый младший - ', Sn, ' Его возраст ', Ss);
Writeln;
```

В нашем случае должен получиться ответ:

```
Самый младший - Maggie      Его возраст  2
```

Заметим, что в данной задаче первоначальное значение переменной *Ss* (минимум) можно было принять равным, например, 1000, а не значению возраста первого персонажа, так как значения всех возрастов в нашем случае явно меньше 1000.

**Пример 14.4.** *Определить, есть ли в базе данных школьники.*

Сначала, конечно же, открываем файл для чтения, иначе его компоненты будут недоступны (ведь мы только что при печати таблицы прочитали весь файл до конца, дальше считывать нечего). Считываем из файла записи, присваивая логической переменной *Sch* значение соответствующего поля считанной записи. Цикл заканчиваем, как только найдем первого школьника, или же просматриваем весь файл, пока не закончится. В первом случае переменная *Sch* будет равна **True** (мы ведь вышли из цикла потому, что нашли школьника), во втором — **False** (это значение поля у последней записи). Остается только по окончании цикла напечатать ответ.

```
Reset(F);
If Not Eof(F) Then Repeat
      Read(F, Simp);
      Sch:=Simp.School
Until Eof(F) or Sch;
Writeln('Наличие школьников ', Sch);
```

При наших данных напечатается:

```
Наличие школьников True
```

**Пример 14.5.** *Если в базе данных есть школьники, определить их средний возраст.*

Наличие школьников у нас определяется значением переменной *Sch*, вычисления надо производить, только если она равна **True**. В этом случае надо опять открыть файл для чтения, чтобы начать работу с первой записи. Обнуляем переменные для подсчета суммы возрастов и количества школьников. В цикле будем изменять значения этих переменных, если значение поля «школьник» в прочитанной записи удовлетворяет нашим условиям. По окончании цикла надо вычислить ответ, поделив найденную сумму на количество. Здесь это можно делать, не опасаясь возникновения деления на ноль, ведь мы будем делать эти вычисления только в том случае, когда в базе данных есть хотя бы один школьник, поэтому знаменатель у нас не может быть равным 0.

```
If Sch Then Begin Reset(F);
                Ss:=0; K:=0;
                While Not Eof(F) do
                Begin Read(F,Simp);
                    If Simp.School Then Begin Ss:=
                                                Ss+Simp.Vozr;
                                                K:=K+1
                                                End
                End;
                Writeln ('Средний возраст школьников ',
                        Ss/K:7:1);
            End;
```

У нас получается:

Средний возраст школьников 9.0

#### **Пример 14.6.** Подсчитать средний IQ каждого персонажа.

И опять начинаем работу с открытия файла. Также вначале печатаем заголовок для таблички: в нее мы выпишем имена героев и их средний IQ.

В цикле после считывания очередной записи приступаем к подсчету результата для данного персонажа. Обнуляем значение суммы и в цикле (ведь в поле *IQ* у нас массив, надо сложить 4 значения) складываем компоненты массива. По окончании этого (вложенного) цикла печатаем ответ: имя персонажа и его средний IQ. Решение этой задачи похоже на работу с матрицей, где каждая строка соответствует очередному персонажу, а столбцы — его баллам.

```
ReSet(F);
Writeln('      Имя      IQ');
While Not Eof(F) do
Begin Read(F,Simp); Sum:=0;
    For j:=1 to 4 do Sum:=Sum+Simp.IQ[j];
    Writeln(Simp.Name:10, Sum/4:10:2);
End;
Writeln;
```

У нас получится:

Имя	IQ
Homer	10.75
Marge	45.00
Bart	19.00
Lisa	68.75
Maggie	4.50

## Задачи 14.1–14.5. Работа с записями

- 14.1. Поработать с уже созданной базой данных «Симпсоны» (лучше внести в нее побольше персонажей).
  - 14.1.1. Напечатать имена всех персонажей старше 30 лет.
  - 14.1.2. Напечатать имена персонажей, у которых хотя бы в одном испытании IQ было набрано менее 10 баллов.
  - 14.1.3. Напечатать имена персонажей, у которых во всех испытаниях IQ набрано не менее 20 баллов.
- 14.2. Создать программу для пункта обмена валют. О каждой валюте известны следующие сведения: название, стоимость покупки и продажи (в рублях), есть ли в наличии. Ввести данные в файл. Напечатать таблицу, соответствующую сведениям, хранящимся в базе данных.
  - 14.2.1. Напечатать названия всех валют, которые имеются в обменном пункте.
  - 14.2.2. Написать названия валют (их может быть несколько), у которых наименьшая разница между стоимостью продажи и покупки.
  - 14.2.3. С клавиатуры вводится некоторое число — количество наличных рублей. Напечатать, сколько каждой валюты (из имеющейся в наличии) можно купить на эти деньги.
- 14.3. Описать тип данных «запись» для хранения дат в виде: день (число от 1 до 31), месяц (число от 1 до 12), год (число от 1 до 3000). Написать следующие программы для работы с переменными этого типа.
  - 14.3.1. Сравнить две даты: какой день наступит раньше?
  - 14.3.2. Вводится некоторая дата. Вычислить, какой датой будет следующий день.
  - 14.3.3. Вводится некоторая дата. Вычислить, какой датой был предыдущий день.
  - 14.3.4. Вводится некоторая дата и количество дней. Вычислить, какая дата наступит через это количество дней.
  - 14.3.5. Вводится две даты. Подсчитать, сколько дней пройдет от одной даты до другой.



- 14.4. У записи два поля. Значениями полей являются слова: слово на английском языке и его перевод на русский язык. Имеется массив таких записей.
- 14.4.1. Реализовать программу «словарь»: вводится слово на английском или русском языке, программа либо отыскивает и печатает перевод слова, либо сообщает, что его нет в словаре.
  - 14.4.2. Реализовать программу «контрольная работа»: выводятся английские слова, человек должен ввести перевод. Компьютер оценивает правильность перевода, подсчитывает количество ошибок.
- 14.5. Создать базу данных, содержащую следующие сведения об учениках вашего класса: имя, фамилия, пол, возраст, оценки по основным предметам.
- 14.5.1. Напечатать отдельно фамилии всех мальчиков и всех девочек.
  - 14.5.2. Найти самых младших мальчика и девочку.
  - 14.5.3. Найти средний балл для каждого ученика. Напечатать фамилии учеников с самым высоким средним баллом.
  - 14.5.4. Найти средний балл отдельно мальчиков и девочек.
  - 14.5.5. Напечатать фамилии мальчиков, которые старше 16 лет, и при этом у них есть хотя бы одна двойка.
  - 14.5.6. Напечатать фамилии отличников.
  - 14.5.7. Отличников больше среди девочек или среди мальчиков?
  - 14.5.8. Напечатать отдельно фамилии мальчиков и девочек, которые учатся без троек.
  - 14.5.9. Есть ли в классе ученики с одинаковыми именами?

## Глава 15

# Некоторые дополнительные процедуры и функции языка Турбо Паскаль

---

Здесь мы расскажем о некоторых процедурах и функциях языка Турбо Паскаль, не входящих в программу ЕГЭ. Соответственно, и задачи, рассматриваемые здесь, не могут встретиться на экзамене. Однако работа с этим материалом позволит вам еще раз повторить пройденное, по-новому взглянуть на некоторые темы, наглядно проследить работу программ.

### Функция **Random**

Иногда бывает, что данные в задаче не нужно вводить, а желательно, чтобы им были присвоены случайные значения из некоторого диапазона (особенно часто это случается в игровых задачах).

В Паскале для создания случайных чисел (точнее, псевдослучайных, так как они получаются по некоторой математической формуле) служит функция **Random**. Это довольно необычная функция: ее значение может быть вещественным (если она используется без параметров) и целым, если используется с параметром.

При использовании без параметров (например, **X:=Random**; **Writeln(Random)**) результатом функции будет случайное вещественное число из диапазона  $[0, 1)$ . Если же функция используется с параметром ( $N$  — целое число), то результатом **Random(N)** будет псевдослучайное целое число из диапазона  $[0, N)$ . Обратите внимание, в обоих случаях значение верхней границы диапазона результатом работы функции не является.

***Пример 15.1.** Напечатать 10 целых случайных чисел от 0 до 4 включительно.*

```
Var I:Integer;
Begin
  For I:=1 to 10 do
    Write(Random(5):4);
  Writeln
End.
```

Результатом работы программы будут числа: 0 0 4 1 1 3 1 0 1 2.

Если же процедуру печати заменить на **Write(Random:6:2)**, напечатаются вещественные числа: 0.00 0.03 0.86 0.20 0.27 0.67 0.32 0.16 0.37 0.43.

Если в той же программе повторим этот цикл (напечатаем еще одну цепочку чисел), первая и вторая цепочки будут разными. Если же просто выполнить программу несколько раз, мы все время будем получать одинаковые цепочки чисел. Это сделано специально, для удобства отладки программ, чтобы в случае ошибки можно было проследить, при каких числах происходит ошибка. По окончании отладки надо в программу (обычно в начало) один раз вставить процедуру **Randomize** (у нее нет параметров), тогда каждый раз будут генерироваться разные последовательности.

Пользуясь функцией **Random** в арифметических выражениях, можно получить случайные числа разнообразных видов:

**10\*Random** — вещественные числа из диапазона [0, 10);

**Random/2** — вещественные числа из диапазона [0, 0.5);

**Random(10)+5** — целые числа из диапазона [5, 14];

**Random(K-N+1)+N** — целые числа из диапазона [N, K];

**2\*Random(45)+10** — двузначные четные числа (т. е. из диапазона [10, 98]).

В учебных задачах функцию генерации случайных чисел иногда используют вместо ввода значений массивов, матриц.

***Пример 15.2.** Заполнить матрицу  $N \times M$  маленькими латинскими буквами случайным образом.*

Здесь единственная сложность — надо задать функции **Random** правильные параметры, чтобы она генерировала только коды маленьких латинских букв (причем всех букв!). Как же это сделать, если кодов мы не знаем? Воспользуемся тем, что для генерации целых чисел из промежутка [N, K] функция должна выглядеть так: **Random(K-N+1)+N**. В этой задаче у нас N (левая граница отрезка) — это код буквы 'a', K (правая граница отрезка) — код буквы 'z'. Подставив эти значения в формулу, мы получим случайное число из нужного отрезка — код случайной буквы. Элементом матрицы является не код, а сама буква, ее можно получить из кода с помощью функции **Chr**.

```
Const N=5;M=4;
Var A : Array [1..N, 1..M] of Char;
    I,J : Integer;
Begin
  Randomize;
  For I:=1 to N do
    For J:=1 to M do
      A[i,j]:=chr(random(ord('z')-ord('a')+1)+ord('a'));
```

```
For I:=1 to N do
Begin
  For J:=1 to M do
    Write (A[i,j]:2);
  Writeln
End
End.
```

Заметим, что для того, чтобы просто напечатать на экране табличку из латинских букв, заводить массив вовсе не нужно. Это вспомогательная задача, предполагается, что с этой матрицей потом будут производиться какие-то действия (например, вывод номеров строк, в которых нет буквы 'а').

## Задачи 15.1–15.11. Работа с генератором случайных чисел

- 15.1. Напечатать фразу «На контрольной я получу оценку ...». Оценка (от 2 до 5) выводится случайным образом.
- 15.2. Напечатать фразу «Я наберу на ЕГЭ ... баллов». Количество баллов вывести случайным образом (в нужных пределах).
- 15.3. В массив из строк поместить имена своих знакомых. Организовать вывод фразы «Мой лучший друг ...», где имя выводится случайным образом из этого массива.
- 15.4. Напечатать 20 выбранных случайным образом больших латинских букв.
- 15.5. Напечатать 5 случайных двузначных чисел с «0» на конце.
- 15.6. Напечатать 10 случайных трехзначных чисел, делящихся на 12.
- 15.7. Некоторый прибор имеет шкалу от 1 до 5 с ценой деления 0,1. Напечатать 10 случайных чисел, которые могут быть показателями этого прибора.
- 15.8. Заполнить матрицу размером  $M * N$  случайными целыми числами из диапазона 1..30 так, чтобы сумма элементов в каждой строке была  $>100$ .
- 15.9. Заполнить матрицу размером  $M * N$  случайными целыми числами так, чтобы сумма элементов в каждой строке была  $=100$ . Попробуйте написать наиболее эффективную программу!
- 15.10\*. Напечатать числа от 1 до 10 в случайном порядке так, чтобы каждое число было выведено ровно 1 раз.
- 15.11\*. В массиве записаны фамилии участников соревнования. Случайным образом организовать очередность выступлений: напечатать список, в порядке которого будут выступать участники (понятно, что выступить должен каждый, причем 1 раз).

## Модуль CRT

Процедуры и функции, с которыми мы далее познакомимся, используются не часто. Они нужны для написания весьма определенных программ, поэтому и собраны в отдельный модуль, в специальную библиотеку, называемую **CRT**.

Модуль **CRT** содержит средства для работы с консолью (то есть с дисплеем и клавиатурой). Аббревиатура **CRT** (*cathode-ray tube*) на русский язык переводится как электронно-лучевая трубка (ЭЛТ). Хотя в настоящее время вместо ЭЛТ в дисплеях используются жидкокристаллические, плазменные и другие устройства визуализации информации, историческое название этого модуля сохранилось.

Чтобы воспользоваться средствами **CRT** (то есть описанными в этом модуле процедурами, функциями, константами и т. п.), следует в начале текста программы сразу за ее заголовком поместить строку:

```
uses crt;
```

## Работа со звуком

Модуль **CRT** дает возможность «озвучить» программу. Заметим здесь, что вся «музыка» воспроизводится не подключенными к компьютеру динамиками (с помощью которых вы слушаете, например, музыкальные диски), а специальным устройством (динамиком), вмонтированным в системный блок. Иногда компьютеры (например, некоторые ноутбуки) этого устройства не имеют, в этом случае предложенные в этом разделе программы выполняться будут (не будут давать никаких ошибок), однако их результата вы не увидите (точнее, не услышите).

### Процедуры Sound, NoSound

Звук характеризуется громкостью, частотой (высотой) и длительностью.

Громкостью звука управлять в Паскале невозможно. Звук определенной частоты можно «создать» с помощью процедуры **Sound**. Она имеет один параметр — целое число — и генерирует звук соответствующей частоты (в герцах). Никаких специальных процедур для регулирования длительности звука в Паскале нет. Звук заданной частоты будет слышаться до тех пор, пока не выполнится следующая процедура **Sound** (тогда звук сменится на другой) или процедура отмены звуков **Nosound** (у нее нет параметров).

Например, процедура **Sound (440)** генерирует звук «ля» первой октавы. Для тех, кто с нотами не знаком, скажем, что примерно так звучит женский голос. Если параметр процедуры взять побольше, получатся более высокие звуки, поменьше — более низкие, например **Sound (165)** дает «ми»

малой октавы — мужской голос. Очень высокие звуки слышатся как неприятный писк, низкие звуки не музыкальны, с их помощью можно моделировать различные шумы: скрип, тархтение. Звуки слишком низкой (ниже 20) и высокой (свыше 20 000) частоты человеческое ухо не различает (у разных людей разные пороги чувствительности).

Длительность можно задавать разными способами, два самых распространенных: управление с клавиатуры и задержка.

**Пример 15.3.** «Сыграть» гамму.

```
uses crt;
Procedure Nota(Ch:Integer);
Begin
    Sound(Ch); Readln;
End;
Begin
    Nota(262); Nota(294); Nota(330);
    Nota(349); Nota(392); Nota(440);
    Nota(494); Nota(524);
    Nosound
End.
```

В этой программе управление длительностью звуков ведется с клавиатуры: при нажатии клавиши **Enter** начинает звучать другой звук. Чтобы не писать каждый раз **Readln**, мы написали процедуру *Nota*, которая воспроизводит звук заданной частоты. При нажатии клавиши ввода процедура завершает свою работу. Если в программе после сыгранной «ноты» идет следующая, процедура вызывается еще раз, и звучит следующая нота. В конце программы стоит обращение к процедуре **NoSound** — она прекращает звучание. Отметим, что звук прекращается благодаря действию именно этой процедуры, а не процедуры **Readln**, как это может показаться в этой программе. **Readln** просто иницииирует вызов следующей процедуры, звук не прекращается, а меняет частоту.

К сожалению, частоты нот в гамме не вычисляются с помощью простой математической формулы, их проще выписывать.

В программах со звуком часто возникают неприятные ситуации, когда программа завершилась аварийно (в результате ошибки или неправильного ввода) во время действия процедуры **Sound**. Звук продолжает звучать даже после завершения программы! Это очень неприятное явление, оно очень раздражает и самого программиста, и окружающих. Прервать звучание можно двумя способами. Во-первых, можно выйти из Паскаля (или в крайнем случае даже выключить компьютер). Во-вторых, можно заставить про-

работать процедуру **NoSound**, запустив программу, в которой она есть, например, такую:

```
uses crt;
Begin
    Nosound
End.
```

Советуем на всякий случай иметь эту нехитрую программку в своей рабочей папке.

## Процедура Delay

В предложенной выше программе мы управляли длительностью звуков с клавиатуры. А как сделать, чтобы звук длился определенное, заранее заданное время? Для этой цели служит процедура **Delay**.

С помощью процедуры **Delay** можно на некоторое время приостановить (задержать) выполнение программы (ее можно использовать в любых программах, не только при работе со звуком). Эта процедура имеет один параметр — целое число. В большинстве справочников говорится, что этот параметр — время в миллисекундах, на которое будет приостановлена программа. На самом деле это не совсем так. На разных компьютерах (разных трансляторах) задержки, задаваемые этой функцией с одним и тем же параметром, могут различаться в десятки, а то и сотни раз. На современных компьютерах часто, чтобы задержка была ощутимой, приходится писать несколько процедур **Delay** подряд — диапазона целых чисел не хватает. Именно поэтому не рекомендуется без особой надобности использовать эту процедуру, особенно многократно. Лучше организовывать приостановку программы, управляемую пользователем. Скажем, для того, чтобы картинка держалась на экране необходимое для ее просмотра время, можно использовать процедуру **Read (Readln)**: выполнение программы будет приостановлено, пока пользователь не нажмет клавишу ввода.

В случаях же, когда использование этой процедуры является необходимым (например, для создания мелодий), рекомендуем в качестве параметров использовать не числа, а буквенные константы. Например, **Delay (ZZ) ; Delay (2\*ZZ)**. Таким образом, в самой программе мы задаем лишь относительные задержки, конкретное же время задается путем определения константы (**Const Z=1000**) и может быть изменено при работе с программой на разных компьютерах.

Перепишем начало нашей программы следующим образом:

```
Const De=XXX;
Procedure Nota(Ch:Integer);
Begin
    Sound(Ch); Delay(De);
End;
```

Мы здесь специально не написали числовое значение константы *De*. Для того чтобы эта программа работала на компьютере, вместо «XXX» надо обязательно поставить какое-нибудь целое число, вы его сами должны подобрать для своего компьютера так, чтобы каждую ноту можно было слышать и чтобы она не звучала слишком долго.

Естественно, нет смысла выписывать на Паскале вручную частоты разных звуков, чтобы сочинять какие-то мелодии. Попробуем использовать в звуковых программах оператор цикла.

**Пример 15.4.** «Сыграть» гамму вверх и вниз.

Для решения этой задачи нам понадобится завести массив, в который надо записать частоты 7 нот октавы. Можно это сделать, как мы это делали раньше: в разделе описаний массив описать, а в начале программы с помощью семи операторов присваивания присвоить его элементам нужные значения. Можно сделать это короче: задать константу-массив. При этом способе в разделе описаний сразу выписывается тип массива и все его элементы (в скобках через запятую).

```
Const Gamma: array [1..7]  
      of Integer=(262,294,330,349,392,440,494);
```

Далее надо описать процедуру *Nota* — в любом ее варианте (длительность задается с клавиатуры или с помощью процедуры задержки). Чтобы программа не только издавала звуки, но еще и печатала частоту текущего звука на экране, можно включить соответствующий оператор вывода в процедуру *Nota*.

Ну а сама программа будет выглядеть в таком случае следующим образом:

```
Var I : Integer;  
Begin  
  For I:=1 to 7 do  
    Nota(Gamma[I]);  
  Nosound;  
  Readln;  
  For I:=7 downto 1 do  
    Nota(Gamma[I]);  
  Nosound;  
End.
```

**Пример 15.5.** «Сыграть» гамму на четырех октавах.

Зная частоты нот первой октавы, частоты нот всех остальных октав можно вычислить. Дело в том, что частоты одинаковых нот соседних октав отличаются друг от друга в 2 раза (это правило музыки, Паскаль здесь совершенно ни при чем). Частоты нот второй октавы в два раза больше час-



тот соответствующих нот первой октавы. Это же выполняется для третьей октавы, только здесь частоты больше частот первой октавы в 4 раза. Чтобы получить ноты октав ниже первой, частоты надо делить на 2, 4 и т. д. Следует иметь в виду, что параметр процедуры **Sound** — целое число, поэтому делить надо нацело (использовать операцию **div**).

Для решения этой задачи удобно использовать описанный в предыдущей задаче константу-массив и процедуру *Nota*:

```
Var I, K : Integer;
Begin
    K:=1;
    Repeat
        For I:=1 to 7 do
            Nota (K*Gamma [I] );
        K:=K*2
    Until K>=8;
    Nosound
End.
```

Мы заставили компьютер генерировать ноты из разных октав, умножая частоты на значение переменной *K*. При  $K = 1$  звучит первая октава, потом  $K$  становится равным 2, звучит вторая октава, при следующем значении  $K = 4$  получаем ноты третьей октавы и последнее  $K = 8$  дает нам четвертую октаву. (Звуки более высоких частот генерировать не стоит — они неприятны, режут слух.)

**Пример 15.6.** Создать «пианино»: при нажатии на клавиши от 1 до 7 должна звучать соответствующая нота.

В этой программе тоже воспользуемся заведенным нами ранее массивом частот нот в октаве. Введенное число надо воспринимать как индекс массива и в качестве параметра процедуры **Sound** задавать компоненту с соответствующим индексом. Проблема здесь в том, что, если с клавиатуры будет введено неправильное число (не входящее в диапазон 1..7), компоненты массива с введенным индексом не найдется, возникнет ошибка. Ошибка также возникнет, если вместо цифры будет случайно введен другой символ.

Чтобы подстраховаться от таких ошибок, будем вводить в переменную типа **Char**. Тогда случайно нажатая вместо цифры буква нам не страшна — прежде чем обращаться к процедуре, проверим, входит ли введенное значение в нужный диапазон. Если входит, преобразуем введенное символьное значение в нужное для работы числовое.

Таким образом, при вводе чисел от 1 до 7 наша программа заставит компьютер издавать звуки — ноты первой октавы (и печатать на экране частоты

этих звуков), при вводе любых других символов она не будет ничего делать. Остается только договориться, какой символ вводить для окончания работы программы. Пусть это будет 0.

Напишем здесь программу полностью (константе *De* надо придать конкретное значение).

```
uses crt;
Const De=XXX;
Const Gamma: array [1..7] of Integer=
                    (262,294,330,349,392,440,494);
Procedure Nota(Ch:Integer);
Begin
    Sound(Ch);
    Writeln(Ch:16);
    Delay(De);
    Nosound
End;
Var I : Integer;
    K : Char;
Begin
    Writeln('При вводе цифр 1-7 будет звучать
                    соответствующая нота');
    Writeln('Конец работы программы - ввод 0');
    Repeat
        Readln(K);
        If (K>='1') and (K<='7') Then
            Nota(Gamma[ord(k)-ord('0')]);
        Until K='0';
    Nosound
End.
```

Ну а как быть, если все же захочется сочинять музыку? Слишком сложно выписывать значения частот разных нот, хотелось бы записывать ноты привычными словами: «до», «ре» и т. п., да и длительности обозначать не миллисекундами, а более привычными значениями.

**Пример 15.7.** Написать программу, читающую из файла запись музыкального произведения (в формате: <нота> пробел <длительность>) и воспроизводящую его.

Для работы этой программы надо подготовить текстовый файл (напомним, это можно сделать в Блокноте или в Паскаль-редакторе) и поместить его в рабочую директорию. В файле в каждой строчке должны быть сведения об очередной ноте: ее название и через пробел длительность (целое число).

С величиной длительности поступим так: пусть у самой короткой ноты в нашем произведении будет длительность 1, тогда у всех нот, которые в 2 раза длиннее ее, будет длительность 2, и так далее. Важно правильно сформировать текстовый файл (не только в плане верного мотива, но и в плане правильного формата заполнения строк: в названиях нот нельзя ошибаться, нота от длительности должна отделяться ровно одним пробелом).

Например, начало песни «Пусть всегда будет солнце» в нашей кодировке будет выглядеть так:

до 1  
до 1  
фа 2

В программе надо будет завести константу-массив частот звуков (назовем его *GInt* — гамма-целые числа) и массив названий нот, названия будем представлять в виде строк (назовем этот массив *GStr* — гамма-строки).

Процедуру *Nota* придется несколько изменить. Теперь у нее будет два параметра: частота и длительность.

И еще одно изменение. Дело в том, что во всех предыдущих программах наша «мелодия» звучала непрерывно (пока звук не прекращался процедурой **NoSound**). Частоты нот менялись (при вызове процедуры **Sound** с новым параметром), звуки плавно перетекали один в другой. При такой организации несколько подряд идущих звуков одинаковой частоты сольются, будут слышны как один длинный звук. Чтобы они слышались как несколько звуков, надо между ними сделать небольшой перерывчик, паузу, т. е. после «исполнения» каждой ноты будем делать небольшую задержку (чуть-чуть «помолчим»).

```
uses crt;
Const De=XXX;
GInt:array[1..7] of Integer =(262, 294, 330, 349,
                             392, 440, 494);
GStr:array[1..7] of String[4]=('до', 'ре', 'ми', 'фа',
                              'соль', 'ля', 'си');
Var F: Text;
    S,N,D: String[80];
    K:Integer;
Procedure Nota(Ch,P:Integer);
Begin
    Sound(Ch);
    Delay(De*P);
    Nosound;
    Delay(De div 100);
End;
```

В основной программе нам сначала надо будет проделать стандартные операции с файлом (написать его имя и открыть для чтения), затем в цикле (до конца файла) будем считывать его строки. Каждая строка несет информацию о названии ноты и ее длительности. Выделим эти части. Так как они разделены пробелом, определим, где он находится, с помощью функции **Pos**. Теперь два раза воспользуемся процедурой **Copy** и в переменную *N* запишем часть строки до пробела (это название ноты), а в переменную *D* — часть строки после пробела (это длительность). Правда, так как мы извлекали длительность из строки, она у нас представлена не в виде числа, а в виде символа. Процедура **Val** преобразует символ *D* в соответствующее ему число *P* (параметр *Er* будет равен 1, если ввиду ошибки преобразование невозможно; в случае правильных записей в файле он всегда будет равен 0, поэтому обращать на него внимание не будем).

Теперь мы имеем название ноты и длительность. Надо найти частоту, соответствующую этой ноте. Это задача, которую мы уже решали, — «найти, на каком месте в массиве стоит заданный элемент». Будем сравнивать нашу ноту со всеми элементами массива названий нот и, как только найдем равный, выйдем из цикла. В переменной *I* при этом у нас будет номер найденного элемента (заметим, что при правильном написании файла элемент найдется всегда). Теперь надо из массива частот звуков элемент с найденным номером подставить в качестве параметра в процедуру *Nota*.

```
Var I,P,Er : Integer;
Begin
  Assign(F,'zvuk.txt');
  Reset(F);
  While not Eof(F) do
  Begin Readln(F,S);
    K:=Pos(' ',S);
    N:=Copy(S,1,K-1);
    D:=Copy(S,K+1,Length(S));
    Val(D,P,Er);
    Writeln(N:8, P:8);
    For I:=1 to 7 do
      If N=GStr[I] Then Break;
    Nota(GInt[I],P)
  End;
  Nosound
End.
```

## Задачи 15.12–15.18. Работа со звуком

- 15.12. Смоделировать полицейскую сирену.
- 15.13. Проиграть звуки разной частоты (от маленькой к большой) в цикле: получится звук, моделирующий взлет ракеты.
- 15.14. Проиграть 5 октав (2 ниже первой, ноты которой описаны, первую и 2 выше нее) вверх и вниз.
- 15.15. Создать «пианино», где с клавиатуры задается октава и нота. Заданная нота озвучивается.
- 15.16. Написать программу со звуковой реакцией на ошибки. Человека просят ввести определенную букву. Если он ошибается, звучит предупреждающий сигнал. (Не забудьте этот сигнал выключить!)
- 15.17. Написать программу, заставляющую человека быстро нажимать нужные клавиши. Пока пользователь не введет нужный символ, звучит «метроном» (звуки через равные промежутки времени).
- 15.18. Изменить программу предыдущего задания: пусть промежутки времени, через которые раздается стук метронома, постепенно уменьшаются.

## Работа с экраном

При работе с дисплеем с помощью модуля **CRT** весь экран разбивается на отдельные строки, а каждая строка — на отдельные позиции, в каждую из которых можно поместить один символ (в том числе и пробел). Таким образом, весь экран состоит из отдельных неделимых прямоугольных элементов. Для каждого элемента можно задать цвет фона (задний план) и цвет символа (передний план). Символ, в случае необходимости, можно сделать мерцающим.

Сколько всего таких прямоугольников умещается на экране? Это зависит от заданного режима работы. По умолчанию обычно устанавливается режим, в котором используется 25 строк по 80 позиций в каждой и 16 цветов. Режим можно изменить с помощью процедуры **TextMode**. Процедура имеет один параметр типа **Word**, с помощью которого и задается режим (в некоторых режимах в строке не 80 позиций, а 40; в некоторых работа ведется не с цветным, а с монохромным экраном). Во всех наших программах вполне достаточно работы с режимом, который устанавливается по умолчанию.

До сих пор мы печатали результаты наших программ белыми буквами на черном фоне (такие цвета приняты в Паскале для вывода результатов по умолчанию). Теперь же, с помощью процедур модуля **CRT**, мы сможем

изменять цвет фона и символов (причем можно задавать цвет фона для каждой позиции и для каждого символа).

Символ может быть изображен с помощью одного из 16 цветов, цвета нумеруются целыми числами от 0 до 15, особое место занимает «Blink» (номер 128) — мерцание. Для закрашивания фона допустимо только 8 цветов с номерами от 0 до 7.

Для более наглядного обозначения цветов в модуле **CRT** описаны константы: английским названиям цветов соответствуют целые числа.

Значение	Константа	Цвет	Значение	Константа	Цвет
0	black	черный	8	darkgray	темно-серый
1	blue	синий	9	lightblue	светло-синий
2	green	зеленый	10	lightgreen	салатовый
3	cyan	сине-зеленый	11	lightcyan	голубой
4	red	красный	12	lightred	светло-красный
5	magenta	малиновый	13	lightmagenta	светло-малиновый
6	brown	коричневый	14	yellow	желтый
7	lightgray	серый	15	white	белый

В программе можно обозначать цвета как числами (это удобно, например, в циклах), так и словами (это делает программу более наглядной, из текста понятно, о каком изображении идет речь).

Процедура Window

Эта процедура задает координаты активного окна относительно левого верхнего угла экрана. Она имеет 4 параметра типа **Byte** (в этом типе разрешены целые значения от 0 до 255):

```
Window(X1,Y1, X2,Y2)
```

Здесь (X1,Y1) — координаты левого верхнего угла окна, а (X2,Y2) — соответственно, правого нижнего.

Термин «активное окно» означает, что отныне (до переопределения координат окна) процедуры работы с экраном будут работать только с этим окном (только в него будет производиться вывод текста, только в нем будет

меняться цвет, координаты курсора будут отмеряться от его левого верхнего угла) (рис. 15.1).

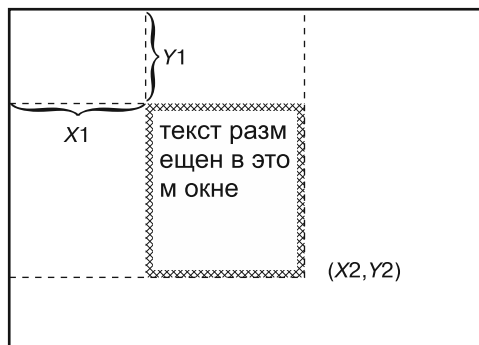


Рис. 15.1

На параметры этой процедуры накладываются достаточно жесткие ограничения: они не должны выходить за границы экрана (стандартный экран  $80 \times 25$ ), окно должно иметь длину и ширину не меньше 1, т. е.  $0 < X1 \leq X2 < 81$ ;  $0 < Y1 \leq Y2 < 26$ . Если хотя бы один из параметров не удовлетворяет этим условиям, процедура не работает, т. е. новое активное окно не устанавливается, однако никаких сообщений об этом не выдается (ошибка не фиксируется, программа не прерывается). В связи с этим надо внимательно следить за параметрами. Для того чтобы активным окном снова стал полный экран (как в начале работы любой программы), надо вызвать процедуру **Window(1,1, 80,25)**.

Отметим сразу, что само по себе обращение к процедуре **Window** не приводит к каким-либо видимым изменениям картинки на экране (все линии, изображенные на приведенном здесь чертеже — невидимые), понять, что процедура проработала, можно только по дальнейшим действиям других процедур.

Результаты работы процедуры **Window** можно увидеть, если вывести после нее что-либо на экран процедурой **Write**. Текст начнет печататься с левого верхнего угла окна, дойдя до его правой границы — прервется, перейдет на следующую строку. Когда будет заполнена нижняя строка окна, будет осуществлена «прокрутка» содержимого окна.

Более наглядно можно продемонстрировать работу процедуры **Window**, совмещая ее с процедурами изменения цвета и очистки экрана.

## Изменение цвета фона и цвета текста (**TextBackGround** и **TextColor**)

Процедура **TextColor** устанавливает цвет выводимых (после нее) символов. У нее один параметр — номер цвета (или соответствующая константа) типа **Byte**.

Как мы уже говорили, цветов для текста — 16, они нумеруются от 0 до 15. Если при обращении задать параметр больше 15, ошибки не происходит, в качестве цвета автоматически подставляется остаток от деления параметра на 16.

Примерно так же работает процедура **TextBackGround**, устанавливающая цвет фона, на котором будут выводиться символы. Разница в том, что цветом фона может быть не любой цвет из 16, а только первые 8 (в случае неверного задания этого параметра пересчет идет по модулю 8).

После обращения к этой процедуре никаких видимых изменений на экране не происходит, их можно заметить только после работы, например, процедур вывода. Символы будут печататься заданным цветом текста и на фоне заданного цвета, т. е. в тех позициях, где выводятся символы, фон будет того цвета, который определен процедурой **TextBackGround**, остальные позиции окна останутся прежнего цвета.

## Очистка экрана **ClrScr**

Чтобы «закрасить» объявленным цветом фона все окно целиком, надо воспользоваться процедурой очистки экрана **ClrScr**. Эта процедура очищает экран или активное окно (если оно задано). Параметров она не имеет. При очистке экран (окно) заполняется текущим цветом фона, а курсор перемещается в левый верхний угол экрана (окна).

### *Важные замечания*

1. Если по какой-либо причине (например, из-за ошибки в программе) цвет фона совпадет с цветом выводимых букв, никакого текста на экране видно не будет, эффект будет тот же, как если бы вывод вообще не производился.
2. По окончании Паскаль-программы первоначально задаваемые по умолчанию цвета фона и текста не восстанавливаются, остаются такими, какими были в последний раз заданы в программе. В следующей программе, выполняемой в этом сеансе работы, при выводе будут использоваться именно они, из-за чего возможны различные накладки. Поэтому рекомендуется в конце каждой программы восстанавливать стандартные цвета, а в начале своей программы задавать нужные цвета фона и текста.

Окна на экране накладываются друг на друга как в аппликации, причем рисуются они очень быстро, незаметно для глаза, поэтому, если из-за ошибки в программе на экране случайно нарисовалось слишком большое окно, закрывшее первоначальную картинку, эффект будет такой, как если бы до этого на экран ничего и не выводилось.



**Пример 15.8.** Напечатать все латинские буквы следующим образом: каждая буква окрашена в свой цвет и стоит на фоне, также имеющем оригинальную окраску.

Сначала зададим цвет экрана — черный. Цвет фона для наших букв будем изменять по порядку от 0 до 7, а цвет буквы зададим случайным образом. При этом будем следить, чтобы цвет фона не совпал с цветом буквы (в этом случае мы букву не увидим). Если такое произошло, цвет изображения надо изменить. Сделаем это с помощью цикла **Repeat-Until**, в котором будем случайным образом подбирать цвет изображения до тех пор, пока оно не станет неравным цвету фона.

Цвет фона каждый раз увеличиваем на 1. Конечно, так как букв 26, цвет фона у нас очень скоро превысит число 7, однако напомним, что ошибки не произойдет, для задания цвета будет браться число по модулю 8.

```
Uses CRT;
Var L:Char;
    CL,CF:Integer;{цвет буквы и цвет фона}
Begin
    TextBackGround(Black);  ClrScr; Randomize;
    CF:=0;
    For L:='A' to 'Z' do
        Begin
            CF:=CF+1;
            Repeat
                CL:=random(16);
            Until CF<>CL;
            TextBackGround(CF); TextColor(CL);
            Write(L:3) ;
        End
    End.
End.
```

В этой программе мы пользуемся тем, что процедура **TextBackGround** окрашивает заданным цветом только тот участок экрана (активного окна), куда непосредственно производится вывод. Именно поэтому на экране получаются разноцветные окошки без использования процедуры **Window**.

## Рисование «прямоугольника»

Таким образом, вызвав три процедуры: задать окно, задать цвет фона, очистить окно, мы получим на экране прямоугольник заданного размера и цвета. Оказывается, это довольно мощный инструмент. Изображая на экране прямоугольники разных цветов и размеров, можно создавать изображения, движущиеся картинки, моделировать мультипликацию.

**Пример 15.9.** Нарисовать на экране прямоугольную «мишень»: вложенные друг в друга прямоугольники разных цветов.

Для решения задачи сначала надо обязательно закрасить экран цветом, отличным от цвета первого прямоугольника (мы выбрали для экрана черный цвет, это вообще очень удобно, так как у него номер 0). Далее в цикле изображаем прямоугольники разного цвета, каждый раз уменьшая размер.

```
Uses CRT;
Var I,N:Integer;
    X1,Y1,X2,Y2:Integer; {координаты первого
                           (самого большого) прямоугольника}
Begin
    X1:=15; Y1:=2; X2:=65;Y2:=25;
    TextBackGround(Black); ClrScr;
    Write('Введите количество прямоугольников
                                                (от 2 до 15) ');
    Readln(N);
    For I:=1 to N do
        Begin Window(X1+(I-1)*2, Y1+(I-1),X2-(I-1)*2,
                                                              Y2-(I-1));
               TextBackGround(I+2);
               ClrScr
        End
    End.
End.
```

При рисовании прямоугольников часто бывает удобно пользоваться следующей процедурой:

```
Procedure Prym(X1,Y1, X2,Y2, Cf : Integer);
Begin
    Window(X1,Y1,X2,Y2);
    TextBackGround(Cf);
    ClrScr
End;
```

В параметрах этой процедуры первые 4 числа — координаты окна, последнее — его цвет. Процедура нарисует в нужном месте прямоугольник заданного цвета. Из нескольких прямоугольников можно создать рисунок. Следует учитывать, что рисунок будет создаваться по принципу аппликации: если прямоугольник, нарисованный позже, накладывается на часть нарисованного раньше, последняя пропадает.

Если в эту процедуру вставить еще параметры — цвет текста и строку, и два оператора — задание цвета текста и печать строки, рисунок будет с подписью. Текст будет выводиться, начиная с первой позиции соответствующего окна.

*Пример 15.10. Нарисовать домик.*

Большими прямоугольниками изобразим стену и крышу, маленьким — трубу, совсем маленькими — в цикле — нарисуем дым. Чтобы домик оказался посередине экрана, будем задавать его координаты  $X$  относительно середины экрана ( **$X0:=40$** ).

Опустим здесь раздел описаний, приведем только основную программу.

```
Begin
  X0:=40;
  TextBackGround (Black);  ClrScr;
  Pym(X0-20,12,X0+20,24,Brown); {стена}
  Pym(X0-25,10,X0+25,11,Red);  {крыша}
  Pym(X0+15,7,X0+17,9,Blue);   {труба}
  X:=X0+15; Y:=6;
  For I:=1 to 5 do
    Begin
      Pym(X-1,Y,X,Y,LightGray); {дым}
      X:=X-5;
      Y:=Y-1
    End
  End.
```

**Мультипликация**

Если изображения на экране плавно заменять одно другим, получится эффект движения — мультипликация.

*Пример 15.11. Нарисовать движущийся прямоугольник.*

Можно просто рисовать прямоугольник, потом стирать его (для этого надо ту же картинку нарисовать цветом фона) и рисовать новый с небольшим смещением. Однако такой рисунок будет «прыгать», мерцать. Эффект «хорошего», плавного движения возникает, когда перерисовывается лишь часть объекта. Поэтому поступим следующим образом.

Нарисуем на черном фоне красный прямоугольник, а затем будем справа пририсовывать к нему красный вертикальный отрезок (по длине равный вертикальной стороне прямоугольника), а слева будем такой же отрезок стирать (т. е. рисовать на месте красного — черный) — возникнет эффект движения прямоугольника слева направо. При этом скорость зависит от быстройдействия компьютера и может оказаться настолько велика, что мы не успеем заметить процесс движения — только начальное и конечное положения объекта. В этом случае движение необходимо каким-либо образом замедлить. В приводимой ниже программе замедление производится за счет последнего оператора (**Readln**) — для продолжения дальнейшей работы программа ждет нажатия клавиши ввода. Соответственно, скорость движения регулируется с клавиатуры — зависит от скорости нажатия **Enter**.

```
Uses CRT;
Var x,y:Integer;{начальные координаты прямоугольника}
    a,b:Integer;{длина и ширина прямоугольника}
    I:Integer;
Procedure Prym (X1,Y1,X2,Y2,C:Integer);
Begin Window(X1,Y1,X2,Y2); TextBackGround(C); ClrScr End;
Begin
  X:=10; Y:=15; A:=20; B:=7;
  TextBackGround (Black); ClrScr;
  Prym (X,Y,X+A,Y+B,Red);
  For I:=X to 80-A-1 do
  Begin
    Prym(I,Y,I,Y+B,Black);{стирание}
    Prym(I+A,Y,I+A,Y+B,Red);{рисование}
    Readln {задержка}
  End
End.
```

## Управление положением курсора (GoToXY)

Процедуры **Write** (**Writeln**) производят вывод информации последовательно, начиная с левого верхнего края окна слева направо, сверху вниз. Конечно, если хочется начать ввод информации в другом месте, например в середине экрана, можно напечатать несколько пустых строк (чтобы сместиться вниз), несколько пробелов (чтобы сместиться вправо), но это не очень удобно. А если хочется выводить текст каким-либо необычным способом (справа налево, сверху вниз)? Для этого можно воспользоваться процедурой, которая устанавливает курсор в заданное место на экране — **Gotoxy** (так и пишется в одно слово, это не команда, а имя процедуры).

Процедура имеет два параметра (целые числа) — координаты нового положения курсора на экране. Первая координата — смещение по горизонтали, вторая — по вертикали. Отсчет производится от левого верхнего угла текущего окна (или экрана, если никакое окно не объявлялось). После применения этой процедуры курсор смещается в заданную позицию, и последующий ввод и вывод производятся, начиная с указанного места окна.

У процедуры есть одно неприятное свойство. Если какая-то координата выходит за границы окна (экрана), никаких сообщений об ошибке не выдается, в большинстве случаев работа процедуры игнорируется, однако возможны ошибки (передвижение курсора в некоторую позицию, появление на экране посторонних символов). Иногда после такой ошибки посторонние символы появляются на экране не только во время работы программы, но и во время ее редактирования. В этом случае надо выйти из Паскаль-системы и снова войти в нее: неправильные символы исчезнут (до выполнения очередной программы с ошибкой подобного типа).

**Пример 15.12.** *Смоделировать движение бильярдного шарика по столу (шарик отскакивает от бортиков по правилу «угол падения равен углу отражения»). Действием силы трения пренебречь.*

Сначала зададим начальное положение шарика и введем направление начального движения. Движение шарика можно разбить на две составляющих: по горизонтали и по вертикали (вспомните векторы из геометрии!), которые мы назовем  $Dx$  и  $Dy$ .

Шарик можно изображать точкой, причем можно написать программу так, чтобы, двигаясь по экрану, он оставлял за собой «след» в виде точек. Можно изображать шарик пробелом, по экрану будет перемещаться курсор, никаких следов оставаться не будет.

Отообразим 200 шагов шарика. Для этого в цикле устанавливаем курсор на нужную позицию и вычисляем новое положение шарика. Если получается, что горизонтальная координата не помещается на экране (у нас в программе выбраны границы чуть меньше экрана: 2 и 77), значит, шарик «докатился» до бортика и должен катиться обратно, смещение по горизонтали поменяет знак. Так же изменяется смещение по вертикали.

Понятно, что если в этой программе не сделать задержку, никакого движения увидеть не удастся. У нас задержка сделана с помощью процедуры ввода **Readln**, но можно воспользоваться и **Delay**.

```
Uses CRT;
Var X,Y,Dx,Dy,I: Integer;
Begin
  TextBackGround (Black);  ClrScr;
  X:=5; Y:=5;
  Write ('<--> '); Readln(Dx);
  Write (' | '); Readln(Dy);
  For I:=1 to 200 do
  Begin  Gotoxy(X,Y); Write('.');
        X:=X+Dx; If (X<=2) or (X>=77) Then Dx:=-Dx;
        Y:=Y+Dy; If (Y<=2) or (Y>=24) Then Dy:=-Dy;
        Readln;
  End
End.
```

**Пример 15.13.** *Напечатать на экране сообщение «бегущей строкой».*

Если напечатать на экране строку, а затем стирать ее и печатать вновь с некоторым смещением (обычно влево или вправо), получим эффект движения строки по экрану. Однако мы уже говорили, что в этом случае изображение будет «прыгать». Лучше организовать смещение строки по отдельным буквам.

Сначала сдвинем влево первую букву напечатанной строки. Для этого достаточно на одну позицию левее, чем печаталась строка, напечатать ее первую букву и пробел (этот пробел как раз и сотрет ранее напечатанную

букву). Получится строка, где первая буква уже стоит на новом месте, а после нее — через пробел — остальные буквы. Теперь точно так же сдвигаем вторую букву (ведь место для нее у нас есть — пробел), за ней третью и т. п. Выполнив столько шагов, сколько букв в строке, мы сдвинем всю строку на 1 позицию влево. Ну а для того, чтобы произвести сдвиг на  $N$  позиций, надо произвести эти действия в цикле  $N$  раз.

```
Uses Crt;
Const De=XXX;
Var S:String[60];
    X0,I,J:Integer;
Begin
  ClrScr; TextBackGround(Blue); TextColor(White);
  S:='Привет всем участникам мероприятия!';
  X0:=40;
  ClrScr;
  Gotoxy(x0,10); Writeln(S); {Начальная печать строки}
  For I:=X0 Downto 1 do {Сдвиг всей строки к левому
                                                                краю экрана}
    Begin For J:=1 to Length(S) do
      {Сдвиг каждой буквы на одну позицию влево}
      Begin Gotoxy(I+J-1,10); Writeln(S[J], ' ');
                                                                Delay(De)
      End
    End
  End
End.
```

## Задачи 15.19–15.42. Работа с экраном

- 15.19. Нарисовать на экране какой-либо «рисунок» из прямоугольников. При этом пусть каждая деталь рисунка будет подписана. (Удобно в процедуру рисования прямоугольника вставить еще и печать заданного текста в нем. Следите, чтобы цвет текста не совпал с цветом фона!)
- 15.20. Нарисовать «пирамидку»: разноцветные прямоугольники стоят друг на друге. Высота у них одинаковая, а ширина уменьшается от нижнего к верхнему.
- 15.21. Организовать смену экранного фона по нажатию клавиши ввода.
- 15.22. Закрасить экран горизонтальными или вертикальными полосками. Количество полосок и количество разных цветов задается с клавиатуры. Надо рассчитать ширину полоски и организовать правильную смену цветов.
- 15.23. Нарисовать на экране двухцветную «шахматную доску»  $M * N$  клеток. Величина клеток зависит от введенных значений  $M$  и  $N$ .

- 15.24. Нарисовать на экране  $K$ -цветную «шахматную доску»  $M * N$  клеток. Величина клеток зависит от введенных значений  $M$  и  $N$ .  $K$  тоже вводится с клавиатуры.
- 15.25. Прямоугольник движется вправо, доходит до границы экрана и движется влево — до левой границы.
- 15.26. Прямоугольник «падает» вниз. Долетев до границы экрана, он «разбивается» (исчезает) со стуком.
- 15.27. Два прямоугольника движутся навстречу друг другу. Когда они сталкиваются, происходит «взрыв»: экран окрашивается в их цвет, раздается соответствующий звук.
- 15.28. Нарисовать «змейку», которая движется по периметру экрана (заворачивает, дойдя до границы).
- 15.29. На черном фоне вывести какой-нибудь текст всеми возможными цветами (кроме черного). После нажатия клавиши ввода поменять фон и проделать то же самое уже на новом фоне (следить, чтобы цвет фона не совпал с цветом изображения).
- 15.30. Пользователь вводит символ и номера цветов фона и изображения. Символ печатается заданным цветом на заданном фоне.
- 15.31. С клавиатуры вводится фраза. Напечатать ее какими-либо двумя цветами (буквы, стоящие на четных местах, печатать одним цветом, соседние — другим).
- 15.32. Вывести на экран целочисленную матрицу, печатая отрицательные числа синим цветом, положительные — красным, ноль — зеленым.
- 15.33. Написать программу, которая помогает подобрать хорошие сочетания цвет–фон. Рисовать прямоугольники всех возможных цветов, в них выводить разным цветом текст. Пользователь оценивает цвета, вводя соответствующие числа. Цветовые сочетания, получившие наиболее высокую оценку, запоминаются и в конце программы выводятся.
- 15.34. С клавиатуры вводится фраза из нескольких слов. Напечатать каждое слово своим цветом.
- 15.35. Заполнить весь экран, напечатав на нем много раз одну и ту же фразу (например, «Учись хорошо!») разными цветами.
- 15.36. Нарисовать «звездное небо»: расположите на экране случайным образом точки (при желании «звезды» можно сделать разных цветов).
- 15.37. Напечатать все цифры на экране: а) снизу вверх; б) справа налево.
- 15.38. Напечатать заданную фразу «сверху вниз».
- 15.39. Напечатать все латинские буквы «ходом быка» (пусть очередная буква появляется при нажатии пробела) в поле заданного размера.
- 15.40. Организовать печать символов при вводе «снизу вверх» (после ввода надо переставлять курсор в нужную позицию).
- 15.41. Напечатать символы по диагоналям экрана.
- 15.42. Напечатать символы по краям экрана (создав рамку вокруг него).

## Работа с клавиатурой

Сначала присмотримся к клавиатуре и разберем, как происходит ввод информации. При нажатии клавиши вырабатывается некоторый соответствующий ей код. Коды нажатых на клавиатуре клавиш запоминаются в специальной буферной памяти и по запросу центрального процессора передаются ему для обработки в том порядке, в каком они туда поступали.

На клавиатуре расположено несколько типов клавиш:

- алфавитно-цифровые — это те клавиши, каждой из которых соответствует какой-либо символ (часто не один), этот символ (пробел в том числе) может быть отображен на экране, принтере;
- функциональные (**F1 — F12**) и управляющие клавиши (**Esc**, **Tab**, **Enter**, **Backspace**, клавиши управления курсором — «стрелочки»). Этим клавишам не соответствуют никакие отображаемые на экране символы, они служат для управления движением курсора и других функций;
- клавиши смещения (**Shift**, **Alt**, **Ctrl**), которые обычно сами по себе не используются, а предназначены для расширения возможностей клавиатуры. Одновременное нажатие клавиши смещения и алфавитно-цифровой клавиши приводит к выработке другого кода;
- клавиши-переключатели (**Ins**, **CapsLock** и т. п.), имеющие специальное назначение.

Число клавиш на современной клавиатуре может быть около 100, но это не значит, что, нажимая различные клавиши, можно получить всего 100 различных кодов. Ведь, нажимая «буквенную» клавишу в сочетании с клавишами смещения, переключателями, можно напечатать русскую и латинскую, большую и маленькую буквы. Таким образом, можно получить порядка 400 различных кодов. Исторически так сложилось, что клавиатурой вырабатываются коды только от 0 до 255. Что же делать, чтобы всем различным сочетаниям клавиш хватало кодов? Из этой ситуации выходят таким образом: есть клавиши (их большинство), нажатие которых дает «обычный» код — число от 1 до 255, а есть и такие, которым соответствуют коды из так называемого расширенного набора: вырабатывается два числа, первое всегда 0, а второе в пределах от 0 до 255.

К клавишам, вырабатывающим код из одного числа, относятся все алфавитно-цифровые (неважно, нажимается такая клавиша одна или в сочетании с клавишей смещения). С кодами символов мы уже знакомы (напомним, код символа выдает функция **Ord**), именно такой код и вырабатывается при нажатии клавиши. Клавиши **Esc**, **Tab**, **Enter**, **Backspace** также вырабатывают код из одного числа. Функциональные клавиши, клавиши управления курсором (стрелочки) и некоторые другие вырабатывают коды из расширенного набора.



До сих пор для ввода информации с клавиатуры мы пользовались стандартными процедурами ввода **Read** и **Readln**. Они удобны только для ввода алфавитно-цифровой информации, с их помощью можно вводить символы, строки, целые и вещественные числа. Однако, пользуясь ими, невозможно распознать, например, клавиши управления курсором, функциональные. Эти процедуры работают так, что реакция на клавиши **Backspace** и **Enter** задана изначально: **Backspace** стирает введенный символ, а **Enter** служит признаком окончания ввода. К тому же ввод всегда сопровождается печатью вводимых символов на экране («эхо»).

Существенно расширить возможности работы Паскаль-программы с клавиатурой помогают процедуры и функции модуля **CRT Keypressed** и **Readkey**.

### Функция Keypressed

Логическая функция **Keypressed** (без параметров) имеет результат **True**, если после последнего чтения из буфера или с момента начала программы была нажата какая-нибудь клавиша. То есть она выдает **True**, если буфер клавиатуры не пуст, и значение **False**, если он пуст (т. е. с момента начала программы ни одна клавиша не нажималась или буфер прочитан процедурами **Read/Readln** или функцией **Readkey**).

Сама по себе эта функция не задерживает работу программы, не ждет нажатия клавиши, но она часто используется для того, чтобы приостановить работу программы до нажатия пользователем любой клавиши:

```
Writeln ('текст ');
Repeat until Keypressed;
ClrScr;
```

При работе этого фрагмента написанный на экране текст сотрется только после того, как пользователь нажмет любую клавишу. Заметим, что выработанное в этом фрагменте значение функции **True** изменится на **False** только после обращения к **Read/Readln**, **Readkey**. Таким образом, если этот фрагмент написать два раза подряд, второй раз уже никакой задержки не будет — ведь клавиша уже была нажата до печати текста.

Так, в результате работы фрагмента:

```
Writeln ('текст1');
Repeat until Keypressed;
ClrScr; { ***** }
Writeln('текст2');
Repeat until Keypressed;
ClrScr;
```

пользователь увидит «текст1», нажмет клавишу, но «текст2» не увидит, программа «проскочит» дальше, так как значение **Keypressed** равно **True** из-за того, что была нажата клавиша после вывода «текст1». Чтобы увидеть

оба текста, надо очистить буфер клавиатуры. Это можно сделать, например, поставив после первой очистки экрана (место помечено звездочками) процедуру **Readln**. Казалось бы использование фрагмента **Repeat until Keypressed** теряет смысл — вместо него можно сразу писать **Readln**. Однако в этом случае пользователю для движения дальше придется нажимать не любую клавишу, а обязательно клавишу **Enter**. Этого недостатка лишено использование функции **Readkey**.

## Функция **Readkey**

Функция **Readkey** (без параметров) имеет тип **Char**, ее результат — код очередной нажатой клавиши (первой из буфера клавиатуры). Как и **Read/Readln**, она приостанавливает работу программы до нажатия клавиши, однако по многим параметрам существенно отличается от этих процедур. Перечислим эти отличия.

1. Функции **ReadKey** достаточно нажатия любой (одной!) клавиши, не требуется завершать ввод нажатием **Enter** (клавиша **Enter** выступает здесь как обычная клавиша, вырабатывающая свой код).
2. Другим отличием этой функции от **Read/Readln** является способность распознать практически любую клавишу, в том числе **Enter**, **Esc**, **Backspace**, функциональные клавиши. Исключение составляют клавиши смещения, некоторые специфические переключатели типа **CapsLock**.
3. Отсутствие «эха»: вводимый символ не отображается на экране.

Работают с этой функцией обычно так: при обращении ее значение присваивается некоторой переменной типа **Char**, например **C:=ReadKey**, а далее анализируют значение переменной **C**.

Если работа идет с буквенно-цифровыми клавишами, можно непосредственно задавать нужные значения (**IF C='Q'**). Если работа идет с клавишами, не имеющими графического изображения, надо пользоваться их кодом:

```
Repeat ... Until Ord(C)=13
```

— оператор цикла, работающий, пока не будет нажата клавиша **Enter** (ее код — 13). Часто вместо записи **Ord(C)=13** используют **C=#13** (эту форму записи можно применять для всех кодов).

Если клавиша имеет расширенный код, надо предусмотреть повторное обращение к функции **ReadKey**:

```
C:=ReadKey;  
If Ord(C)=0 Then C:= ReadKey;
```

Именно таким образом работают со стрелочками и функциональными клавишами.

А откуда же узнать коды клавиш? Какие клавиши вырабатывают коды из двух чисел, какие из одного? Можно, конечно, посмотреть в справочнике,

а можно иметь под рукой небольшую программку, которая всегда подскажет ответ на эти вопросы.

**Пример 15.14.** Написать программу, которая при нажатии клавиши сообщает, код из скольких чисел она вырабатывает, и печатает эти коды.

Договоримся, что для окончания ввода нажмем клавишу 0 на основной клавиатуре (с помощью этой программы вы можете убедиться, что коды клавиш цифровой клавиатуры совсем другие).

Программа очень простая. Считываем клавишу. Если ее код 0, значит, это специальная клавиша, которая вырабатывает два числа, считываем второе число и печатаем их. Если же код отличен от нуля, это обычная клавиша, печатаем ее код. Обратите внимание: при нажатии клавиши на клавиатуре никакой печати на экране не происходит (как это обычно бывает при использовании процедур **Read/Readln**, весь вывод на экран осуществляется только процедурой **Writeln**).

```
Uses Crt;
Var C:Char;
Begin
  TextBackGround(Black); TextColor(White); ClrScr;
  Repeat
    C:=ReadKey;
    If Ord(C)=0 Then Begin C:=ReadKey;
                        Writeln ('2 числа: 0 и ',Ord(C))
                        End
                    Else Writeln('1 число ',Ord(C))
  Until C='0'
End.
```

С помощью этой программы вы можете увидеть, что обычная «буквенная» клавиша, будучи нажата вместе с **Alt** или **Ctrl**, вырабатывает совсем другие коды.

Для тех, кому такую программку писать лень, приведем здесь коды некоторых полезных клавиш.

Клавиши (в том числе управляющие), вырабатывающие одно число:

<b>Enter</b> — 13	<b>Backspace</b> — 8
<b>Esc</b> — 27	<b>Tab</b> — 9 <b>Пробел</b> — 32

Управляющие клавиши, вырабатывающие коды из расширенного набора (здесь указано только второе число):

<b>Del</b> — 83	<b>Ins</b> — 82
<b>Home</b> — 71	<b>End</b> — 79

Стрелки также вырабатывают два числа:

↑ — 72	↓ — 80	← — 75	→ — 77
--------	--------	--------	--------

Два числа вырабатывают и функциональные клавиши:

**F1** — 59      **F2** — 60      ...      **F10** — 68

Перечислим коды некоторых символов, которые не видны на клавиатуре (их можно напечатать, зная их код: **Write (Chr (<код>))**). Это коды из «обычного» набора, состоящие из одного числа:

218: ▮	196: —	191: ▮	201: ▮	205: =	187: ▮
179:	197: +	179:	186:	206: ▮	186:
192: L	196: —	217: J	200: L	205: =	188: J

Это так называемые символы псевдографики, их можно использовать для рисования рамок (например, вокруг окон).

Мы уже говорили, что с помощью функции **KeyPressed** можно организовывать задержку (менять картинку при нажатии любой клавиши). Используя функцию **Readkey**, тоже можно управлять выдачей информации на экран. Она удобнее, чем **Readln**, тем, что здесь «управляющей» может быть не только клавиша **Enter**, а любая.

***Пример 15.15.** Написать программу, печатающую текст, переливающийся всеми цветами.*

```
Uses Crt;
Var S : String[60];
    I, Cvet, Cvet0 : Integer;
    C : Char;
Begin
  S:='Как здорово, что все мы здесь сегодня собрались!';
  TextBackGround(Black); ClrScr;
  Cvet0:=0;
  Repeat
    gotoxy(20,10);
    Cvet0:=Cvet0+1; If Cvet0=16 Then Cvet0:=1;
    Cvet:=Cvet0;
    For I:=1 to Length (S) do
      Begin {Печать строки разными цветами}
        TextColor(Cvet); Write(S[i]);
        Cvet:=Cvet+1; If Cvet=16 Then Cvet:=1
      End;
      c:=readkey
    Until ord(c)=27
  End.
```

В цикле **For I:=1 to Length (S) do** мы печатаем строку по буквам — каждую букву своим цветом. Обратите внимание, после печати каждой буквы мы увеличиваем значение цвета на 1, но проверяем, не стал ли цвет рав-

ным 16. В этом случае мы снова делаем его равным 1 (иначе цвет стал бы равен 0, буквы бы было не видно).

Первая буква строки печатается цветом *Cvet*. Если теперь значение этой переменной (а мы его запомним в *Cvet0*) увеличить на 1, каждая буква будет напечатана тем цветом, которым была напечатана ее «соседка» справа, — возникнет эффект «цветовой волны», сами буквы стоят на месте, а вот цвета, которыми они окрашены, как бы бегут справа налево.

Управление задержкой в этой программе осуществляется функцией **Readkey** — для того, чтобы картинка изменилась, надо нажать любую клавишу. Эта же процедура помогает и остановить программу: для остановки надо нажать клавишу **Esc** (именно она имеет код 27, который используется в операторе цикла).

В некоторых задачах, особенно в игровых, бывает нужно очистить буфер клавиатуры (стереть оттуда все, что было введено до некоторого момента). Это можно сделать следующим образом:

```
While KeyPressed do C:=ReadKey;
```

Используя функцию **KeyPressed** для ввода вместо **Read**, можно выводить на экран не то, что вводит пользователь, а другие символы (например, этот прием используется при вводе пароля).

***Пример 15.16.** Написать программу — ввод пароля. Вместо вводимого символа на экране печатается звездочка. Ввод заканчивается нажатием **Enter**. Если пользователь считает, что допустил ошибку, он может нажать **Backspace**, — все введенные символы сотрутся, ввод начнется сначала.*

Пароль, правильность ввода которого будем проверять, запишем в начале программы в виде строковой константы. Напомним, правильно введенным будет считаться только пароль, полностью совпадающий с заданным (у нас первая буква — большая, остальные — маленькие).

В этой же программе мы также покажем, как использовать приведенные выше коды клавиш для прорисовывания границ окна: пароль будем выписывать в окошке, обведенном двойной рамкой.

Сначала зададим начальные условия для экрана (черный цвет фона и белый для изображения), так как рамка для окна пароля будет рисоваться на основном экране. Выбрав нужные размеры окна, прорисовываем вокруг него (воображаемого) рамку с помощью процедуры вывода (иногда используя формат для вывода нужного количества пробелов) и цикла.

Затем рисуем окно для ввода пароля (зеленое) и приступаем собственно к вводу пароля.

Ввод будем производить по одной букве, приписывая их справа к переменной *S* (в начале она равна пустой строке), в которой и должно сложиться введенное слово (мы уже делали такой ввод с клавиатуры при работе со строками, при вводе информации из файла). Для контроля длины введенно-

го слова используем переменную  $I$ , значение которой при вводе очередной буквы будем увеличивать на 1. Длину введенного слова надо проверять обязательно: если она будет слишком большой, текст не уместится в зеленом окне, картинка станет некрасивой (к тому же слишком длинное слово и не может быть правильным паролем).

Нажатая клавиша считывается с помощью функции **ReadKey** (на экране при этом ничего не отображается). Сначала проверим, не нажата ли клавиша **Backspace** (ее код 8), ведь по условию задачи этой клавишей стираются ранее введенные символы и ввод начинается сначала. Таким образом, при нажатии клавиши с кодом 8 надо стереть введенное слово. В нашем случае для этого надо сделать три действия: сделать слово равным пустой строке (изменить значение переменной  $S$ ), обнулить его длину и удалить слово с экрана (очистить окно). При очистке окна курсор устанавливается в левый верхний угол окна, так что новый ввод будет производиться с начала.

Если же была нажата другая клавиша, проверим, не является ли она управляющей. Нажатие всех управляющих клавиш здесь отслеживать не будем, проверим лишь, чтобы код был больше 13 (13 — это код клавиши **Enter**, при нажатии которой наш цикл завершается; у других управляющих клавиш коды меньше 13, на них программа реагировать не будет).

Введенный символ приписываем справа к вводимому слову, длину слова увеличиваем на 1, а в окне ввода пароля печатаем звездочку.

Цикл завершаем, когда пользователь нажмет клавишу **Enter**, или если он пытается ввести слишком длинный пароль (мы ведь уже знаем, что такой пароль — неправильный).

Печать сообщения о правильности ввода пароля производим в другом окне (чтобы не стереть только что напечатанные в зеленом окне звездочки), устанавливаем его параметры и выписываем ответ.

```
Uses Crt;
```

```
Const PassWord: String='Parol';
```

```
Var S      : String[10];
```

```
    I      : Integer;
```

```
    C      : Char;
```

```
Begin
```

```
    TextBackGround(Black); ClrScr;  TextColor(White);
```

```
    Writeln('Введите пароль');
```

```
    Write(chr(201):19){Левый верхний угол}
```

```
    For I:=1 to 20 do Write(Chr(205));{Верхняя линия}
```

```
    Writeln(chr(187)); {Правый верхний угол}
```

```
    For I:=1 to 3 do
```

```
        Writeln(chr(186):19, chr(186):21){Боковые линии}
```

```
    Write(chr(200):19){Левый нижний угол}
```

```
    For I:=1 to 20 do Write(Chr(205));{Нижняя линия}
```

```

Writeln(chr(188)); {Правый нижний угол}
Window(20,3,39,5);
TextBackGround(Green); ClrScr; TextColor(Yellow);
I:=0; S:='';
Repeat
  C:=ReadKey;
  If ord(c)=8 Then Begin ClrScr; I:=0;S:='' End
    Else If Ord(C)>13 Then Begin Write('*');
      S:=S+C; I:=I+1
      End;
Until (Ord(C)=13) Or (I>15);
Window(20,10,80,20); TextBackGround(Black);
                                                    TextColor(Red);
If S=PassWord Then Writeln('Пароль верный')
  Else Writeln('Пароль набран неправильно')
End.

```

**Пример 15.17.** Написать программу, рисующую на экране «изображение» с помощью стрелок. Пользователь нажимает стрелочку — курсор сдвигается в заданном направлении, оставляя на экране символ (таким образом, нажав несколько раз стрелочку «вверх», можно получить вертикальную линию). Для смены цвета символа использовать функциональную клавишу **F1**, для окончания работы — клавишу **Enter**.

Эта несложная и короткая программа очень эффектно выглядит в работе: с ее помощью действительно можно создавать на экране разноцветные рисунки.

Рисовать будем в цикле **Repeat** (цикл закончится, когда будет нажата клавиша **Enter**).

Нам надо проверить код введенной клавиши. При этом мы знаем, что нам нужны только управляющие клавиши, которые вырабатывают коды из расширенного набора. Первое из чисел расширенного набора — всегда 0, поэтому выписываем в программе только реакцию на нажатие клавиш, вырабатывающих 0, нажатие остальных игнорируем (никаких изменений на экране не производим). Если код введенной клавиши оказался нулем, считываем второе число. Теперь остается только выписать действия программы, которые будут различаться в зависимости от введенного кода (точнее, от второго числа в этом коде). Это удобно сделать с помощью оператора **Case**. Если была нажата одна из стрелочек, надо изменить координату положения курсора (при этом проверив, что рисунок не будет выходить за границы экрана), если же была нажата клавиша изменения цвета, увеличим на 1 цвет.

Теперь остается только установить в новое положение курсор, поменять цвет изображения и напечатать символ. Отметим, что если цвет символа

будет выбран черным (равным цвету фона), вместо рисования будет производиться стирание изображения.

```
Uses Crt;
Var S,C:Char;
    X,Y:Integer;
    Cvet:Integer;
Begin
  X:=40;Y:=10;  Cvet:=15; {Начальный цвет}
  TextBackGround(Black);
  ClrScr;  S:='@';{Будем рисовать с помощью этого символа}
  TextColor(Cvet);
  gotoxy(X,Y);  Writeln(S); {печатаем символ}
  Repeat
    C:=ReadKey;{вводим управляющий символ}
    If Ord(C)=0 Then Begin C:=ReadKey;
      {Производим изменение координат или цвета}
      Case Ord(C) of
        80: If Y<24 then Y:=Y+1 ;
        72: if Y>1  then Y:=Y-1;
        75: If X>1 Then X:=X-1;
        77: If X<79 Then X:=X+1;
        59: If Cvet=15 Then Cvet:=0
        Else Cvet:=Cvet+1
      End
    End;
    GotoXY(X,Y); TextColor(Cvet); {Устанавливаем
                                   новые параметры}
    Write(S) {Печатаем символ}
  Until Ord(C)=13 {Окончание работы при нажатии
                                   клавиши Enter}
End.
```

Эта программа обладает большими возможностями для совершенствования. Например, можно более удобно сделать изменение цвета: клавишей **F1** увеличивать значение цвета на 1, а клавишей **F2** — уменьшать. С помощью какой-нибудь другой клавиши можно менять символ, который печатается на экране, изменять цвет фона и т. п.

Теперь у вас достаточно знаний, чтобы рисовать «интерактивные мультфильмы»: управлять с клавиатуры появлением, исчезновением, движением объектов на экране.

***Пример 15.18.** Написать программу, рисующую на экране «мордочку», которая умеет открывать и закрывать глаза. Управление движением глаз осуществлять с помощью клавиатуры (самая левая и самая правая нижние буквенные клавиши соответственно для правого и левого глаз).*



Каждый глаз у нас в каждый момент времени может быть открыт или закрыт, поэтому статус глаз удобно регистрировать с помощью логических переменных: *R* — для правого глаза и *L* — для левого. Переменные равны **True**, если глаз открыт (будем это изображать на рисунке большим синим прямоугольником), и **False**, если закрыт (на рисунке — узкий черный прямоугольник).

Сначала выпишем для пользователя подсказку и нарисуем саму мордочку (большой коричневый прямоугольник) с открытыми глазами (два синих прямоугольника). Статусы глаз вначале установим **True** (открыты).

Теперь в цикле — пока пользователь не прекратит работу нажатием клавиши «q» — будем «подмигивать».

Присваиваем переменной *Y* значение нажатой клавиши. Реагировать будем только на ввод символов «q», «z» и «/». Записать это удобно с помощью оператора **Case**.

Если введен символ «z», надо изменить состояние левого глаза. Проверяем, какое оно сейчас (статус хранится в переменной *L*). Если глаз был открыт (**True**), его надо закрыть, для чего стираем голубой прямоугольник (рисуем на его месте коричневый, цвета «мордочки») и рисуем узкий черный. Если надо, наоборот, открыть глаз, рисуем большой голубой прямоугольник (стирать находящийся на его месте черный узкий не надо, он «пропадет» под нарисованным голубым). В обоих случаях статус глаза (значение переменной *L*) меняется на противоположный.

С правым глазом, естественно, поступаем точно так же, описывая реакцию на ввод символа «/».

```
Uses CRT;
```

```
Var Y:Char; {переменная для управления движением  
с клавиатуры}
```

```
R,L:Boolean; {статус соответственно правого и левого  
глаз - открыт или закрыт}
```

```
Procedure Rectangl (X1,Y1,X2,Y2,C:Integer);
```

```
Begin Window(X1,Y1,X2,Y2); TextBackGround(C); ClrScr End;
```

```
Begin
```

```
TextBackGround (Black); ClrScr;
```

```
Writeln('q - окончание работы');
```

```
Writeln('z - подмигивание левым глазом');
```

```
Writeln('/ - подмигивание правым глазом');
```

```
Rectangl (25,4,60,20,Brown);
```

```
Rectangl (35,10,40,14,Blue); L:=True;
```

```
Rectangl (45,10,50,14,Blue); R:=True;
```

```
Repeat
```

```
Y:=ReadKey;
```

```
Case Y of
  'z':Begin If L Then Begin Rectangl(35,10,40,14,Brown);
                                Rectangl(35,12,40,12,Black)
                                End
                                else Rectangl(35,10,40,14,Blue);
                                L:=Not(L)
                                End;
  '/':Begin If R Then Begin Rectangl(45,10,50,14,Brown);
                                Rectangl(45,12,50,12,Black)
                                End
                                else Rectangl(45,10,50,14,Blue);
                                R:=Not(R)
                                End
End;
Until Y='q';
End.
```

А теперь вспомним, как мы в гл. 8 изучали логику, и попробуем с помощью логических операций и логических переменных *L* и *R* описать, в каком состоянии находятся глаза нашей «мордочки». Например, высказыванию «Оба глаза открыты» соответствует логическое выражение **L and R**, а «Только один глаз открыт» — **L and Not R or R and Not L**.

Если мы организуем в конце программы (в новом окне) вывод этих фраз в зависимости от значения логических выражений, можно использовать эту программу для проверки правильности составления логических выражений. В самом деле, установив какое-либо состояние глаз, останавливаем программу и смотрим, что же выведено. Если вывод соответствует состоянию глаз — выражение составлено правильно.

Новое окончание программы (добавим его перед **End.**):

```
Window(1,20,79,25);
if L and R Then Writeln('Оба глаза открыты');
If Not L and Not R Then Writeln ('Оба глаза закрыты');
If L or R Then Writeln('Хотя бы один глаз открыт');
If L and Not R or R and Not L Then Writeln('Только один
                                глаз открыт')
```

Имея некоторое количество времени и желание, можно создавать достаточно сложные «мультфильмы», игровые программы. Например, чтобы заставить объект «делать зарядку», надо сначала нарисовать неподвижные части фигурки, а потом в цикле после нажатия определенных клавиш, стирать и рисовать в новом положении подвижные части тела (например: руки вниз, руки в стороны, руки вверх). Если при этом всю картинку чуть сдвигать (например, по горизонтали), фигурка будет двигать руками «на бегу».

Дополнив картинку «музыкой» с помощью процедуры **Sound**, получим «звуковое кино».

Понятно, что средств псевдографики и генерации звука, описанных здесь, совершенно недостаточно для создания полноценных мультфильмов или игр. Однако понятие о том, как эти программы создаются, такие средства дают. И — можете проверить — игровая программа, созданная самостоятельно, доставит вам не меньше удовольствия, чем покупная, несмотря на ее слабые изобразительные средства — ведь она будет создана точно в соответствии с вашими желаниями (и возможностями)!

## Задачи 15.43–15.52. Работа с клавиатурой

- 15.43. «Обманная печать». Человек нажимает цифры, а вместо них печатаются слова — их названия. (Названия цифр удобно хранить в массиве строк.)
- 15.44. Независимо от того, большие или маленькие латинские буквы вводит пользователь, печатаются всегда маленькие.
- 15.45. При нажатии стрелочки появляется слово — название соответствующего направления (например: «вправо»).
- 15.46. В центре экрана — прямоугольник синего цвета. Стрелочками его цвет можно изменять (вправо — увеличивать, влево — уменьшать).
- 15.47. В центре экрана маленький квадратик. При нажатии клавиши «стрелка вверх» он увеличивается во всех направлениях, «стрелка вниз» — уменьшается.
- 15.48. Организовать постепенное увеличение размеров нарисованного на экране прямоугольника при нажатии соответствующей стрелочки (например, если нажата стрелочка вверх — изменяется верхняя граница, «ползет» вверх). Следите, чтобы рисунок не вышел за пределы экрана!
- 15.49. Изобразить квадратик в рамке. Клавишами «стрелка вверх» и «стрелка вниз» пусть меняется цвет квадратика, а клавишами «стрелка вправо», «стрелка влево» — цвет рамки. **Esc** — конец работы.
- 15.50. Организовать движение прямоугольника в соответствующем направлении при нажатии на клавиатуре стрелок «вправо» и «влево». Прямоугольник не должен «выходить» за границы экрана!
- 15.51. Организовать движение какого-либо слова на экране в соответствующем направлении при нажатии на клавиатуре стрелок «вправо» и «влево».
- 15.52. При нажатии стрелки «влево» появляется полоса около левой границы экрана, «вправо» — у правой.

## Задачи 15.53–15.61. Общие задачи с модулем CRT

- 15.53. Дополнить программу «ввод пароля» (пример 15.16), чтобы при нажатии клавиши **Backspace** стиралось не все введенное сообщение, а только последний символ.
- 15.54. Дополнить программу «рисование изображений с помощью стрелок» (пример 15.17). Уменьшить границы поля для рисования, в освободившемся месте выписывать текущие параметры рисования: цвет изображения, цвет фона, символ. Сделать больше доступных опций.
- 15.55. По типу «подмигивающей мордочки» (пример 15.18) создать программу, проверяющую правильность логических выражений из трех переменных.
- 15.56. Экран разделен на 4 прямоугольника. При переходе (с помощью стрелочек) из одного прямоугольника в другой в новом прямоугольнике появляется надпись (элемент заранее определенного массива или файла), а из старого надпись, которая там была, стирается. Конец программы — **Esc**.
- 15.57. Экран разделен на 4 прямоугольника (три — синего цвета, один — красного). При переходе (с помощью стрелочек) из красного в синий прямоугольник цвета меняются. Конец программы — **Esc**.
- 15.58. В центре экрана появляются арифметические примеры на сравнение (числа от 1 до 100, разделенные знаками < или >). По краям экрана надписи «Правильно» и «Неправильно». Пользователь должен нажать соответствующую стрелочку, указывающую на соответствующую надпись. Подсчитывается число правильных ответов.
- 15.59. На экране рамка — прямоугольник, стороны которого раскрашены четырьмя цветами. В центре экрана случайным образом появляется квадрат одного из этих четырех цветов. Пользователь должен нажать стрелку, направленную на цвет, совпадающий с цветом квадрата. Если он ошибается, раздается звуковой сигнал. Подсчитывается количество ошибок.
- 15.60. Игра на тренировку памяти. На экране несколько прямоугольников разных цветов, через какое-то время они исчезают, а на экране появляются «незакрашенные» прямоугольники. Меняя некоторой клавишей цвета прямоугольников, человек должен все прямоугольники «раскрасить» в первоначальные цвета.
- 15.61\*. Игра «Змейка». Прямоугольник движется, управляемый стрелочками, изгибаясь на поворотах. При столкновении с границей экрана подается звуковой сигнал.

## Глава 16

# Разные задачи

---

В предыдущих главах мы разбирали в основном учебные задачи: в реальной жизни никто нам не «задает массив», не определяет значения каких-то переменных. Обычно мы имеем дело с результатами каких-то измерений, подсчетов, и сами должны решать, как их представить в программе, с каким типом данных работать. Здесь мы хотим предложить несколько таких задач. Некоторые из них похожи на те, которые бывают на экзаменах, на олимпиадах, некоторые — «игровые». Объединяет эти задачи то, что программист сам должен выбрать не только алгоритм решения, но и способ представления данных (от которого этот алгоритм может зависеть).

Мы не будем детально разбирать все задачи, оставим написание полной программы читателю в качестве самостоятельной работы, дадим лишь некоторые советы. При этом заметим, что в большинстве случаев существует несколько способов как представления данных, так и способов решения.

### *Пример 16.1. Голосование*

*Создать систему электронного голосования. Пусть в некоторой организации (школе, поселке и т. п.) происходят выборы. Имеется несколько кандидатов (их количество и имена или названия партий, естественно, известны заранее). А вот количество избирателей, которые захотят проголосовать, неизвестно. Каждый избиратель имеет право проголосовать за одного кандидата, причем в системе не должны запоминаться сведения, как проголосовал очередной отдельный избиратель (соблюдаем тайну голосования!). По окончании голосования программа печатает количество проголосовавших избирателей и список кандидатов в порядке убывания количества отданных за них голосов. Также печатается, сколько голосов набрал каждый кандидат (абсолютное количество и в процентах от общего числа проголосовавших).*

Какие данные понадобятся для решения задачи? Количество кандидатов — это константа, целое число. Имена кандидатов — строки. Видимо, удобно их заранее занести в файл. Никакого «массива избирателей» создавать нельзя. Во-первых, по условию задачи необходимо обеспечить тайну голосования, а, во-вторых, мы и не можем это сделать — ведь количество избирателей неизвестно.

Как организовать голосование? Список кандидатов печатается на экране, каждый избиратель имеет право указать, за какого кандидата он голосует, например, нажав кнопку с номером своего кандидата в списке (или выбрав его из списка, управляя стрелочками). Будем считать, что, если избиратель нажимает цифру с номером, которого нет в списке, значит, он голосует «против всех», его мнение тоже будем учитывать (увеличивать количество проголосовавших). По окончании голосования «дежурный» нажимает специальную кнопку, означающую конец голосования.

Какой метод выбрать для решения задачи? Это следует из ее формулировки: надо подсчитать количество голосов — метод подсчета. Организуем массив (его размер — количество кандидатов), после волеизъявления очередного избирателя добавляем единичку к «его» кандидату, не забывая добавить единичку и к количеству проголосовавших. Остается подсчитать результаты в процентах и отсортировать кандидатов.

Чтобы очередной избиратель не видел, как проголосовал предыдущий, после каждого выбора (нажатия кнопки с цифрой) будем очищать экран — это процедуру мы уже писали.

Вот программа, выполняющая эту задачу в несколько облегченном виде (использованы только средства языка, обязательно изучаемые в школе). Файлы мы здесь не используем, поэтому имена кандидатов в начале придется ввести (они будут храниться в массиве).

```
const Kp=5;
Procedure Clear;
  Var I:Integer;
  Begin For I:=1 to 25 do Writeln
  End;
Var Sk:Array [1..Kp] of Integer; {Для подсчета, сколько
                                человек проголосовало за этого кандидата}
  Name:Array[1..Kp] Of String[15];
  Dop: String[15];
  I,J,N,Max,NMax, Kvo : Integer; {общее количество
                                голосовавших}
Begin
  Kvo:=-1;{Так как в конце будет признак конца, его
                                добавлять не надо}
  Clear;
  For I:=1 to Kp do begin Sk[i]:=0;
                                Write('Кандидат ', i, ' - ');
                                Readln(Name[i])
                                end;
  Repeat Clear;
    For I:=1 to Kp do
      Writeln(i:3, ' . ', Name[i]);
```

```

Write('Номер от 1 до ', Kp, ' (0 для завершения) ');
Readln(N);
Kvo:=Kvo+1;
If (N>0) and (N<Kp) Then Sk[N]:=Sk[N]+1
    {голосов у кандидата N стало больше}
Until N=0;
Clear;
Writeln('Всего проголосовало человек: ', Kvo);
{Сортировка методом: максимальный переставляется
    на первое место}
For I:=1 to Kp-1 do {Найдем и переставим Kp-1 максимумов}
Begin Max:=0; Nmax:=0;
  For J:=I to Kp do
    If Sk[j]>Max Then Begin Max:=Sk[j]; NMax:=j End;
    {Ищем максимум и его место}
  if Max=0 Then Break; {Остались только кандидаты,
    не набравшие голосов, их переставлять не надо}
  Sk[NMax]:=Sk[i]; {Переставляем}
  Dop:=Name[NMax]; Name[NMax]:=Name[i];
  Sk[i]:=Max; Name[i]:=Dop
End;
Writeln(' Кандидаты набрали в процентах ');
For I:=1 to Kp do
  Writeln(i:3, Name[i]:16, Sk[i]:6, Sk[i]/Kvo*100:7:1, '%')
End.

```

В этой программе выполняется сортировка массива *Sk* (в нем хранятся данные, сколько голосов набрал каждый кандидат), но переставлять надо не только элементы этого массива, но и соответствующие элементы массива *Name* — имена кандидатов. В таких случаях удобно использовать тип «Запись».

Если использовать тип «Файл», фамилии кандидатов можно не вводить во время работы программы, а заранее занести в файл. Результаты в этом случае можно также не только выводить на экран, но и записывать в файл, чтобы потом, например, распечатать.

Если использовать средства для управления экраном и клавиатурой, можно сделать, чтобы голосующие нажимали не цифру с номером своего кандидата, а выбирали нужную строчку из списка стрелочками.

Вот еще две похожие задачи, их можно решать тоже с использованием метода подсчета.

### **Пример 16.2. Олимпиада**

*В городе Мухинске проводилась олимпиада по труду. Сведения об участниках олимпиады подавались следующим образом: N строк (количество*

школ, пославших своих учеников на олимпиаду), в каждой из которых содержится 12 чисел: первое число — номер школы, второе — количество участников олимпиады из 1-го класса этой школы, второе число — из 2-го класса и т. д. Например, строка:

347 0 0 0 1 1 22 1 0 0 0 3

означает, что школа 347 послала на олимпиаду по 1 ученику из 4, 5 и 7 классов, 22 — из 6-го, 3 — из 11-го, а из других классов от этой школы участников не было. Известно, что в олимпиаде участвовали ученики не менее чем из двух школ и каждая из школ-участниц прислала хотя бы по одному человеку.

Определить номер школы, которая послала больше других самых младших учеников. Если таких школ несколько, напечатать все номера.

Кажется, можно определить, какая школа послала больше всего учеников 1-го класса, но это неверный путь. А вдруг первоклассников на олимпиаде вообще не было? Подсчитать то же для второклассников? А вдруг и их не было?

### Пример 16.3. Экзамен

При проведении экзамена решили реализовать систему оценок, зависящую от того, сколько человек справились с каждой задачей.  $N$  экзаменуемых получают  $R$  задач. За правильное решение каждой задачи начисляется 1 балл — таким образом, число первичных баллов — это количество решенных каждым участником задач. Далее первичные баллы переводятся во вторичные: каждой задаче присваивается коэффициент, обратно пропорциональный количеству решивших ее. Вычисляется он по формуле  $K_i = N/Z_i$ , где  $i$  — номер задачи,  $Z_i$  — количество экзаменуемых, решивших  $i$ -ю задачу. Таким образом, если какую-то задачу решили все ученики, для нее коэффициент пересчета будет равен 1, если половина учеников — 2, если же задачу решил только один ученик, он за нее получит  $N$  баллов.

Путем сложения коэффициентов за каждую задачу получаем вторичный балл для каждого ученика. А теперь надо перевести результаты в привычную столбальную систему. Вторичный балл самого лучшего ученика примем за 100 и соответствующим образом пересчитаем баллы всем остальным.

Нельзя сказать, что эта задача на алгоритм подсчета в чистом виде. Здесь придется хранить в памяти результаты учеников, так как они используются два раза: при расчете коэффициентов и при подсчете вторичных баллов. Удобно хранить данные в матрице (таблице): в каждой строчке записывать, как решал задачи данный ученик. Сначала надо заполнить таблицу 1 и 0 (решена или не решена задача), потом подсчитать сумму элементов каждого столбца — получим числа  $Z_i$ , далее подсчитать  $K_i$  и умножить на него все



элементы  $i$ -го столбца. Сумма строчек даст нам вторичный балл. Эти баллы надо запомнить в массиве, так как надо найти максимальный и пересчитать их по столбальной системе.

### **Пример 16.4. График функции**

Конечно, чтобы на экране построить график функции так, как мы это привыкли видеть в учебнике, наших знаний Паскаля недостаточно (нужны специальные процедуры по работе с графическим экраном). Мы можем построить некоторую «схему» — отметим на экране звездочками позиции некоторых точек графика. Надо выбрать функцию, отрезок, на котором будет изображаться график, масштаб (проверьте, чтобы значения функции на выбранном отрезке не были слишком маленькими или слишком большими!).

**Вариант 1.** Печать «графика» без хранения значений функции. На экране мы можем печатать только «сверху вниз», поэтому график придется повернуть на  $90^\circ$  по часовой стрелке: ось  $OX$  будет направлена сверху вниз, а ось  $OY$  — слева направо. В верхней строке экрана значение  $X$  будет равно началу выбранного интервала, в нижней — концу. Оси можно изображать символами «|» и «-». В позиции, соответствующей очередному значению функции, надо ставить «\*» (не забудьте, что значения функции вещественные, а номер позиции — целое число из определенного диапазона).

**Вариант 2.** Чтобы напечатать график без поворота, надо сначала вычислить значения функции на заданном отрезке, запомнить их в таблице, а потом печатать. В этом случае можно автоматически вычислять масштаб (брать его таким, чтобы все значения функции умещались на экране).

### **Пример 16.5. Иностранные слова**

Пусть компьютер «знает» некоторое количество иностранных слов и их перевод.

**Вариант 1.** «Словарь»: человек с клавиатуры вводит иностранное слово; ему либо сообщается, что такого слова в словаре нет, либо выдается перевод.

**Вариант 2.** «Контрольная работа»: в случайном порядке выводятся иностранные слова, человек должен ввести перевод. Компьютер оценивает правильность, подсчитывает количество ошибок.

**Вариант 3.** «Обучение»: выводится русское слово, человек должен дать перевод. Если человек не знает ответа, компьютер подсказывает ему (например, выводит первую букву слова), если человек печатает слово с ошибкой (какая-либо буква неправильна), компьютер указывает место ошибки и предлагает исправить ее. Обучение считается законченным, когда человек каждое слово из имеющегося списка перевел правильно.

Для хранения данных в этой задаче понадобятся файлы или массивы одинаковой длины. Например, в одном массиве хранятся русские слова, в другом — их переводы (русский эквивалент  $k$ -го английского слова стоит на  $k$ -м месте в массиве русских слов). Можно также слово и перевод хранить в виде записи. Таким образом, реализация варианта 1 представляет собой обычный поиск в массиве или в файле.

В программе варианта 2 надо сравнивать соответствующие компоненты массивов (компоненты файлов или поля записи). Строки можно сравнивать целиком, так что это никаких трудностей не вызовет.

А вот вариант 3 посложнее. Надо придумать, как предлагать человеку слова в случайном порядке, но без повторов (ведь правильно переведенные слова второй раз предлагать не надо!). Для этого можно завести еще один массив (или поле в записи) — логический. В начале заполнить его значениями **False**. Если слово переведено правильно, надо заменять **False** на **True**, а при выборе слова проверять этот признак, и слова с **True** не предлагать. Также для реализации этого варианта понадобится работа со строковыми функциями (чтобы отлавливать небольшие ошибки), давать подсказки.

### *Пример 16.6. Игра «Угадай слово»*

*Компьютер выбирает слово, которое предлагает человеку для угадывания и печатает на экране значки «-» — по числу букв в слове. Угадывающий называет букву. Если она есть в слове, соответствующая «черточка» (или несколько «черточек») заменяется буквой. Узнали? Далее правила могут быть разнообразны: можно подсчитывать количество неудачных попыток — в таком случае игра заканчивается, когда их становится определенное число. Можно предлагать угадывающему подсказки, объясняющие, что за слово загадано — получится что-то типа телепередачи «Поле чудес». Придется учесть и некоторые особенности игры с компьютером: предусмотреть некоторую защиту от неправильного ввода: вдруг играющий по ошибке нажмет не букву, а цифру или какой-то другой значок, вдруг он нажмет букву, которую уже называл?*

Как хранить данные? Слова удобно записать в файл, например, каждое слово с новой строки (выбирать случайное слово можно с помощью функции **Random**). Выбранное слово можно хранить в переменной типа **String**, причем придется завести две такие переменные: в одной хранить заданное слово, а в другой — ту «картинку» которая печатается на экране (сначала слово из черточек, потом — слово, где некоторые черточки заменены буквами).

Какие процедуры и функции понадобятся? Естественно, одна из основных — это функция, определяющая, входит ли буква в строку. Она нам по-

надобится и для того, чтобы определить, входит ли названная буква в загаданное слово, и для того, чтобы посмотреть, является ли введенный символ буквой, и чтобы определить, не вводилась ли уже такая буква. Можно использовать во всех этих ситуациях встроенную функцию **Pos**, а можно для некоторых случаев написать собственный аналог (учитывая, что нам нужно не просто определить, входит ли буква в слово, но и подсчитать, сколько раз входит, заменить в нужной позиции черточку на букву и т. д.).

Программа будет состоять из двух частей: подготовительной (выбор слова, присвоение начальных значений переменным) и основной (угадывание слова), которую, понятно, надо оформлять в виде цикла.

Какой цикл использовать? Если мы устанавливаем правила, по которым на угадывание слова отводится определенное количество попыток, то **For**. Если же игра продолжается до тех пор, пока слово не угадано, то, например, **Repeat**. В обоих случаях надо не забыть предусмотреть выход для ситуации, когда угадывающий хочет назвать слово целиком. В цикле **For** это будет процедура **Break**, в цикле **Repeat** нужно вставить дополнительное условие для выхода.

Эта задача легко решается с помощью операторов языка, которые в обязательном порядке изучаются в школе (только набор слов для отгадывания в этом случае придется хранить не в файле, а в массиве), однако за счет дополнительных средств по работе с экраном ее можно значительно украсить: напечатать на экране алфавит, буквы из него выбирать стрелочками, при ошибках издавать какие-либо звуки и т. п.

### *Пример 16.7. Игра «Крестики-нолики»*

*Вариант 1. Игра для двоих игроков. Компьютер следит за правильностью ходов, после каждого хода проверяет, не закончилась ли игра (один из игроков выиграл или не осталось свободных клеток), сообщает результат.*

Для хранения данных можно использовать символьную матрицу. В начале во все клетки запишем пробелы. Игрок вводит номер позиции, в которую он хочет поставить свой «крестик» или «нолик». Компьютер, если клетка занята, просит повторить ход, если свободна — ставит в нее нужный символ. Для того чтобы проверить, не принес ли очередной ход игроку победу, надо посмотреть, не появились ли в матрице строки, столбцы или диагонали, заполненные одинаковыми символами.

*Вариант 2 сложнее. Человек играет с компьютером. Конечно, можно сделать компьютер «глупым» игроком, запрограммировать, что он будет делать ход на любую случайную пустую клетку — это несложно. Если же хочется снабдить компьютер некоторым «интеллектом», придется научить его выбирать позицию для хода. Например, проверять, нет ли строки, столбца, диагонали, где уже стоят два символа противника.*

### *Пример 16.8. Игра «Морской бой»*

Правила этой популярной игры всем известны.

Как хранить данные? Очевидно, в виде матриц (двумерных массивов), причем нумеровать позиции удобно так, как это принято в игре: буквами (правда, латинскими) и числами — язык позволяет это сделать. Каждому игроку понадобится две матрицы: для хранения мест расположения своих кораблей и для хранения своего представления о ситуации противника. Нужно решить, какими значками (символами или числами) помечать разные виды клеток: пустые, занятые кораблем (здесь будет два вида клеток: не обнаруженная противником и «убитая»), клетки, «содержимое» которых пока неизвестно и т. п.

В программе можно выделить две части: расстановка кораблей и собственно игра (угадывание позиции).

В игре человек вводит позицию, куда он хочет стрелять. Компьютер проверяет правильность ввода, смотрит, занята ли эта клетка, отвечает «мимо», «ранен» или «убит» (для этого придется посмотреть, что стоит в соседних клетках). А как компьютер будет выбирать, куда ему стрелять? Даже если он не будет «умным», не будет иметь никакой особой стратегии, а будет, например, проверять все клетки подряд, все равно он должен уметь реагировать на ответ человека о результате своего выстрела (изменить статус клетки, в которую стрелял), должен уметь «добивать» раненые корабли. И наконец, компьютер должен уметь оценивать сложившуюся ситуацию — проверять, не победил ли один из игроков.

Расстановка кораблей, пожалуй, более сложная задача. Компьютер должен уметь правильно расставлять свои корабли, «помогать» человеку в расстановке кораблей. (Нестрашно, что «компьютер» узнает, как стоят корабли противника — все будет «по-честному», он не будет пользоваться полученными знаниями — мы не будем их учитывать в программе). Здесь надо будет научиться рассматривать клетки, соседние с заданной (и при этом не выходить за границы поля!).

### *Пример 16.9. Игра «15»*

*Помните: коробочка размером  $4 \times 4$  с 15 фишками-квадратиками, пронумерованными с 1 до 15. Одно место не занято, на него можно передвигать соседние фишки (из того же ряда или из того же столбца). Путем многократных передвижений нужно добиться, чтобы фишки стояли в правильном порядке.*

Работаем с целочисленной матрицей, в которой располагаем случайным образом числа от 0 до 15. Ноль будет обозначать пустое место. Надо написать процедуры, которые позволяют менять местами числа в соседних строках и соседних столбцах матрицы. Человеку доступны 4 варианта хода: поменять с нулем число сверху, снизу, слева или справа от него. Вводить нужный ход

можно стрелочками или (если пользоваться только операторами, входящими в школьную программу) буквами (каждый ход обозначить определенной буквой). После ввода хода программа сначала проверяет, возможен ли такой ход, и далее надо либо вывести сообщение о том, что ход неправильный, либо произвести передвижение и напечатать измененную матрицу. Потом программа должна проверить, не достигнута ли цель, не стоят ли числа правильно. Можно подсчитывать количество сделанных ходов.

Есть в этой задаче один неприятный момент: не всякую начальную комбинацию чисел можно, действуя по правилам, преобразовать в полностью упорядоченную; в некоторых случаях не удастся поставить на место два последних числа. Как выходить из этой ситуации — решайте сами. Можно постановить, что в разных ситуациях будут разные окончательные комбинации, а можно изначально располагать числа таким образом, чтобы преобразование было возможным.

### **Пример 16.10.** Игра «2048» (или «2048+»)

*Имеем квадратную матрицу заданного размера, изначально заполненную нулями (так будем обозначать свободные места). В начале хода на одно из свободных мест случайным образом ставится число «2» или «4». Человек может выбрать «сдвиг» вверх, вниз, вправо или влево. Процедуры сдвига будут довольно сложные, гораздо сложнее, чем в предыдущей задаче. Во-первых, они будут затрагивать все строки или столбцы матрицы, во-вторых, по-разному работать с числами: нули, сколько бы их не было, будут «пропадать», а ненулевые числа, если по соседству оказались пара одинаковых, суммироваться. При этом «свободные» места надо не забывать заполнять нулями. В остальном программа похожа на предыдущую: она также должна проверять, возможен ли выбранный вариант хода, делать ход, печатать измененную матрицу, проверять, не достигнут ли конец игры. Программа должна подсчитывать очки (например, суммировать числа, появляющиеся в начале хода).*

Здесь приведено несколько задач из разных сфер жизни. Надеемся, вы теперь сами сможете придумать немало задач, которые можно решить, написав программу на Паскале, и знание программирования поможет вам при изучении геометрии, физики, химии...

Для успешной сдачи экзамена по информатике надо знать не только программирование, но и другие разделы информатики. Вам теперь легче будет с ними разобраться, так как многие задачи вы сможете запрограммировать. Мы разобрали, как написать программы для работы с разными системами счисления, посмотрели, как программировать игры. Задачи на подсчет можно решить с помощью электронных таблиц — разделы информатики тесно связаны между собой, попробуйте решать задачи школьного курса разными способами, с применением разных инструментов.

# Оглавление

---

<b>ВВЕДЕНИЕ</b>	<b>3</b>
<b>ГЛАВА 1. ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ</b>	<b>5</b>
Программирование	5
Этапы решения задачи	5
Что такое алгоритм?	6
Словесная формулировка алгоритма	6
Блок-схема. Основные конструкции	7
Переменная. Присваивание	9
Условие. Виды разветвлений	10
Цикл	14
Массив	20
Подпрограммы	22
Тестирование	26
Исполнитель алгоритма	27
Оптимальный алгоритм. Сложность алгоритма	29
Задачи 1.1–1.26. Составление алгоритмов	30
<b>ГЛАВА 2. ПЕРВАЯ ПРОГРАММА НА ЯЗЫКЕ ПАСКАЛЬ</b>	<b>35</b>
Понятие об алфавите языка	35
Принципы записи и «внешний вид» программы	38
<b>ГЛАВА 3. ЭТАПЫ ПОДГОТОВКИ ПРОГРАММЫ. ПАСКАЛЬ-СРЕДА</b>	<b>39</b>
Этапы подготовки программы	39
Основные функции Паскаль-среды	40
Задачи 3.1–3.4. Работа в редакторе	50
<b>ГЛАВА 4. СТРУКТУРА ПАСКАЛЬ-ПРОГРАММЫ</b>	<b>51</b>
<b>ГЛАВА 5. ОСНОВНЫЕ ТИПЫ ДАННЫХ.</b>	
<b>ОПИСАНИЯ ПЕРЕМЕННЫХ. ПРИСВАИВАНИЕ</b>	<b>53</b>
Некоторые типы данных и работа с ними	54
Оператор присваивания	63

Пример программы с разными типами данных и операторами присваивания . . . . .	65
Задачи 5.1–5.17. Числа и формулы . . . . .	66
<b>ГЛАВА 6. ВВОД С КЛАВИАТУРЫ И ВЫВОД НА ЭКРАН . . . . .</b>	<b>68</b>
Оператор ввода . . . . .	69
Оператор вывода . . . . .	70
Форматный вывод . . . . .	71
Грамотное использование операторов ввода и вывода . . . . .	72
Примеры программ с вводом-выводом . . . . .	73
Задачи 6.1–6.27. Ввод и вывод . . . . .	76
<b>ГЛАВА 7. РАЗВЕТВЛЕНИЯ . . . . .</b>	<b>78</b>
Условный оператор . . . . .	78
Составной оператор . . . . .	81
Решение задач с условным оператором . . . . .	83
Оператор выбора . . . . .	91
Задачи 7.1–7.35. Программы с разветвлениями . . . . .	93
<b>ГЛАВА 8. ТИП BOOLEAN. ЛОГИЧЕСКОЕ ВЫРАЖЕНИЕ . . . . .</b>	<b>96</b>
Логические значения, логические константы . . . . .	96
Булева алгебра, алгебра логики . . . . .	97
Логические операции . . . . .	98
Составление логических выражений . . . . .	100
Задачи с логическими выражениями . . . . .	101
Программы с логическими выражениями . . . . .	107
Задачи 8.1–8.11. Логическое выражение . . . . .	113
<b>ГЛАВА 9. ОПЕРАТОРЫ ЦИКЛА . . . . .</b>	<b>118</b>
Циклы с предусловием и с постусловием . . . . .	118
Решение задач с помощью циклов с постусловием и с предусловием . . . . .	122
Задачи 9.1–9.12. Циклы While и Repeat . . . . .	129
Оператор цикла с параметром . . . . .	130
Решение задач с помощью оператора цикла с параметром . . . . .	132
Задачи 9.13–9.21. Цикл For . . . . .	135
Задачи 9.22–9.24. Разные циклы . . . . .	136
Цикл со сложным условием. Досрочный выход из цикла . . . . .	137
Процедура Break . . . . .	141
Обработка последовательностей . . . . .	142

Задачи 9.25–9.55. Работа с последовательностью . . . . .	152
Вокруг максимума . . . . .	155
Задачи 9.56–9.65. Поиск наибольших и наименьших значений . . . . .	160
Вложенные циклы. . . . .	161
Задачи 9.66–9.70. Вложенные циклы . . . . .	163
Решение задач методом перебора . . . . .	164
Задачи 9.71–9.74. Метод перебора . . . . .	166
Работа с таблицами . . . . .	166
Задачи 9.75–9.81. Работа с таблицами . . . . .	169
Задачи 9.82–9.115. Оператор цикла. Разные задачи . . . . .	171

## **ГЛАВА 10. МАССИВ. . . . . 174**

Задание типов . . . . .	174
Тип данных «Массив» . . . . .	176
Задачи 10.1–10.21. Массив. Заполнение, печать . . . . .	188
А нужен ли массив? . . . . .	190
Перестановка элементов массива . . . . .	193
Задачи 10.22–10.29. Перестановка элементов . . . . .	197
Сортировка . . . . .	197
Задачи 10.30–10.35. Сортировка . . . . .	205
Поиск в массиве . . . . .	206
Вспомогательный массив. . . . .	211
Метод подсчета . . . . .	214
Задачи 10.36–10.45. Метод подсчета . . . . .	222
Строки . . . . .	223
Задачи 10.46–10.58. Символьные массивы, строки . . . . .	237
Матрицы . . . . .	238
Решение задач с матрицами . . . . .	241
Задачи 10.59–10.71. Работа с матрицей . . . . .	247

## **ГЛАВА 11. ПРОЦЕДУРЫ И ФУНКЦИИ . . . . . 248**

Описание процедур и функций . . . . .	250
Обращение к подпрограмме. Фактические параметры . . . . .	251
Принцип локализации . . . . .	253
Задачи 11.1–11.3. Вызов процедуры и функции . . . . .	255
Работа с процедурами. . . . .	256
Задачи 11.4–11.12. Процедуры с входными параметрами и функции. . . . .	263
Параметры-переменные и параметры-значения . . . . .	264
Примеры использования процедур и функций . . . . .	267
Задачи 11.13–11.21. Процедуры и функции с входными и выходными параметрами. . . . .	270



<b>ГЛАВА 12. РЕКУРСИЯ</b> .....	<b>272</b>
Работа рекурсивных процедур и функций .....	272
Задачи 12.1–12.5. Работа рекурсивных процедур и функций .....	277
Рекурсивные алгоритмы .....	278
Задачи 12.6–12.19. Написать рекурсивную процедуру или функцию .....	288
<b>ГЛАВА 13. РАБОТА С ФАЙЛАМИ</b> .....	<b>290</b>
Описание файла .....	291
Стандартные процедуры и функции для работы с файлами .....	291
Примеры работы с файлами .....	297
Задачи 13.1–13.17. Типизированные файлы .....	301
Текстовые файлы .....	303
Задачи 13.18–13.30. Текстовые файлы .....	310
<b>ГЛАВА 14. КОМБИНИРОВАННЫЙ ТИП (ЗАПИСЬ)</b> .....	<b>312</b>
Работа с типом «запись» .....	313
Задачи 14.1–14.5. Работа с записями .....	319
<b>ГЛАВА 15. НЕКОТОРЫЕ ДОПОЛНИТЕЛЬНЫЕ ПРОЦЕДУРЫ И ФУНКЦИИ ЯЗЫКА ТУРБО ПАСКАЛЬ</b> .....	<b>321</b>
Функция Random .....	321
Задачи 15.1–15.11. Работа с генератором случайных чисел .....	323
Модуль CRT .....	324
Работа со звуком .....	324
Задачи 15.12–15.18. Работа со звуком .....	332
Работа с экраном .....	332
Задачи 15.19–15.42. Работа с экраном .....	341
Работа с клавиатурой .....	343
Задачи 15.43–15.52. Работа с клавиатурой .....	354
Задачи 15.53–15.61. Общие задачи с модулем CRT .....	355
<b>ГЛАВА 16. РАЗНЫЕ ЗАДАЧИ</b> .....	<b>356</b>

*Минимальные системные требования определяются соответствующими требованиями программы Adobe Reader версии не ниже 11-й для платформ Windows, Mac OS, Android, iOS, Windows Phone и BlackBerry; экран 10"*

*Учебное электронное издание*

Серия: «ВМК МГУ — школе»

**Грацианова Татьяна Юрьевна**

## **ПРОГРАММИРОВАНИЕ В ПРИМЕРАХ И ЗАДАЧАХ**

Ведущий редактор *М. С. Стригунова*. Художник *В. Е. Шкерин*

Компьютерная верстка: *С. А. Янковая*

Подписано к использованию 06.09.16.

Формат 145×225 мм

Издательство «Лаборатория знаний»

125167, Москва, проезд Аэропорта, д. 3

Телефон: (499) 157-5272, e-mail: [info@pilotLZ.ru](mailto:info@pilotLZ.ru), <http://www.pilotLZ.ru>



**Факультет вычислительной математики и кибернетики  
МГУ имени М. В. Ломоносова**

## **ПОДГОТОВИТЕЛЬНЫЕ КУРСЫ**

проводят обучение

**по**

**МАТЕМАТИКЕ**

**ФИЗИКЕ**

**ИНФОРМАТИКЕ**

**РУССКОМУ ЯЗЫКУ**

учащихся 9-х (*трехгодичная программа*), 10-х (*двухгодичная программа*)  
и 11-х классов (*девятимесячная, шестимесячная и трехмесячная программы*)  
в целях подготовки к сдаче школьных выпускных экзаменов (ЕГЭ)  
и вступительных испытаний в вузы.

Для жителей Подмосковья и ближайших областей организуются  
группы выходного дня (*только для 11-х классов*) с занятиями по субботам.

Занятия на подготовительных курсах  
проходят в вечернее время  
с 18.00 до 21.10

в учебных аудиториях факультета вычислительной математики и кибернетики  
в группах по 15–16 человек (*метро «Университет»*).

Набор на трехгодичную, двухгодичную и на девятимесячную программы  
проходит с 10 по 20 мая и с 1 сентября по 20 сентября,  
на шестимесячную программу – в конце декабря,  
на трехмесячную – в конце марта.



**<http://www.vmk-edu.ru>**

Справки по телефону  
932-98-08

с 16 часов до 19 часов в рабочие дни.

*Учащимся, не имеющим возможности приехать на занятия,  
предлагаются дистанционные подготовительные курсы:*

**[www.ecmc.ru](http://www.ecmc.ru)**



**Факультет вычислительной математики и кибернетики  
МГУ имени М. В. Ломоносова**

## **КОМПЬЮТЕРНЫЕ КУРСЫ**

### **Курсы для школьников:**

работа на компьютере для школьников 3–5 кл., программирование для школьников младшего возраста, базовая подготовка для начинающих (6–11 кл.), создание сайтов, Интернет, Flash (основы, мультстудия), графика (Photoshop, CorelDraw), программирование (Паскаль, DELPHI, C, C++, Java,), создание домашней компьютерной сети.

Организованным группам школьников предоставляется скидка.

### **Компьютер для начинающих и углубленно:**

Windows, офисные программы, Internet, электронная почта. Компьютер для работы в офисе.

### **Построение сайтов:**

HTML, CSS, JavaScript, управление сайтами, администрирование веб-сервера Apache, разработка веб-приложений на языке PHP.

### **Компьютерная графика и верстка:**

Photoshop, CorelDraw, Flash, QuarkXPress, AutoCAD, основы цифровой фотографии.

### **Профессиональные курсы:**

NET, SQL Server, Windows Server 2003, ISA Server 2004, Oracle 10G, Comp TIA Security, управление IT-процессами, C, C++, C#, Java, Flash, 1C, создание малой компьютерной сети для офиса и дома.



Будни и выходные

**[www.vmk-edu.ru](http://www.vmk-edu.ru)**

(495) 939-54-29, 939-36-04

м. «Университет»

*Занятия в течение учебного года 1–2 раза в неделю*

*Интенсивные курсы в июне*



ВМК МГУ – ШКОЛЕ



Развитие и широкое распространение компьютеров вызывают насущную потребность в высококвалифицированных специалистах в области прикладной математики, вычислительных методов и информатики. Сегодня наш факультет – один из основных факультетов Московского университета, ведущий учебный и научный центр России в области фундаментальных исследований и образования по прикладной математике, информатике и программированию.

Высокая квалификация преподавателей и сотрудников факультета, сочетание их глубокого теоретического и практического опыта являются залогом успешной работы наших выпускников в ведущих научных центрах, промышленных, коммерческих и других учреждениях.

Факультет не только учит студентов, но и ведет большую работу со школьниками и учителями:

- на факультете работают вечерняя математическая школа, подготовительные курсы и компьютерные курсы для школьников;
- для учителей есть курсы повышения квалификации и ежегодно проводятся летние школы по математике и информатике;
- сотрудники факультета и преподаватели других факультетов МГУ, работающие на подготовительных курсах факультета, готовят учебные и методические пособия по математике, информатике и физике как для школьников, так и для учителей.

Мы рады видеть новых студентов и приветствуем новых партнеров в научном сотрудничестве и инновационной деятельности.

*Декан факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова,  
академик РАН **Е. И. Мусеев***

Сайт факультета ВМК МГУ:

<http://www.cs.msu.ru>

