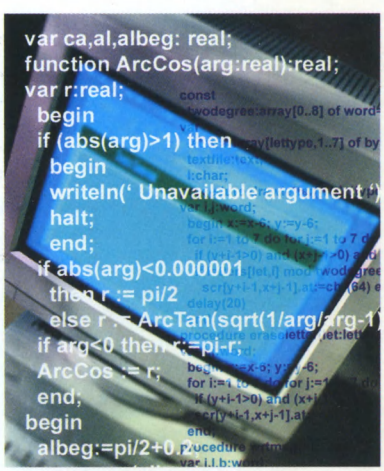
Полное руководство

ШАГ ЗА ШАГОМ

Паскаль – это лучший язык для начинающих, отличающийся четкой структурой и богатыми возможностями. Освоив его, вы получите прекрасное представление о программировании и составлении алгоритмов.

Турбо Паскаль 7.0

- Программирование «с нуля» на основе самого популярного языка для начинающих
- Множество примеров программ, иллюстрирующих решение реальных задач программирования
- Все операторы и конструкции языка, которые понадобятся вам при работе с любым проектом



Алексеев Е. Р., Чеснокова О. В.

Шаг за шагом

Алексеев Е. Р., Чеснокова О. В.

Турбо Паскаль 7.0

NT Press
Москва, 2005

УДК 004.438
ББК 32.973.26-018.1
А47

Подписано в печать 16.11.04. Формат 70x90/16. Гарнитура "Баскервиль".
Печать офсетная. Усл. печ. л. 23,3. Доп. тираж 4000 экз. Заказ № 4886.

Алексеев, Е.Р.

А47 Турбо Паскаль 7.0 / Алексеев Е.Р., Чеснокова О.В. — М.: НТ Пресс, 2005. — 314, [6] с.: ил. — (Шаг за шагом).

ISBN 5-477-00012-0

Книга адресована изучающим алгоритмизацию и программирование. Прочитав ее, вы научитесь составлять алгоритмы и программы. Книга посвящена языку программирования Турбо Паскаль версии 7.0. Приведено большое количество практических примеров программирования. Подробно описаны такие этапы программирования, как работа с подпрограммами, модулями, файлами, экраном дисплея в текстовом и графическом режимах.

Издание предназначено для школьников и студентов, начинающих изучать программирование, а также для всех желающих познакомиться с языком Турбо Паскаль.

УДК 004.438
ББК 32.973.26-018.1

Содержание

Предисловие

Цель авторов данной книги – научить читателей программированию. Основные проблемы при написании программ возникают на этапе разработки алгоритма решения задачи. В качестве средства обучения составлению алгоритмов авторы выбрали язык блок-схем. Языком программирования был выбран Турбо Паскаль, разработанный фирмой Borland. Все примеры и задачи, приведенные в книге, работают как в среде Borland Pascal, так и в среде Turbo Pascal; без серьезных изменений эти программы могут быть перенесены в среду компилятора Free Pascal. Все программы тестировались авторами, но если читатель обнаружит ошибки, просьба сообщить об этом по адресу teacher@teacher.dn-ua.com.

Введение

В настоящее время существует множество подходов к изучению программирования. По мнению авторов, на первом этапе необходимо ознакомиться с методами составления алгоритмов без привязки к конкретному языку. Одним из наиболее наглядных методов составления алгоритмов является язык *блок-схем.* Об этом свидетельствует и опыт преподавания программирования. Однако, несмотря на огромное количество хороших книг по программированию, появившихся в последнее время, практически неохваченным остался такой раздел учебной литературы, как изложение методов составления алгоритмов и обучение составлению блок-схем.

Авторы постарались заполнить этот пробел, написав книгу, которая соединила в себе учебник по алгоритмизации и программированию. Насколько нам удалось решить поставленную задачу – судить читателю.

Авторы надеются, что читатель умеет обращаться с персональным компьютером, знаком с файловой системой, имеет опыт работы в Windows и помнит школьный курс математики.

Мы выбрали язык программирования Турбо Паскаль, который представляется нам ясным, логичным и гибким и приучает к хорошему стилю программирования. Освоив его, можно перейти к серьезному профессиональному программированию в среде Borland Delphi.

Книга состоит из двенадцати глав и двух приложений.

В главе 1 читатель узнает о структуре программы на языке Паскаль, типах данных и некоторых встроенных функциях языка. В этой же главе кратко описана оболочка Турбо Паскаля.

В главе 2 изложены основные элементы языка (переменные, выражения, операторы) Турбо Паскаль. Описаны простейшие операторы языка: присваивания и ввода-вывода, приведена структура программы на языке Паскаль, а также примеры простейших программ линейной структуры.

Глава 3 является одной из ключевых в изучении программирования. В ней изложена методика составления алгоритмов с помощью блок-схем. Приведено большое количество примеров блок-схем алгоритмов и программ различной сложности. Авторы рекомендуют внимательно разобрать все примеры и выполнить упражнения данной главы и только после этого приступать к изучению последующих глав книги.

Главы 4 и 5 посвящены изучению алгоритмов обработки массивов и матриц. Здесь же читатель познакомится с реализацией рассмотренных алгоритмов в Паскале. Эти главы совместно с главой 3 являются ключом к пониманию принципов программирования.

В главе 6 читатель познакомится с подпрограммами. Описан механизм передачи параметров между подпрограммами с использованием открытых массивов и процедурных типов. Последний раздел посвящен рекурсивным подпрограммам.

Глава 7 знакомит читателя с использованием файлов в Турбо Паскале. На практических примерах показан механизм прямого и последовательного доступа к файлам и обработки ошибок ввода-вывода. Описана работа с бестиповыми и текстовыми файлами.

Глава 8 посвящена обработке строк.

В главе 9 описаны принципы редактирования таблиц с помощью специального типа данных Турбо Паскаля – записей. Все объяснения подкреплены примерами.

В главе 10 описаны указатели, работа с динамическими массивами и матрицами. Приведен текст подпрограмм, позволяющих работать с массивами более 64 Кб.

Глава 11 посвящена модулям Турбо Паскаля, описаны стандартные модули CRT, PRINTER, DOS. Кратко изложен механизм создания собственных модулей.

В главе 12 рассмотрены графические средства Турбо Паскаля, содержится подробное описание алгоритма построения графиков непрерывных функций на экране дисплея. Приведены тексты программ изображения графиков непрерывных и разрывных функций с подробными комментариями, а также текст модуля построения графиков функций и поверхностей.

В приложении 1 кратко описан отладчик среды Турбо Паскаля.

В приложении 2 приведены некоторые ресурсы Internet, посвященные алгоритмизации и программированию на языке Паскаль.

Авторы надеются, что большое количество упражнений, содержащихся в книге, поможет использовать ее не только начинающим самостоятельно изучать программирование, но и преподавателям. Нам было бы интересно получить отзывы от коллег.

Система обозначений

В книге выделены:

- программные элементы языка Паскаль и имена переменных – моноширинным шрифтом;
- определения, новые понятия – *курсивом*;
- элементы файловой системы, сочетания клавиш, элементы оболочки Турбо Паскаля – **полужирным шрифтом**;
- в текстах программ комментарии выделены фигурными скобками {}.

Глава 1 Знакомимся с Паскалем

В этой главе читатель начнет знакомство с Турбо Паскалем. Авторы предлагают начать изучение языка с написания простой программы. На этом примере читатель узнает о структуре программы на языке Паскаль, типах данных и некоторых встроенных функциях языка. В этой же главе читатель познакомится с оболочкой Турбо Паскаля, научится вводить программу в компьютер, обнаруживать и исправлять синтаксические ошибки, а также узнает, как запустить программу и увидеть результаты ее работы.

Язык Паскаль был разработан Николасом Виртом в шестидесятые годы прошлого века как учебный язык для студентов. Современный Турбо Паскаль сохранил его простоту и структуру. Это достаточно мощное средство программирования, предназначенное для написания программ различной сложности. На Турбо Паскале можно выполнить простые расчеты, составить программы для реализации сложных инженерных задач, обучающие и тестирующие программы, программы-оболочки и драйверы.

В книге будет рассматриваться седьмая версия Турбо Паскаля (TP 7.0). Эта разработка американской фирмы Borland дала развитие языку Object Pascal, который лежит в основе системы визуального программирования для Windows – Delphi. Познакомившись с программированием на Турбо

Паскале, вы сможете перейти к программированию для Windows с использованием Delphi.

1.1. Первая программа на Паскале

Знакомство с языком начнем с решения следующей задачи.

Пример 1.1. Заданы длины двух катетов прямоугольного треугольника *a*, *b*. Вычислить длину гипотенузы *c* и величины двух его углов α и β (рис. 1.1). Значения *a*, *b*, *c* ввести с клавиатуры.

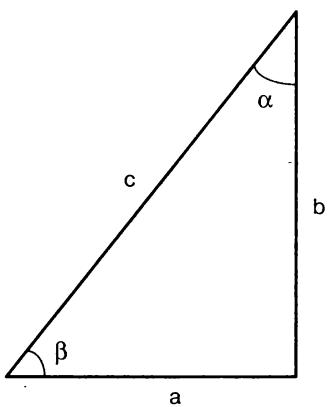


Рис. 1.1 ▼ Прямоугольный треугольник

Перед написанием программы давайте вспомним основные формулы, которые нам понадобятся. Гипотенуза *c* вычисляется по формуле

$$c = \sqrt{a^2 + b^2},$$

углы треугольника α и β рассчитываются следующим образом:

$$\alpha = arctg\left(\frac{a}{b}\right),\ \beta = arctg\left(\frac{b}{a}\right).$$

Решение задачи можно разбить на следующие этапы:

1. Определение значений *a*, *b* (ввод величин *a* и *b* с клавиатуры в память компьютера).
2. Расчет значений *c*, α и β по приведенным выше формулам.
3. Вывод значений *c*, α и β на экран дисплея.

Ниже приведен текст программы. Сразу заметим, что в фигурных скобках в тексте присутствуют комментарии, которые не являются обязательными элементами программы и ничего не сообщают компьютеру. Они поясняют человеку, читающему текст, назначение отдельных элементов программы. В книге комментарии будут широко использоваться для пояснения отдельных участков программы.

```
{ Заголовок программы Pr_1. }
Program Pr_1;
{ Раздел описаний. }
var
  a,b,c,alf,bet:real;
{ Тело программы. }
begin
  write('a=');read(a);
  write('b=');read(b);
  c:=sqrt(a*a+b*b);
  alf:=arctan(a/b);
  bet:=arctan(b/a);
  writeln('c=',c:6:2);
  writeln('Радианы');
  writeln('alf=',alf:6:2,' bet=',bet:6:2);
  writeln('Градусы');
  writeln('alf=',alf*180/pi:3:0,' bet=',bet*180/pi:3:0)
end.
```

1.2. Неформальное введение в Паскаль

В классическом языке Паскаль программа должна начинаться с *заголовка*, в котором первым словом будет слово `program`. За ним следует *имя* программы. В нашем примере заголовок имеет вид:

```
Program Pr_1;
```

Имя[1] состоит из латинских букв[2], цифр и символа подчеркивания. Оно не может начинаться с цифры. В Турбо Паскале нет обязательного требования наличия заголовка программы. Следовательно, строку `Program Pr_1;` можно не использовать.

После заголовка идет *раздел описаний.* В нем описываются все объекты, которые будут использоваться в программе. В нашем случае в программе присутствует пять переменных: *a, b* (значения катетов) – исходные данные; *c* (значение гипотенузы), α и β (величины углов) – результаты. Для их описания используется служебное слово `var`, после которого перечисляются переменные и через двоеточие указывается их тип. Служебное слово `real` означает, что переменные относятся к вещественным (действительным) числам.

За разделом описаний следует *исполняемая часть программы*[3]. Она начинается со служебного слова `begin`, заканчивается словом `end` (и точкой). В исполняемой части следуют операторы языка Паскаль, предназначенные для реализации задачи. Друг от друга они отделяются точкой с запятой.

Рассмотрим основные операторы, используемые для решения поставленной задачи. В нашей программе используются операторы ввода (`read`), вывода (`write`, `writeln`) и присваивания.

Оператор вывода может выводить значения переменных, выражений и текстовую информацию. Для вывода текста его необходимо заключить в кавычки. Оператор `write` просто выводит информацию на экран дисплея, а оператор `writeln` после вывода информации переводит курсор в следующую строку. Оператор `read` предназначен для ввода. Если он встречается в теле программы, то ее выполнение приостанавливается до тех пор, пока пользователь не введет необходимые значения. Теперь последовательно опишем назначение каждого оператора.

```
write('a=');read(a);
```

Оператор `read(a)` предназначен для ввода значения переменной `a` (целую и дробную часть числа в Паскале следует разделять точкой). После ввода пользователь должен нажать клавишу **Enter**. Оператор `write('a=')` выводит на экран два символа `a=`, которые подсказывают пользователю, что он должен ввести значение переменной `a`.

```
write('b=');read(b);
```

---

[1] В специальной литературе вместо имени иногда используется термин «идентификатор».

[2] В языке Паскаль большие и малые буквы равнозначны, имена `PR_1`, `pr_1`, `Pr_1` и `pR_1` эквивалентны.

[3] В литературе по Паскалю вместо выражения «исполняемая часть программы» используется термин «тело программы». В нашей книге мы также будем его употреблять.

Эти два оператора выводят подсказку b= и осуществляют ввод значения b.

```
c:=sqrt(a*a+b*b);
```

Этот оператор присваивания выполняет следующее: с помощью операции умножения (*) вычисляются b^2 и a^2; из их суммы (операция +)[1] извлекается квадратный корень с помощью функции sqrt. Полученный результат записывается в переменную c[2]:

```
alf:=arctan(a/b);
bet:=arctan(b/a);
```

Эти операторы присваивания вычисляют значения углов прямоугольного треугольника с помощью функций arctan (арктангенс) и записывают в переменные alf и bet. Следует учесть, что при использовании тригонометрических функций углы вычисляются в радианах. Поэтому в дальнейшем тексте программы предусмотрен вывод значений углов как в радианах, так и градусах.

```
writeln('Радианы');
```

Выводит на экран слово Радианы и переводит курсор в новую строку.

```
writeln('alf=',alf:6:2,' bet=',bet:6:2);
```

Этот оператор предназначен для вывода на экран значений углов в радианах. Оператор выводит строку alf=, значение переменной alf, затем – строку bet= и значение переменной bet. Переменные alf и bet будут занимать шесть позиций на экране, из них две предназначены для вывода дробной части числа (для этого в операторе writeln используется формат 6:2)[3]. На этом написание программы можно было бы закончить, но более привычным является представление значений углов не в радианах, а в градусах. Для этого величину в радианах надо умножить на $\frac{180}{\pi}$. Для вычисления α в Турбо Паскале предусмотрена встроенная функция pi. Для вывода на экран значения alf и bet в градусах в формате 3:0 (без дробной части) в нашей программе используется оператор writeln.

```
writeln('alf=',alf*180/pi:3:0,' bet=',bet*180/pi:3:0)
```

Еще одним важным моментом при написании программ на Турбо Паскале является использование точки с запятой. Символ ; отделяет один оператор в теле программы от другого. Поэтому после последнего оператора он отсутствует.

---

[1] Кроме этих операций в Паскале присутствуют операции вычитания (-) и деления (/).

[2] Переменной c присваивается значение, поэтому и оператор присваивания.

[3] Подробно о форматах вывода вещественных будет рассказано во второй главе.

Кроме того, заголовок программы и раздел описаний так же заканчиваются этим символом. Подробно об использовании точки с запятой будет рассказано в главе 2.

Таким образом, наша программа вводит значение катетов, затем вычисляет значения гипотенузы и углов, после чего выводит их на экран, причем величины углов указываются как в радианах, так и градусах.

1.3. Ввод программы в компьютер

После того как программа написана, ее необходимо ввести в компьютер. Для этого необходимо запустить оболочку Турбо Паскаля с помощью файла turbo.exe. Ее экран представлен на рис. 1.2.

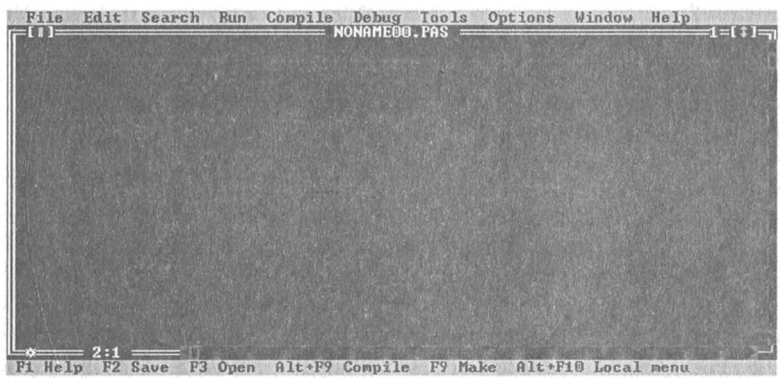


Рис. 1.2 ▼ Внешний вид экрана оболочки Турбо Паскаля

В состав оболочки входит текстовый редактор, транслятор[1] и отладчик. В этой главе мы расскажем о текстовом редакторе и трансляторе. С отладчиком читатель познакомится в приложении 1.

Параллельно с набором текста нашей программы давайте рассмотрим основные возможности редактора оболочки Турбо Паскаля.

[1] Транслятор – программа, которая переводит текст программы с Паскаля на машинный язык.

1.3.1. Основные приемы работы с текстовым редактором Турбо Паскаля

Редактор оболочки Турбо Паскаля предоставляет пользователю удобные средства создания и редактирования текстов программ. Признаком того, что оболочка находится в состоянии редактирования, является наличие в окне редактора курсора в виде небольшого мигающего прямоугольника. Данный режим автоматически устанавливается после загрузки Турбо Паскаля. Из него можно перейти к любому другому режиму работы Турбо Паскаля с помощью функциональных клавиш или команд главного меню. Если оболочка находится в состоянии выбора режима в меню, курсор исчезает, а в строке меню появляется цветной указатель – прямоугольник, выделяющий один из пунктов меню. Для перехода из главного меню в состояние редактирования нужно нажать клавишу **Esc**, а для перехода из режима редактирования к главному меню – **F10**.

Окно редактора имитирует длинный и достаточно широкий лист бумаги, фрагмент которого виден в окне. Если курсор достиг нижнего края, осуществляется прокрутка окна редактора: его содержимое смещается вверх на одну строку, и снизу появляется новая строка листа. Когда курсор достигает правой границы экрана, окно начинает по мере ввода смещаться вправо, показывая правый край листа. Не следует вводить строки программы длиной более 126 символов.

Для создания текста программы нужно ввести символы с помощью клавиатуры ПК. После заполнения очередной строки следует нажать клавишу **Enter**, которая вставляет специальный символ конца строки, после чего курсор окажется на следующей строке (он всегда показывает то место на экране, куда будет помещен очередной вводимый символ программы).

Перемещать курсор по экрану можно при помощи клавиш передвижения. Если вы ошиблись при вводе очередного символа, сотрите его с помощью клавиши **Backspace**. Клавиша **Delete** удаляет символ, под которым в данный момент находится курсор, а комбинация клавиш **Ctrl+Y** – всю строку, на которой он расположен.

Чтобы «разрезать» строку, следует установить курсор в нужную позицию и нажать клавишу **Enter**; чтобы «склеить» соседние строки, нужно переместить курсор в конец первой строки (для этого удобно использовать клавишу **End**) и

нажать клавишу **Delete** или поставить его в начало следующей строки (клавишей **Home**) и нажать клавишу **Backspace**.

Нормальный режим работы редактора – режим вставки, в котором каждый вновь вводимый символ как бы раздвигает текст на экране, смещая вправо остаток строки. Следует учитывать, что «разрезание» и последующая вставка пропущенных строк возможны только в этом режиме. Редактор может также работать в режиме замены, когда новый символ заменяет собой тот, перед которым стоит курсор, а остаток строки справа от курсора не смещается. Для перехода к режиму наложения нужно нажать клавишу **Insert**; если нажать на нее еще раз, вновь восстановится режим вставки. Понять, в каком режиме работает редактор, можно по форме курсора: в режиме вставки он похож на мигающий символ подчеркивания, а в режиме замены представляет собой крупный мигающий прямоугольник, заслоняющий символ целиком.

И еще об одной возможности редактора. Обычно он работает в режиме автоотступа, когда каждая новая строка начинается в той же позиции на экране, что и предыдущая. Этот режим обеспечивает хороший стиль оформления текстов программ. Отступы от левого края выделяют тело условного или составного оператора и делают программу более наглядной. Отказаться от автоотступа можно с помощью сочетания клавиш **Ctrl+O+1** (удерживая клавишу **Ctrl**, нажмите сначала на клавишу **O**, затем отпустите ее и нажмите клавишу **1**), повторное использование сочетания клавиш **Ctrl+O+1** восстановит данный режим.

Перемещать курсор можно с помощью следующих клавиш:

- **Page Up** – на страницу вверх;
- **Page Down** – на страницу вниз;
- **Home** – в начало строки;
- **End** – в конец строки;
- **Ctrl+Page Up** – в начало текста;
- **Ctrl+Page Down** – в конец текста.

1.3.2. Работа с блоками

При подготовке текстов программ часто возникает необходимость перенести фрагмент текста в другое место или удалить его. Для такого рода операций удобно использовать *блоки* – фрагменты текста, рассматриваемые как единое целое. Длина блока может быть достаточно большой (до 64 Кб), и тогда он занимает несколько экранных страниц. В каждый момент в оболочке может быть объявлен только один блок в одном окне редактора.

Сочетания клавиш, предназначенные для работы с блоком, таковы:

- **Ctrl+K+B** – пометить начало блока;
- **Ctrl+K+K** – пометить конец блока;
- **Ctrl+K+T** – пометить в качестве блока слово слева от курсора;
- **Ctrl+K+Y** – стереть блок;
- **Ctrl+K+C** – копировать блок в позицию, где находится курсор;
- **Ctrl+K+V** – переместить блок в позицию, где находится курсор;
- **Ctrl+K+W** – записать блок в файл;
- **Ctrl+K+R** – прочитать блок из файла;
- **Ctrl+K+P** – напечатать блок;
- **Ctrl+K+H** – снять пометку блока (повторное использование этого клавиатурного сочетания вновь выделит блок).

1.3.3. Работа с файлами

Как уже говорилось, сразу после запуска Турбо Паскаля вы попадаете в режим редактирования текста, в котором можно подготовить новую программу или исправить существующую.

Тексты программ хранятся в виде файлов на диске. После завершения работы с Турбо Паскалем можно сохранить текст новой программы в файле на диске, чтобы иметь возможность использовать его в следующий раз. Для обмена данными между файлами и редактором предназначены клавиши **F2** (запись в файл) и **F3** (чтение из файла).

После нажатия на клавишу **F3** на экране появляется окно выбора файла (рис. 1.3).

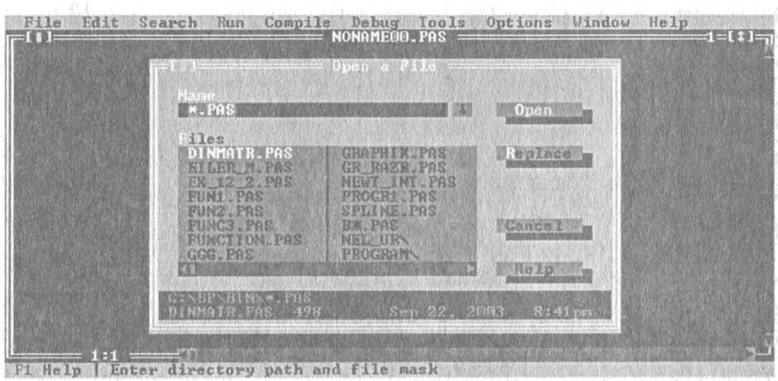


Рис. 1.3 ▾ Диалоговое окно выбора файла

Для загрузки файла в поле ввода (**Name**) нужно ввести его имя. Если в названии опущено расширение, автоматически добавится стандартное расширение .PAS. Имени файла может предшествовать путь. Нужный файл вы можете также выбрать из поля выбора, предварительно активизировав его клавишей **Tab**. Кнопка **Open** (Открыть) используется для открытия файла в новом окне редактирования, **Replace** (Заменить) – для замены существующего в активном окне текста на текст, считанный из файла, **Cancel** (Отменить) – закрыть диалоговое окно без выполнения каких-либо действий.

Если вы создаете новую программу, то по умолчанию тексту программы присваивается стандартное имя NONAMEOO.PAS (no name – нет имени).

Если для сохранения текста программы в файле нажать на клавишу **F2**, то при первом сохранении появится диалоговое окно выбора файла для сохранения, аналогичное изображенному на рис. 1.3. В этом окне необходимо ввести или выбрать имя файла (если расширение опущено, то автоматически добавится стандартное расширение .PAS). Если файл уже сохранялся, то нажатие клавиши **F2** просто сохранит файл под тем же именем без всяких дополнительных вопросов.

Завершение работы с оболочкой Турбо Паскаля осуществляется с помощью комбинации клавиш **Alt+X**. Если при этом не сохранен текст программы на диске, на экране появится окно с запросом:

Filename.PAS[1] has been modified. Save?

(Файл Filename.PAS был изменен. Сохранить?)

В ответ следует нажать на кнопку **Y** (Yes – да), если необходимо сохранить текст в файле, или **N** (No – нет), если делать этого не нужно.

После набора текста нашей программы окно оболочки Турбо Паскаля примет вид, показанный на рис. 1.4.

1.3.4. Работа с окнами

В редакторе Турбо Паскаля можно редактировать не один файл, а несколько. Для создания нового текстового окна необходимо выполнить команды **File/New** (Файл/Новый). Второму окну присвоится стандартное имя **NONAMEO1.PAS**, третьему – **NONAMEO2.PAS** и т.д. Переключиться между окнами можно двумя способами:

- для переключения в окно с номером от первого до девятого нажать комбинацию клавиш (**Alt+клавиша от 1 до 9** в зависимости от номера окна);

[1] Здесь Filename – имя реального файла.

Рис. 1.4 ▾ Окно Турбо Паскаля с текстом программы примера 1.1

➤ для вывода списка окон на экран нажать комбинацию клавиш **Alt+0**, появится список активных окон (рис. 1.5), в котором необходимо будет выбрать нужное и нажать **Enter**.

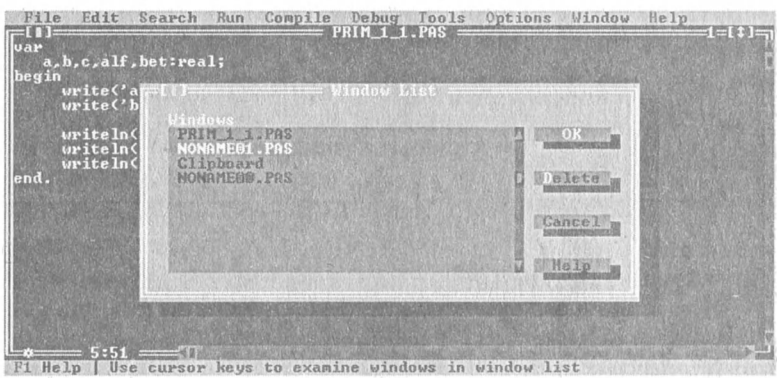


Рис. 1.5 ▾ Окно Турбо Паскаля со списком активных окон редактора

Обмен информацией между окнами возможен через буфер обмена редактора. Для этого предназначен пункт меню редактора Турбо Паскаля **Edit** (Правка). В нем есть следующие пункты:

➤ **Undo** (Отменить) – отменяет последнее изменение в текстовом редакторе Турбо Паскаля, команда может быть вызвана комбинацией клавиш **Alt+Backspace**;

- **Redo** (Восстановить) – отменяет действие команды **Undo**;
- **Cut** (Вырезать) – переносит выделенный блок из окна редактора в буфер обмена; команда может быть вызвана комбинацией клавиш **Shift+Delete**;
- **Copy** (Копировать) – копирует выделенный блок из окна редактора в буфер обмена, команда может быть вызвана комбинацией клавиш **Ctrl+Insert**;
- **Paste** (Вставить) – вставляет содержимое буфера обмена в то место окна редактора, где находится курсор; команда может быть вызвана комбинацией клавиш **Shift+Insert**;
- **Clear** (Очистить) – удаляет выделенный блок, но не помещает его в буфер; команда может быть вызвана комбинацией клавиш **Ctrl+Delete**;
- **Show Clipboard** (Показать буфер обмена) – открывает окно с буфером обмена.

Для того чтобы скопировать блок из одного окна в другое, необходимо выполнить следующие действия:

1. Выделить блок.
2. Скопировать его в буфер обмена с помощью команд **Edit/Copy** (комбинация клавиш **Ctrl+Insert**).
3. Перейти в нужное окно.
4. Вставить блок с помощью команд **Edit/Paste** (комбинация клавиш **Shift+Insert**).

При переносе блока из одного окна в другое вместо команд **Edit/Copy** следует выполнить команды **Edit/Cut** (комбинация клавиш **Shift+Delete**).

1.4. Запуск программы и просмотр результатов

После того как текст программы был набран, его следует перевести в машинный код. Для этого необходимо вызвать транслятор с помощью команды `Compile-Compile` (комбинация клавиш **Alt+F9**). На первом этапе транслятор проверяет наличие синтаксических ошибок. Если в программе их нет, то на экране появляется сообщение о количестве строк транслированной программы и объеме доступной оперативной памяти (рис. 1.6).

Если на каком-либо этапе транслятор обнаружит ошибку, то в окне редактора курсор указывает ту строку программы, в которой при трансляции была

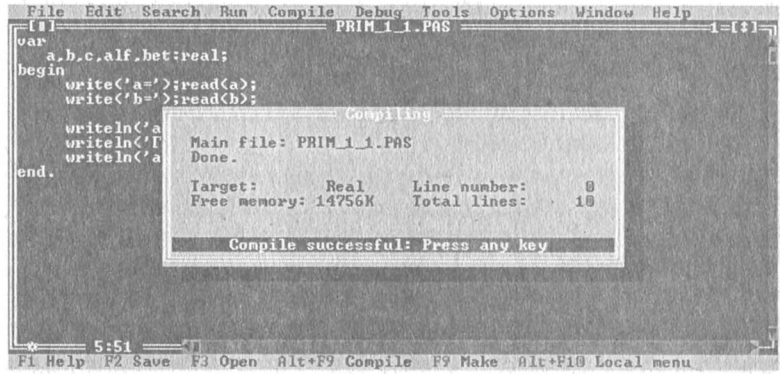


Рис. 1.6 ▾ Окно Турбо Паскаля после трансляции программы

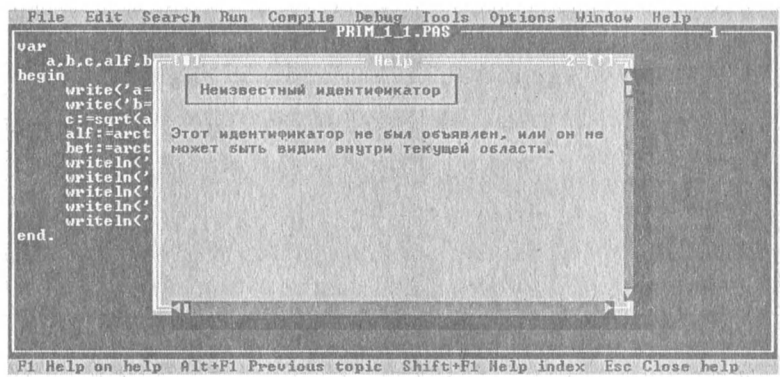


Рис. 1.7 ▾ Сообщение об ошибке трансляции

обнаружена ошибка. При этом в верхней строке редактора появляется краткое диагностическое сообщение о причине сбоя. Нажав клавишу **F1**, пользователь сможет получить более подробную информацию о причинах ошибки (рис. 1.7).

В стандартной поставке вся справочная информация Турбо Паскаля выдается на английском языке. На сайте авторов книги (www.teacher.dn-ua.com/uchebn/file/help_rus.zip) присутствует русскоязычный файл справки. Если читатель заменит им стандартный файл справки Турбо Паскаля, то получит оболочку Турбо Паскаля с русскоязычной справкой.

Для запуска транслированной программы необходимо выполнить команду Run-Run (комбинация клавиш **Ctrl+F9**)[1], после чего на экране появляется окно командной строки Windows, в котором пользователь и осуществляет диалог с программой. После завершения работы программы пользователь опять видит экран оболочки Турбо Паскаля.

Для просмотра результатов работы программы необходимо использовать комбинацию клавиш **Alt+F5**. Для возврата в оболочку следует нажать любую клавишу. Результаты работы программы из примера 1.1 представлены на рис 1.8.

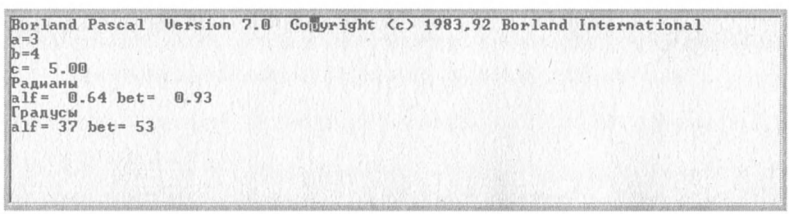


Рис. 1.8 ▼ Результаты работы программы из примера 1.1

[1] Для того чтобы после трансляции программа автоматически была запущена на выполнение, можно сразу вызывать команду Run-Run. В этом случае сначала происходит компиляция программы, а после ее успешного завершения – запуск программы на выполнение.

Глава 2 Общие сведения о языке Паскаль

В этой главе читатель познакомится с основными элементами языка Турбо Паскаль: переменными, константами, их типами, основными операциями и функциями языка. Здесь же описаны простейшие операторы (присваивания и ввода-вывода), приведена структура программы на языке Паскаль.

2.1. Алфавит языка

Программа на языке Паскаль может содержать следующие символы:

- латинские буквы A, B, C, ..., x, y, z;
- цифры 0, 1, 2, ..., 9;
- специальные символы + (плюс), - (минус), / (слэш), = (равно), < > (меньше, больше), [] (квадратные скобки), . (точка), () (круглые скобки), ; (точка с запятой), : (двоеточие), {} (фигурные скобки), $ (символ доллара), # (символ решетки), _ (знак подчеркивания), @ (собачка).

В качестве имен программ, типов, констант, переменных, модулей и других объектов языка используются *идентификаторы*, которые представляют собой совокупность букв, цифр и символа подчеркивания, начинающуюся с буквы или символа подчеркивания. Идентификатор *не* может содержать пробелы. При написании могут быть использованы как прописные, так и строчные буквы. Каждое имя (идентификатор) должно быть уникальным. Его длина не

ограничена. Если в именах первые 63 символа неодинаковые, то имена считаются различными. Большие и маленькие буквы равнозначны.

2.2. Данные в языке Турбо Паскаль

Для решения задачи в любой программе выполняется обработка каких-либо *данных*. Данные могут быть самых различных типов: целые и вещественные числа, символы, строки, массивы. Все данные в языке Паскаль должны быть описаны в начале программы.

Данные языка Паскаль можно разделить на *константы* и *переменные*.

2.2.1. Константы языка Турбо Паскаль

Константы не изменяют своего значения в процессе выполнения программы. Они описываются с помощью служебного слова `const`, за которым следует список имен констант, каждому из которых с помощью символа = присваивается значение. Одна константа от другой отделяется точкой с запятой, например:

```
Const
h=3;
b=-7.5;
c='abcde';
```

2.2.2. Переменные языка Турбо Паскаль

Переменные могут изменять свое значение в процессе выполнения программы неограниченное число раз. Описание переменных начинается со служебного слова `var`, за которым следуют имена переменных и через двоеточие указывается их тип, например:

```
var
a,b: real;
c,d: integer;
```

2.2.3. Типы данных в языке Турбо Паскаль

Типы данных в Паскале можно разделить на *скалярные* и *структурированные*. Существует также возможность вводить пользовательские типы данных.

В скалярных типах данных можно выделить следующие группы.

Целочисленные типы данных занимают от 1 до 4 байт. Все они представлены в табл. 2.1.

Таблица 2.1 ▾ Целочисленные типы данных

Тип	Диапазон	Размер в байтах
Byte	0..255	1
Word	0..65535	2
Integer	–32768..32767	2
Shortint	–128..127	1
Longint	–2147483648..2147483647	4

Пример описания переменных целочисленных типов:

```
Var
a,b:byte;
f:word;
```

Вещественные типы данных занимают от 4 до 10 байт. Они могут быть как с *плавающей* (например, –3.2E–6, –6.42E+2), так и с *фиксированной* (например, 4.12, 6.05, –17.5489) точкой. Вещественные числа в формате с плавающей точкой представлены в экспоненциальной форме `mE±p`, где `m` – мантисса (целое или дробное число с десятичной точкой), `p` – порядок (целое число). Для того чтобы перевести число в экспоненциальной форме к обычному представлению с фиксированной точкой, необходимо мантиссу умножить на 10 в степени порядка. Например:

$-6.42E+2 = -6.42 \cdot 10^{2} = -642;$

$-3.2E-6 = -3.2 \cdot 10^{-6} = -0.0000032.$

Все вещественные типы данных приведены в табл. 2.2.

Таблица 2.2 ▾ Вещественные типы данных

Тип	Диапазон	Мантисса	Размер (в байтах)
Real	2.9E-39..1.7E38	11–12	6
Single	1.5E-45..3.4E38	7–8	4
Double	5.0E-324..1.7E308	15–16	8
Extended	3.4E-4932..1.1E4932	19–20	10

Пример описания переменных вещественных типов:

```
var
a,b,c: real;
d,f: double;
k: single;
```

Символьный тип данных представляет собой любой символ, который может быть отображен на экране дисплея. Он занимает 1 байт и может быть описан с помощью служебного слова char, например:

```
var
a,b: char;
```

В тексте программы значения переменных и константы символьного типа должны быть заключены в апострофы: 'a', 'b', '+'.

Логический (*булевский*) тип данных. Данные этого типа могут принимать одно из двух значений: true (истина) или false (ложь).

Например:

```
var  a,b: boolean;
```

Кроме стандартных скалярных типов в Турбо Паскале можно вводить такие скалярные типы, как перечислимый и интервальный.

Перечислимый тип задается непосредственным перечислением значений, которые может принимать переменная данного типа, например:

```
var
a,c: (red,blue,green);
b: (dog,cat);
```

Можно сначала ввести перечислимый тип данных, а затем описать его переменные. Для создания нового типа используется служебное слово type:

```
type <имя_типа>=<определение_типа>;
```

Например:

```
type
color=(red,blue,green);
var
a,b: color;
```

Интервальный тип данных позволяет задавать две константы, которые определяют границы изменения переменных данного типа. Значение первой константы должно быть меньше значения второй. Сами же они являются целочисленными или символьными, например:

```
var
a,b,c: -7..4;
x: 'a'..'c';
```

Как и в случае перечислимого типа, можно предварительно ввести тип данных с помощью служебного слова type, а затем описывать переменные.

Например:

```
type
x=0..9;
var
a,b: x;
```

Каждая переменная интервального типа занимает 1 байт.

К *структурированным* типам данных относятся: массивы, строки, записи, файлы, множества.

Массив – совокупность данных одного и того же типа. Число элементов массива фиксируется при описании типа и в процессе выполнения программы не изменяется. Для доступа к элементу необходимо указать имя массива и его номер в квадратных скобках. Для описания массивов используется служебное слово array. Описание переменной данного типа имеет следующий вид:

```
<имя_переменной>: array[i..i1,j..j1,…] of <тип_элементов>,
```

где i, i1 – границы первого индекса массива, j, j1 – границы второго индекса массива.

Например:

```
var
a: array[1..10] of integer;
```

Можно сначала определить тип данных массива, а затем описывать переменные этого типа, как и в случае со скалярными типами.

Строки – последовательность символов. При использовании в выражениях строка заключается в апострофы. Ее длина ограничена 255 символами. Для описания переменных строкового типа используется служебное слово string, например:

```
<имя_переменной>: string[n],
```

где n – длина строковой переменной; если n не указана, то длина строки равна 255 символам.

Записи и файлы будут рассмотрены ниже.

2.3. Операции и выражения в языке Паскаль

Выражение задает порядок выполнения действий над данными и состоит из операндов (констант, переменных, обращений к функциям), круглых скобок и знаков операций: `a+b*sin(cos(x))`. Операции делятся на *унарные* (например, `-c`) и *бинарные* (например, `a+b`), а также на ниженазванные группы.

Арифметические операции, применяемые в Турбо Паскале, приведены в табл. 2.3.

Таблица 2.3 ▼ Арифметические операции

Операция	Действие	Тип операндов	Тип результата
+	Сложение	Целый, вещественный	Целый, вещественный
-	Вычитание	Целый, вещественный	Целый, вещественный
*	Умножение	Целый, вещественный	Целый, вещественный
/	Деление	Целый, вещественный	Целый, вещественный
Div	Деление нацело	Целый	Целый
Mod	Остаток от деления	Целый	Целый
And	И	Целый	Целый
Shl	Сдвиг влево	Целый	Целый
Shr	Сдвиг вправо	Целый	Целый
Or	ИЛИ	Целый	Целый
Xor	Исключающее ИЛИ	Целый	Целый
-	Отрицание	Целый	Целый
Not	Логическое отрицание	Целый	Целый

В Турбо Паскале кроме широко известных арифметических операций присутствуют операции целочисленной арифметики и битовой арифметики.

2.3.1. Операции целочисленной арифметики

К операциям целочисленной арифметики относятся:

- целочисленное деление `div`;
- остаток от деления `mod`.

При целочисленном делении операция `div` возвращает целую часть частного (дробная часть отбрасывается), а операция `mod` – остаток от деления. Ниже приведены примеры этих операций:

11 mod 4 = 3;
11 div 4 = 2;
7 mod 3 = 1;
7 div 3 = 2;

26 div 5 = 5;
26 mod 5 = 1.

2.3.2. Операции битовой арифметики

Во всех операциях битовой арифметики действия происходят над двоичным представлением целых чисел.

К операциям битовой арифметики относятся следующие операции Паскаля.

Арифметическое И (and). Оба операнда переводятся в двоичную систему, затем над ними происходит логическое поразрядное умножение операндов по следующим правилам:

1 and 1 = 1;
1 and 0 = 0;
0 and 1 =0;
0 and 0 = 0.

Рассмотрим пример использования арифметического И:

```
Var A, B: integer;
Begin
  A:=13;
  B:=23;
  Writeln(A and B)
End.
```

Этот участок программы работает следующим образом. Число A=13 и B=23 переводятся в двоичное представление 0000000000001101 и 0000000000010111[1]. Затем над ними поразрядно выполняется логическое умножение:

0000000000001101
and
0000000000010111
0000000000000101

Результат переводится в десятичную систему счисления, в нашем случае получится число 5. Таким образом, 13 and 23 = 5.

Арифметическое ИЛИ (or). Здесь также оба операнда переводятся в двоичную систему, после чего над ними происходит логическое поразрядное сложение операндов по следующим правилам:

1 or 1 = 1;
1 or 0 = 1;
0 or 1 =1;
0 or 0 = 0.

[1] Переменные A и B занимают в памяти 2 байта – 16 двоичных разрядов.

Пример:

```
Var A, B: integer;
Begin
  A:=13;
  B:=23;
  Writeln(A or B)
End.
```

Над двоичным представлением значений A и B выполняется логическое сложение:

0000000000001101
or
0000000000010111
0000000000011111

После перевода результата в десятичную систему имеем 13 or 23 =31.

Арифметическое исключающее ИЛИ (xor). Здесь также оба операнда переводятся в двоичную систему, после чего над ними происходит логическая поразрядная операция xor по следующим правилам:

1 xor 1 = 0;
1 xor 0 = 1;
0 xor 1 =1;
0 xor 0 = 0.

Арифметическое отрицание (not). Эта операция выполняется над одним операндом[1]. Применение операции not вызывает побитную инверсию двоичного представления числа. Например, рассмотрим операцию not 13.

0000000000001101
not a 1111111111110010

После перевода результата в десятичную систему получаем not 13 = –14.

Сдвиг влево (*M* shl *L*). Двоичное представление числа *M* сдвигается влево на *L* позиций. Рассмотрим пример использования операции shl: 17 shl 3. Представляем число 17 в двоичной системе: 10001. Сдвигаем его на три позиции влево: 10001000. В десятичной системе это число 136: 17 shl 3 =136. Заметим, что сдвиг на один разряд влево соответствует умножению на 2, на два разряда – умножению на 4, на три – умножению на 8. Таким образом, операция *M* shl *L* эквивалентна $M \cdot 2^L$.

[1] Операции, выполняемые над одним операндом, принято называть унарными, над двумя – бинарными.

Сдвиг вправо (*M* shr *L*). В этом случае двоичное представление числа *M* сдвигается вправо на *L* позиций, что эквивалентно целочисленному делению числа *M* на 2^L. Например, 25 shr 1 =12, 25 shr 3= 3.

Операции отношения выполняют сравнение двух операндов и определяют, истинно выражение или ложно. Их результат – логический. Определены следующие операции отношения: <, >, =, <= ,>= ,<>.

Пример операций отношения:

```
3.14<>2, 6>4.
```

Операции отношения определены и над символьными переменными и строками:

```
'a'<'b';
'abc'<'abd'.
```

Логические операции выполняются над логическими данными. Определены следующие логические операции (табл. 2.4).

Таблица 2.4 ▾ Логические операции

A	*B*	Not *A*	*A* and *B*	*A* or *B*
t	t	f	t	t
t	f	f	f	t
f	t	t	f	t
f	f	t	f	f

Примечание к табл. t – true (истина), f – false (ложь).

В логических выражениях могут использоваться операции отношения, логические и арифметические.

Пример логического выражения:

```
(a+x)>(c+d*cos(y))or(a>b)
```

В сложных выражениях порядок, в котором выполняются операции, соответствует приоритету операций. В Паскале приняты следующие приоритеты:

1. Унарные операции.
2. *, /, div, mod, and, shl, shr.
3. +, -, or, xor.
4. =, <>, >, <, >=, <=.

Использование скобок в выражениях позволяет менять порядок вычислений.

2.4. Стандартные функции в языке Паскаль

В Турбо Паскале определены *стандартные функции* над арифметическими операндами (табл. 2.5).

Таблица 2.5 ▼ Некоторые стандартные функции

Обозначение	Тип аргументов	Тип результата	Действие
`Abs(X)`	Целый, вещественный	Целый, вещественный	Модуль числа
`Sin(X)`	Вещественный	Вещественный	Функция синус
`Cos(X)`	Вещественный	Вещественный	Функция косинус
`Arctan(X)`	Вещественный	Вещественный	Арктангенс
`Pi`		Вещественный	π
`Exp(X)`	Вещественный	Вещественный	e^x
`Ln(X)`	Вещественный	Вещественный	Функция натурального логарифма
`Sqr(X)`	Вещественный	Вещественный	x^2
`Sqrt(X)`	Вещественный	Вещественный	$\sqrt{x}$
`Int(X)`	Вещественный	Вещественный	Целая часть числа
`Frac(X)`	Вещественный	Вещественный	Дробная часть числа
`Round(X)`	Вещественный	Целый	Округление числа *X*
`Trunc(X)`	Вещественный	Целый	Отсекание дробной части числа *X*
`Random`		Вещественный	Случайное число от 0 до 1
`Random(n)`	Целый	Целый	Случайное число от 0 до *n*

Остальные часто встречающиеся функции (тангенс, арксинус и т.д.) моделируются из уже определенных с помощью известных математических соотношений:

$$\mathrm{tg}(x) = \frac{\sin(x)}{\cos(x)},$$

$$\arcsin(x) = \mathrm{arctg}\left(\frac{x}{\sqrt{1-x^2}}\right),$$

$$\arccos(x) = \frac{\pi}{2} - \mathrm{arctg}\left(\frac{x}{\sqrt{1-x^2}}\right).$$

Определенную проблему представляет возведение *X* в степень *n*. Если значение степени *n* – целое положительное число, то можно *n* раз перемножить *X*

(что дает более точный результат и при целом n предпочтительнее) или воспользоваться формулой[1]:

$$\begin{cases} X^n = e^{n\ln(X)}, X > 0 \\ X^n = -e^{n\ln|X|}, X < 0 \end{cases},$$

которая программируется с помощью стандартных функций на языке Паскаль:

- `exp(n*ln(x))` – для положительного X;
- `-exp(n*ln(abs(x)))` – для отрицательного X.

Эту же формулу можно использовать для возведения X в дробную степень n, где n – обыкновенная правильная дробь вида k/l, а знаменатель l нечетный. Если знаменатель l четный, это означает извлечение корня четной степени, следовательно, есть ограничения на выполнение операции.

При возведении числа X в отрицательную степень следует помнить, что

$$X^{-n} = \frac{1}{X^n}.$$

Таким образом, для программирования выражения, содержащего возведение в степень, надо внимательно проанализировать значения, которые могут принимать X и n, так как в некоторых случаях возведение X в степень n невыполнимо.

Строковые функции будут рассмотрены при изучении переменных строкового типа в главе 8. С другими функциями можно ознакомиться в справочной системе Турбо Паскаль.

2.5. Структура программы на языке Турбо Паскаль

Как уже известно читателю из главы 1, программа, написанная на языке Турбо Паскаль, имеет следующую структуру:

- заголовок программы;
- раздел описаний;
- тело программы.

[1] Формула $e^{n\ln(x)}$ получается следующим образом: логарифмируем выражение x^n; получается $n\ln(x)$; затем экспоненцируем последнее.

Заголовок программы состоит из служебного слова program, имени программы, образованного по правилам использования идентификаторов Паскаля, и точки с запятой. Например:

```
program my_prog001;
```

Раздел описаний включает следующие подразделы:

- раздел описания констант;
- раздел описания типов;
- раздел описания переменных;
- раздел описания процедур и функций.

В языке Турбо Паскаль должны быть описаны все переменные, типы, константы, которые будут использоваться программой. В стандартном Паскале порядок следования разделов в программе жестко установлен, в Турбо Паскале такого строгого требования нет. В программе может быть несколько разделов описания констант, переменных и т.д. Более подробно *структуру программы* на языке Паскаль можно представить следующим образом:

```
program имя_программы;
const описания_констант;
type описания_типов;
var описания_переменных;
begin
операторы_языка;
end.
```

Тело программы начинается со слова begin, затем следуют операторы языка Паскаль, реализующие алгоритм решаемой задачи. *Операторы* в языке Паскаль отделяются друг от друга точкой с запятой и могут располагаться в одну строчку или начинаться с новой строки (в этом случае их также необходимо разделять точкой с запятой). Назначение символа ; – отделение операторов друг от друга. Тело программы заканчивается служебным словом end. Несмотря на то что операторы могут располагаться в строке как угодно, рекомендуется размещать их по одному в строке, а в случае сложных операторов отводить для каждого несколько строк. Рассмотрим более подробно структуру программы:

```
program имя_программы;
const описания_констант;
type описания_типов;
var описания_переменных;
```

```
begin
  оператор_1;
  оператор_2;
  ...
  оператор_n
end.
```

В текст программы на Паскале могут быть включены комментарии в фигурных скобках ({ это комментарий }) или в круглых скобках в сопровождении символа * (* это тоже комментарий *). Комментарии игнорируются в процессе выполнения программы и служат для пояснения отдельных ее частей. Приведем пример текста программы на Паскале:

```
program one;
const
  a=7;
var
  b,c: real;
begin
  c:=a+2; b:=c-a*sin(a)
end.
```

2.6. Простейшие операторы языка Паскаль

В языке Паскаль есть *простые* и *структурированные* операторы. Рассмотрим сначала первые.

2.6.1. Оператор присваивания

В операторе присваивания слева всегда стоит имя переменной, а справа – значение, например:

```
a:=b;
```

где a – имя переменной или элемента массива, b – значение как таковое, выражение, переменная, константа или функция.

Типы переменных a и b должны совпадать или быть совместимыми для присваивания, то есть тип, к которому принадлежит переменная b, должен находиться в границах типа переменной a.

В результате выполнения оператора a:=b переменной a присваивается значение b, например:

```
var
  a,b,c,d:real;
begin
c:=pi/2;
d:=sin(pi*c)*cos(c)*ln(c);
a:=(c+d)/(c-d)*exp(-c);
d:=sqrt(c)*exp(1/9*ln(c));
end.
```

Обращаем внимание читателя на то, что оператор присваивания неэквивалентен математическому знаку равенства. Запись `b:=b+2` *является корректным оператором Паскаля и означает: к значению переменной* `b` *добавить 2 и результат записать в переменную* `b`. *Фактически речь идет об увеличении значения* `b` *на 2. В то же время, запись* `b+2:=b` *не является оператором – это бессмыслица.*

2.6.2. Операторы ввода-вывода

Ввод информации с клавиатуры осуществляется с помощью оператора `read`. Он может иметь один из следующих форматов:

```
read (x1,x2,…,xn);
```

или

```
readln (x1,x2,…,xn);
```

где `x1,x2,…,xn` – список вводимых переменных.

Когда в программе встречается оператор `read`, ее действие приостанавливается до тех пор, пока не будут введены исходные данные. При вводе числовых значений два числа считаются разделенными, если между ними есть хотя бы один пробел, символ табуляции или конца строки (**Enter**). После ввода последнего значения следует нажать **Enter**.

Так чем же отличаются `read` и `readln`? Оператор `readln` аналогичен оператору `read`. Разница заключается в том, что после считывания последнего в списке значения для одного оператора `readln` данные для следующего оператора `readln` будут считываться с начала новой строки. Но следует помнить, что **Enter** переведет курсор на новую строку независимо от того, как именно происходит считывание данных. При вводе отдельных[1] числовых значений большой разницы между `read` и `readln` нет. При вводе строковых переменных

[1] Существенная разница между `read` и `readln` появляется при вводе матриц, о чем будет рассказано в главе 5.

лучше использовать второй из них. Строковые значения вводятся подряд или отделяются нажатием клавиши **Enter**. Более подробно об этом будет рассказано в главе 8.

Пример:

```
program three;
var
  a,b,c:real;
begin
read(a,b);
c:=a+b
end.
```

Для *вывода информации* (чисел, строк и булевых значений) на экран дисплея служат операторы `write` и `writeln`. В общем случае они имеют вид:

```
write(x1,x2,…,xn); writeln(x1,x2,…,xn);
```

где `x1,x2,…,xn` представляют собой список выводимых переменных, констант, выражений (`x1,x2,…,xn` не могут быть перечислимого типа).

Операторы `write` и `writeln` последовательно выводят все переменные на экран дисплея. Если используется `writeln`, то после вывода информации курсор перемещается в новую строку. Вещественные данные выводятся в формате с плавающей точкой[1]. Ширина поля вывода в этом случае составляет 18 символов: ±#.##########E±##, где # – любая десятичная цифра от 0 до 9, например:

0.344300000000E-01 0.03443

–5.44317180000E+02 –544.31718

Рассмотрим фрагмент программы на Паскале:

```
r:=17.42;
c:=-0.0001342;
k:=12;
writeln(r);
writeln(c);
writeln(k);
```

В результате выполнения этих действий на экране появятся следующие числа:

```
1.7420000000E+01
-1.3420000000E-04
12
```

[1] Чтобы выводить числа в формате с фиксированной точкой, необходимо использовать форматированный вывод.

Попробуйте самостоятельно разобраться, чем отличается вывод в следующих трех программах:

```
var a,b,c:real;
begin
a:=174.256;
b:=-13.6671512;
c:=24316.1196673;
write(a);
write(b);
write(c);
end.
```

```
var a,b,c:real;
begin
a:=174.256;
b:=-13.6671512;
c:=24316.1196673;
write(a);
writeln(b);
write(c);
end.
```

```
var a,b,c:real;
begin
a:=174.256; b:=-13.6671512; c:=24316.1196673;
writeln(a); writeln(b); writeln(c);
end.
```

После знакомства с операторами `read` и `write` и оператором присваивания не составит труда написать программу решения следующей задачи.

Пример 2.1. Заданы длины трех сторон треугольника *a*, *b*, *c*. Вычислить периметр и площадь треугольника. Значения *a*, *b*, *c* ввести с клавиатуры.

Решение задачи можно разделить на несколько этапов:

1. Ввод значений *a*, *b* и *c*.
2. Вычисление полупериметра по формуле $p=\frac{a+b+c}{2}$.
3. Вычисление площади прямоугольника по формуле Герона $s=\sqrt{p(p-a)(p-b)(p-c)}$.
4. Вывод площади треугольника *s* и периметра $2p$.

Текст программы с комментариями приведен ниже.

```
program four;
{ Описаны переменные вещественного типа: a, b, c - стороны треугольника,
s - площадь, p - полупериметр. }
var
  a,b,c,s,p:real;
begin
{ Ввод исходных данных. }
write('a='); readln(a);
write('b='); readln(b);
write('c='); readln(c);
```

```
{ Вычисление полупериметра треугольника. }
p:=(a+b+c)/2;
{ Вычисление площади треугольника. }
s:=sqrt(p*(p-a)*(p-b)*(p-c));
{ Вывод периметра и площади в экспоненциальной форме. }
writeln('периметр треугольника', 2*p);
writeln('площадь треугольника', s);
end.
```

Форматированный вывод информации

В операторах `write` (`writeln`) имеется возможность указать константу (или выражение) целочисленного типа, определяющую ширину поля вывода. Для целых и строковых выражений она указывается через двоеточие после имени выводимой переменной или выражения, например:

```
var
  a: string[15];
  b,c: integer;
  d: word;
begin
readln(c,b);
readln(a); readln(d);
writeln(a:18);
writeln(c:6,b:5,d:7)
end.
```

При выводе вещественных значений кроме ширины поля вывода через двоеточие надо указывать количество позиций, необходимых для дробной части числа. Форматированный вывод вещественных чисел позволяет увидеть значения на экране в формате с фиксированной точкой.

Пример 2.2. Заданы радиус основания и высота цилиндра. Вычислить площадь основания и объем. Площадь основания вычисляется по формуле, объем цилиндра равен $V = \pi r^2 h = Sh$.

Решение задачи можно разделить на несколько этапов:

1. Ввод значений r и h.
2. Вычисление площади основания $S = \pi r^2$.
3. Вычисление объема цилиндра $V = Sh$.
4. Вывод площади основания и объема цилиндра.

Текст программы с комментариями приведен ниже:

```
{ Описание переменных. }
var
  s,v,r,h:real;
```

```
begin
{ Ввод исходных данных. }
  writeln('Введите R и H');
  readln(r,h);
{ Вычисление площади основания. }
  s:=pi*sqr(r);
{ Вычисление объема цилиндра. }
  v:=s*h;
{ Вывод результатов в формате с фиксированной точкой. }
  writeln(' v=', v:6:2);
  writeln(' s=', s:8:3);
end.
```

Рассмотрим одну особенность форматированного вывода вещественного числа. Пусть есть оператор `write(a:m:n)`, где `m` – ширина поля вывода, `n` – количество знаков в дробной части числа. Если число `a` не помещается в `m` позиций, то поле вывода расширяется до минимально необходимого. В связи с этим допустимыми являются следующие форматы: a:2:2, a:1:2, a:0:2 и даже a:–1:2. В этом случае вы фактически указываете только количество разрядов в дробной части числа, а ширина поля вывода определяется в момент вывода вещественного числа на экран.

2.6.3. Составной оператор

Составной оператор – группа операторов, отделенных друг от друга точкой с запятой, начинающихся со служебного слова `begin` и заканчивающихся служебным словом `end`.

```
begin
  оператор_1;
  ...
  оператор_n
end;
```

Транслятор воспринимает составной оператор как единый.

Глава 3 Операторы управления

В этой главе описаны основные операторы языка: условный оператор if, оператор выбора case, операторы while...do, repeat...until, for...do. Изложена методика составления алгоритмов с помощью блок-схем. Приводится большое количество примеров составления программ различной сложности. Отметим, что данная глава является ключевой в изучении программирования.

3.1. Использование условного оператора

Решение большинства задач редко сводится к простому последовательному расчету по формулам. Чаще порядок вычислений зависит от определенных условий, например от исходных данных или от промежуточных результатов, полученных на предыдущих шагах программы. Для организации вычислений в зависимости от какого-либо условия в Турбо Паскале используется условный оператор if, который в общем виде записывается следующим образом:

```
if условие then оператор_1 else оператор_2;
```

В качестве условия должно использоваться логическое значение, представленное константой, переменной или выражением, например:

```
a:=2; b:=8;
if a>b tnen
  writeln('a больше b')
else writeln('a меньше b');
```

Если условие истинно, то выполняется оператор (оператор_1), следующий за словом then. Но если условие ложно, то будет выполняться оператор (оператор_2), следующий за словом else.

И по ветке then, *и по ветке* else *должен выполняться единственный оператор. Если по смыслу задачи необходимо выполнить несколько операторов, тогда следует использовать составной оператор. Поэтому в общем случае оператор* if *имеет следующую структуру*[1]*:*

```
if условие then
begin
 оператор_1;
 оператор_2;
 ...
 оператор_n
end
else
begin
 оператор_1;
 оператор_2;
 ...
 оператор_n
end
```

Здесь оператор_1, оператор_2, оператор_n – любые операторы языка Паскаль.

Альтернативная ветвь else может отсутствовать, если в ней нет необходимости. В таком «усеченном» условном операторе в случае невыполнения условия ничего не происходит, и выполняется оператор, следующий за условным. Например:

```
a:=2; b:=8;c:=0;
if a>b tnen
begin
  writeln('a>b');
  c:=a+b
end;
c:=c+12;
writeln('c=',c:2);
```

Условные операторы могут быть *вложены* друг в друга:

```
if условие then
  if подусловие then
```

[1] Обратите внимание на использование точки с запятой в операторе if.

```
  begin
  ..........
  end
  else
  begin
  .........
  end
else
begin
..........
end;
```

При вложениях условных операторов всегда действует правило: альтернатива `else` считается принадлежащей ближайшему условному оператору `if`, имеющему ветвь `else`. Следовательно, есть риск допустить ошибку, например:

```
If условие_1 then
if условие_2 then
оператор_А
else
оператор_Б;
```

По записи похоже, что `оператор_Б` будет выполняться только при невыполнении `условия_1`. Но в действительности он будет отнесен к `условию_2`. Причем точка с запятой после `оператора_А` только ухудшит положение. Выход таков: нужно представить вложенное условие как составной оператор:

```
if условие_1 then
  begin
     if условие_2 then
        оператор_А
     end
  else
  оператор_Б;
```

В этом случае для ветви `else` ближайшим незакрытым оператором `if` окажется оператор с `условием_1`.

Рассмотрим использование оператора `if` на примере следующей задачи.

Пример 3.1. Написать программу решения квадратного уравнения $ax^2 + bx + c = 0$. Можно выделить следующие этапы решения:

1. Ввести числовые значения переменных a, b и c.
2. Вычислить значения дискриминанта d по формуле $d = b^2 - 4ac$.
3. Если $d < 0$, то будет выведено сообщение «Корней нет», поэтому необходимо перейти к выполнению пункта 4. Иначе произвести вычисление корней

$$X_1 = \frac{-b+\sqrt{d}}{2a},\ X_2 = \frac{-b-\sqrt{d}}{2a}$$

и вывод их на экран.

4. Прекратить вычисления.

На третьем этапе решения задачи воспользуемся оператором `if`.

Исходные данные: вещественные числа `a`, `b` и `c` – коэффициенты квадратного уравнения.

Результаты работы программы: вещественные числа `x1`, `x2` (корни квадратного уравнения) либо сообщение о том, что корней нет.

Для вычисления дискриминанта квадратного уравнения введем вещественную переменную `d`.

Текст программы с комментариями приведен ниже.

```
Program kvadrat;
{ Описание всех переменных. }
var
  a,b,c,d,x1,x2: real;
begin
{ Ввод значения коэффициентов квадратного уравнения. }
  writeln('Введите коэффициенты квадратного уравнения');
  readln(a,b,c);
{ Вычисление дискриминанта. }
  d:=b*b-4*a*c;
{ Если дискриминант отрицателен, }
  if d<0 then
{ то вывод сообщения, что корней нет. }
     writeln('Корней нет')
  else
{ По ветке else использован составной оператор, потому, что при
положительном дискриминанте надо выполнить три оператора, }
  begin
{ иначе вычисление корней x1, x2 }
     x1:=(-b+sqrt(d))/2/a;
     x2:=(-b-sqrt(d))/(2*a);
{ и вывод их на экран. }
     writeln('X1=',x1:6:3, ' X2=',x2:6:3)
  end
end.
```

При разработке простейших программ несложно перейти от словесного описания к написанию программы на Паскале. Однако большинство реально разрабатываемых программ довольно сложные, поэтому трудно сразу написать их на каком-либо языке программирования. Словесное описание подобных задач может занять не одну страницу. Каждый программист будет по-своему описывать решение одной и той же задачи, да еще на своем «родном» языке. После словесного описания не всегда просто создать программу на Паскале. Поэтому на этапе разработки программы вводится еще один промежуточный этап – построение *блок-схемы алгоритма*[1]. Блок-схема поможет перейти от описания решения задачи на обычном языке к написанию программы на алгоритмическом языке (в нашем случае на Паскале). Все этапы решения задачи достаточно стандартны: ввод исходных данных, расчет по формулам, проверка условий, вывод результатов. Поэтому при составлении блок-схемы решения задачи все эти этапы изображаются при помощи различных геометрических фигур, называемых блоками, а связи между этапами (последовательность выполнения этапов) указываются при помощи стрелок, соединяющих эти фигуры. Блоки сопровождаются надписями.

Типичные этапы решения задачи изображаются следующими геометрическими фигурами:

- *блок начала-конца* (рис. 3.1). Надпись внутри блока: «начало» («конец»);
- *блок ввода-вывода данных* (рис. 3.2). Надпись внутри блока: «ввод (вывод или печать)»; далее следует список вводимых (выводимых) переменных;
- *блок решения*, или *арифметический* (рис. 3.3). Внутри блока записывается действие, вычислительная операция или группа;
- *условный блок* (рис. 3.4). Условие записывается внутри блока. В результате проверки условия осуществляется выбор одного из возможных путей (ветвей) вычислительного процесса. Если условие истинно, то следующим выполняется этап, соответствующий ветви +, если условие ложно, то выполняется этап, соответствующий ветви –.

[1] Алгоритм (от algorithmi, algorismus – первоначально латинская транслитерация имени математика аль-Хорезми) – способ решения задач, точно предписывающий, как и в какой последовательности получить результат, однозначно определяемый исходными данными [3].

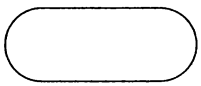


Рис. 3.1 ▼ Блок начала-конца алгоритма

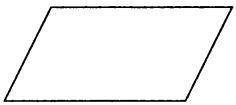


Рис. 3.2 ▼ Блок ввода-вывода данных

Рис. 3.3 ▼ Арифметический блок

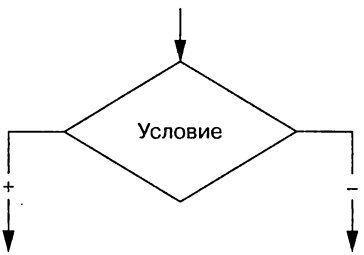


Рис. 3.4 ▼ Условный блок

Использование блок-схем рассмотрим на примере следующей задачи.

Пример 3.2. Составить программу нахождения действительных и комплексных корней квадратного уравнения. Можно выделить следующие этапы решения задачи:

1. Ввод коэффициентов квадратного уравнения a, b и c.
2. Вычисление дискриминанта d по формуле $d = b^2 - 4ac$.
3. Проверка знака дискриминанта. Если $d \geq 0$, то вычисление действительных корней

$$X_1 = \frac{-b+\sqrt{d}}{2a}, \quad X_2 = \frac{-b-\sqrt{d}}{2a}$$

и вывод их на экран. При отрицательном дискриминанте выводится сообщение о том, что действительных корней нет, и вычисляются комплексные корни[1]

$$X_1 = \frac{-b}{2a} + i\frac{\sqrt{|d|}}{2a}, \quad X_2 = \frac{-b}{2a} - i\frac{\sqrt{|d|}}{2a}.$$

У обоих комплексных корней действительные части одинаковые, а мнимые отличаются знаком. Поэтому можно в переменной x1 хранить действительную

[1] Комплексные числа записываются в виде $a + ib$, где a – действительная часть комплексного числа, b – мнимая часть комплексного числа, i – мнимая единица ($i = \sqrt{-1}$).

часть числа $-\frac{b}{2a}$, в переменной x2 – модуль мнимой части $\frac{\sqrt{|d|}}{2a}$, а в качестве корней вывести x1+ix2 и x1-ix2.

На рис. 3.5 изображена блок-схема решения задачи. Блок 1 предназначен для ввода коэффициентов квадратного уравнения. В блоке 2 осуществляется вычисление дискриминанта. Блок 3 осуществляет проверку знака дискриминанта, если дискриминант отрицателен, то корни комплексные, их расчет происходит в блоке 4 (действительная часть корня записывается в переменную x1, модуль мнимой – в переменную x2), а вывод – в блоке 5 (первый корень x1+ix2, второй – x1-ix2). Если дискриминант положителен, то вычисляются действительные корни уравнения (блок 6) и выводятся на экран (блок 7).

Текст программы приведен ниже.

```
Program kvadrat;
{ Описание всех переменных. }
var
a,b,c,d,x1,x2: real;
begin
{ Ввод значения коэффициентов квадратного уравнения. }
writeln('Введите коэффициенты квадратного уравнения');
readln(a,b,c);
{ Вычисление дискриминанта. }
d:=b*b-4*a*c;
{ Проверка знака дискриминанта. }
if d<0 then
{ По ветке then (как, впрочем, и по ветке else) в операторе if
используется составной оператор, потому что при любом знаке дискриминанта
надо выполнить несколько операторов. }
begin
{ Если дискриминант отрицателен, то вывод сообщения, что действительных
корней нет, и вычисление комплексных корней. }
  writeln('Действительных корней нет');
  { Вычисление действительной части комплексных корней. }
  x1:=-b/(2*a);
  { Вычисление модуля мнимой части комплексных корней. }
  x2:=sqrt(abs(d))/(2*a);
  { Сообщение о комплексных корнях уравнения вида ax^2+bx+c=0. }
  writeln('Комплексные корни уравнения ',
  a:1:2,'x^2+',b:1:2,'x+',c:1:2,'=0');
  { Вывод значений комплексных корней в виде x1±ix2. }
  writeln(x1:1:2,'+i*',x2:1:2);
  writeln(x1:1:2,'-i*',x2:1:2);
end
```

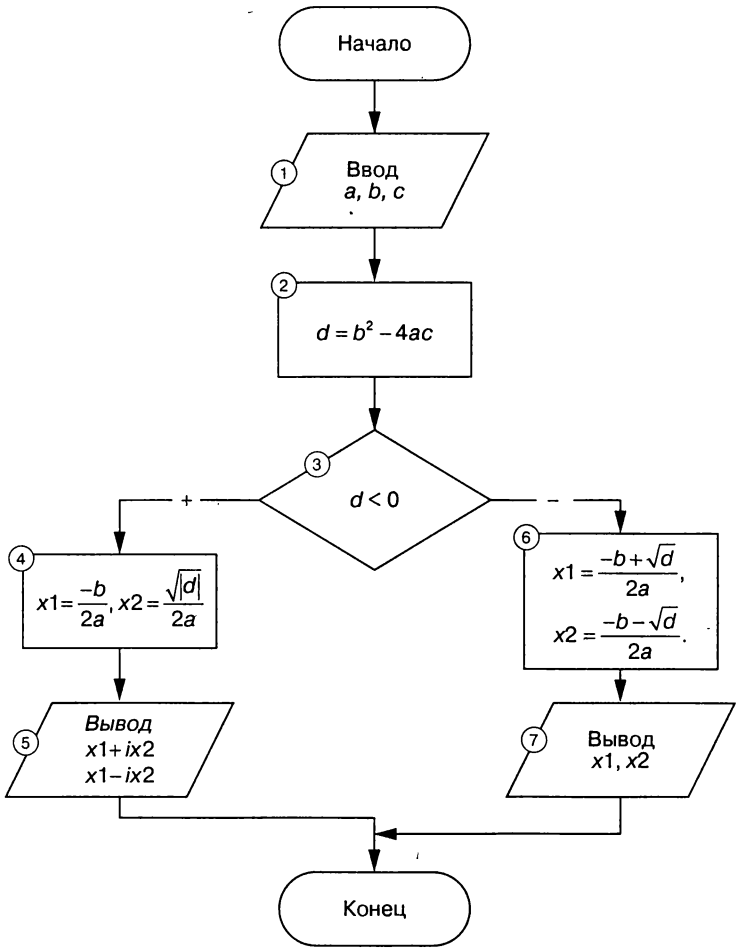


Рис. 3.5 ▼ Блок-схема решения квадратного уравнения

```
else
begin
{ иначе вычисление действительных корней x1, x2 }
  x1:=(-b+sqrt(d))/2/a;
  x2:=(-b-sqrt(d))/(2*a);
{ и вывод их на экран. }
  writeln('Действительные корни уравнения ',
   a:1:2,'x^2+',b:1:2,'x+',c:1:2,'=0');
```

```
  writeln('X1=',x1:1:2, ' X2=',x2:1:2)
end
end.
```

Рассмотрим еще две задачи с использованием оператора `if`.

Пример 3.3. Заданы коэффициенты *a*, *b* и *c* биквадратного уравнения $ax^4 + bx^2 + c = 0$. Найти все его действительные корни. Для решения биквадратного уравнения необходимо заменой $y = x^2$ привести его к квадратному и решить это уравнение.

Входные данные задачи решения биквадратного уравнения: *a*, *b*, *c*.

Выходные данные задачи решения биквадратного уравнения: x_1, x_2, x_3, x_4.

Блок-схема решения задачи представлена на рис. 3.6.

Можно выделить следующие этапы решения:

1. Ввод коэффициентов биквадратного уравнения *a*, *b* и *c* (блок 1).
2. Вычисление дискриминанта уравнения *d* (блок 2).
3. Если $d < 0$ (блок 3), вывод сообщения, что корней нет (блок 4), а иначе определяются корни соответствующего квадратного уравнения y_1 и y_2 (блок 5).
4. Если y_1 и $y_2 < 0$ (блок 6) , то вывод сообщения, что корней нет (блок 7).
5. Если y_1 и $y_2 \geq 0$ (блок 8), то вычисляются четыре корня по формулам $\pm\sqrt{y_1}$, $\pm\sqrt{y_2}$ (блок 9) и выводятся значения корней (блок 10).
6. Если условия 4 и 5 не выполняются, то необходимо проверить знак y_1. Если y_1 i 0 (блок 11), то вычисляются два корня по формуле $\pm\sqrt{y_1}$ (блок 12), иначе (если y_2 i 0) вычисляются два корня по формуле $\pm\sqrt{y_2}$ (блок 13). Вывод вычисленных значений корней (блок 14).

Текст программы решения биквадратного уравнения приведен ниже.

```
Program bikvadrat;
{ Описание всех переменных:
a,b,c - коэффициенты биквадратного уравнения,
d - дискриминант,
x1,x2,x3,x4 - корни биквадратного уравнения,
y1,y2 - корни квадратного уравнения ay^2+by+c=0,
Все используемые переменные имеют вещественный тип (real). }
Var
a,b,c,d,x1,x2,x3,x4,y1,y2: real;
begin
{ Ввод коэффициентов уравнения. }
```

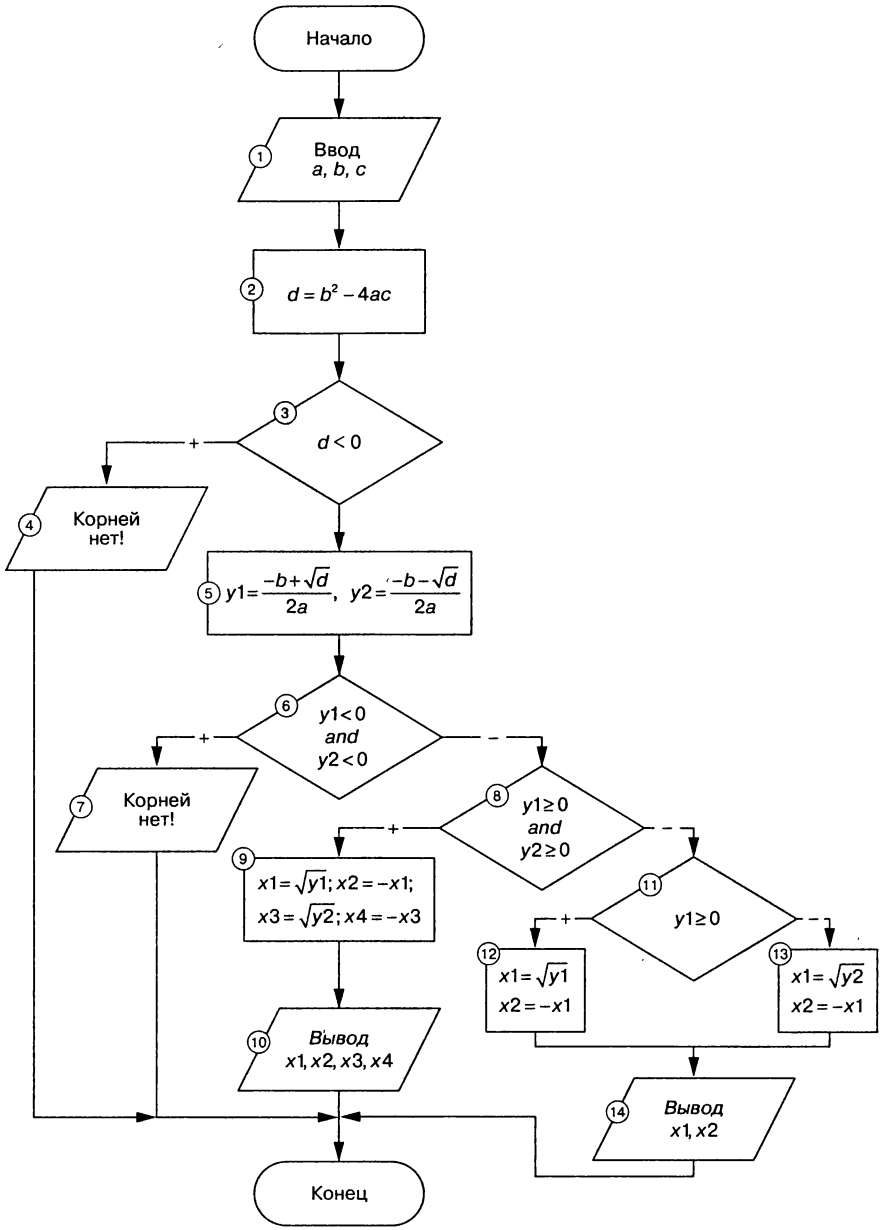


Рис. 3.6 ▼ Блок-схема решения биквадратного уравнения

```
writeln('Введите коэффициенты биквадратного уравнения');
readln(a,b,c);
{ Вычисление дискриминанта. }
d:=b*b-4*a*c;
{ Если он отрицателен, }
if d<0 then
{ вывод сообщения "Корней нет". }
writeln('Корней нет')
{ Если дискриминант ≥ 0, }
else
begin
  { вычисление корней соответствующего квадратного уравнения. }
  y1:=(-b+sqrt(d))/2/a;
  y2:=(-b-sqrt(d))/(2*a);
  { Если оба корня квадратного уравнения < 0, }
  if (y1<0) and (y2<0) then
  { вывод сообщения "Корней нет". }
     writeln('Корней нет')
  { Если оба корня квадратного уравнения ≥ 0, }
  else if (y1>=0) and (y2>=0) then
  begin
     { вычисление четырех корней биквадратного уравнения. }
     x1:=sqrt(y1);
     x2:=-x1;
     x3:=sqrt(y2);
     x4:=-sqrt(y2);
     { Вывод корней биквадратного уравнения на экран. }
     writeln('X1=',x1:6:3,' X2=',x2:6:3);
     writeln('X3=',x3:6:3,' X4=',x4:6:3);
     end
     { Если не выполнились оба условия
     1. y1<0 И y2<0
     2. y1>=0 И y2>=0,
     то проверяем условие y1>=0. }
     else if (y1>=0) then
     { Если оно истинно, }
     Begin
     { для вычисления корней биквадратного уравнения извлекаем корни из y1. }
        x1:=sqrt(y1);
        x2:=-x1;
        writeln('X1=',x1:6:3,'   X2=',x2:6:3);
     end
     else
     { Если условие y1>=0 ложно, то }
     Begin
        { для вычисления корней биквадратного уравнения извлекаем корни из y2. }
        x1:=sqrt(y2);
```

```
        x2:=-x1;
        writeln('X1=',x1:6:3,'   X2=',x2:6:3);
    end
end
end.
```

Читателю предлагается самостоятельно модифицировать программу таким образом, чтобы она находила все корни (как действительные, так и комплексные) биквадратного уравнения.

Пример 3.4. Решить кубическое уравнение $ax^3 + bx^2 + cx + d = 0$.

Кубическое уравнение имеет вид

$$ax^3 + bx^2 + cx + d = 0. \quad (3.1)$$

После деления на a уравнение (3.1) принимает канонический вид:

$$x^3 + rx^2 + sx + t = 0, \quad (3.2)$$

где $r = \frac{b}{a}$, $s = \frac{c}{a}$, $t = \frac{d}{a}$.

В уравнении (3.2) сделаем замену $x = y - \frac{r}{3}$ и получим приведенное уравнение (3.3) [5]:

$$y^3 + py + q = 0, \quad (3.3)$$

где $p = \frac{3s - r^2}{3}$, $q = \frac{2r^3}{27} - \frac{rs}{3} + t$.

Число действительных корней приведенного уравнения (3.3) зависит от знака дискриминанта $D = \left(\frac{p}{3}\right)^3 + \left(\frac{q}{2}\right)^2$ [5] (табл. 3.1).

Таблица 3.1 ▼ Количество корней кубического уравнения

Дискриминант	Количество действительных корней	Количество комплексных корней
$D \geq 0$	1	2
$D < 0$	3	–

Корни приведенного уравнения могут быть рассчитаны по формулам Кардано (3.4) [5].

$$
\begin{aligned}
y_1 &= u + v,\\
y_2 &= -\frac{u+v}{2} + \frac{u-v}{2} i\sqrt{3},\\
y_3 &= -\frac{u+v}{2} - \frac{u-v}{2} i\sqrt{3},
\end{aligned}
\qquad (3.4)
$$

где $u = \sqrt[3]{-\frac{q}{2} + \sqrt{D}}$, $v = \sqrt[3]{-\frac{q}{2} - \sqrt{D}}$.

При отрицательном дискриминанте уравнение (3.1) имеет три действительных корня, но они будут вычисляться через вспомогательные комплексные величины. Чтобы избавиться от этого, можно воспользоваться формулами (3.5) [5]:

$$
\begin{aligned}
y_1 &= 2\sqrt[3]{\rho}\cos\left(\frac{\varphi}{3}\right),\\
y_2 &= 2\sqrt[3]{\rho}\cos\left(\frac{\varphi}{3} + \frac{2\pi}{3}\right),\\
y_3 &= 2\sqrt[3]{\rho}\cos\left(\frac{\varphi}{3} + \frac{4\pi}{3}\right),
\end{aligned}
\qquad (3.5)
$$

где $\rho = \sqrt{\frac{-p^3}{27}}$, $\cos(\varphi) = -\frac{q}{2\rho}$.

Таким образом, при положительном дискриминанте кубического уравнения (3.3) расчет корней будем вести по формулам (3.4), а при отрицательном – по формулам (3.5). После расчета корней приведенного уравнения (3.3) по формулам (3.4) или (3.5) необходимо по формулам $x_k = y_k - \frac{r}{3}$, $k = 1, 2, 3$ перейти к корням заданного кубического уравнения (3.1).

Блок-схема решения кубического уравнения представлена на рис. 3.7.

Описание блок-схемы. В блоке 1 вводятся коэффициенты кубического уравнения, в блоках 2–3 рассчитываются коэффициенты канонического и приведенного уравнений. Блок 4 предназначен для вычисления дискриминанта. В блоке 5 проверяется знак дискриминанта кубического уравнения. Если он

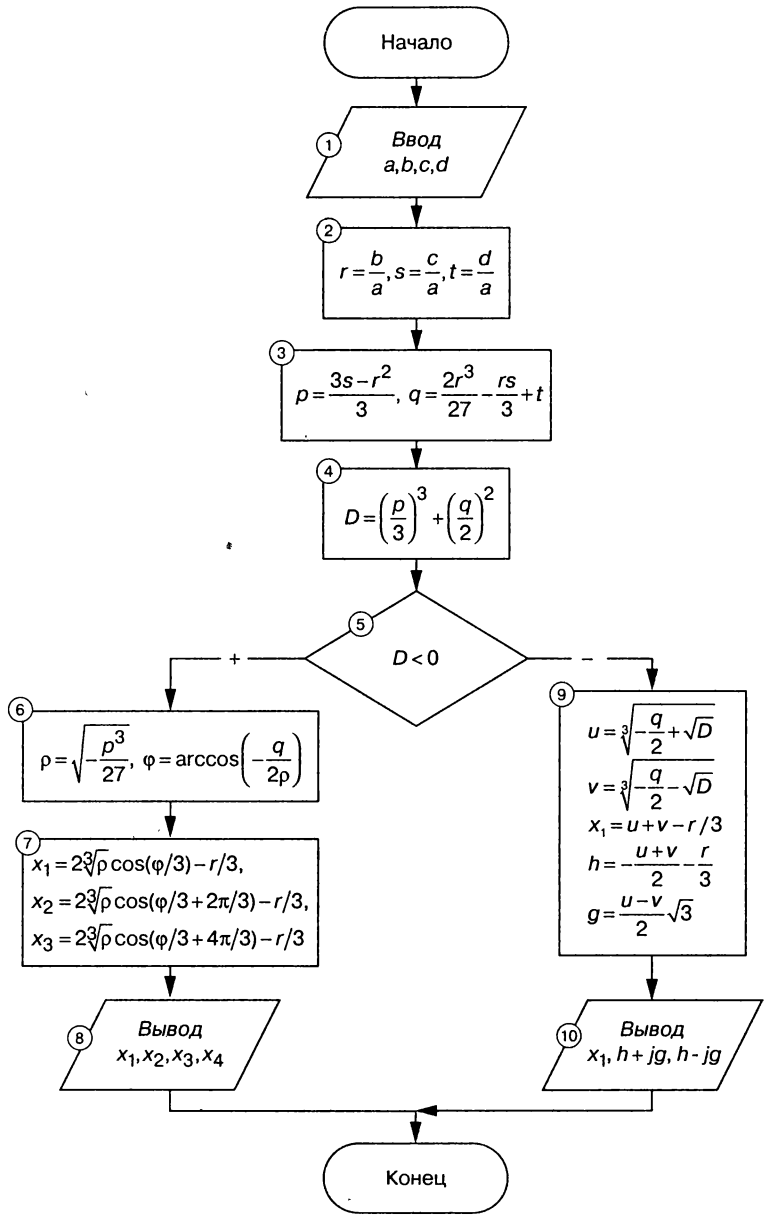


Рис. 3.7 ▼ Блок-схема задачи решения кубического уравнения

отрицателен, то корни вычисляются по формулам (3.5) – блоки 6–7. При положительном значении дискриминанта расчет идет по формулам (3.4) – блок 9. Блоки 8 и 10 предназначены для вывода результатов на экран.

Программа решения кубического уравнения на Турбо Паскале с комментариями приведена ниже. Обращаем внимание, что при расчете величин $u=\sqrt[3]{-\frac{q}{2}+\sqrt{D}}$, $v=\sqrt[3]{-\frac{q}{2}-\sqrt{D}}$ в программе предусмотрена проверка знака $-\frac{q}{2}\pm\sqrt{D}$. Если величина $-\frac{q}{2}\pm\sqrt{D}$ положительна, то расчет величин u и v идет по формуле $e^{\frac{1}{3}\ln\left(-\frac{q}{2}\pm\sqrt{D}\right)}$. При отрицательной величине $-\frac{q}{2}\pm\sqrt{D}$ u и v рассчитываются по формуле $-e^{\frac{1}{3}\ln\left|-\frac{q}{2}\pm\sqrt{D}\right|}$.

```
{ Описание переменных; все они вещественные. }
var
  a,b,c,d,r,s,t,p,q,ro,fi,x1,x2,x3,u,v,h,g:real;
begin
{ Ввод коэффициентов кубического уравнения. }
  write('a=');readln(a);
  write('b=');readln(b);
  write('c=');readln(c);
  write('d=');readln(d);
{ Расчет коэффициентов канонического уравнения (см. формулу (3.2)). }
  r:=b/a; s:=c/a; t:=d/a;
{ Вычисление коэффициентов приведенного уравнения (см. формулу (3.3)). }
  p:=(3*s-r*r)/3;
  q:=2*r*r*r/27-r*s/3+t;
{ Вычисление дискриминанта кубического уравнения. }
  d:=(p/3)*sqr(p/3)+sqr(q/2);
{ Проверка знака дискриминанта, ветка then реализует формулы (3.5), ветка
else - формулы (3.4). }
  if d<0 then
  begin
     ro:=sqrt(-p*p*p/27);
{ Следующие два оператора реализуют расчет угла fi, сначала вычисляется
величина косинуса угла, затем вычисляется его арккосинус через арктангенс. }
     fi:=-q/(2*ro);
     fi:=pi/2-arctan(fi/sqrt(1-fi*fi));
{ Вычисление действительных корней уравнения x1, x2 и x3. }
     x1:=2*exp(1/3*ln(ro))*cos(fi/3)-r/3;
```

```
    x2:=2*exp(1/3*ln(ro))*cos(fi/3+2*pi/3)-r/3;
    x3:=2*exp(1/3*ln(ro))*cos(fi/3+4*pi/3)-r/3;
    writeln('x1=',x1:1:3,' x2=',x2:1:3,' x3=',x3:1:3);
  end
  else
  begin
{ Вычисление u и v с проверкой знака подкоренного выражения. }
    if -q/2+sqrt(d)>0 then
      u:=exp(1/3*ln(-q/2+sqrt(d)))
    else
    if -q/2+sqrt(d)<0 then
      u:=-exp(1/3*ln(abs(-   q/2+sqrt(d))))
    else u:=0;
    if -q/2-sqrt(d)>0 then
      v:=exp(1/3*ln(-q/2-sqrt(d)))
    else
    if -q/2-sqrt(d)<0 then
      v:=-exp(1/3*ln(abs(-q/2-sqrt(d))))
    else v:=0;
{ Вычисление действительного корня кубического уравнения. }
    x1:=u+v-r/3;
{ Вычисление действительной мнимой части комплексных корней. }
    h:=-(u+v)/2-r/3;
    g:=(u-v)/2*sqrt(3);
    writeln('x1=',x1:1:3,' x2=',h:1:3,'+i*',g:1:3,
    ' x3=',h:1:3,'-i*',g:1:3);
  end
end.
```

3.2. Использование оператора варианта

Оператор варианта `case` необходим в тех случаях, когда в зависимости от значений какой-либо переменной надо выполнить те или иные операторы.

```
case управляющая_переменная of
  набор_значений_1: оператор_1;
  набор_значений_2: оператор_2;
  …
  набор_значений_N: оператор_N
else
  альтернативный_оператор
end;
```

Оператор работает следующим образом. Если `управляющая_переменная` принимает значение из `набора_значений_1`, то выполняется `оператор_1`. Если `управляющая_переменная` принимает значение из `набора_значений_2`, то

выполняется оператор_2. Если управляющая_переменная принимает значение из набора_значений_N, то выполняется оператор_N. Если управляющая переменная не принимает ни одно значение из имеющихся наборов, то выполняется альтернативный_оператор.

Тип управляющей_переменной, которая стоит между служебными словами case и of, должен быть только перечислимым (включая char и boolean), диапазоном или целочисленным. Набор_значений – это конкретное значение управляющей переменной или выражение, при котором необходимо выполнить соответствующий оператор, игнорируя остальные варианты. Значения в каждом наборе должны быть уникальными, то есть они могут появляться только в одном варианте. Пересечение наборов значений для разных вариантов является ошибкой. Ключевое слово else может отсутствовать.

Рассмотрим применение оператора варианта на следующих примерах.

Пример 3.5. Вывести на печать название дня недели, соответствующее заданному числу D, при условии, что в месяце 31 день и 1-е число – понедельник.

Для решения задачи воспользуемся операцией mod, позволяющей вычислить остаток от деления двух чисел, и условием, что 1-е число – понедельник. Если в результате остаток от деления заданного числа D на семь будет равен единице, то это понедельник, двойке – вторник, тройке – среда и т.д. Следовательно, при построении алгоритма необходимо использовать семь условных операторов, как показано рис. 3.8.

Решение задачи упростится, если при написании программы воспользоваться оператором варианта:

```
var d:byte;
begin
  write('Введите число D=');
  readln(D);
  case D mod 7 of { Вычисляется остаток от деления D на 7. }
{ В зависимости от полученного значения }
{ на печать выводится название дня недели. }
     1:writeln('ПОНЕДЕЛЬНИК');
     2:writeln('ВТОРНИК');
     3:writeln('СРЕДА');
     4:writeln('ЧЕТВЕРГ');
     5:writeln('ПЯТНИЦА');
     6:writeln('СУББОТА');
     0:writeln('ВОСКРЕСЕНЬЕ');
  end;
end.
```

В предложенной записи оператора варианта отсутствует ветвь else. Это объясняется тем, что выражение D mod 7 может принимать только одно из

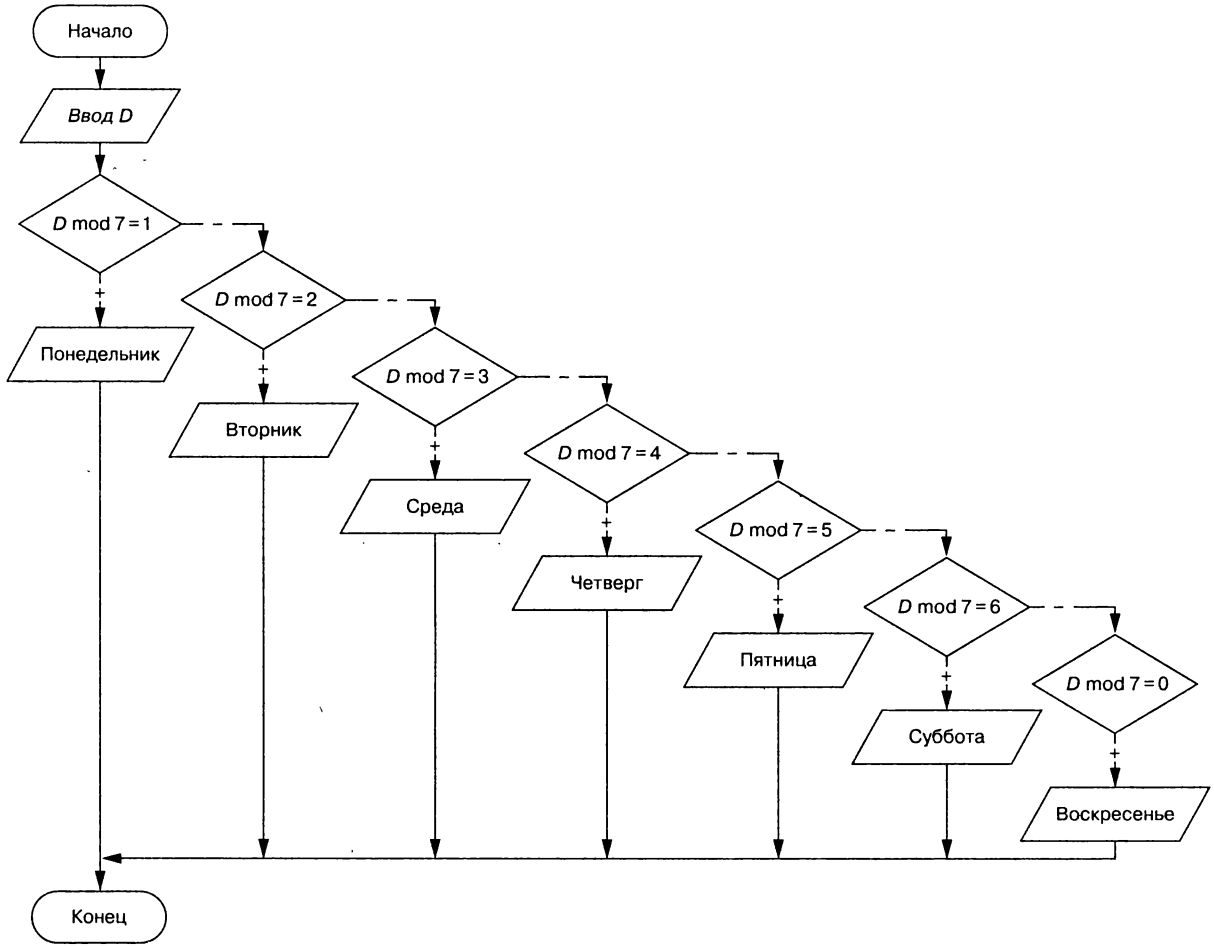


Рис. 3.8 ▾ Блок-схема определения дня недели по заданному числу

указанных значений, то есть 1, 2, 3, 4, 5, 6 или 0. Если же предположить, что ошибочно было введено число D, превышающее 31, и значение управляющего выражения выходит из области возможных значений, то оператор case можно переписать следующим образом:

```
case D mod 7 of
  1:writeln('ПОНЕДЕЛЬНИК');
  2:writeln('ВТОРНИК');
  3:writeln('СРЕДА');
  4:writeln('ЧЕТВЕРГ');
  5:writeln('ПЯТНИЦА');
  6:writeln('СУББОТА');
  0:writeln('ВОСКРЕСЕНЬЕ')
{ Если результат управляющего выражения превышает 6, }
{ выдается сообщение о ошибке. }
else writeln('ОШИБКА ПРИ ВВОДЕ!!!');
end;
```

Пример 3.6. По заданному номеру месяца m вывести на печать название времени года.

Для решения данной задачи необходимо проверить выполнение четырех условий. Если заданное число m равно 12, 1 или 2, то это зима, если m попадает в диапазон от 3 до 5, то – весна. Лето определяется принадлежностью числа m диапазону от 6 до 8, и соответственно, при равенстве переменной m 9, 10 или 11 – это осень. Понятно, что область возможных значений переменной m находится в диапазоне от 1 до 12.

```
Var m:byte;
begin
write('Введите номер месяца m=');
readln(m);
{ Проверка области допустимых значений переменной m. }
if (m>=1) and (m<=12) then
  case m of
  { В зависимости от значения m на печать }
  { выводится название времени года. }
  12,1,2:  writeln('ЗИМА');
  3..5: writeln('ВЕСНА');
  6..8: writeln('ЛЕТО');
  9..11:   writeln('ОСЕНЬ');
  end
{ Если значение переменной m выходит за пределы области
 допустимых значений, то выдается сообщение об ошибке. }
else writeln('ОШИБКА ПРИ ВВОДЕ!!!');
end.
```

3.3. Использование операторов цикла

Циклом в программировании называют повторение одних и тех же действий (шагов). Последовательность действий, которые повторяются в цикле, называют *телом цикла.*

Существует несколько типов *алгоритмов циклической структуры.*

На рис. 3.9 изображен цикл с предусловием, а на рис. 3.10 – цикл с постусловием, которые называют *условными циклическими алгоритмами* [6]. Нетрудно заметить, что эти циклы взаимозаменяемы и обладают некоторыми отличиями:

- в цикле с предусловием условие проверяется до тела цикла, в цикле с постусловием – после тела цикла;
- в цикле с постусловием тело цикла выполняется хотя бы один раз, в цикле с предусловием тело цикла может не выполниться ни разу;
- в цикле с предусловием проверяется условие продолжения цикла, в цикле с постусловием – условие выхода из цикла.

При написании условных циклических алгоритмов следует помнить следующее. Во-первых, чтобы цикл имел шанс когда-нибудь закончиться, содержимое

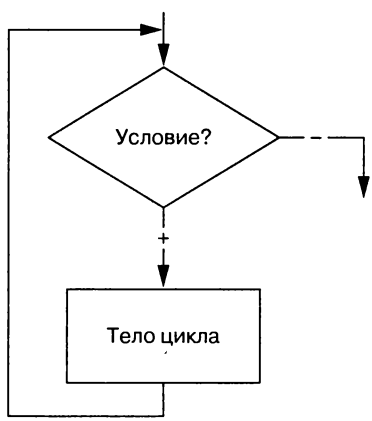


Рис. 3.9 ▼ Алгоритм циклической структуры с предусловием

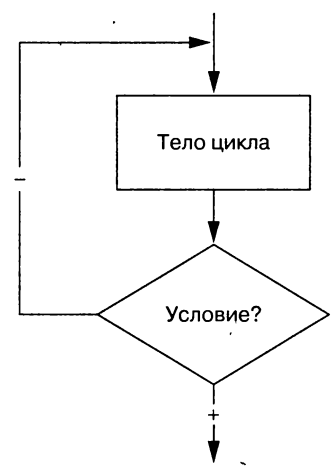


Рис. 3.10 ▼ Алгоритм циклической структуры с постусловием

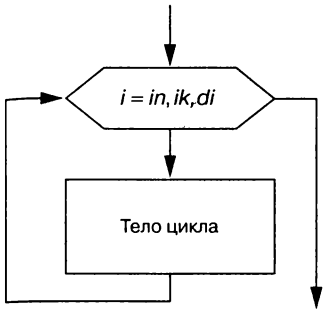


Рис. 3.11 ▾ Алгоритм циклической структуры без условия

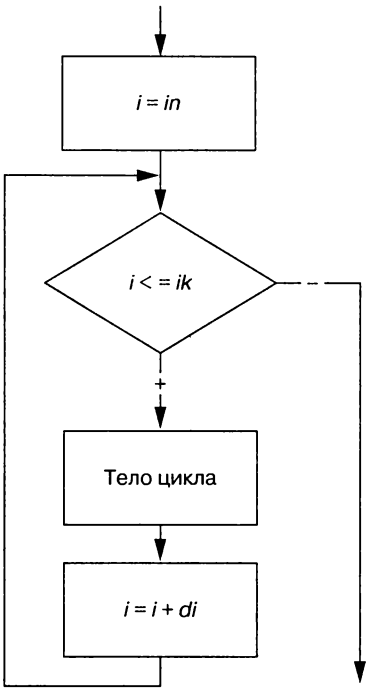


Рис. 3.12 ▾ Условный циклический алгоритм с известным числом повторений

его тела должно обязательно влиять на условие цикла. Во-вторых, условие должно состоять из корректных выражений и значений, определенных еще до первого выполнения тела цикла.

Кроме того, существует так называемый безусловный циклический алгоритм (рис. 3.11) [6], который удобно использовать, если известно, сколько раз необходимо выполнить тело цикла.

Выполнение безусловного циклического алгоритма начинается с присвоения переменной *i* стартового значения *in*. Затем следует проверка, не превосходит ли переменная *i* конечное значение *ik*. Если это так, то цикл считается завершенным и управление передается следующему за телом цикла оператору. В противном случае выполняется тело цикла, и переменная *i* меняет свое значение в соответствии с указанным шагом *di*. Далее снова производится проверка значения переменной *i*, и алгоритм повторяется. Понятно, что безусловный циклический алгоритм можно заменить любым условным. Например, так, как показано на рис. 3.12.

Отметим, что переменную *i* называют *параметром цикла*, так как это переменная, которая изменяется внутри цикла по определенному закону и влияет на его окончание.

3.3.1. Оператор цикла while ... do

Обращение к оператору while ... do переводится как «пока ... делать» и выглядит так:

```
while условие do оператор;
```

Оператор, стоящий после служебного слова do и называемый телом цикла, будет выполняться циклически, пока логическое условие истинно. Само условие может быть логической константой, переменной или логическим выражением.

Если тело цикла состоит более чем из одного оператора, необходимо использовать операторные скобки:

```
while условие do
begin
  оператор 1;
  оператор 2;
  ...
  оператор n;
end;
```

Условие выполнения тела цикла while проверяется до начала каждой итерации. Поэтому, если условие сразу не выполняется, то тело цикла игнорируется и управление передается первому оператору, стоящему за телом цикла.

Оператор while ... do предназначен для реализации циклов с предусловием.

3.3.2. Оператор цикла с постусловием repeat ... until

В цикле с предусловием предварительной проверкой определяется, выполнять тело цикла или нет, до первой итерации. Если это не соответствует логике алгоритма, то можно использовать цикл с постусловием, то есть цикл, в котором проверяется, делать или нет очередную итерацию, лишь после завершения предыдущей. Это имеет принципиальное значение только на первом шаге, а далее циклы ведут себя идентично. Оператор repeat ... until реализует цикл с постусловием. Цикл с постусловием всегда будет выполнен хотя бы один раз.

```
repeat
  оператор_1;
  оператор_2;
  ............
  оператор_N;
until<условие>;
```

В цикле while подразумевается такой алгоритм: пока условие истинно, выполнять операторы тела цикла. В цикле repeat действует другой алгоритм: выполнять тело цикла, пока не станет истинным условие, то есть пока условие ложно, выполняется цикл.

3.3.3. Оператор цикла for ... do

Операторы цикла с условием обладают значительной гибкостью, но не слишком удобны для организации «строгих» циклов, которые должны быть выполнены заданное число раз. Оператор цикла for ... do используется именно в таких случаях.

```
for параметр_цикла:=начальное_значение to конечное_значение do
    <оператор>;
for параметр_цикла:=конечное_значение downto начальное_значение do
    <оператор>;
```

Оператор, представляющий собой тело цикла, может быть простым или составным. Параметр цикла, а также диапазон его изменения могут быть только целочисленного или перечислимого типа. Параметр описывается совместно с другими переменными. Шаг цикла for всегда постоянный и равен интервалу между двумя ближайшими значениями типа параметра цикла. Например:

```
var  { Описание параметров цикла. }
i: integer; c:char; b: boolean;
begin   { Вывод на печать целых чисел от 1 до 10. }
  for i:=1 to 10 do writeln(i);{ Шаг цикла равен 1. }
{ Вывод на печать чисел от 10 до -10. }
  for i:=10 downto -10 do{ Шаг цикла равен -1. }
  writeln(i);
{ Вывод на печать латинских символов от a до r. }
{ Параметр цикла изменяется от a до r в алфавитном порядке. }
  for c:='a' to 'r' do writeln(c);

end.
```

Выполнение цикла начинается с присвоения параметру стартового значения. Затем следует проверка, не превосходит ли параметр конечное значение. Если результат проверки утвердительный, то цикл считается завершенным и управление передается следующему за телом цикла оператору. В противном случае выполняется тело цикла и параметр меняет свое значение на следующее согласно заголовку цикла. Далее снова производится проверка значения параметра цикла, и алгоритм повторяется.

3.3.4. Операторы break, continue, exit, halt

Операторы `break` и `continue` введены в язык Турбо Паскаль версии 7.0.

Оператор `break` осуществляет немедленный выход из циклов `repeat`, `while`, `for`. Его можно использовать только внутри циклов.

Оператор `continue` начинает новую итерацию цикла, даже если предыдущая не была завершена. Его можно использовать только внутри цикла.

Оператор `exit` осуществляет выход из подпрограммы.

Оператор `halt` прекращает выполнение программы и возвращает код завершения в операционную систему.

Структура оператора: `halt[(e)];` где `e` – переменная типа `word` (код завершения). Если `e` отсутствует, то код завершения – 0.

Рассмотрим использование операторов `exit` и `halt` на примере вычисления факториала.

3.3.5. Решение задач с использованием циклов

Рассмотрим использование циклических операторов на конкретных примерах.

Пример 3.7. Найти наибольший общий делитель (НОД) двух натуральных чисел *A* и *B*.

Входные данные: `A` и `B`.

Выходные данные: `A` – НОД.

Для решения поставленной задачи воспользуемся алгоритмом Евклида [5]: будем уменьшать каждый раз большее из чисел на величину меньшего до тех пор, пока оба значения не станут равными, так, как показано в табл. 3.2.

Таблица 3.2 ▼ Поиск НОД для чисел *A* = 25 и *B* = 15

Исходные данные	Первый шаг	Второй шаг	Третий шаг	НОД (A, B) = 5
A = 25	*A* = 10	*A* = 10	*A* = 5	
B = 15	*B* = 15	*B* = 5	*B* = 5	

В блок-схеме решения задачи, представленной на рис. 3.13, используется цикл с предусловием, то есть тело цикла повторяется до тех пор, пока `A` не равно `B`. Следовательно, при создании программы воспользуемся циклом `while ... do`:

```
var a,b:word;
begin
  writeln('введите два натуральных числа');
  write('A='); readln(A);
```

```
  write('B='); readln(B);
{ Если числа не равны, выполнять тело цикла. }
  while a<>b do
{ Если число A больше, чем B, то уменьшить его значение на B, }
  if a>b then
     a:=a-b
{ иначе уменьшить значение числа B на A. }
  else
     b:=b-a;
  writeln('НОД=',A);
end.
```

Результат работы программы не изменится, если для ее решения воспользоваться циклом с постусловием `repeat ... until`:

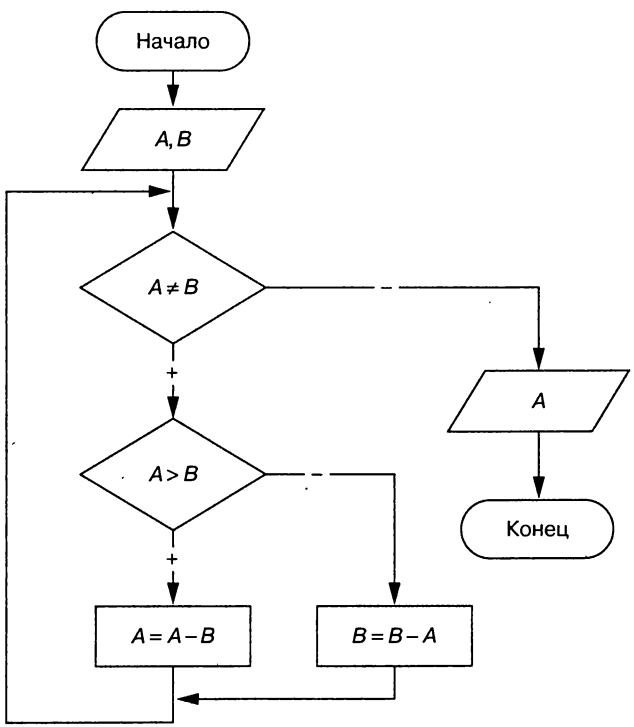


Рис. 3.13 ▾ Поиск наибольшего общего делителя двух чисел

```
var a,b:word;
begin
  writeln('введите два натуральных числа');
  write('A='); readln(a);
  write('B='); readln(b);
  repeat
     if a>b then
       a:=a-b
     else
        b:=b-a;
  until a=b;{ Закончить работу цикла, при условии, что A равно B. }
  writeln('НОД=',a);
end.
```

Пример 3.8. Вводится последовательность чисел, 0 – конец последовательности. Определить, содержит ли последовательность хотя бы два равных соседних числа [6].

Входные данные: X0 – текущий член последовательности, X1 – следующий член последовательности.

Выходные данные: сообщение о наличии в последовательности двух равных соседних элементов.

Вспомогательные переменные: Fl – логическая переменная; сохраняет значение «истина», если в последовательности есть равные рядом стоящие члены, и «ложь» – иначе.

Блок-схема решения задачи приведена на рис. 3.14. Применение здесь цикла с постусловием обосновано тем, что необходимо вначале сравнить два элемента последовательности, а затем принять решение об окончании цикла.

При составлении программы воспользуемся циклом repeat ... until:

```
var X0,X1:real;
Fl:boolean;
Begin
{ Предположим, что в последовательности нет равных }
{ рядом стоящих элементов. }
Fl:=false;
write('X='); { Ввод текущего элемента последовательности. }
readln(X0);
write('X='); { Ввод следующего элемента последовательности. }
readln(X1);
repeat
{ Если в последовательности есть равные рядом стоящие элементы, }
{ то логической переменной присвоить значение «истина». }
if X0=X1 then Fl:=true;
X0:=X1;    { Следующий элемент становится текущим. }
write('X='); { Ввод следующего элемента. }
```

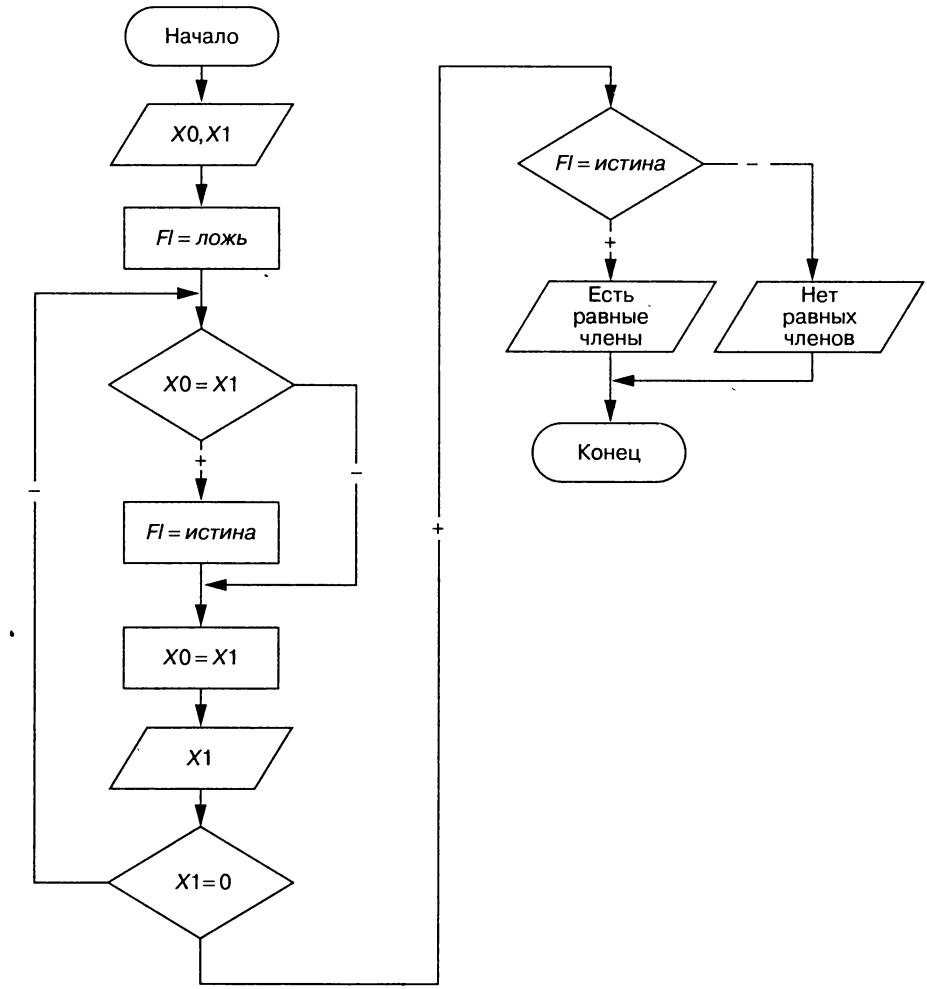


Рис. 3.14 ▾ Поиск равных соседних элементов последовательности

```
readln(X1);
{ Если следующий элемент равен нулю, то выполнение цикла прекратить. }
until X1=0;
if Fl then
writeln('В последовательности есть равные соседние члены.')
else
writeln('В последовательности нет равных соседних членов.');
end.
```

В приведенных примерах условие задачи таково, что неизвестно, сколько раз повторится тело цикла. Такие циклы называют циклами с неизвестным числом повторений (на самом деле неизвестно, закончится ли цикл вообще). Цикл, количество повторений которого известно заранее или его можно определить по исходным данным, называют циклом с известным числом повторений. Рассмотрим несколько примеров.

Пример 3.9. Составить таблицу значений функции $y = e^{\sin(x)}\cos(x)$ на отрезке $[0;\pi]$ с шагом 0,1.

Входные данные: начальное значение аргумента – 0, конечное значение аргумента – π, шаг изменения аргумента – 0,1.

Выходные данные: множество значений аргумента X и соответствующее им множество значений функции Y.

В условии задачи количество повторений цикла явно не задано, но известно, как изменяется параметр цикла X и каковы его начальное и конечное значения. Поэтому решим эту задачу, используя цикл с предусловием (рис. 3.15).

Текст программы, соответствующий составленной блок-схеме, будет иметь вид:

```
var
x,y:real;
begin
{ Присвоение начального значения параметру цикла. }
x:=0;
{ Перед каждым выполнением тела цикла проверяется условие x ≤ π, }
{ если оно истинно, то тело цикла выполняется, иначе }
{ управление передается следующему за циклом оператору. }
while x<=pi do
begin
  { Расчет значения функции для соответствующего значения аргумента X. }
  y:=exp(sin(x))*cos(x);
  { Печать значений функции и аргумента. }
  writeln(x:4:2,' ',y:8:5);
  { Вычисление нового значения аргумента. }
  x:=x+0.1;
end; { Конец цикла while. }
end.
```

Теперь решим задачу другим способом, используя циклический оператор for … do, предварительно определив количество повторений тела цикла n.

```
var x,y:real;
n,i:integer;
begin
  x:=0;       { Присвоение начального значения аргументу. }
```

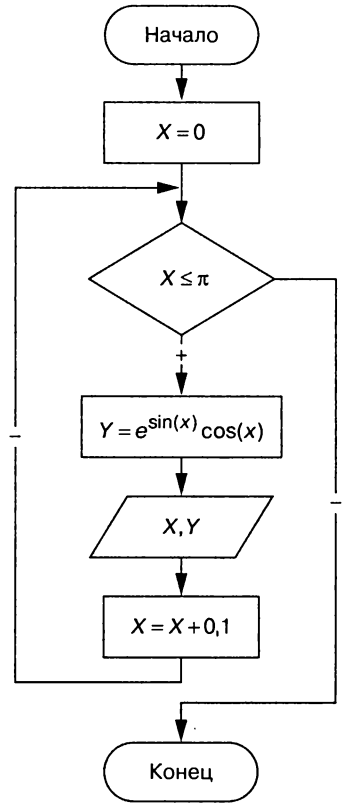


Рис. 3.15 ▾ Создание таблицы значений функции $y = e^{\sin(x)}\cos(x)$

```
  n:=round((pi-0)/0.1)+1; { Расчет количества повторений цикла. }
{ Функция round округляет результат деления до целого. }
  for i:=1 to n do     { Тело цикла выполнится ровно n раз. }
  begin
{ Расчет значения функции для соответствующего значения аргумента X. }
     y:=exp(sin(x))*cos(x); { Печать значений функции и аргумента. }
     writeln(x:4:2,' ',y:8:4);
{ Вычисление нового значения аргумента. }
     x:=x+0.1;
  end;           { Конец цикла for ... do. }
end.
```

Итак, если параметр цикла *x* принимает значения в диапазоне от *xn* до *xk*, изменяясь с шагом *dx*, то количество повторений тела цикла можно определить по формуле:

$$n = \frac{xk - xn}{dx} + 1, \tag{3.6}$$

округлив результат деления до целого числа.

Пример 3.10. Вычислить факториал числа N ($N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$).

Входные данные: `N` – целое число, факториал которого необходимо вычислить.

Выходные данные: `factorial` – значение факториала числа `N`, произведение чисел от 1 до `N`, целое число.

Промежуточные данные: `i` – целочисленная переменная, принимающая значения от 2 до `N` с шагом 1, параметр цикла.

Блок-схема приведена на рис. 3.16.

Итак, вводится число `N`. Переменной `factorial`, предназначенной для хранения значения произведения последовательности чисел, присваивается начальное значение, равное единице. Затем организуется цикл, параметром которого выступает переменная `i`. Если значение параметра цикла меньше или равно `N`, то выполняется оператор тела цикла, в котором из участка памяти с именем `factorial` считывается предыдущее значение произведения, умножается на текущее значение параметра цикла, а результат снова помещается в участок памяти с именем `factorial`. Когда параметр `i` становится больше `N`, цикл заканчивается и на печать выводится значение переменой `factorial`, которая была вычислена в теле цикла.

Теперь рассмотрим текст программы вычисления факториала на языке Турбо Паскаль.

```
var
  factorial, n, i:integer;
begin
write('n='); readln(n);
factorial:=1; { Стартовое значение. }
for i:=2 to n do
  factorial:=factorial*i;
writeln(factorial);
end.
```

Пример 3.11. Вычислить a^n ($n > 0$).

Входные данные: `a` – вещественное число, которое необходимо возвести в целую положительную степень `n`.

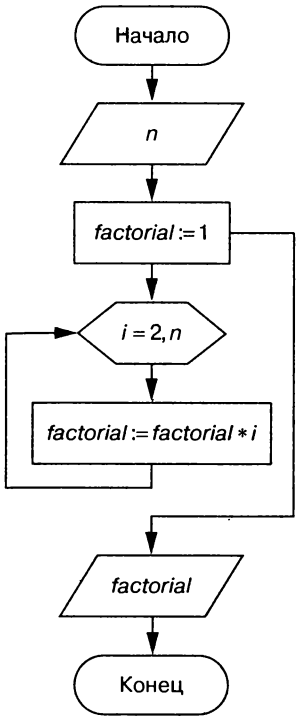


Рис. 3.16 ▾
Вычисление факториала

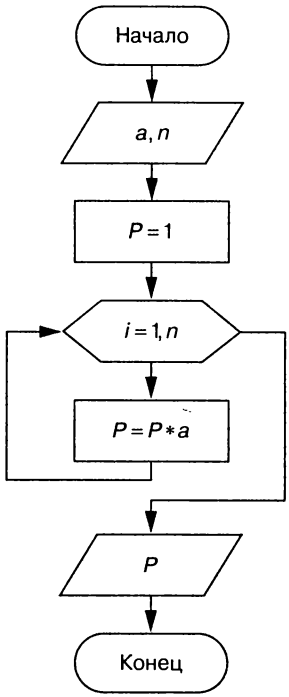


Рис. 3.17 ▾ Возведение вещественного числа в целую степень

Выходные данные: p (вещественное число) – результат возведения вещественного числа a в целую положительную степень n.

Промежуточные данные: i – целочисленная переменная, принимающая значения от 1 до n с шагом 1, параметр цикла.

Блок-схема приведена на рис. 3.17.

Известно, что для того чтобы получить целую степень n числа a, нужно умножить его само на себя n раз. Результат будет храниться в участке памяти с именем p. При выполнении очередного цикла из этого участка предыдущее значение будет считываться, умножаться на основание степени a и снова записываться в участок памяти p. Цикл выполняется n раз.

В табл. 3.3 отображен протокол выполнения алгоритма при возведении числа 2 в пятую степень: a = 2, n = 5. Подобные таблицы, заполненные вручную, используются для тестирования – проверки всех этапов работы программы.

Таблица 3.3 ▼ Процесс возведения числа a в степень n

i		1	2	3	4	5
P	1	2	4	8	16	32

Далее приведен текст программы, написанной для решения поставленной задачи.

```
var
  a,p:real;
  i,n:word;
begin
  write('Введите основание степени a=');
  readln(a);
  write('Введите показатель степени n=');
  readln(n);
  p:=1;
  for i:=1 to n do
     p:=p*a;
  writeln('P=',P:1:3);
end.
```

Пример 3.12. Вычислить сумму натуральных четных чисел, не превышающих *N*.

Входные данные: N – целое число.

Выходные данные: S – сумма четных чисел.

Промежуточные данные: i – переменная, принимающая значения от 2 до N с шагом 2, следовательно, также имеет целочисленное значение.

При сложении нескольких чисел необходимо накапливать результат в определенном участке памяти, каждый раз считывая оттуда предыдущее значение суммы и прибавляя к нему следующее слагаемое. Для выполнения первого оператора накапливания суммы из участка памяти необходимо взять такое число, которое не влияло бы на результат сложения. Иными словами, перед началом цикла переменной, предназначенной для накапливания сумы, необходимо присвоить значение нуль. Блок-схема решения этой задачи представлена на рис. 3.18.

Так как параметр цикла i изменяется с шагом 2, в блок-схеме, построенной для решения данной задачи (см. рис. 3.18), использован цикл с предусловием, который реализуется при составлении программы с помощью оператора while ... do:

```
var n,i,S:word;
begin
  write('n=');
  readln(n);
```

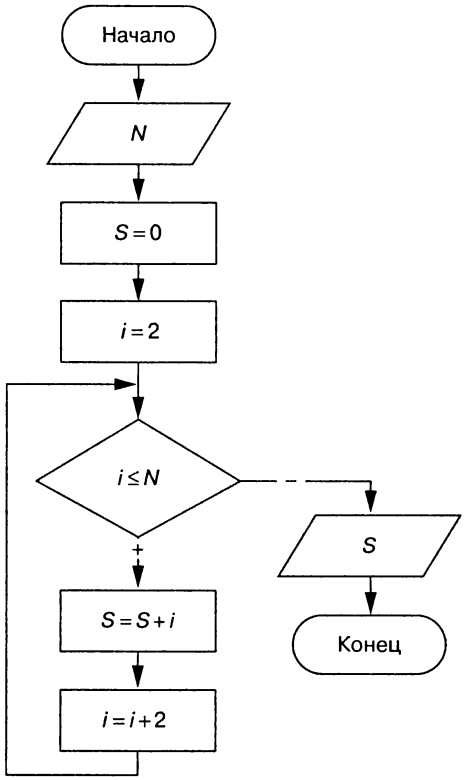


Рис. 3.18 ▾ Вычисление суммы четных натуральных чисел

```
  S:=0;
  i:=2;
  while i<=n do
  begin
     S:=S+i;
     i:=i+2;
  end;
  writeln('S=',S);
end.
```

Эту же задачу можно решить иначе, используя цикл `for ... do`:

```
var n,i,S:word;
begin
  write('n=');
```

```
  readln(n);
  S:=0;
  for i:=1 to n do
{ Если остаток от деления параметра цикла на 2 равен 0, то это число
четное. Следовательно, происходит накапливание суммы. }
     if i mod 2 = 0 then
        S:=S+i;
  writeln('S=',S);
end.
```

В табл. 3.4 приведены результаты тестирования программы для n = 7. Несложно заметить, что при нечетных значениях параметра цикла значение переменной, предназначенной для накапливания суммы, не изменяется.

Таблица 3.4 ▼ Суммирование четных чисел

i		1	2	3	4	5	6	7
S	0	0	2	2	6	6	12	12

Пример 3.13. Дано натуральное число *N*. Определить *K* – количество делителей этого числа, не превышающих его (*N* = 12, его делители 1, 2, 3, 4, 6, *K* = 5).

Входные данные: N – целое число.

Выходные данные: целое число K – количество делителей N.

Промежуточные данные: i – параметр цикла, возможные делители числа N.

В блок-схеме, изображенной на рис. 3.19, реализован следующий алгоритм: в переменную K, предназначенную для подсчета количества делителей заданного числа, помещается значение, которое не влияло бы на результат, то есть нуль. Далее организовывается цикл, в котором изменяющийся параметр i выполняет роль возможных делителей числа N. Если заданное число делится нацело на параметр цикла, это означает, что i является делителем N, и значение переменной K следует увеличить на единицу. Цикл необходимо повторить N/2 раз.

В табл. 3.5 отображены результаты тестирования алгоритма при определении делителей числа N = 12.

Таблица 3.5 ▼ Определение делителей числа N

i		1	2	3	4	5	6
K	0	1	2	3	4	4	5

```
var N,i,K:word;
begin
  write('N=');
  readln(N);
```

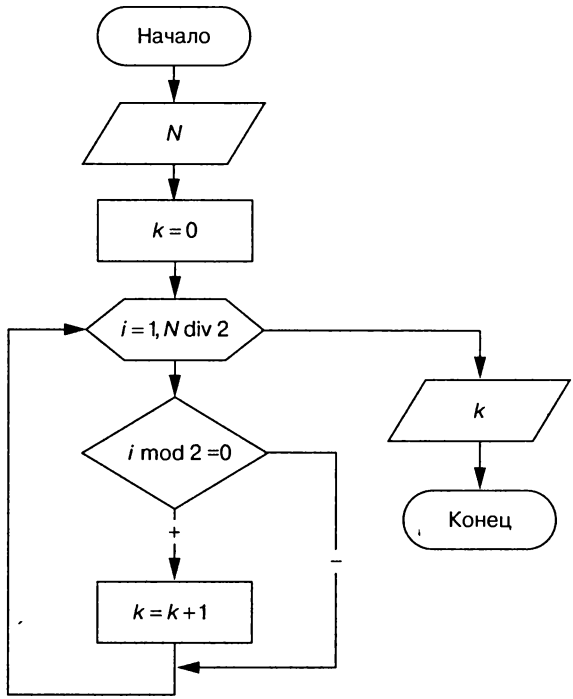


Рис. 3.19 ▾ Определение количества делителей натурального числа

```
  K:=0;       { Присвоение начального значения. }
  for i:=1 to N div 2 do
     if N mod i= 0 then      { Если N делится нацело на i, то }
        k:=K+1;         { увеличить счетчик на единицу. }
  writeln(' K=',K);
end.
```

Пример 3.14. Дано натуральное число *N*. Определить, является ли оно простым. Натуральное число *N* называется простым, если оно делится нацело без остатка только на единицу и *N*. Число 13 – простое, так как делится только на 1 и 13, *N* = 12 не является простым, так как делится на 1, 2, 3, 4, 6 и 12.

Входные данные: `N` – целое число.

Выходные данные: сообщение.

Промежуточные данные: `i` – параметр цикла, возможные делители числа `N`.

Алгоритм решения этой задачи (рис. 3.20) заключается в том, что число `N` делится на параметр цикла `i`, изменяющийся в диапазоне от 2 до `N`/2. Если

среди значений параметра не найдется ни одного числа, делящего заданное число нацело, то N – простое число, иначе оно таковым не является. Обратите внимание на то, что в алгоритме предусмотрено два выхода из цикла. Первый – естественный, при исчерпании всех значений параметра, а второй – досрочный. Нет смысла продолжать цикл, если будет найден хотя бы один делитель из указанной области изменения параметра.

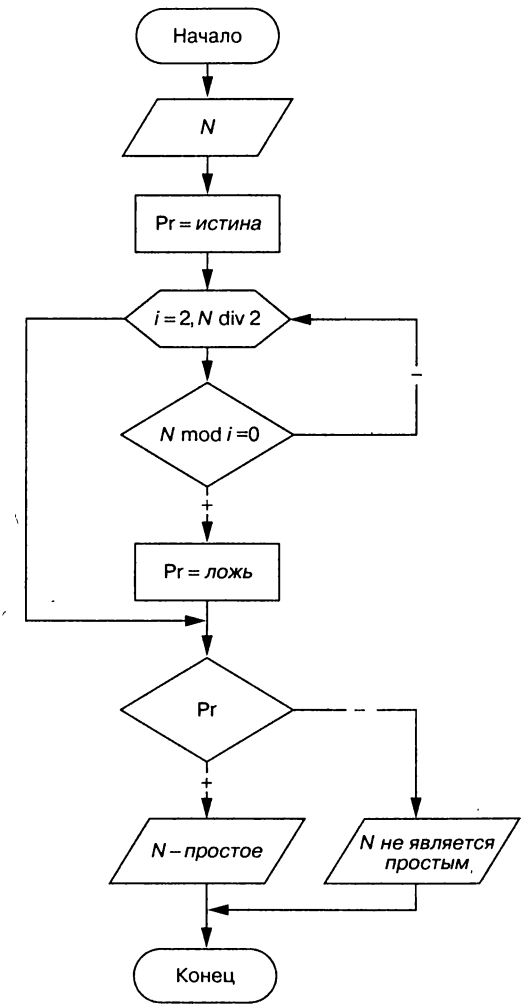


Рис. 3.20 ▼ Определение простого числа

При составлении программы на языке Турбо Паскаль досрочный выход из цикла удобно выполнять при помощи оператора break:

```
var N,i:integer;
Pr:boolean;
begin
  write('N=');
  readln(N);
  Pr:=true;          { Предположим, что число простое. }
  for i:=2 to N div 2 do
     if N mod i = 0 then     { Если найдется хотя бы один делитель, то }
     begin
        Pr:=false;         { число простым не является, и }
        break;          { досрочный выход из цикла. }
     end;
  if Pr then         { Проверка значения логического параметра и }
{ вывод на печать соответствующего сообщения. }
     writeln('Число ',N,' - простое')
  else
     writeln('Число ',N,' простым не является');
end.
```

Пример 3.15. Дано натуральное число *N*. Определить самую большую цифру и ее позицию в числе (*N* = 573863, наибольшей является цифра 8, ее позиция – четвертая слева).

Входные данные: N – целое число.

Выходные данные: max – значение наибольшей цифры в числе, pos – позиция этой цифры в числе.

Промежуточные данные: i – параметр цикла, kol – количество цифр в числе, M – переменная для временного хранения значения N.

Разобьем решение этой задачи на два этапа. Вначале найдем количество цифр в заданном числе (рис. 3.21), а затем определим наибольшую цифру и ее позицию (рис. 3.22).

Для того чтобы подсчитать количество цифр в числе, необходимо определить, сколько раз заданное число можно разделить на десять нацело. Например, пусть N = 12345, тогда количество цифр kol = 5. Результаты вычислений сведены в табл. 3.6. Процесс определения текущей цифры числа N = 12345 представлен в табл. 3.7.

Если цифры числа известны, определить наибольшую из них не составит труда. Алгоритм поиска максимального значения в некоторой последовательности цифр заключается в следующем. В ячейку, в которой будет храниться максимальный элемент (max), записывают значение, меньшее любого из элементов последовательности (в нашем случае max = –1, так как цифры числа находятся в

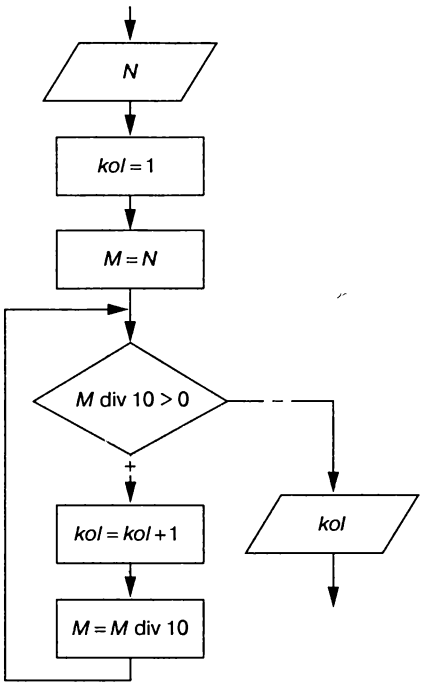


Рис. 3.21 ▼ Определение количества цифр в числе

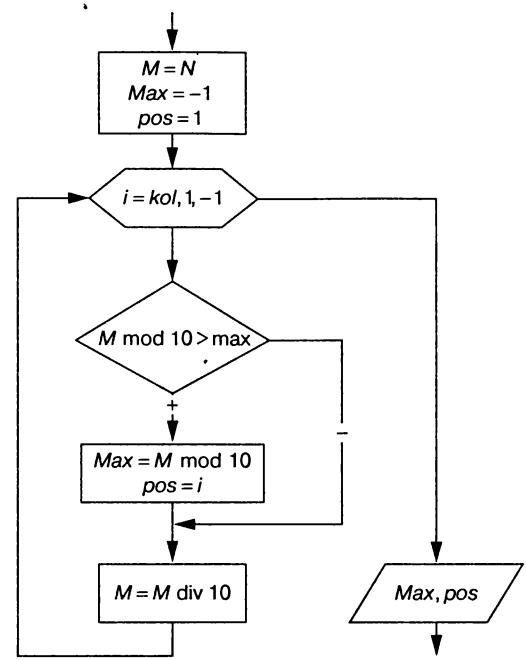


Рис. 3.22 ▼ Определение максимальной цифры в числе и ее позиции

Таблица 3.6 ▼ Определение количества цифр числа

kol	N
1	12345
2	12345 div10 = 1234
3	1234 div 10 = 123
4	123 div 10 = 12
5	12 div 10 = 1
	1 div 10 = 0

Таблица 3.7 ▾ Определение текущей цифры числа

i	Число M	Цифра
1	12345	12345 mod 10 = 5
2	12345 div 10 = 1234	1234 mod 10 = 4
3	1234 div 10 = 123	123 mod 10 = 3
4	123 div 10 = 12	12 mod 10 = 2
5	12 div 10 = 1	1 mod 10 = 1

диапазоне от 0 до 9). Затем сравнивают элементы последовательности со значением ячейки max, если найдется элемент, превышающий значение предполагаемого максимума, то ячейке max необходимо присвоить значение этого элемента и, соответственно, запомнить его номер в последовательности (в нашем случае переменной pos присваивается значение параметра цикла i)[1].

Текст программы, реализующей данную задачу, можно записать так:

```
var M,N,max:longint;
  i,kol,pos:word;
begin
{ Так как речь идет о натуральных числах, }
{ при вводе предусмотрена проверка. }
{ Закончить цикл, если введено положительное число, }
{ иначе повторить ввод. }
  repeat
     write('N=');
     readln(N);
  until N>0;
  M:=N;    { Сохранить значение переменной N. }
  kol:=1;     { Предположим, что число состоит из одной цифры. }
  while M div 10 > 0 do
{ Выполнять тело цикла, пока число делится нацело на 10. }
  begin
     kol:=kol+1;    { Счетчик количества цифр. }
     M:=M div 10;   { Изменение числа. }
  end;
  max:=-1;    { Присвоение максимуму начального значения. }
  pos:=1;     { Предполагаемая позиция максимума в числе. }
```

[1] В алгоритме поиска минимума вначале в переменную min записываем значение, заранее большее любого элемента последовательности. Затем все элементы последовательности сравниваем с min. Если встретится значение меньшее, чем min, переписываем его в переменную min.

```
  M:=N;
  for i:=kol downto 1 do { Нумерация цифр в числе слева направо. }
  begin
     if M mod 10 > max then
{ Если цифра больше предполагаемого максимума, то }
     begin
{ заменить максимум этой цифрой и }
        max:=M mod 10;
{ запомнить ее позицию в числе. }
        pos:=i;
     end;
     M:=M div 10;      { Изменение числа. }
  end;
  writeln('max=',max,' pos=',pos);
end.
```

Пример 3.16. Определить количество простых чисел в интервале от N до M, где N и M – натуральные числа (рис. 3.23).

```
var N,M,i,j,k: longint;
  Pr:boolean;
begin
  repeat
     write('N=');
     readln(N);
     write('M=');
     readln(M);
  until (n>0) and (m>0) and (n<M);
{ Перед вычислением количество простых чисел (k) равно 0. }
  k:=0;
{ Перебираем все числа от N до M. }
  for i:=N to M do
  begin
{ Проверяем, является ли число простым. }
     Pr:=true;
     for j:=2 to i div 2 do
        if i mod j = 0 then
        begin
           Pr:=false;
           break;
        end;
{ Если число простое, увеличиваем количество простых чисел на 1. }
     if Pr then
     k:=k+1;
  end;
  if k=0 then
     writeln('Простых чисел в диапазоне нет')
```

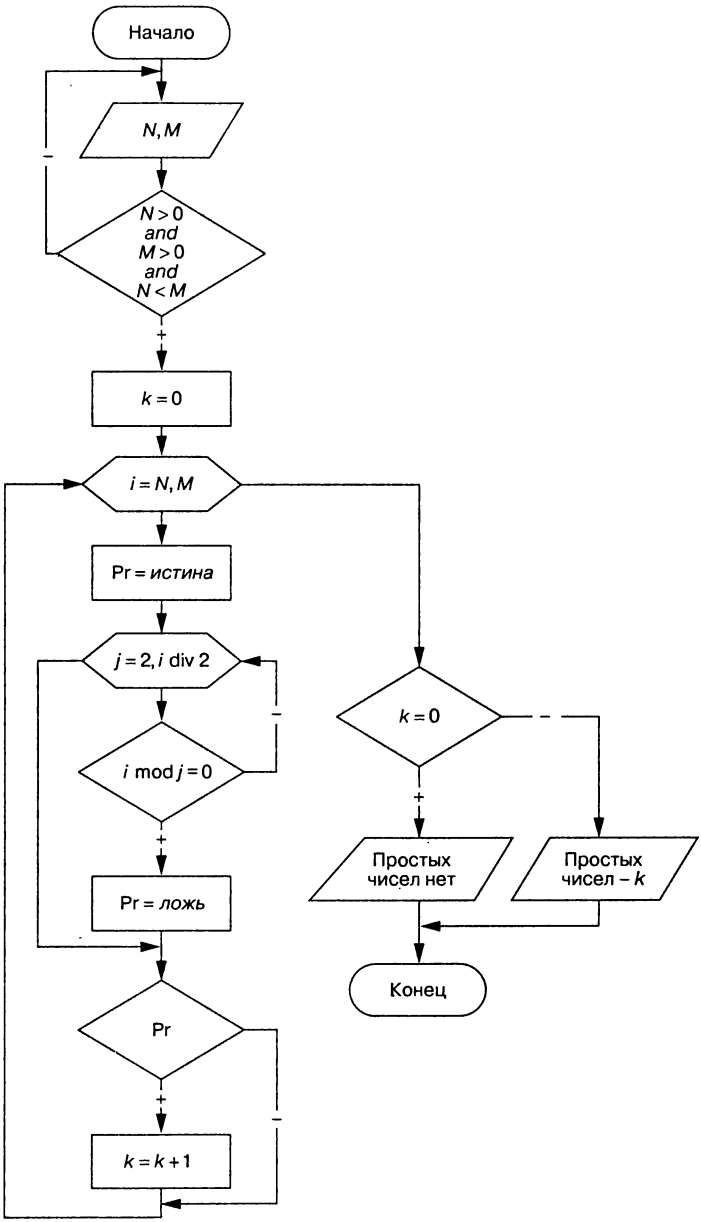


Рис. 3.23 ▾ Определение количества простых чисел в заданном интервале

```
  else
     writeln('Простых чисел в диапазоне ',k);
end.
```

Пример 3.17. Вводится последовательность целых чисел, 0 – ее конец. Найти минимальное среди положительных значений; если их несколько, определить количество.

Блок-схема решения задачи приведена на рис. 3.24.

Заметим, что логическая переменная Pr определяет наличие положительных чисел в последовательности (Pr=false, если положительных чисел нет). Текст программы с комментариями приведен ниже.

```
var min, n, k: integer;
  pr:boolean;
begin
{ Вводим первое число последовательности. }
  write('N=');readln(N);
{ Предполагаем, что в последовательности нет положительных значений. }
  pr:=false;
{ Количество минимумов равно 0. }
  k:=0;
{ Если текущее значение не равно 0, то входим в цикл. }
  while(N<>0) do
  begin
{ Если число положительное. }
     if N>0 then
{ Проверяем, если это первое положительное число, }
        if not Pr then
        begin
{ в последовательности есть положительные числа, }
           Pr:=true;
{ первое положительное число записываем в переменную min. }
           min:=N;
{ количество минимумов =1. }
           k:=1
        end
{ Если это не первое положительное число, }
        Else
{ если число меньше min, }
        if min>N then
        begin
{ то его переписываем в переменную min, }
           min:=N;
{ количество минимумов =1. }
           k:=1
        end
```

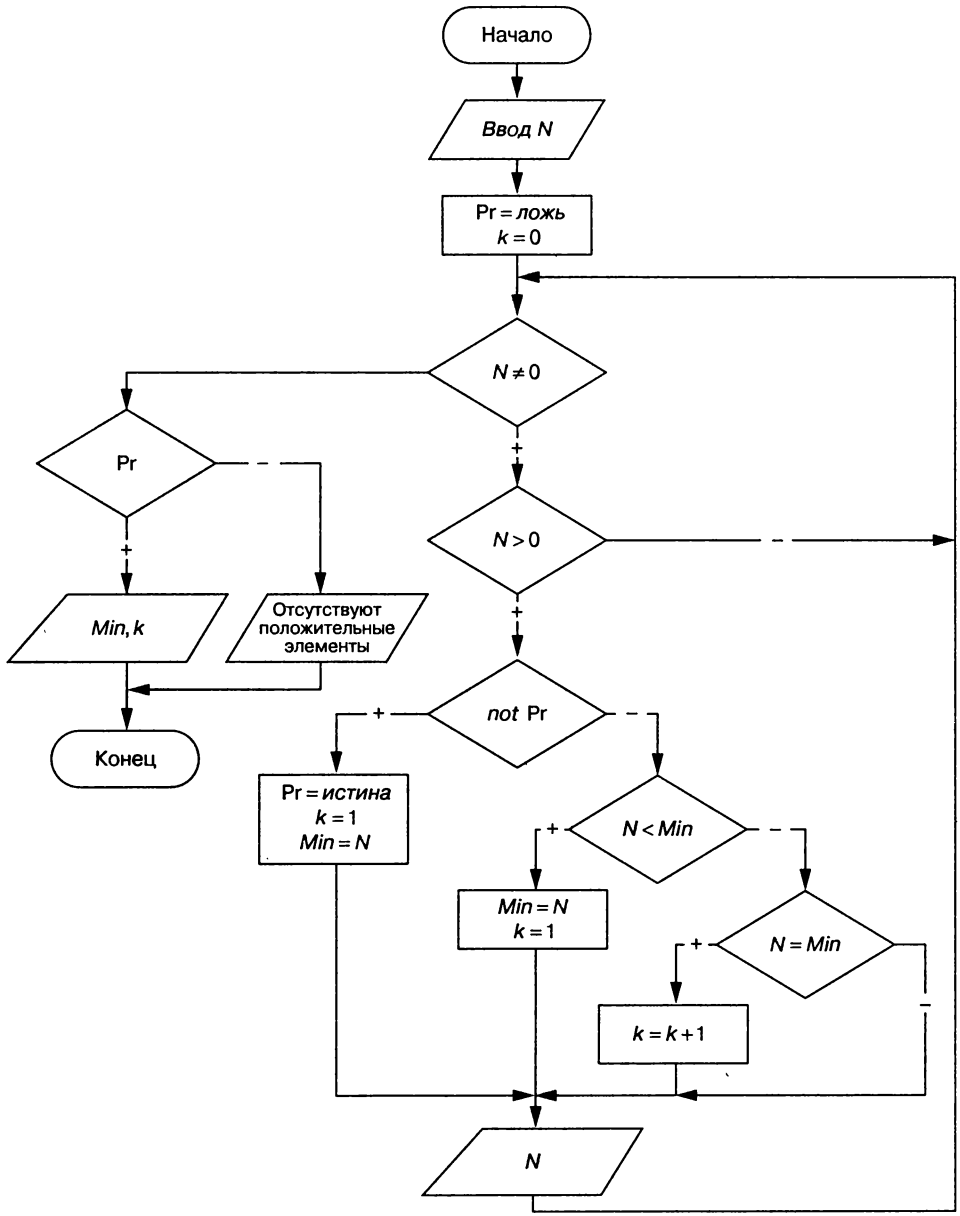


Рис. 3.24 ▾ Блок-схема нахождения минимального положительного числа последовательности

```
        else
{ Если текущий элемент последовательности равен min, количество минимумов
увеличиваем на 1. }
        if min=N then k:=k+1;
{ Вводим очередное значение последовательности. }
     write('N=');readln(N);
  end;
  if Pr then
     writeln('min=',min,'. Количество минимумов =',k)
  else
     writeln('Отсутствуют положительные элементы.')
end.
```

3.4. Упражнения

Изобразите блок-схему решения задачи и напишите программу на языке Турбо Паскаль.

1. Вычислить сумму натуральных нечетных чисел, не превышающих N.
2. Вычислить произведение натуральных чисел, кратных трем и не превышающих N.
3. Определить значение выражения $P = a^{(n-1)}/(n+1)!$.
4. Вводится последовательность ненулевых чисел, 0 – ее конец. Определить, содержит ли последовательность хотя бы два соседних числа с одинаковыми знаками.

Произведение двух чисел с одинаковыми знаками – положительное число.

5. Определить, сколько раз последовательность из N произвольных чисел меняет знак.

Произведение двух чисел с разными знаками – отрицательное число.

6. Вычислить сумму положительных элементов последовательности из N произвольных чисел.
7. В последовательности из N произвольных чисел подсчитать количество нулей.
8. Вводится последовательность ненулевых чисел, 0 – ее конец. Определить наибольшее число в последовательности.

9. Дано натуральное число P. Определить все простые числа, не превосходящие P.
10. Определить количество четных цифр в заданном натуральном числе M.
11. Вычислить сумму цифр натурального числа N.
12. Определить, является ли число L совершенным. Совершенное число L равно сумме всех своих делителей, не превосходящих L. Например, $6 = 1 + 2 + 3$ или $28 = 1 + 2 + 4 + 7 + 14$.

В основе решения задачи лежит алгоритм из примера 3.13.

13. Вводится последовательность из M элементов. Каждый элемент последовательности – цифра (то есть находится в диапазоне от 0 до 9). Сформировать число N, считая первый элемент последовательности младшим разрядом. Например, дана последовательность 5, 4, 3, 2, 1, тогда десятичное число формируется следующим образом:
$5 + 4 \cdot 10 + 3 \cdot 100 + 2 \cdot 1000 + 1 \cdot 10000 = 12345$.
14. Дано натуральное число N. Записать его цифры в обратном порядке. Например, $12345 \rightarrow 54321$.

Определите разрядность данного числа так, как показано в табл. 3.8. Затем сформируйте новое число, воспользовавшись примером из табл. 3.9.

Таблица 3.8 ▾ Процесс определения старшего разряда числа

Число	Разрядность
12345	1
12345 div 10 = 1234	10
1234 div 10 = 123	100
123 div 10 = 12	1000
12 div 10 = 1	10000

Таблица 3.9 ▾ Процесс записи цифр числа в обратном порядке

Цифра	Разрядность	Число S = 0
12345 mod 10 = 5	10000 (десятки тысяч)	S = S + 5 · 10000 = 50000
1234 mod 10 = 4	1000 (тысячи)	S = S + 4 · 1000 = 54000
123 mod 10 = 3	100 (сотни)	S = S + 3 · 100 = 54300
12 mod 10 = 2	10 (десятки)	S = S + 2 · 10 = 54320
1 mod 10 = 1	1 (единицы)	S = S + 1 · 1 = 54321

15. Дано натуральное число N. Определить, является ли оно автоморфным. Автоморфное число N равно последним разрядам числа N^2. Например, 5 – 25, 6 – 36, 25 – 625.

Определите количество разрядов R в числе N. Если остаток от деления числа N^2 на $(10 \cdot R)$ равен числу N, то оно автоморфное. Например, старший разряд числа 25 – десятки (см. табл. 3.8). Значит, 25^2 mod 100 = 625 mod 100 = 25, следовательно, число автоморфно.

Глава 4 Обработка массивов в Турбо Паскале

Часто для работы с множеством однотипных данных (целочисленными значениями, строками, датами и т.п.) оказывается удобным использовать массивы. Например, можно создать массив для хранения списка студентов, обучающихся в одной группе. Вместо того, чтобы задавать переменные для каждого студента, например `Студент1`, `Студент2` и т.д., достаточно создать один массив, где каждой фамилии из списка будет присвоен порядковый номер. Таким образом, можно дать следующее определение. *Массив* – структурированный тип данных, состоящий из фиксированного числа элементов одного типа [4].

Массив на рис. 4.1 имеет 8 элементов, каждый из которых сохраняет число вещественного типа. Элементы в массиве пронумерованы от 1 до 8. Такого рода массив, представляющий собой список данных одного и того же типа, называют *простым*, или *одномерным*. Для доступа к данным, хранящимся в определенном элементе массива, необходимо указать имя массива и порядковый номер этого элемента, называемый *индексом*.

12.1	0.13	-1.5	0	21.9	-3.7	5.0	121.7
1-й элемент массива	2-й элемент массива	3-й элемент массива	4-й элемент массива	5-й элемент массива	6-й элемент массива	7-й элемент массива	8-й элемент массива

Рис. 4.1 ▾ Одномерный числовой массив

Если возникает необходимость хранения данных в виде таблиц, в формате строк и столбцов, то необходимо использовать *многомерные массивы*. На рис. 4.2

приведен пример массива, состоящего из четырех строк и четырех столбцов. Это *двумерный массив*. Строки в нем можно считать первым измерением, а столбцы – вторым. Для доступа к данным, хранящимся в этом массиве, необходимо указать его имя и *два индекса*, первый из которых должен соответствовать номеру строки, а второй – номеру столбца, в котором хранится необходимый элемент.

		Номера столбцов			
		1	2	3	4
Номера строк	1	3.5	7.8	1.3	0.6
	2	-1.4	0.3	0	12.1
	3	-5.7	-0.78	5.0	6.9
	4	45.1	124.0	-24.7	0.96

Рис. 4.2 ▼ Двумерный числовой массив

Рассмотрим работу с массивами в языке Турбо Паскаль.

4.1. Описание массивов

Для описания массива служат служебные слова `array of`. Сама же процедура может выполняться двумя способами:

1. Ввести новый тип данных, а потом описать переменные нового типа. В этом случае формат команды `type` следующий:

```
type
имя_типа = array [тип_индекса] of тип_компонентов;
```

В качестве `типа_индекса` следует использовать перечислимый тип. `Тип_компонентов` – это любой ранее определенный тип данных, например:

```
type
massiv=array[0..12] of real;
dabc=array[-3..6] of integer;
```

```
var
x,y:massiv;
z: dabc;
```

2. Можно не вводить новый тип, а просто описать переменную следующим образом:

```
var переменная : array [тип_индекса] of тип_переменных;
```

Например:

```
var
z,x: array[1..25] of word;
g:array[-2..7] of real;
```

Для описания массива можно использовать предварительно определенные константы:

```
const
n=10;
m=12;
var
a: array[1..n] of real;
b: array[0..m] of byte;
```

Константы должны быть определены до использования, так как массив не может быть переменной длины!

Размер массива не может превышать 64 Кб. Это ограничение относится и к прочим структурам данных.

Двумерный массив можно описать, применив в качестве базового типа (типа компонентов) одномерный:

```
type
massiv=array[1..20] of real;
matrica=array[1..20] of massiv;
var
a:matrica;
```

Такая же структура получается и при использовании другой формы записи:

```
Type
matrica = array [1..20,1..20] of real;
var
a:matrica;
```

или

```
var
a:array [1..20,1..20] of real;
```

Аналогично можно ввести трехмерный массив или массив большего числа измерений:

```
type
abc=array [1..4,0..6,-7..8,3..11] of real;
var
b:abc;
```

4.2. Операции над массивами

Для работы с массивом как с единым целым надо использовать его идентификатор без указания индекса в квадратных скобках. Доступ к каждому элементу осуществляется с помощью индекса, то есть порядкового номера элемента массива. Для обращения к элементу массива надо указать имя массива и порядковый номер элемента: `x[1]`, `y[5]`, `c[25]`, `A[8]`. Если указано только имя массива, то речь идет обо всем массиве (`A`, `C` и `F`). Для обработки массива нужно организовать цикл и уже в нем работать с элементами.

В Турбо Паскале определена также операция присваивания для массивов, идентичных по структуре, то есть с одинаковыми типами индексов и компонентов. Например, если массивы `C` и `D` описаны как

```
var c,d: array [0..30] of real;
```

то можно записать оператор

```
c:=d;
```

Такая операция сразу всем элементам массива `C` присвоит значения элементов массива `D`, соответствующих им по номерам.

4.3. Ввод-вывод элементов массива

Паскаль не имеет специальных средств ввода-вывода всего массива, поэтому данную операцию следует организовывать поэлементно. При вводе массива необходимо последовательно вводить 1-й, 2-й, 3-й и т. д. его элементы и аналогичным образом поступить при выводе. Следовательно, для ввода-вывода необходимо организовать цикл, в котором практически все операции с массивами необходимо проводить поэлементно. Для обработки элементов массива удобно использовать цикл `for ... do`.

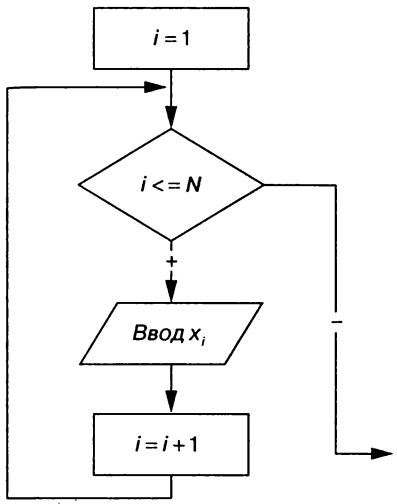


Рис. 4.3 ▾ Алгоритм ввода массива с использованием цикла с предусловием

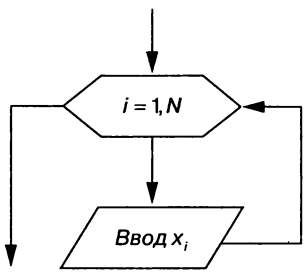


Рис. 4.4 ▾ Алгоритм ввода массива с использованием блока модификации

Блок-схемы алгоритмов ввода элементов массива изображены на рис. 4.3 и 4.4.

```
{ Ввод элементов массива X с помощью цикла while. }
var
x: array [1..100] of real;
  i,n: integer;
begin
  writeln ('введите размер массива');
  readln(N);
  i:=1;
  while (i<=N) do
  begin
        write('x(',i,')= ');
        readln(x[i]);
        i:=i+1
  end;
end.
{ Ввод элементов массива X с помощью цикла for. }
var
  x: array [1..100] of real;
  i,n: integer;
```

```
begin
  readln(N);
  for i:=1 to N do
  begin
        write('x(',i, ')= ');
        readln(x[i])
  end;
end.
```

Как видно, цикл for ... do удобно использовать для обработки всего массива, и в дальнейшем при выполнении таких операций мы будем применять именно его.

Вывод массива организуется аналогично вводу.

Предлагаем читателю рассмотреть несколько вариантов вывода массива и самостоятельно разобраться, чем они отличаются друг от друга и какой из них лучше.

```
{ Вариант 1 }
for i: = 1 to n do write (a[i]:6:2);
{ Вариант 2 }
for i: = 1 to n do write (a[i]:6:2,' ');
{ Вариант 3 }
writeln ('массив A');
for i:=1 to n do writeln (a[i]:6:2);
{ Вариант 4 }
for i:=1 to n do write ('a[',i,']=',a[i]:6:2,' ')
{ Вариант 5 }
for i:=1 to n do writeln ('a[',i, ']= ',a[i]:6:2);
```

Рассмотрим несколько примеров обработки массивов. Алгоритмы, с помощью которых обрабатывают одномерные массивы, похожи на обработку последовательностей (вычисление суммы, произведения, поиск элементов по определенному признаку, выборки и т.д.). Отличие заключается в том, что в массиве одновременно доступны все его компоненты, поэтому становится возможной, например, сортировка его элементов и другие более сложные преобразования.

4.4. Вычисление суммы элементов массива

Дан массив X, состоящий из n элементов. Найти сумму его элементов. Процесс накапливания суммы элементов массива достаточно прост и практически ничем не отличается от суммирования значений некоторой числовой

последовательности. Переменной S присваивается значение, равное нулю, затем последовательно суммируются элементы массива X. Блок-схема алгоритма расчета суммы приведена на рис. 4.5.

Соответствующий алгоритму фрагмент программы будет иметь вид:

```
s:=0;
for i:=1 to N do
  s:=s+x[i];
```

4.5. Вычисление произведения элементов массива

Дан массив X, состоящий из n элементов. Найти произведение элементов этого массива. Решение задачи сводится к тому, что значение переменной P, в которую предварительно была записана единица, последовательно умножается на значение i-го элемента массива. Блок-схема алгоритма приведена на рис. 4.6.

Соответствующий фрагмент программы будет иметь вид:

```
p:=1;
for i:=1 to N do
p:=p*x[i];
```

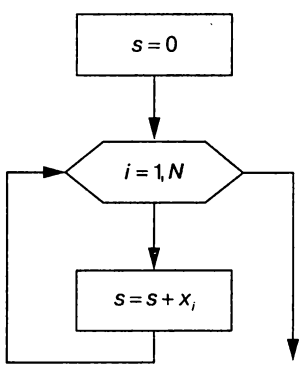


Рис. 4.5 ▾ Алгоритм вычисления суммы элементов массива

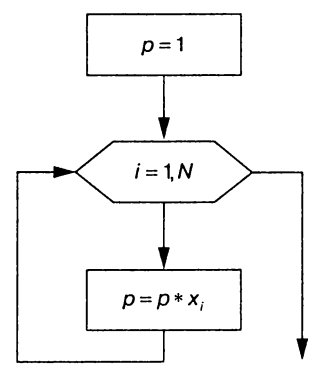


Рис. 4.6 ▾ Вычисление произведения элементов массива

4.6. Поиск максимального элемента и его номера в массиве

Дан массив X, состоящий из n элементов. Найти максимальный элемент и номер, под которым он хранится в массиве.

Алгоритм решения задачи следующий. Пусть в переменной с именем Max хранится значение максимального элемента массива, а в переменной с именем Nmax – его номер. Предположим, что первый элемент массива является максимальным, и запишем его в переменную Max, а в Nmax – его номер (то есть 1). Затем все элементы, начиная со второго, сравниваем в цикле с максимальным. Если текущий элемент массива оказывается больше максимального, то записываем его в переменную Max, а в переменную Nmax – текущее значение индекса i. Процесс определения максимального элемента в массиве приведен в табл. 4.1 и изображен при помощи блок-схемы на рис. 4.7.

Таблица 4.1 ▼ Определение максимального элемента и его номера в массиве

	Номера элементов						
	1	2	3	4	5	6	7
Исходный массив	4	7	3	8	9	2	5
Значение переменной Max	4	7	7	8	9	9	9
Значение переменной Nmax	1	2	2	4	5	5	5

Соответствующий фрагмент программы имеет вид:

```
Max:=X[1];
Nmax:=1;
for i:=2 to N do
  if X[i]>Max then
  begin
     Max:=X[i];
     Nmax:=i;
  end;
writeln('Max=',Max:1:3, ' Nmax=', Nmax);
```

Алгоритм поиска минимального элемента в массиве будет отличаться от приведенного выше лишь тем, что в условном блоке и, соответственно, в конструкции `if` *текста программы знак поменяется с* > *на* <.

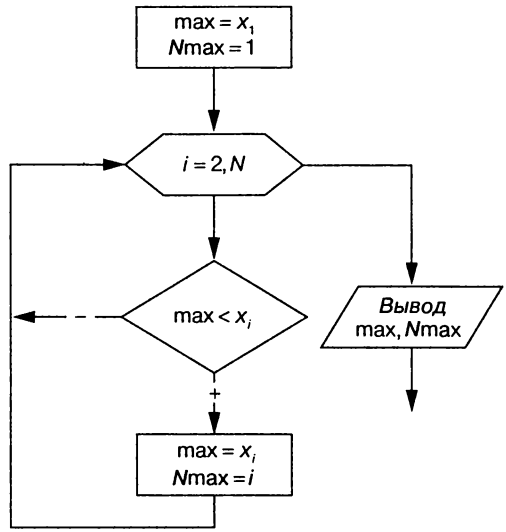


Рис. 4.7 ▾ Поиск максимального элемента и его номера в массиве

4.7. Сортировка элементов в массиве

Сортировка представляет собой процесс упорядочения элементов в массиве по возрастанию или убыванию их значений . Например, массив `X` из `n` элементов будет отсортирован в порядке возрастания значений его элементов, если

$$X[1] \le X[2] \le \ldots \le X[n],$$

и в порядке убывания, если

$$X[1] \ge X[2] \ge \ldots \ge X[n].$$

Существует большое количество алгоритмов сортировки, но все они базируются на трех основных [12]:

- сортировка обменом;
- сортировка выбором;
- сортировка вставкой.

Представим, что нам необходимо разложить по порядку карты в колоде. Для сортировки карт *обменом* можно разложить их на столе лицевой стороной вверх и менять местами те, которые расположены в неправильном порядке, делая это до тех пор, пока колода не станет упорядоченной.

Для сортировки *выбором* из разложенных на столе карт выбирают самую младшую (старшую) карту и держат ее в руках. Затем из оставшихся вновь выбирают наименьшую (наибольшую) по значению карту и помещают ее позади той, которая была выбрана первой. Этот процесс повторяется до тех пор, пока вся колода не окажется в руках. Поскольку каждый раз выбирается наименьшая (наибольшая) по значению карта из оставшихся на столе, по завершении такого процесса карты будут отсортированы по возрастанию (убыванию).

Для сортировки *вставкой* из колоды берут две карты и располагают их в необходимом порядке по отношению друг к другу. Каждая следующая карта, взятая из колоды, должна быть установлена на соответствующее место по отношению к уже упорядоченным.

Итак, решим следующую задачу. Задан массив `Y` из `n` целых чисел. Расположить элементы массива в порядке возрастания их значений.

4.7.1. Сортировка методом «пузырька»

Сортировка пузырьковым методом является наиболее известной. Ее популярность объясняется запоминающимся названием, которое обусловлено подобием процессу движения пузырьков в резервуаре с водой, когда каждый пузырек находит свой собственный уровень, и простотой алгоритма. Сортировка методом «пузырька» использует метод обменной сортировки и основана на выполнении в цикле операций сравнения и при необходимости обмена соседних элементов. Рассмотрим алгоритм пузырьковой сортировки более подробно.

Сравним первый элемент массива со вторым. Если первый окажется больше второго, то поменяем их местами. Те же действия выполним для второго и третьего, третьего и четвертого, `I`-го и (`I` + 1)-го, (`n` – 1)-го и `n`-го элементов. В результате этих действий самый большой элемент станет на последнее (`n`-е) место. Теперь повторим данный алгоритм сначала, но последний (`n`-й) элемент, рассматривать не будем, так как он уже занял свое место. После проведения данной операции самый большой элемент оставшегося массива встанет на (`n` – 1)-е место. Так повторяем до тех пор, пока не упорядочим весь массив.

В табл. 4.2 подробно расписан процесс упорядочивания элементов в массиве. Нетрудно заметить, что для преобразования массива, состоящего из n элементов, необходимо просмотреть его (n – 1) раз, каждый раз уменьшая диапазон просмотра на один элемент. Блок-схема описанного алгоритма приведена на рис. 4.8. Обратите внимание на то, что для перестановки элементов (блок 4) используется буферная переменная b, в которой временно хранится значение элемента, подлежащего замене.

Таблица 4.2 ▾ Процесс упорядочивания элементов в массиве по возрастанию

	Номер элемента				
	1	**2**	**3**	**4**	**5**
Исходный массив	7	3	5	4	2
Первый просмотр	3	5	4	2	7
Второй просмотр	3	4	2	5	7
Третий просмотр	3	2	4	5	7
Четвертый просмотр	2	3	4	5	7

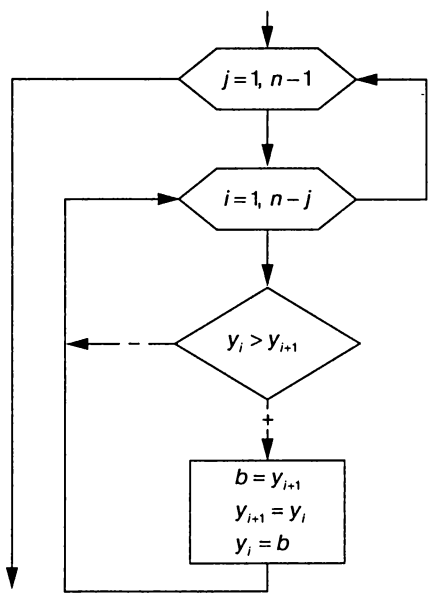


Рис. 4.8 ▾ Сортировка массива пузырьковым методом

```
{ Упорядочивание элементов в массиве по возрастанию их значений. }
var
i,n,j: integer;
b:word;
y: array [1..100] of word;
begin
writeln ('введите размер массива ');
readln (n);
for i:=1 to n do
begin
  write('y[',i,']=');
  readln (y[i]);
end;
writeln ('массив y ');
for i:=1 to n do write (y [i],' ');
writeln;
for j:=1 to n-1 do
for i:=1 to n-j do
  if y[i] > y[i+1] then  { Если текущий элемент больше следующего, то }
  begin { поменять их местами. }
     b:=y[i]; { Сохранить значение текущего элемента. }
     y[i]:=y[i+1];  { Заменить текущий элемент следующим. }
     y[i+1]:=b;  { Заменить следующий элемент текущим. }
  end;
writeln('упорядоченный массив');
for i:=1 to n do
  write (y[i],' ');
writeln;
end.
```

Для перестановки элементов в массиве по убыванию их значений необходимо при их сравнении заменить знак > на <.

4.7.2. Сортировка выбором

Алгоритм приведен в виде блок-схемы на рис. 4.9. Найдем в массиве самый большой элемент (блоки 3–7) и поменяем его местами с последним элементом (блок 8). Повторим алгоритм поиска максимального элемента, уменьшив количество просматриваемых элементов на единицу (блок 9), и поменяем его местами с предпоследним элементом (блоки 3–7). Описанную выше операцию поиска проводим до полного упорядочивания элементов в массиве. Так как в блоке 9 происходит изменение переменной n, то в начале алгоритма ее значение необходимо сохранить (блок 1).

Для упорядочивания массива по убыванию необходимо перемещать минимальный элемент.

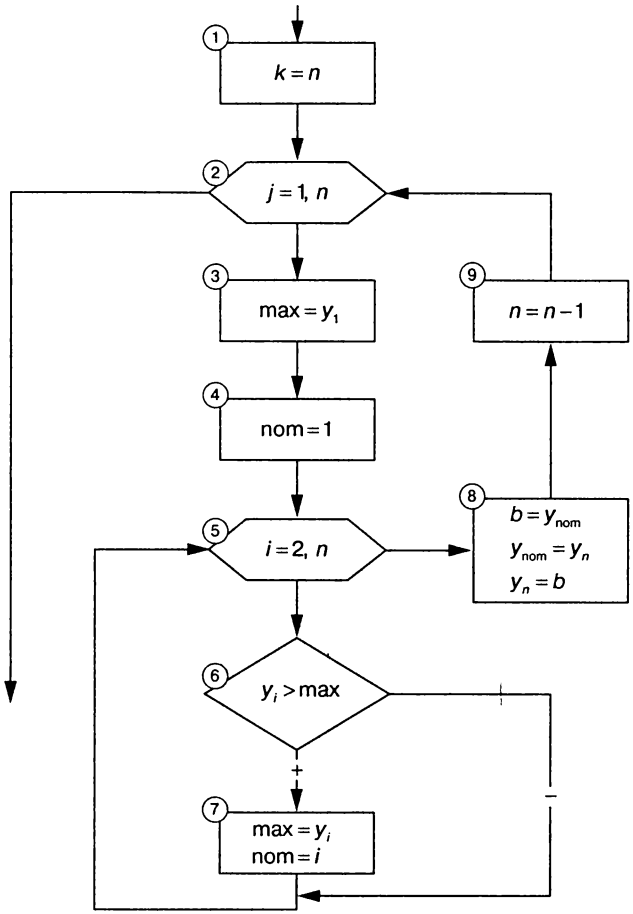


Рис. 4.9 ▾ Сортировка массива выбором наибольшего элемента

4.7.3. Сортировка вставкой

Сортировка вставкой заключается в том, что сначала упорядочиваются два элемента массива. Затем делается вставка третьего элемента в соответствующее место по отношению к первым двум. Четвертый элемент помещают в список из уже упорядоченных трех элементов. Этот процесс повторяется до тех пор, пока все элементы не будут упорядочены.

Прежде чем приступить к составлению блок-схемы, рассмотрим следующий пример. Пусть известно, что в массиве из восьми элементов первые шесть уже упорядочены, а седьмой элемент нужно вставить между вторым и четвертым. Сохраним его во вспомогательной переменной, как показано на рис. 4.10, а на его место запишем шестой. Далее пятый элемент переместим на место шестого, четвертый – на место пятого, а третий – на место четвертого, тем самым выполнив сдвиг элементов массива на одну позицию вправо. Записав содержимое вспомогательной переменной в третью позицию, достигнем нужного результата.

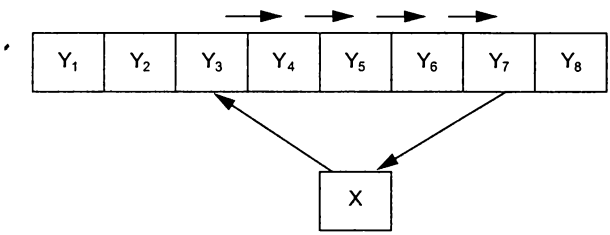


Рис. 4.10 ▼ Процесс вставки элемента в массив

Составим блок-схему алгоритма (рис. 4.11), учитывая, что описанные выше действия, возможно, придется выполнить неоднократно. Организуем цикл для просмотра всех элементов массива, начиная со второго (блок 4). Сохраним значение текущего I-го элемента во вспомогательной переменной X, так как оно может быть потеряно при сдвиге элементов (блок 5) и присвоим переменной j значение индекса предыдущего (I – 1)-го элемента массива (блок 6). Далее движемся по массиву влево в поисках элемента, меньшего, чем текущий, и пока он не найден, сдвигаем элементы вправо на одну позицию. Для этого организуем цикл (блок 7), который прекратится, как только будет найден элемент меньше текущего. Если такого элемента в массиве не найдется и переменная j станет равной нулю, это будет означать, что достигнута левая граница массива, и текущий элемент необходимо установить в первую позицию. Смещение элементов массива вправо на одну позицию выполняется в блоке 8, а изменение счетчика j – в блоке 9. Блок 10 выполняет вставку текущего элемента в соответствующую позицию.

Далее приведен фрагмент программы, реализующей сортировку массива методом вставки.

```
for i := 2 to n do
begin
  x := y[i];{ Сохраним текущий элемент массива. }
```

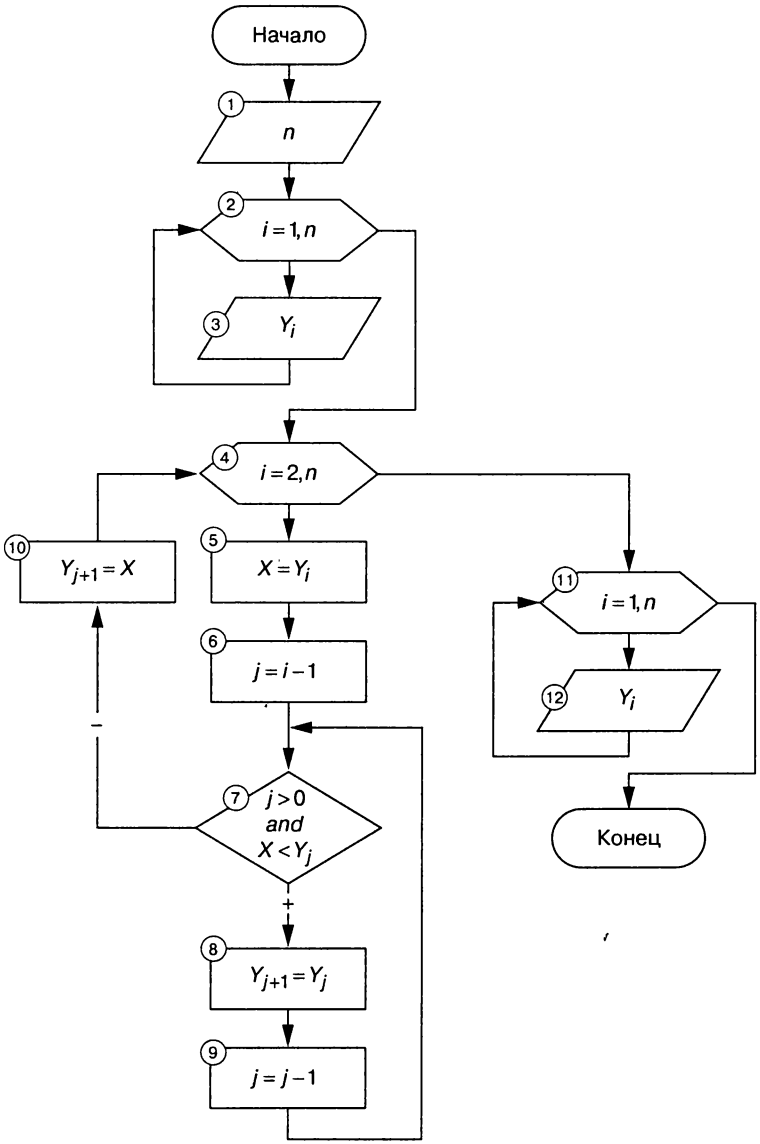


Рис. 4.11 ▾ Сортировка массива вставкой

```
{ В переменной j будем хранить номера элементов, предшествующих текущему. }
  j := i-1;
{ Сдвиг массива на одну позицию вправо до тех пор, пока }
{ текущий элемент меньше предшествующих и не достигнут левый край массива. }
  while (x<y[j]) and (j>0) do
  begin
     y[j+1] := y[j];
     j := j-1;
  end;
  { Запись текущего элемента на соответствующую позицию, }
  { то есть перед элементами, превышающими его. }
  y[j+1] := x;
end;
```

4.8. Удаление элемента из массива

Необходимо удалить из массива X, состоящего из n элементов, m-й по номеру элемент. Для этого достаточно записать элемент (m + 1) на место элемента m, (m + 2) – на место (m + 1) и т.д., n – на место (n – 1) и при дальнейшей работе с этим массивом использовать (n – 1)-й элемент (рис. 4.12). Алгоритм удаления из массива X размерностью n элемента с номером m приведен на рис. 4.13.

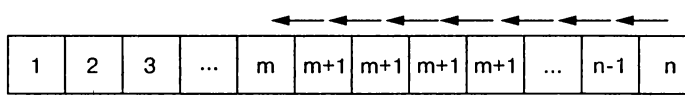


Рис. 4.12 ▼ Процесс удаления элемента из массива

4.9. Примеры программ

Пример 4.1. Дан массив A, состоящий из k целых положительных чисел. Записать все четные по значению элементы массива A в массив B.

Решение задачи заключается в следующем. Последовательно перебираются элементы массива A. Если среди них находятся четные, то они записываются в массив B. На рис. 4.14 видно, что первый четный элемент хранится в массиве A под номером 3, второй и третий – под номерами 5 и 6 соответственно, а четвертый – под номером 8. В массиве B этим элементам присваиваются совершенно иные номера. Поэтому для их формирования необходимо определить

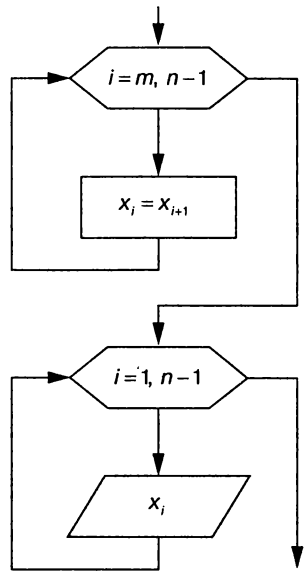


Рис. 4.13 ▾ Алгоритм удаления элемента из массива

дополнительную переменную. В блок-схеме, приведенной на рис. 4.15, такой переменной является m. Операция, выполняемая в блоке 2, означает, что в массиве может не быть искомых элементов. Если же условие в блоке 5 выполняется, то переменная m увеличивается на единицу, а значение элемента массива A записывается в массив B под номером m (блок 6). Условный блок 7 необходим для того, чтобы проверить, выполнилось ли хотя бы раз условие поиска (блок 5).

Приведенная ниже программа реализует описанный алгоритм.

```
var a,b:array [1..20] of word;
k,m,i:byte;
begin
write('введите размерность массива k=');
readln(k);
m:=0;
for i:=1 to k do
begin
write('A[',i,']=');
readln(A[i]);
if A[i] mod 2 =0 then
```

i	1	2	3	4	5	6	7	8
A	3	7	4	9	2	8	1	6

B	4	2	8	6
m	1	2	3	4

Рис. 4.14 ▼ Процесс формирования массива B из элементов массива A

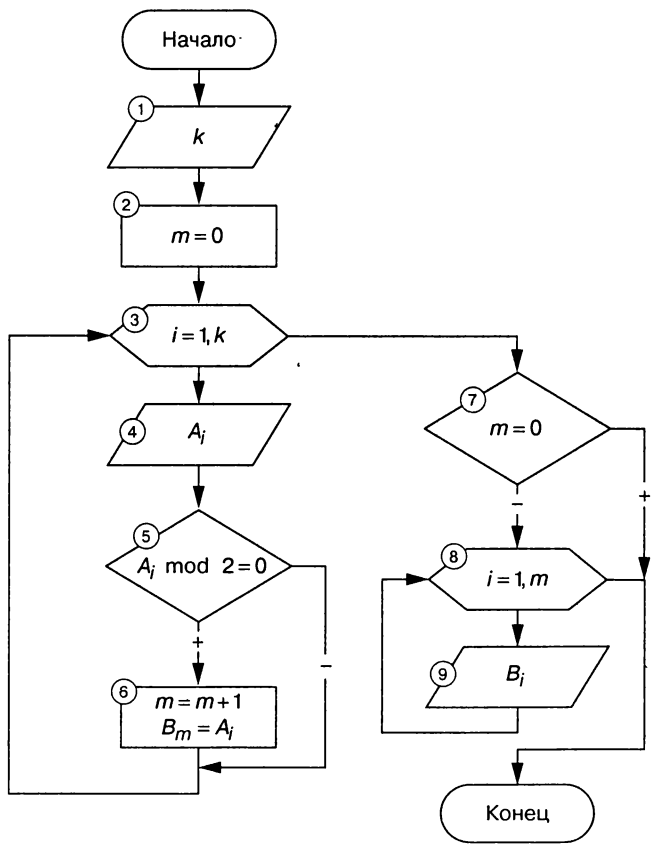


Рис. 4.15 ▼ Формирование массива B из соответствующих элементов массива A

```
begin
m:=m+1;
B[m]:=A[i];
end;
end;
if m<>0 then
for i:=1 to m do
write(B[i],' ')
else
write('В массиве нет четных элементов!!!');
end.
```

Пример 4.2. Задан массив y из n целых чисел. Сформировать массив z таким образом, чтобы вначале шли отрицательные элементы массива y, затем положительные и наконец нулевые. Блок-схема представлена на рис. 4.16.

```
{ Пример 4.2.}
program mas_four;
var
y,z:array [1..50] of integer;
i,k,n: integer;
begin
writeln ('введите n<=50');
readln (n);
for i:=1 to n do
begin
  write('y[',i,']=');
  readln(y[i]);
end;
k:=0;
for i:=1 to n do
  if y[i] < 0 then
  begin
     k:=k+1;
     z[k]:=y[i];
  end;
for i:=1 to n do
  if y[i] >0 then
  begin
     k:=k+1;
     z[k]:=y[i]
  end;
for i:=1 to n do
  if y[i]=0 then
  begin
     k:=k+1;
     z[k]:= y[i];
  end;
```

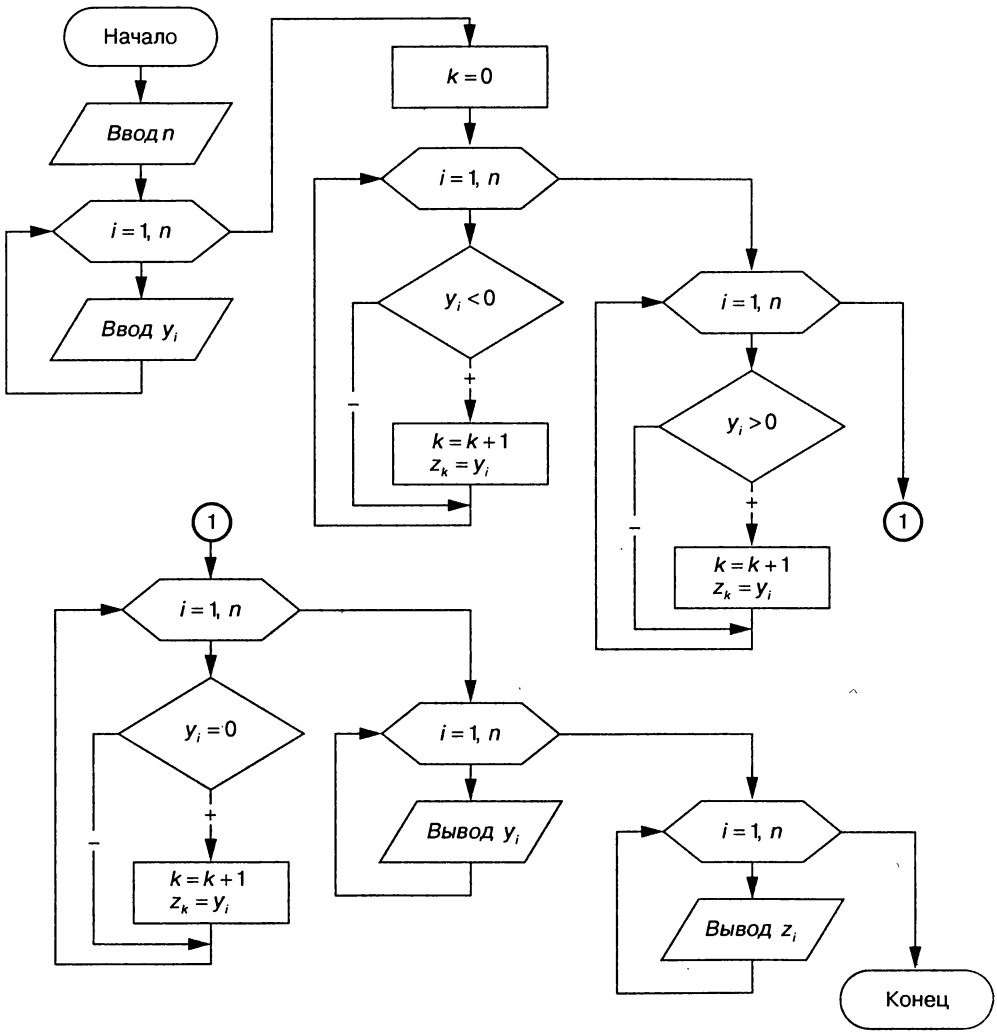


Рис. 4.16 ▾ Алгоритм примера 4.2

```
writeln ('Массив Y:');
for i:=1 to n do
  write(y[i],' ');
writeln;
writeln ('Массив Z:');
```

```
for i:=1 to n do
  write (z[i],' ');
writeln;
end.
```

Пример 4.3. Переписать элементы массива x в обратном порядке. Блок-схема представлена на рис. 4.17. Алгоритм состоит в следующем: меняем местами 1-й и n-й элементы, затем 2-й и (n – 1)-й и т. д. до середины массива (элемент с номером i следует обменять с элементом (n + 1 – i)).

```
{ Пример 4.3.}
program mas_five;
type
massiv=array [1..100] of real;
var
x:massiv;
i,n:integer;
b:real;
begin
writeln ('введите размер массива');
readln(n);
for i:=1 to n do
begin
  write ('x[',i,']=');
  readln(x[i]);
end;
{ Элементы массива меняются местами: 1-й - с n-м, 2-й - с (n-1) и т.д. }
for i:=1 to n div 2 do
begin
  b:=x[n+1-i];
  x[n+1-i]:=x[i];
  x[i]:=b;
end;
writeln('преобразованный массив');
for i:=1 to n do
  write(x[i]:1:2,' ');
end.
```

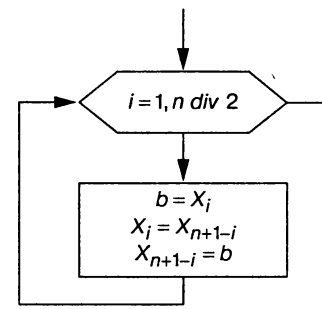


Рис. 4.17 ▾ Фрагмент блок-схемы к примеру 4.3

Пример 4.4. Задан массив из n элементов. Сформировать массивы номеров положительных и отрицательных элементов. Блок-схема представлена на рис. 4.18.

```
{ Пример 4.4. }
program mas_six;
type
massiv=array [1..20] of real;
massiv1=array[1..20] of integer;
```

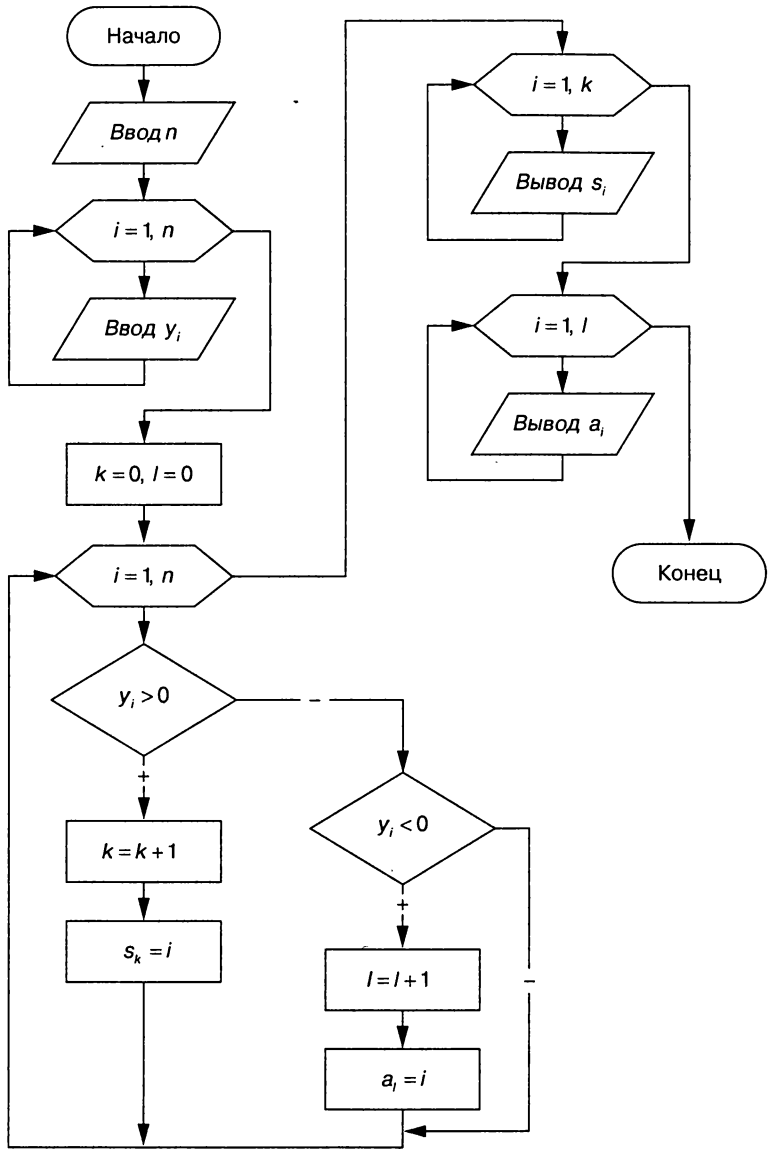


Рис. 4.18 ▾ Алгоритм примера 4.4

```
var
a,s:massiv1;
y: massiv;
i,k,l,n:integer;
begin
write('n=');
readln (n);
for i:=1 to n do
begin
  write('y[',i,']=');
  readln (y[i]);
end;
k:=0;   { Счетчик положительных элементов. }
l:=0;   { Счетчик отрицательных элементов. }
for i:=1 to n do
begin
  if y[i] >0 then   { Если элемент положительный, то }
  begin
     k:=k+1;  { нарастить счетчик k на 1 и }
     s[k]:=i; { записать номер элемента в массив S. }
  end;
  if y[i] <0 then   { Если элемент отрицательный, то }
  begin
     l:=l+1;  { нарастить счетчик l на 1 и }
     a[l]:=i; { записать номер элемента в массив A. }
  end;
end;
writeln('массив индексов положительных элементов');
for i:=1 to k do
  write (s[i],' ');
  writeln;
writeln ('массив индексов отрицательных элементов');
for i:=1 to l do write(a[i],' ');
writeln
end.
```

Пример 4.5. Удалить из массива X, состоящего из n элементов, первые четыре нулевых элемента. Изначально количество нулевых элементов равно нулю (k = 0). Последовательно перебираем все элементы массива. Если встречается нулевой элемент, то количество нулевых элементов увеличиваем на 1 (k = k + 1). Если количество нулевых элементов меньше или равно 4, то удаляем очередной нулевой элемент, иначе аварийно выходим из цикла (встретился пятый нулевой элемент, и дальнейшая обработка массива бесполезна). Блок-схема представлена на рис. 4.19.

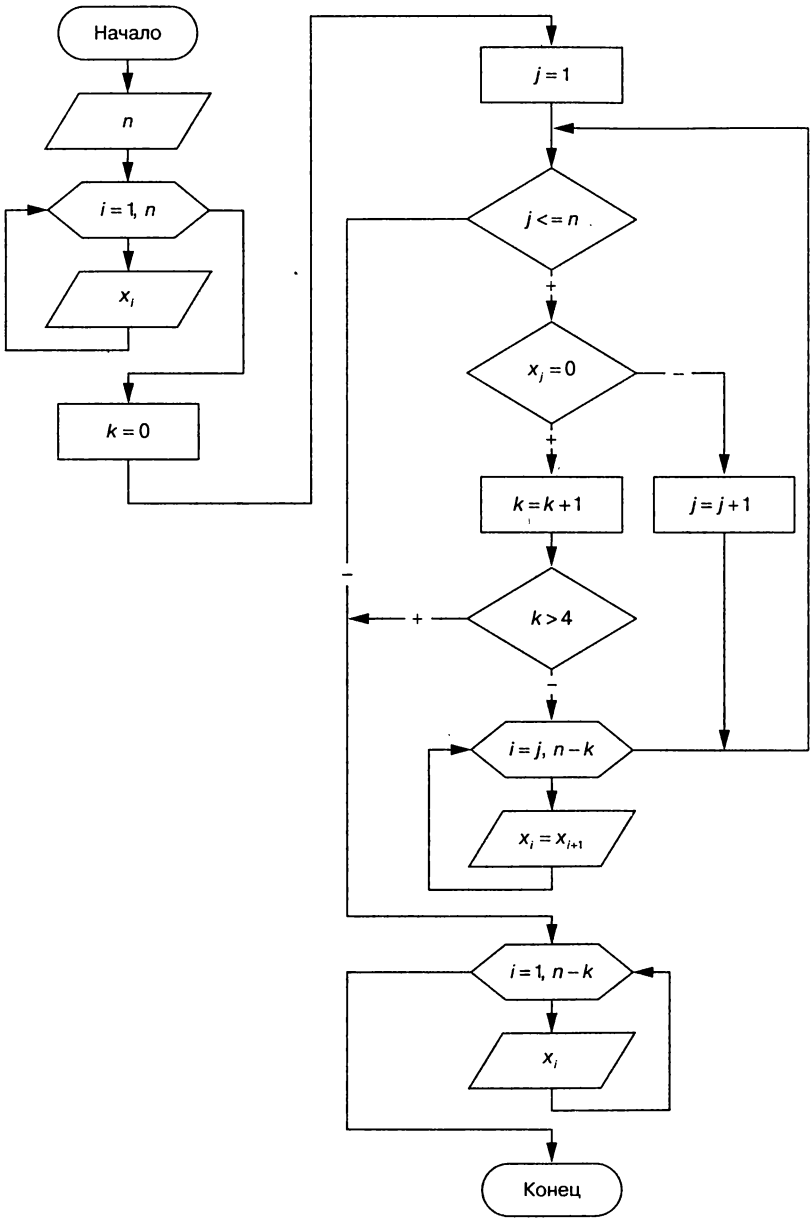


Рис. 4.19 ▾ Алгоритм примера 4.5

```
const n=20;
var X:array [1..n] of byte;
k,i,j:integer;
begin
for i:=1 to n do
  readln(X[i]);
k:=0;   { Количество нулевых элементов. }
j:=1;   { Номер элемента в массиве X. }
while j<=n do { Пока не конец массива. }
begin
  if x[j]=0 then { Если нашли нулевой элемент, то }
  begin
     k:=k+1;  { посчитать его номер. }
     if k>4 then break { Если превышает 4, то выйти из цикла. }
     Else     { Иначе удалить элемент из массива, }
        for i:=j to n-k do   { начиная с j-й позиции. }
           X[i]:=X[i+1];
End
{ Возвращаемся на начало и просматриваем массив с j-й позиции, так как в
ней оказался новый элемент. }
  else j:=j+1;   { Если элемент ненулевой, переходим к следующему. }
end;
{ Вывод на печать измененного массива. }
for i:=1 to n-k do
  write(X[i],' ');
end.
```

Пример 4.6. Массив целых чисел C состоит из N элементов, найти сумму простых чисел, входящих в него. Идея алгоритма состоит в следующем. Сначала сумма равна 0. Последовательно перебираем все элементы, если очередной элемент простой, то добавляем его к сумме. Блок-схема алгоритма изображена на рис. 4.20. Ниже приведен текст программы, реализующей этот алгоритм.

```
var c:array [1..50] of word;
i,j,n:byte; S:word;
Pr:boolean;
begin
write('Введите размерность массива n=');readln(n);
for i:=1 to n do
begin
write('Введите ',i,'-й элемент массива ');
readln(C[i]);
end;
S:=0;
for i:=1 to n do
```

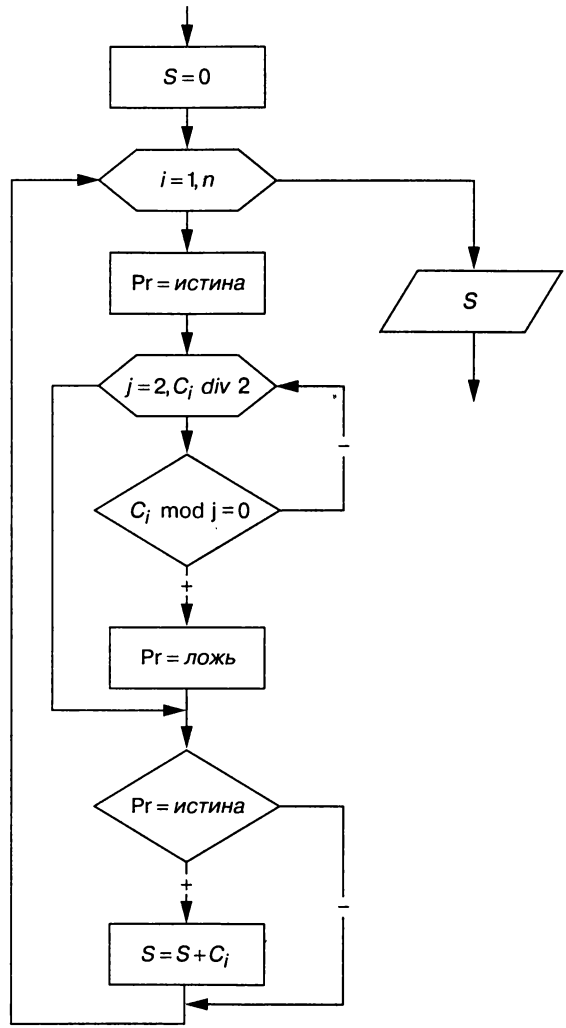


Рис. 4.20 ▾ Алгоритм примера 4.6

```
begin
pr:=true;{ Предположим, что текущее число простое. }
for j:=2 to C[i] div 2 do
{ Если найдется хотя бы один делитель кроме единицы и самого числа, то }
if C[i] mod j =0 then
```

```
begin
{ изменить значение логической переменной }
pr:=false;
{ и досрочно выйти из цикла. }
break;
end;
{ Если число простое, то накапливать сумму. }
if pr then
S:=S+C[i];
{ Иначе перейти к следующему элементу массива. }
end;
writeln('Сумма простых чисел массива S=',S);
end.
```

Пример 4.7. Определить, есть ли в заданном массиве серии элементов, состоящих из знакочередующихся чисел (рис. 4.21). Если есть, то вывести на экран количество таких серий.

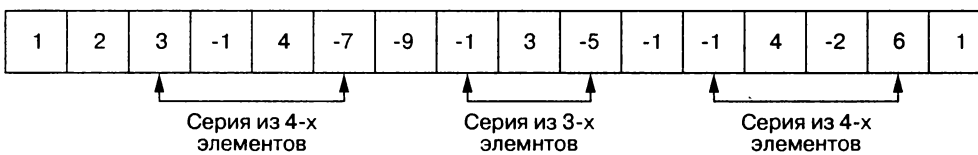


Рис. 4.21 ▾ Иллюстрация к примеру 4.7

На рис. 4.22 изображена блок-схема алгоритма для решения поставленной задачи. Здесь переменная k – количество элементов, попадающих в серию, kol – количество знакочередующихся серий в массиве.

```
const n=15;
var x:array[1..n] of real;
i,j,k,kol:integer;
begin
for i:=1 to n do
read(x[i]);
{ Так как минимальная серия состоит из двух элементов, }
{ k присвоим значение 1. }
k:=1; { Длина серии. }
kol:=0; { Количество серий в массиве. }
for i:=1 to n-1 do
{ Если при умножении двух соседних элементов }
{ результат - отрицательное число, то элементы имеют разный знак. }
if x[i]*x[i+1]<0 then
k:=k+1 { Показатель продолжения серии. }
```

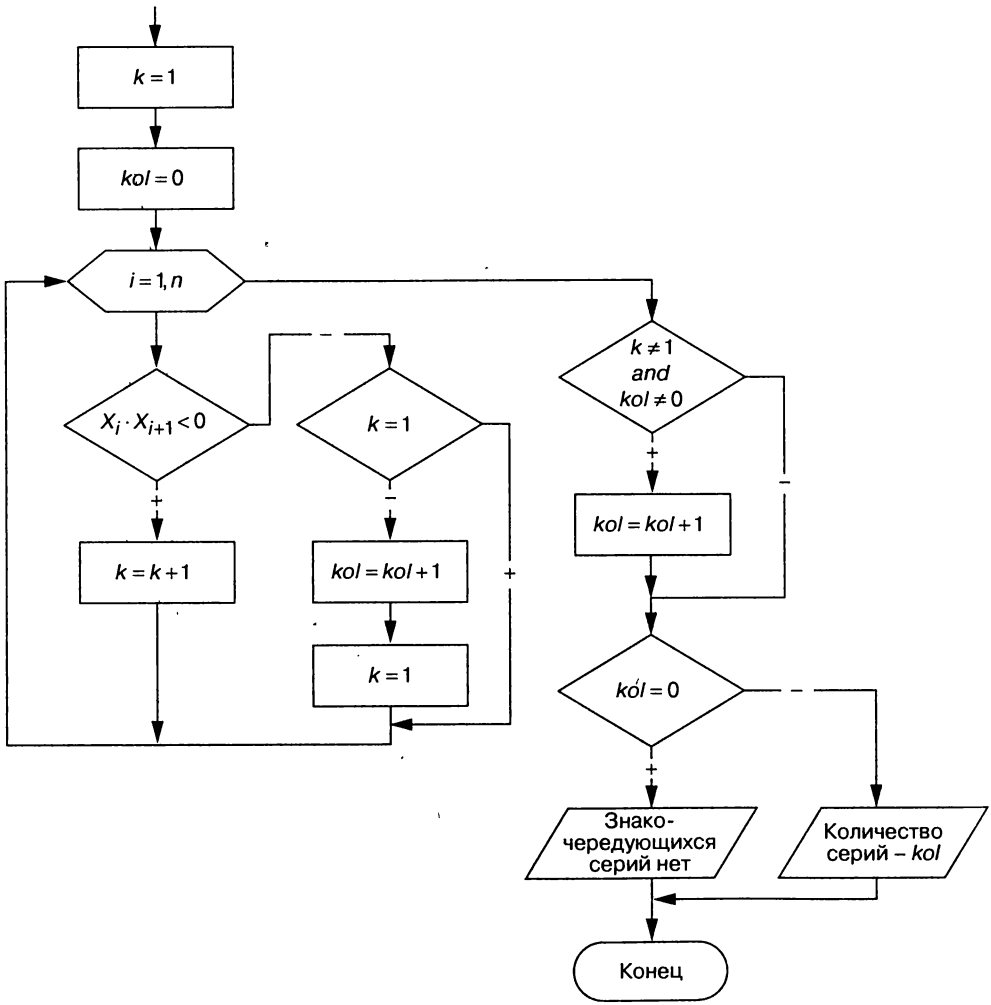


Рис. 4.22 ▾ Блок-схема алгоритма для решения задачи примера 4.7

```
else
{ Если серия разорвалась, }
{ то увеличить счетчик подсчета количества серий. }
if k<>1 then
begin
kol:=kol+1;
```

```
{ Подготовить показатель продолжения серии }
{ к возможному появлению следующей серии. }
k:=1;
end;
{ Проверка, не было ли серии в конце массива. }
if (k<>1) and (kol<>0)then
{ Если да, увеличить счетчик еще на единицу. }
kol:=kol+1;
if kol=0 then
write('Знакочередующихся серий нет')
else
write('Количество знакочередующихся серий',kol);
end.
```

Пример 4.8. В заданном массиве найти самую длинную серию элементов, состоящую из единиц.

Эта задача похожа на предыдущую, поэтому предлагаем читателю самостоятельно разобраться в ее решении, ознакомившись с приведенным далее текстом программы.

```
const n=15;
var x:array[1..n] of real;
i,k,max:integer;
Fl:boolean;
begin
for i:=1 to n do
read(x[i]);
k:=1; Fl:=true;
for i:=1 to n-1 do
if (x[i]=1) and (x[i+1]=1) then
k:=k+1
else if k<>1 then
begin
if fl then
begin
max:=k;
Fl:=false;
end
else if k>max then
max:=k;
k:=1;
end;
if (k<>1) and (k>max) then
max:=k;
write(max);
end.
```

4.10. Упражнения по теме «Массивы»

1. Записать положительные элементы массива $X = (x_1, x_2, ..., x_n)$ подряд в массив $Y = (y_1, y_2, ..., y_k)$. Определить k (количество положительных элементов). Вычислить сумму элементов массива X и произведение элементов массива Y.
2. Сформировать массив $B = (b_1, b_2, ..., b_k)$, записав в него элементы массива $A = (a_1, a_2, ..., a_n)$ с четными индексами. Вычислить среднее арифметическое элементов массива B и удалить из него пятый элемент.
3. Дан массив $X = (x_1, ..., x_n)$. Переписать пять первых положительных элементов массива подряд в массив $Y = (y_1, y_2, ..., y_5)$. Найти максимальный элемент массива X.
4. Записать элементы массива $X = (x_1, x_2, ..., x_n)$, удовлетворяющие условию $x_i \in [1, 2]$, подряд в массив $Y = (y_1, y_2, ..., y_k)$. Определить минимальный элемент массива X.
5. Переписать элементы массива целых чисел $X = (x_1, x_2, ..., x_n)$ в обратном порядке в массив $Y = (y_1, y_2, ..., y_n)$. Вычислить количество четных, нечетных и нулевых элементов массива Y.
6. Определить максимальный и минимальный элементы среди положительных нечетных элементов целочисленного массива $X = (x_1, x_2, ..., x_n)$. Удалить из массива все нулевые элементы.
7. Переписать элементы массива $X = (x_1, x_2, ..., x_{12})$ в массив $Y = (y_1, y_2, ..., y_{12})$, сдвинув элементы массива X вправо на три позиции. При этом три элемента с конца массива X перемещаются в начало: $(y_1, y_2, ..., y_{12}) = (x_{10}, x_{11}, x_{12}, x_1, x_2, ..., x_9)$. Определить номера максимального и минимального элементов в массивах X и Y.
8. Записать элементы массива $X = (x_1, x_2, ..., x_{15})$ в массив $Y = (y_1, y_2, ..., y_{15})$, сдвинув элементы массива X влево на четыре позиции. При этом четыре элемента, стоящие в начале массива X, перемещаются в конец: $(y_1, y_2, ..., y_{15}) = (x_5, x_6, ..., x_{15}, x_1, x_2, x_3, x_4)$. Поменять местами минимальный и максимальный элементы массива Y.
9. В массиве $X = (x_1, x_2, ..., x_n)$ определить количество элементов, меньших среднего арифметического значения. Не упорядочивая массив, удалить из него элементы, расположенные между максимальным и минимальным.
10. Вычислить среднее арифметическое элементов массива $X = (x_1, x_2, ..., x_n)$, расположенных между его минимальным и максимальным значениями. Если минимальный элемент размещается в массиве раньше максимального, то упорядочить массив на данном промежутке по возрастанию его элементов (возможна и обратная ситуация).

11. Определить, содержит ли заданный массив группы элементов, расположенных в порядке возрастания их значений. Если да, то определить количество таких групп.
12. В заданном массиве целых чисел найти самую маленькую серию подряд стоящих нечетных элементов.
13. Удалить из массива целых чисел все элементы, являющиеся простыми числами.
14. Удалить из массива последнюю группу элементов, представляющих собой знакочередующийся ряд.
15. Преобразовать заданный массив целых положительных чисел таким образом, чтобы цифры каждого его элемента были записаны в обратном порядке. Определить количество простых чисел в массиве до и после преобразования.

Глава 5 Обработка матриц в Турбо Паскале

Матрица – это двумерный массив, каждый элемент которого имеет два индекса: номер строки `i` и номер столбца `j`. Поэтому для работы с элементами матрицы необходимо использовать два цикла. Если значениями параметра первого цикла будут номера строк матрицы, то значениями параметра второго – столбцы (или наоборот). Обработка матрицы заключается в том, что вначале поочередно рассматриваются элементы первой строки (столбца), затем второй и т.д. до последней. Рассмотрим основные операции, выполняемые над матрицами при решении задач.

5.1. Ввод-вывод матриц

Матрицы, как и массивы, нужно вводить (выводить) поэлементно. Блок-схема ввода элементов матрицы представлена на рис. 5.1.

Вывод можно осуществлять по строкам или по столбцам, но лучше, если элементы располагаются построчно. Например:

```
 6  -9   7  13
 5   8   3   8
 3   7  88  33
55  77  88  37
```

Алгоритм построчного вывода элементов матрицы представлен на рис. 5.2.

Ниже приведен пример программы ввода-вывода матрицы.

```
var
  a: array [1..20,1..20] of real;
  i,n,m: integer;
begin
  readln(N,M);
{ Ввод элементов матрицы. }
  for i:=1 to N do
     for j:=1 to m do
     begin
        write('A(',i,',',j,')=');
        readln(A[i,j])
     end;
{ Вывод элементов матрицы. }
  writeln ( 'матрица A ' );
  for i:=1 to n do
  begin
     for j:=1 to m do
        write(a[i,j]:8:3, ' '); { Печатается строка. }
     Writeln     { Переход на новую строку. }
  end;
end.
```

Для организации построчного ввода матрицы в двойном цикле по строкам и столбцам можно использовать оператор read.

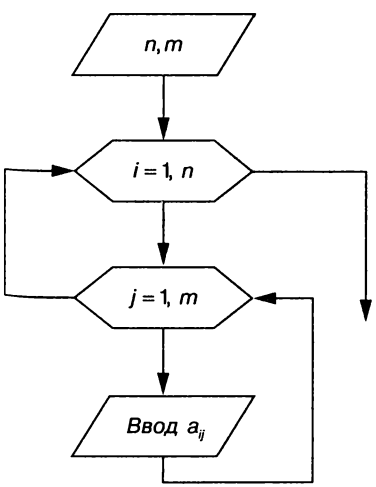


Рис. 5.1 ▾ Ввод элементов матрицы

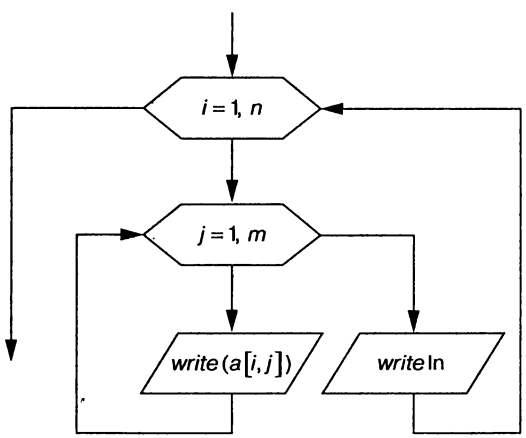


Рис. 5.2 ▾ Блок-схема вывода элементов матрицы по строкам

```
for i:=1 to N do
  for j:=1 to m do
     read(A[i,j]);
```

В этом случае элементы каждой строки матрицы можно разделять символами пробела или табуляции и только в конце строки нажимать клавишу **Enter**.

Рассмотрим несколько задач обработки матриц. Для их решения предварительно напомним читателю некоторые свойства матриц (рис. 5.3):

- если номер строки элемента совпадает с номером столбца (`i = j`), это означает, что элемент лежит на главной диагонали матрицы;
- элемент находится ниже главной диагонали, если номер строки превышает номер столбца (`i > j`);
- если номер столбца больше номера строки (`i < j`), то элемент находится выше главной диагонали;
- элемент лежит на побочной диагонали, если его индексы удовлетворяют равенству `i + j – 1 = n`;
- неравенство `i + j – 1 < n` характерно для элемента, находящегося выше побочной диагонали;
- соответственно, элементу, лежащему ниже побочной диагонали, соответствует выражение `i + j – 1 > n`.

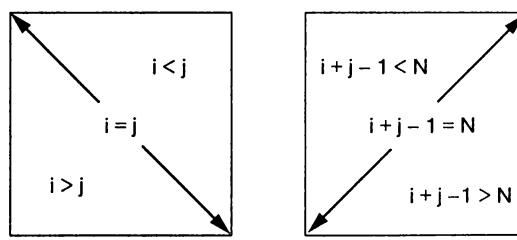


Рис. 5.3 ▼ Свойства элементов матрицы

5.2. Алгоритмы и программы работы с матрицами

Пример 5.1. Найти сумму элементов матрицы, лежащих выше главной диагонали (рис. 5.4).

Алгоритм решения данной задачи (рис. 5.5) построен следующим образом: обнуляется ячейка для накапливания суммы (переменная `s`). Затем с помощью

двух циклов (первый по строкам, второй по столбцам) просматривается каждый элемент матрицы, но суммирование происходит только в том случае, если этот элемент находится выше главной диагонали, то есть выполняется свойство i < j.

```
program one;
var
  a:array [1..15,1..10] of real;
  i,j,n,m: integer;
  s: real;
begin
  writeln('введите размеры матрицы');
  writeln('n - количество строк, m - количество столбцов');
  readln (n,m);
  for i:=1 to n do
     for j:=1 to m do
     begin
        write('a[',i,',',j,']=');
```

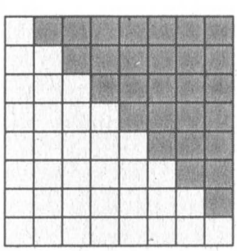


Рис. 5.4 ▾ Рисунок к условию задачи из примера 5.1

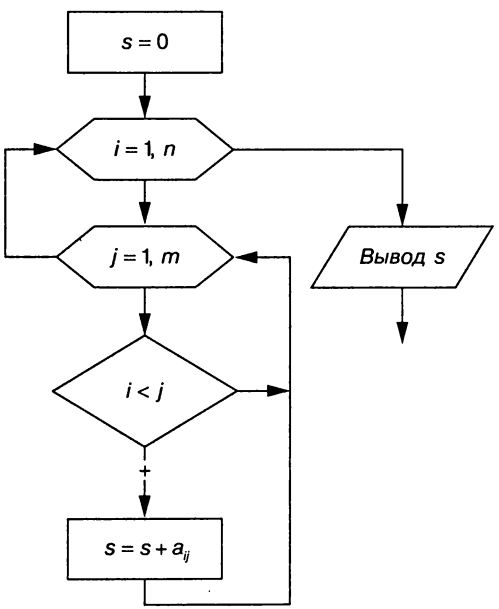


Рис. 5.5 ▾ Блок-схема примера 5.1 (алгоритм 1)

```
        readln(a[i,j]);
    end;
  s:=0;
  for i:=1 to n do
    for j:=1 to n do
      if j>i then { Если элемент лежит выше главной диагонали, то }
      s:=s+a[i,j];{ наращиваем сумму. }
    writeln('матрица A');
  for i:=1 to n do
  begin
    for j:=1 to m do
      { Здесь важен формат, особенно общая ширина поля! }
      write(a[i,j]:8:3,' ');
      writeln
    end;
  writeln('сумма элементов матрицы', s:8:3);
end.
```

На рис. 5.6 представлен еще один вариант решения данной задачи. В нем проверка условия `i < j` не выполняется, однако здесь также суммируются элементы матрицы, находящиеся выше главной диагонали. Для того чтобы понять, как работает этот алгоритм, вернемся к рис. 5.4. В первой строке заданной матрицы необходимо сложить все элементы, начиная со второго. Во второй – все, начиная с третьего, в `i`-й строке процесс начнется с `(i + 1)`-го элемента и т.д. Таким образом, первый цикл работает от 1 до `N`, а второй от `i + 1` до `M`. Предлагаем читателю самостоятельно составить программу, соответствующую описанному алгоритму.

Пример 5.2. Вычислить количество положительных элементов квадратной матрицы, расположенных по ее периметру и на диагоналях. Напомним, что в данной матрице число строк равно числу столбцов.

Прежде чем преступить к решению задачи, обратимся к рис. 5.7, на котором изображена схема квадратных матриц различной размерности. Из условия задачи понятно, что рассматривать все элементы заданной матрицы не нужно. Достаточно просмотреть первую и последнюю строки, первый и последний столбцы, а также диагонали. Все эти элементы отмечены на схеме, причем черным цветом выделены те, обращение к которым может произойти дважды. Например, элемент с номером (1,1) принадлежит как к первой строке, так и к первому столбцу, а элемент с номером (N,N) находится в последней строке и последнем столбце одновременно. Кроме того, если N – число нечетное (на рис. 5.7 эта матрица расположена слева), то существует элемент с номером (N div 2 + 1, N div 2 + 1), который находится на пересечении главной и побочной

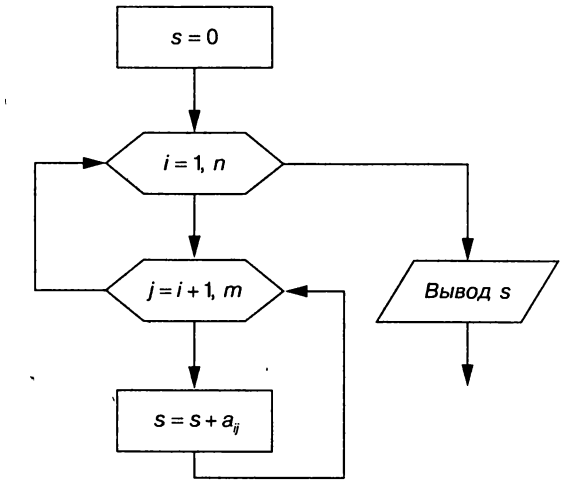
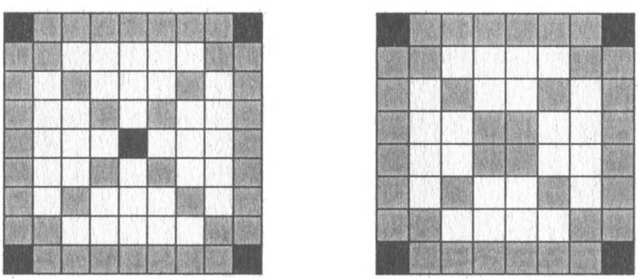


Рис. 5.6 ▾ Блок-схема примера 5.1 (алгоритм 2)

Матрица из *N* строк и *N* столбцов



N – нечетное *N* – четное

Рис. 5.7 ▾ Рисунок к условию задачи из примера 5.2

диагоналей. При нечетном значении *N* (матрица справа на рис. 5.7) диагонали не пересекаются.

Итак, разобрав подробно постановку задачи, рассмотрим алгоритм ее решения. Для обращения к элементам главной диагонали вспомним, что номера их строк всегда равны номерам столбцов. Поэтому, если параметр `i` изменяется

циклически от 1 до N, то $A_{i,i}$ – элемент главной диагонали. Воспользовавшись свойством, характерным для элементов побочной диагонали, получим:

$i + j - 1 = n \rightarrow j = n - i + 1$,

следовательно, для i = 1,2, ..., n элемент $A_{i,n-i+1}$ – элемент побочной диагонали. Элементы, находящиеся по периметру матрицы, записываются следующим образом: $A_{1,i}$ – первая строка, $A_{N,i}$ – последняя строка, и соответственно $A_{i,1}$ – первый столбец, $A_{i,N}$ – последний столбец.

Блок-схема описанного алгоритма изображена на рис. 5.8. В блоке 1 организуется цикл для обращения к диагональным элементам матрицы. Причем в блоках 2–3 подсчитывается количество положительных элементов на главной диагонали, а в блоках 5–6 – на побочной. Цикл в блоке 6 задает изменение параметра i от 2 до $N - 1$. Это необходимо для того, чтобы не обращаться к элементам, которые уже были рассмотрены: $A_{1,1}$, $A_{1,N}$, $A_{N,1}$ и $A_{N,N}$. Блоки 7–8 подсчитывают положительные элементы в первой строке, 9 и 10 – в последней; 11 и 12 – в первом столбце, а 13 и 14 – в последнем. Блок 15 проверяет, не был ли элемент, находящийся на пересечении диагоналей, подсчитан дважды. Напомним, что это могло произойти только в том случае, если N является нечетным числом и этот элемент был положительным. Эти условия и проверяются в блоке 16, который уменьшает вычисленное количество положительных элементов на единицу.

```
var  a:array [1..10,1..10] of integer;
  i,j,n,k:integer;
begin
write('n=');readln(n);
for i:=1 to n do
  for j:=1 to n do
     readln(a[i,j]);
writeln('Была введена матрица:');
for i:=1 to n do
begin
  for j:=1 to n do
     write(a[i,j,' ']);
     writeln;
end;
k:=0;
for i:=1 to n do
begin
  if (a[i,i]>0) then   { Элемент лежит на главной диагонали. }
     k:=k+1;
  if a[i,n-i+1]>0 then { Элемент лежит на побочной диагонали. }
     k:=k+1;
end;
```

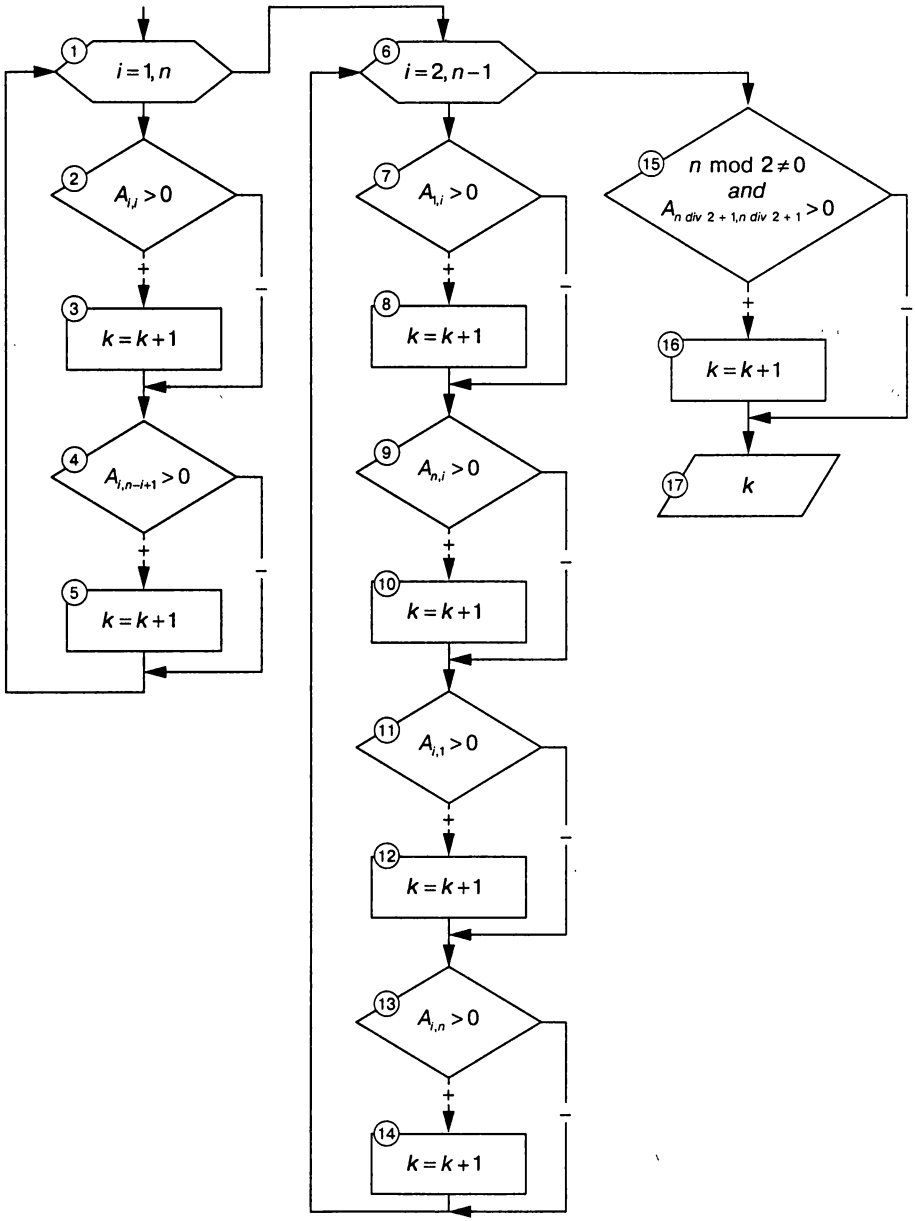


Рис. 5.8 ▾ Блок-схема алгоритма примера 5.2

```
for i:=2 to n-1 do
begin
  if (a[1,i]>0) then   { Элемент находится в первой строке. }
     k:=k+1;
  if (a[n,i]>0) then   { Элемент находится в последней строке. }
     k:=k+1;
  if (a[i,1]>0) then   { Элемент находится в первом столбце. }
     k:=k+1;
  if (a[i,n]>0) then   { Элемент находится в последнем столбце. }
     k:=k+1;
end;
{ Если элемент, находящийся на пересечении диагоналей, подсчитан дважды, }
{ то уменьшить вычисленное значение k на один. }
if (n mod 2 <>0) and (a[(n div 2)+1,(n div 2)+1]>0) then k:=k-1;
write(k);
end.
```

Пример 5.3. Проверить, является ли заданная квадратная матрица единичной. Напомним, что единичной называют матрицу, у которой элементы главной диагонали – единицы, а все остальные – нули. Например:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Решать задачу будем следующим образом. Предположим, что матрица единичная, и попытаемся доказать обратное. Если окажется, что хотя бы один диагональный элемент не равен единице или любой из элементов вне диагонали не равен нулю, то матрица единичной не является. Воспользовавшись логическими операциями языка, все эти условия можно соединить в одно и составить программу:

```
var a:array[1..10,1..10] of integer;
i,j,n:integer;
Fl:boolean;
begin
  readln(n);
  for i:=1 to n do
     for j:=1 to n do
        read(a[i,j]);
```

```
{ Предположим, что матрица единичная, }
{ и присвоим логической переменной значение «истина». }
{ Если значение этой переменной при выходе из цикла не изменится, }
{ это будет означать, что матрица действительно единичная. }
  Fl:=true;
  for i:=1 to n do
     for j:=1 to n do
     if ((i=j) and (a[i,j]<>1)) or ((i<>j) and (a[i,j]<>0)) then
{ Если элемент лежит на главной диагонали и не равен единице или }
{ элемент лежит вне главной диагонали и не равен нулю, то }
  begin
{ присвоим логической переменной значение «ложь». }
{ Это будет означать, что матрица единичной не является. }
     Fl:=false;
{ Выйти из цикла. }
     break;
  end;
{ Проверка значения логической переменной и печать результата. }
  if fl then
     writeln('Матрица единичная')
  else writeln('Матрица не является единичной');
end.
```

Пример 5.4. Преобразовать исходную матрицу так, чтобы первый элемент каждой строки был заменен средним арифметическим элементов этой строки.

Для решения данной задачи необходимо найти в каждой строке сумму элементов, а затем разделить ее на их количество. Полученный результат записать в первый элемент соответствующей строки. Блок-схема алгоритма решения представлена на рис. 5.9.

```
program two;
type
  matrica=array[1..15,1..15] of real;
var
  a:matrica;
  i,j,n,m: integer;
  s: real;
begin
  write('Введите n и m:');
  readln(n,m);
  for i:=1 to n do
     for j:=1 to m do
     begin
        write ('a[',i,',',j,']=');
        read (a[i,j]);
     end;
```

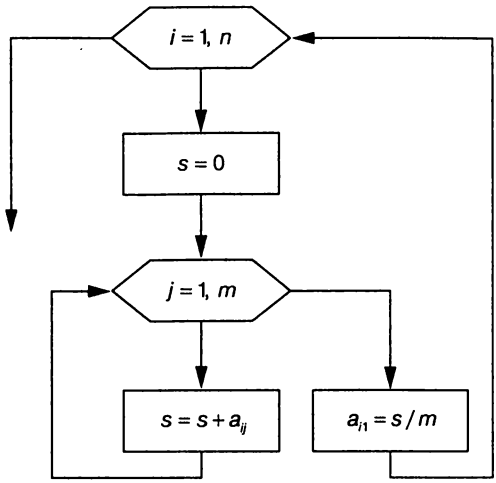


Рис. 5.9 ▼ Блок-схема алгоритма примера 6.2

```
  writeln ('Исходная матрица:');
  for i:=1 to n do
  begin
     for j:=1 to m do
        write (a[i,j]:8:3,' ');
     writeln
  end;
  for i:=1 to n do
  begin
     s:=0;
        for j:=1 to m do
           s:=s+a[i,j];   { Вычисление суммы элементов строки. }
           a[i,1]:=s/m;   { Запись среднего элемента в первый элемент
                            строки. }
  end;
  writeln ('Преобразованная матрица:');
  for i:=1 to n do
  begin
     for j:=1 to m do
        write (a[i,j]:8:3,' ');
        writeln
  end;
end.
```

Пример 5.5. Задана матрица $A(n,m)$. Сформировать вектор $P(m)$, в который записать номера строк максимальных элементов каждого столбца. Алгоритм

решения этой задачи следующий: для каждого столбца матрицы находим максимальный элемент и его номер, номер максимального элемента j-го столбца матрицы записываем в j-й элемент массива P. Блок-схема алгоритма приведена на рис. 5.10.

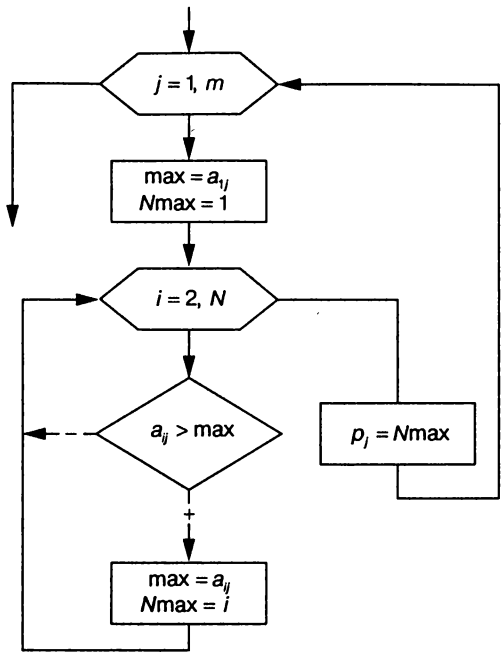


Рис. 5.10 ▾ Блок-схема алгоритма примера 5.5

```
program three;
  type
  massiv = array [1..15,1..15] of real;
var
  i,j,n,nmax,m: byte;
  max: real;
  a: massiv;
  p: array [1..15] of byte;
begin
  write('n=');
  readln(n);
  write('m=');
  readln (m);
```

```
  for i:=1 to n do
     for j:=1 to m do
     begin
        write('a[',i,',',j,']=');
        readln(a[i,j]);
     end;
     for j:=1 to m do  { Цикл по столбцам. }
     begin
     max:= a[1,j];
     nmax:=1;
     for i:=2 to n do  { Цикл по строкам. }
        if a[i,j] > max then
        begin
           max:=a[i,j];
           nmax:=i;
        end;
     p[j]:=nmax; { Запись номера максимального элемента в массив. }
     end;
  writeln('матрица A');
  for i:=1 to n do
  begin
     for j:=1 to m do
        write(a[i,j]:8:3,' ');
     writeln
  end;
  writeln('вектор P');
  for j:=1 to m do
     write (p[i]:2,' ');
  writeln;
end.
```

Пример 5.6. Написать программу умножения двух матриц $A(n,m)$ и $B(m,l)$. Например, необходимо перемножить две матрицы:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{bb} \\ b_{31} & b_{32} \end{pmatrix}.$$

Воспользовавшись правилом «строка на столбец», получим матрицу

$$\begin{pmatrix} a_{11}\cdot b_{11}+a_{12}\cdot b_{21}+a_{13}\cdot b_{31} & a_{11}\cdot b_{12}+a_{12}\cdot b_{22}+a_{13}\cdot b_{32} \\ a_{21}\cdot b_{11}+a_{22}\cdot b_{21}+a_{23}\cdot b_{31} & a_{21}\cdot b_{12}+a_{22}\cdot b_{22}+a_{23}\cdot b_{32} \\ a_{31}\cdot b_{11}+a_{32}\cdot b_{21}+a_{33}\cdot b_{31} & a_{31}\cdot b_{12}+a_{32}\cdot b_{22}+a_{33}\cdot b_{32} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{pmatrix}.$$

В общем виде формула для нахождения элемента C_{ij} матрицы выглядит следующим образом:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj},$$

где $i = 1,N$ и $j = 1,L$. Обратите внимание, что проводить операцию умножения можно только в том случае, если количество строк левой матрицы совпадает с количеством столбцов правой. Кроме того, $A \times B \neq B \times A$. Блок-схема, изображенная на рис. 5.11, реализует расчет каждого элемента матрицы C в виде суммы по вышеприведенной формуле.

```
program four;
type
  matrica=array [1..15,1..15] of real;
```

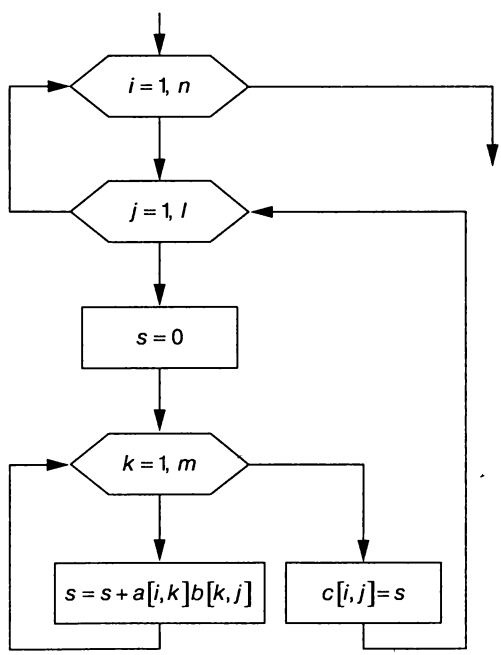


Рис. 5.11 ▾ Алгоритм умножения двух матриц

```
var
  a,b,c:matrica;
  i,j,m,n,l,k:byte;
  s:real;
begin
  writeln('введите n,m и l');
  readln(n, m, l);
  for i:=1 to n do
     for j:=1 to m do
     begin
        write('a[',i,',',j,']=');
        readln(a[i,j]);
     end;
  for i:=1 to m do
     for j:=1 to l do
     begin
        write('b[',i,',',j,']=');
        readln(b[i,j]);
     end;
  for i:=1 to n do
     for j:=1 to l do
     begin
{ В переменной S будет храниться результат скалярного }
{ произведения i-й строки на j-й столбец. }
        S:=0;
        for k:=1 to m do
           s:=s+a[i,k]*b[k,j];
        c[i,j]:=s;
     end;
     writeln('матрица a');
  for i:=1 to n do
  begin
     for j:=1 to m do
        write(a[i,j]:7:3,' ');
     writeln;
  end;
  writeln('матрица b');
  for i:=1 to m do
  begin
     for j:=1 to l do
        write(b[i,j]:7:3,' ');
     writeln;
  end;
  writeln('матрица c=a*b');
  for i:=1 to n do
  begin
```

```
    for j:=1 to l do
       write(c[i,j]:7:3,' ');
    writeln;
  end;
end.
```

Пример 5.7. Преобразовать матрицу $A(m,n)$ таким образом, чтобы каждый столбец был упорядочен по убыванию. Алгоритм решения этой задачи сводится к тому, что уже известный нам по предыдущей главе алгоритм упорядочивания элементов в массиве выполняется для каждого столбца матрицы. Блок-схема представлена на рис. 5.12.

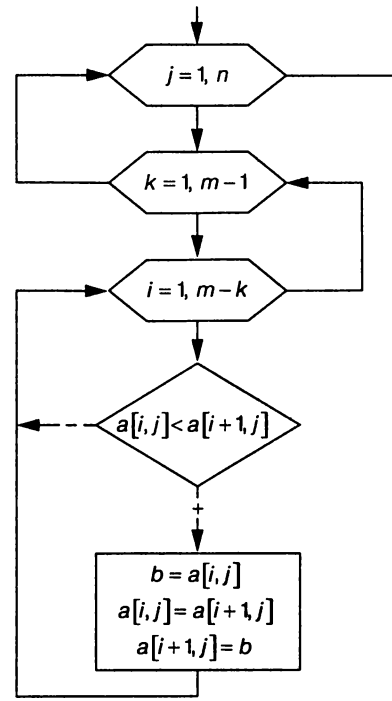


Рис.5.12 ▾ Блок-схема алгоритма примера 5.7

```
program five;
type
  matrica=array [1..15,1..15] of real;
var
  a:matrica;
  i,j,k,m,n:byte;
  b:real;
begin
  write ('m=');
  readln(m);
  write ('n=');
  readln(n);
  for i:=1 to m do
     for j:=1 to n do
     begin
        write ('a[',i,',',j,']=');
        readln(a[i,j]);
     end;
  writeln ('матрица a');
  for i:=1 to m do
  begin
     for j:=1 to n do
        write (a[i,j]:7:3,' ');
     writeln
  end;
  for j:=1 to n do     { j - номер столбца. }
     for k:=1 to m-1 do  { k - номер просмотра. }
        for i:=1 to m-k do  { i - номер строки. }
           if a[i,j] < a[i+1,j] then
           begin
```

```
            b:=a[i,j];
            a[i,j]:=a[i+1,j];
            a[i+1,j]:=b;
         end;
         writeln(' преобразованная матрица a');
   for i:=1 to m do
   begin
   for j:=1 to n do
   write (a[i,j]:7:3,' ');
  writeln;
  end;
end.
```

Пример 5.8. Преобразовать матрицу $A(m,n)$ так, чтобы строки с нечетными индексами были упорядочены по убыванию, с четными – по возрастанию. Блок-схема представлена на рис. 5.13.

```
program six;
var
  a:array [1..15,1..15] of real;
  j,i,k,m,n:byte;
  b:real;
begin
  writeln('введите m и n');
  readln(m,n);
  for i:=1 to m do
     for j:=1 to n do
     begin
        write('a[',i,',',j,']=');
        readln(a[i,j]);
     end;
  writeln ('матрица a');
  for i:=1 to m do
  begin
     for j:=1 to n do
        write(a[i,j]:7:3,' ');
     writeln
  end;
  for i:=1 to m do
     if (i mod 2)=0 then  { Если номер строки четный, то }
     begin       { упорядочить ее элементы по возрастанию. }
        for k:=1 to n-1 do
           for j:=1 to n-k do
           if a[i,j] > a[i,j+1] then
           begin
              b:=a[i,j];
              a[i,j]:=a[i,j+1];
```

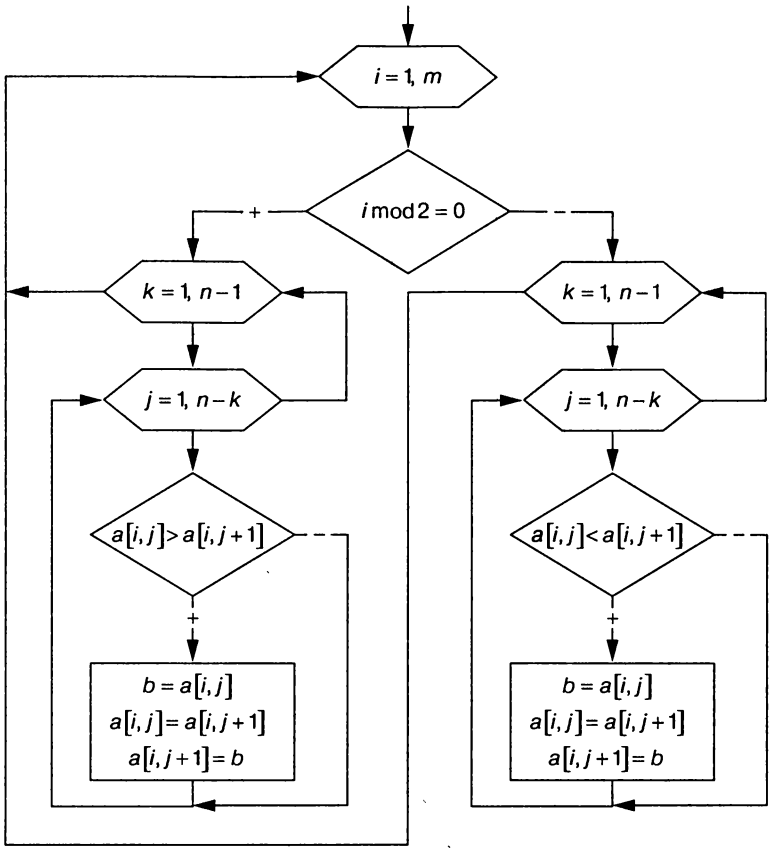


Рис. 5.13 ▾ Блок-схема алгоритма примера 5.8

```
            a[i,j+1]:=b;
         end;
   end
else        { Если номер строки нечетный, то }
for k:=1 to n-1 do   { упорядочить ее элементы по убыванию. }
   for j:=1 to n-k do
   if a[i,j] < a[i,j+1] then
   begin
      b:=a[i,j];
      a[i,j]:=a[i,j+1];
      a[i,j+1]:=b;
   end;
writeln('преобразованная матрица a');
```

```
  for i:=1 to m do
  begin
     for j:=1 to n do
        write (a[i,j]:7:3,' ');
     writeln
  end
end.
```

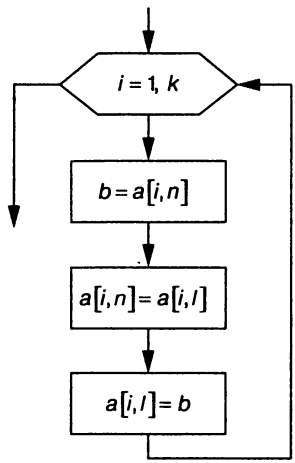


Рис. 5.14 ▼ Блок-схема алгоритма примера 5.9

Пример 5.9. Поменять местами n-й и l-й столбцы матрицы $A(k,m)$. Блок-схема приведена на рис. 5.14.

```
program seven;
type
  matrica=array [1..15,1..15] of real;
var
  a:matrica;
  i,j,k,m,n,l:byte;
  b:real;
begin
  write ('k=');
  readln(k);
  write ('m=');
  readln(m);
  for i:=1 to k do
     for j:=1 to m do
```

```
    begin
      write ('a[',i,',',j,']=');
      readln (a[i,j])
    end;
  repeat
    write('n=');
    readln(n);
    write('l=');
    readln(l);
{ Ввод считается верным, если n и l меньше m и не равны друг другу. }
  until (n<=m) and (l<=m) and (n< >l);
{ Элементы столбца с номером l заменить элементами столбца с номером n. }
  for i:=1 to k do
  begin
    b:=a[i,n];
    a[i,n]:=a[i,l];
    a[i,l]:=b
  end;
  for i:=1 to k do
  begin
    for j:=1 to m do
      write(a[i,j]:7:3,' ');
    writeln;
  end;
end.
```

5.3. Упражнения по теме «Работа с матрицами»

Разработать блок-схему и написать программу на Турбо Паскале:

1. Определить номера строки и столбца максимального элемента прямоугольной матрицы $A(n,m)$. Подсчитать количество нулевых элементов матрицы и напечатать их индексы.
2. Найти сумму элементов квадратной матрицы $X(n,n)$, находящихся по ее периметру и на диагоналях.
3. Сформировать вектор $D = (d_1, d_2, ..., d_k)$, каждый элемент которого представляет собой среднее арифметическое значение элементов строк матрицы $C(k,m)$, и вектор $G = (g_1, g_2, ..., g_m)$ – любой его компонент должен быть равен произведению элементов соответствующего столбца матрицы $C(k,m)$.

4. Задана матрица $A(n,m)$, в каждом столбце которой минимальный элемент необходимо заменить суммой положительных элементов этого же столбца.
5. Задана матрица $A(n,n)$. Определить максимальный элемент среди элементов матрицы, расположенных выше главной диагонали, и минимальный элемент среди тех, которые находятся ниже главной диагонали. Отсортировать каждый столбец матрицы по возрастанию.
6. Заменить строку матрицы $P(n,m)$ с максимальной суммой элементов на первую строку поэлементно.
7. Переместить максимальный элемент матрицы $F(k,p)$ в правый верхний угол, а минимальный элемент – в левый нижний.
8. Проверить, является ли матрица $A(n,n)$ диагональной (все элементы являются нулями, кроме главной диагонали), единичной (все элементы являются нулями; на главной диагонали только единицы) или нулевой (все элементы – нули).
9. Сформировать из некоторой матрицы $A(n,n)$ верхнетреугольную матрицу $B(n,n)$ (все элементы ниже главной диагонали нулевые), нижнетреугольную матрицу $C(n,n)$ (все элементы выше главной диагонали нулевые) и диагональную матрицу (все элементы – нули, кроме главной диагонали) $D(n,n)$.
10. Заданы матрицы $A(M,N)$ и $B(L,P)$. Найти, если возможно, матрицы $C = A \cdot B$ и $D = A + B$.
11. Выполнив действия над матрицами $A(n,n)$ и $B(n,n)$, вычислить матрицу $C(n,n)$ по формуле: $C = (A - B^T)(3A^T + B/2)$, где A^T и B^T – транспонированные матрицы, то есть те, которые получают из исходных путем замены строк на столбцы.
12. Проверить, выполняется ли для двух матриц $A(n,n)$ и $B(n,n)$ условие: $A \cdot B = E$, где E – единичная матрица. Учесть, что матрицы A и B необходимо ввести, а матрицу E сформировать в программе.
13. Вычислить произведение ненулевых элементов матрицы, выделенных на рис. 5.15 темным цветом. Учесть, что матрица квадратная, а размерность ее может иметь как четное, так и нечетное значение.
14. Вычислить минимальный элемент среди элементов белого цвета на рис. 5.15. Учесть, что матрица квадратная, а размерность ее может иметь как четное, так и нечетное значение.
15. Определить количество простых чисел, расположенных вне диагоналей матрицы $B(n,n)$.

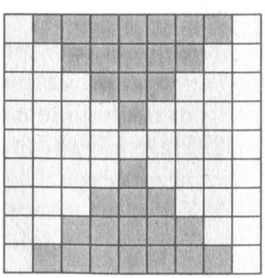


Рис. 5.15 ▼ Матрица для упражнений 13, 14

Подпрограммы в языке Турбо Паскаль

В данной главе читатель познакомится с подпрограммами. Первые два параграфа посвящены использованию процедур. В третьем описана работа с функциями. Новые возможности работы с подпрограммами, появившиеся в Турбо Паскале 7.0, рассмотрены в четвертом параграфе. Последние два параграфа познакомят читателя с процедурными типами и рекурсивными подпрограммами.

В практике программирования часто возникают ситуации, когда одну и ту же группу операторов, реализующих определенную цель, требуется повторить без изменений в нескольких местах программы. Для избавления от столь нерациональной траты времени была предложена концепция подпрограммы.

Подпрограмма – именованная, логически законченная группа операторов языка, которую можно вызвать для выполнения любое количество раз из различных мест программы. В языке Паскаль существуют два вида подпрограмм: *процедуры* и *функции*.

Главное отличие процедур от функций заключается в том, что результатом последних является одно единственное значение. Все процедуры и функции языка Паскаль можно разделить на два класса:

- стандартные процедуры и функции языка Паскаль;
- процедуры и функции, определенные пользователем.

О стандартных функциях языка Паскаль говорилось в разделе 2.4. Рассмотрим некоторые стандартные арифметические процедуры.

`Dec(X[,n])` уменьшает значение целочисленной переменной `X` на n[1]. Если n отсутствует, то значение `X` уменьшается на 1. `Inc(X[,n])` увеличивает значение целочисленной переменной `X` на n.

6.1. Процедуры в языке Турбо Паскаль

Каждая новая *процедура* или функция должна быть предварительно описана в разделе описания процедур и функций. Для использования процедуры необходимо написать оператор вызова.

Описание процедуры состоит из заголовка процедуры и тела процедуры. Заголовок включает служебное слово `procedure`, имя процедуры и заключенный в круглые скобки список формальных параметров *с указанием их типов* (именно типов, а не описаний!):

```
procedure < имя > (<список формальных параметров>);
```

Например:

```
procedure a17 (a:real; b,c:real; var x1,x2:real; var k:integer);
```

Формальные параметры отделяются точкой с запятой (список однотипных параметров может быть перечислен через запятую). После заголовка идут разделы описаний (констант, типов, переменных, процедур и функций, используемых в процедуре) и операторы языка Паскаль, реализующие алгоритм процедуры.

Например:

```
{ Процедура f вычисляет значения факториала числа r }
{ и возвращает результат в переменной r1. }
{ Для хранения значений факториалов (r1) использован тип longint. }
procedure f(r:integer; var r1:longint);
var
  i:integer;
begin
{ Если значение r отрицательно, то r1=0, и процедура завершается. }
  if r<0 then
  begin
        r1:=0;
        exit;
  end;
{ Ниже реализован алгоритм вычисления r!=1·2·3·…·n. }
  r1:=1;
```

[1] Параметр n не является обязательным.

```
  for i:=2 to r do
     r1:=r1*i;
end.
```

Формальные параметры *нельзя* описывать в разделе описаний процедуры.

Для обращения к процедуре необходимо использовать оператор вызова процедуры. Он имеет следующий вид:

```
<имя процедуры> (<список_фактических_параметров>);
```

Например:

```
f (n,fn);
```

Фактические параметры в списке отделяются друг от друга запятой. Механизм применения формальных-фактических параметров обеспечивает замену первых параметров последними, что позволяет выполнять процедуру с различными данными. Между фактическими параметрами в операторе вызова процедуры и формальными параметрами в заголовке процедуры устанавливается взаимно однозначное соответствие.

Количество, типы и порядок следования формальных и фактических параметров должны совпадать.

Например:

```
program abc;
var
  r,r1,r2:real;
procedure f1 (x:real; var y:real; var c:real);
begin
  y:=sin(x)/cos(x);
  c:=ln(x)/ln(10);
end;
begin
  read(r);
  f1(r,r1,r2);
  write (r1,r2);
end.
```

6.2. Формальные и фактические параметры

Можно выделить два основных класса формальных параметров:

- *параметры-значения*;
- *параметры-переменные*.

Параметры-значения используются в качестве входных данных подпрограммы, при обращении к которой фактические параметры передают свое значение формальным и больше не изменяются.

Параметры-переменные могут использоваться как в качестве входных данных, так и в качестве выходных. В заголовке процедуры перед ними необходимо указывать служебное слово `var`. При обращении к подпрограмме фактические параметры замещают формальные[1]. В результате выполнения подпрограммы изменяются фактические параметры.

В качестве входных данных в подпрограмме следует использовать параметры-значения, в качестве выходных – параметры-переменные. Данные, которые являются и входными, и выходными, следует описывать как параметры-переменные.

Пример 6.1. Написать программу решения группы квадратных уравнений $p_i x^2 + q_i x + r_i = 0$, где `p`, `q`, `r` – массивы вещественных чисел, состоящие из `k` элементов. Решение одного уравнения оформить в виде процедуры.

Создание программы начнем с процедуры решения квадратного уравнения. Эта задача подробно была рассмотрена в главе 3. У процедуры решения квадратного уравнения `korni` 6 параметров:

- параметры-значения (`a`, `b`, `c`) типа `real`, которые используются для передачи в процедуру коэффициентов квадратного уравнения;
- параметры-переменные `x1`, `x2` типа `real`, `pr` типа `boolean`; в переменных `x1` и `x2` из процедуры будут возвращаться значения корней уравнения; переменная `pr` возвращает `true`, если в уравнении есть действительные корни, и `false`, если действительных корней нет.

Заголовок процедуры имеет вид:

```
procedure korni(a,b,c:real;var x1,x2:real;var pr:Boolean);
```

На рис. 6.1 приведена блок-схема подпрограммы решения квадратного уравнения. Обращаем внимание читателя на то, что в начальном блоке подпрограммы вместо слова «начало» следует писать имя подпрограммы с указанием списка параметров. Значение параметра `pr` определяется знаком дискриминанта.

Далее рассмотрим решение всей задачи целиком. На рис. 6.2 представлена соответствующая блок-схема, в которой можно выделить следующие этапы:

[1] Это упрощенное объяснение механизма передачи данных между формальными и фактическими параметрами. На самом деле все несколько сложнее – происходит передача адреса параметра в процедуру.

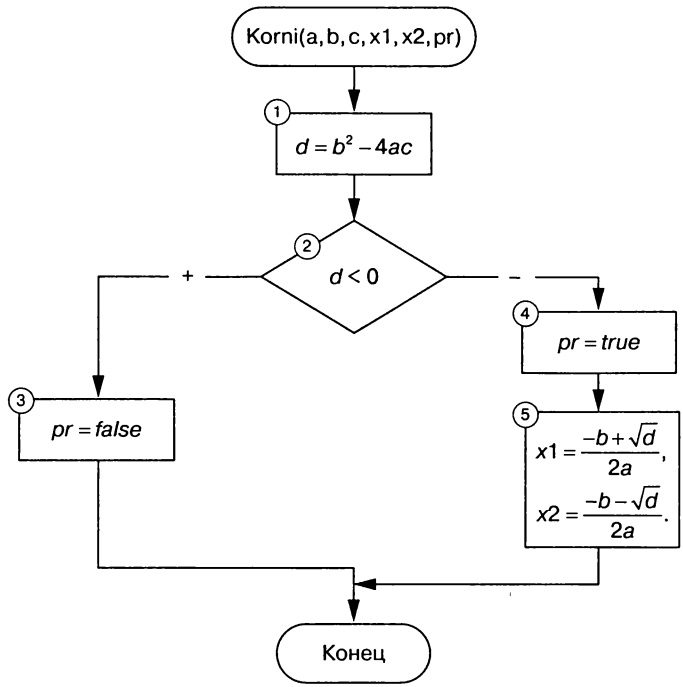


Рис. 6.1 ▾ Блок-схема подпрограммы решения квадратного уравнения

1. Ввод количества уравнений в переменную k (блок 1).
2. Организация цикла по переменной i, которая будет изменяться от 1 до k (блоки 2–7) и определять номера уравнения. Для каждого уравнения необходимо будет выполнить следующее:
   2.1. Ввести значения коэффициентов уравнения p[i], q[i] и r[i] (блок 3).
   2.2. Обратиться к процедуре решения квадратного уравнения (блок 4[1]).
   2.3. В зависимости от значения логической переменной L необходимо вывести значения корней (блок 6) или сообщение о том, что уравнение не имеет решений (блок 7).

Ниже приведен текст программы с комментариями:

```
Program urav;
{ Здесь начинается раздел описаний программы urav. }
var
```

[1] Именно так в дальнейшем мы будем обозначать блок обращения к подпрограммам.

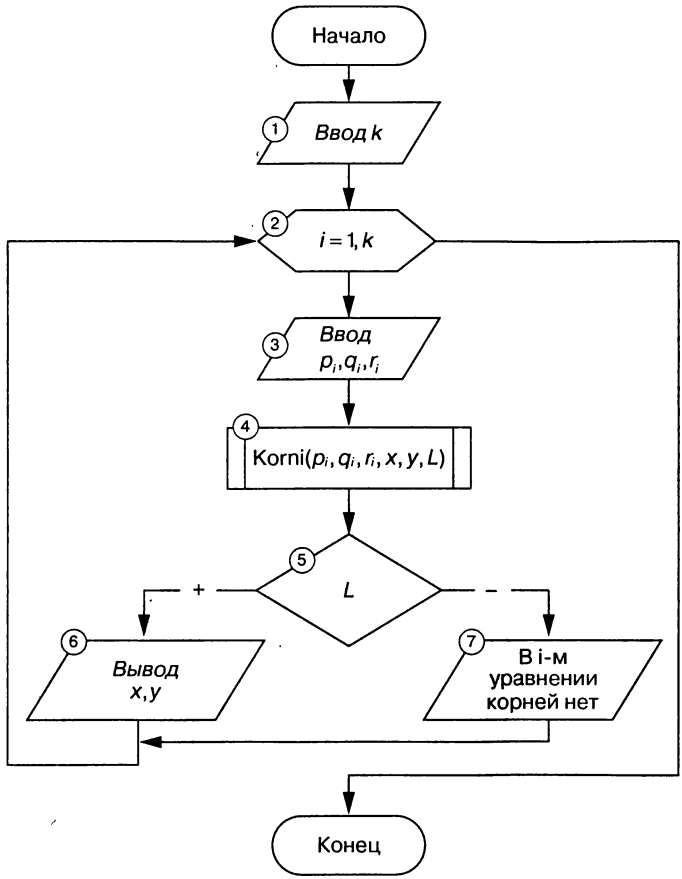


Рис. 6.2 ▼ Блок-схема решения примера 6.1

```
  p,q,r:array [1..50] of real;
  x,y:real;
  i,k:integer;
  L:Boolean;
{ В разделе описаний присутствует процедура korni решения квадратного
уравнения. Коэффициенты уравнения a,b,c - входные параметры-значения.
В качестве выходных выступают параметры-переменные x1, x2 и pr.
В переменных x1 и x2 возвращаются корни квадратного уравнения. Переменная
pr возвращает значение true, если есть действительные корни уравнения.
Значение false возвращается, если корней нет и дискриминант отрицателен. }
procedure korni(a,b,c:real;var x1,x2:real;var pr:Boolean);
```

```
var
  d:real;
begin
  d:=b*b-4*a*c;
  if d>=0 then
  begin
     pr:=true;
     x1:=(-b+sqrt(d))/2/a;
     x2:=(-b-sqrt(d))/2/a;
  end
  else pr:=false
end;
{ Начало основной программы. }
begin
  writeln (' введите количество уравнений');
  read (k);
{ Цикл предназначен для ввода коэффициентов уравнения, вызова процедуры
решения квадратного уравнения korni и, если корни существуют, вывода их на
экран. }
  for i:=1 to k do
  begin
     writeln ('введите коэффициенты',i,'-го уравнения');
     read(p[i],q[i],r[i]);
{ Обращение к процедуре korni. }
     korni(p[i],q[i],r[i],x,y,L);
{ Проверка, существуют ли korni (L=true). }
     if L then
        writeln('корни',i,'-го','уравнения',' ',x:8:5,' ', y:8:5)
     else
        writeln ('в ',i,'-м уравнении корней нет');
  end;
end.
```

Читателю предлагается модифицировать программу для нахождения не только действительных, но и комплексных корней. В этом случае в процедуре при отрицательном дискриминанте `pr` можно записывать значение `false`, в переменную `x1` – действительную часть комплексного корня, в переменную `x2` – модуль мнимой части комплексного корня.

Пример 6.2. Найти максимальные элементы массивов `a`, `b`, `c` и их номера. Подпрограмму поиска оформить в виде процедуры.

Решение этой задачи также начнем с написания процедуры. Поиск максимального элемента и его номера уже был рассмотрен в третьей главе. У процедуры нахождения максимального элемента и его номера `max1` 4 параметра:

- параметры-значения: массив x и его размер L (тип integer). Для указания типа массива необходимо ввести новый тип данных (назовем его vector);
- параметры-переменные: rmax (в ней возвращается максимальный элемент массива x) типа real и n (номер максимального элемента в массиве) типа integer.

Заголовок процедуры можно записать следующим образом[1]:

```
procedure max1(x:vector; L:integer; var rmax:real; var n:integer);
```

В головной программе надо будет ввести размеры массива a и их элементы, обратиться к процедуре поиска максимума, вывести значения максимального элемента массива a и его номера. Эти же действия повторить для массивов b и c.

Ниже приведен текст программы с подробными комментариями:

```
program abc;
{ Тип данных vector, который будет определять тип формального параметра. }
type
  vector=array [1..200] of real;
var
  a,b,c:vector;
  i,k:integer;
  max:real;
  nmax:integer;
{ Процедура max1 предназначена для поиска максимального элемента массива и
его номера. В ней два входных параметра-значения: массив x и его размер L.
При определении типа массива используется тип vector, выходные параметры-
переменные: максимальный элемент rmax и его номер n. }
procedure max1(x:vector; L:integer; var rmax:real; var n:integer);
var
  j:integer;
begin
  rmax:=x[1]; n:=1;
  for j:=2 to L do
     if x[j]>rmax then
     begin
        rmax:=x[j]; n:=j;
     end
end;
begin
{ В данном примере размеры всех элементов одинаковы, но можно вводить
переменную k для каждого массива. }
  writeln ('введите размер массивов'); read (k);
  for i:=1 to k do
     begin
```

[1] Перед заголовком процедуры необходимо описать тип данных vector следующим образом: `type vector = array [1..200] of real;`

```
    write ('a[',i,']='); read (a[i]);
  end;
{ Обращение к процедуре для поиска максимального элемента и его номера в
массиве a. }
  max1(a, k, max, nmax);
  writeln('max=',max:1:4,' nmax=',nmax);
  for i:=1 to k do
  begin
    write ('b[',i,']='); read (b[i]);
  end;
{ Обращение к процедуре для поиска максимального элемента и его номера в
массиве b. }
  max1(b, k, max, nmax);
  writeln (' max=',max:1:4,' nmax=',nmax);
  for i:=1 to k do
  begin
    write('c[',i,']='); read(c[i]);
  end;
{ Обращение к процедуре для поиска максимального элемента и его номера в
массиве c. }
  max1(c, k, max, nmax);
  writeln (' max=',max:1:4,' nmax=',nmax);
end.
```

6.3. Функции в языке Паскаль

Описание функции состоит из заголовка функции и тела. Заголовок содержит служебное слово function, имя функции, список формальных параметров с указанием их типа и типа возвращаемого результата:

```
function <имя> (<список_формальных_ параметров>):<тип>.
```

Здесь <тип> – тип возвращаемого функцией значения. Функции могут возвращать *скалярные* значения целого, вещественного, логического, символьного или ссылочного типа.

Примеры описания функций:

```
function tan (x:real):real;
function max(x,y:real):real;
function a1(x:real; y:real):real;
```

Обращение к функции осуществляется по имени с указанием списка фактических параметров. Количество, типы и порядок следования формальных и фактических параметров должны совпадать:

```
<имя_функции> (<список_фактических_параметров>);
```

В теле функции всегда должен быть *один* оператор, присваивающий значение имени функции.

Рассмотрим пример использования функции:

```
program abc;
var
  x,y:real;
{ Определяем функцию tan, которая возвращает значения тангенса (tg). }
function tan (c:real):real;
begin
  tan:=sin(c)/cos(c);
end;
begin
  read (x,y);
{ Вычисляются значения tg(x) и tg(y) путем обращения к функции tan. }
  writeln ('tg(',x:1:3,')=',tan (x):1:4);
  writeln ('tg(',y:1:3,')=',tan (y):1:4);
end.
```

При использовании процедур и функций переменные объявляются несколько раз в основной программе и в подпрограммах.

Переменные и типы, определенные в основной программе до объявления процедур и функций, называются *глобальными* – они доступны всем функциям и процедурам. Переменные, определенные в какой-либо подпрограмме или основной программе после раздела описаний процедур и функций, называются *локальными*.

Для правильного определения области действия идентификаторов (переменных) необходимо придерживаться следующих правил:

- каждая переменная должна быть описана перед тем, как она будет использована;
- областью действия переменной является та подпрограмма, в которой она описана;
- все переменные в подпрограммах должны быть уникальными;
- одна и та же переменная может быть по-разному определена в каждой из подпрограмм;
- если имя подпрограммы совпадает с названием стандартной подпрограммы, то последняя игнорируется, а выполняется подпрограмма пользователя;
- если внутри какой-либо процедуры встречается переменная с таким же именем, что и глобальная переменная, то внутри процедуры будет действовать *локальное* описание;
- каждая подпрограмма может изменить значение глобальной переменной.

Рассмотрим примеры задач с использованием подпрограмм.

Пример 6.3. Задан массив целых чисел. Необходимо удалить из него все совершенные числа.

Совершенное число представляет собой сумму всех своих делителей, меньших его самого. Для решения поставленной задачи необходимо проверить каждый элемент массива, и, если он представляет собой совершенное число, удалить его (после удаления остается элемент с тем же номером, однако это уже следующий элемент). В противном случае надо перейти к следующему элементу. Блок-схема решения данной задачи представлена на рис. 6.3.

В программе будут использоваться две подпрограммы: функция `Sover`, которая проверяет, является ли число совершенным, и процедура `Udal` удаления одного элемента из массива.

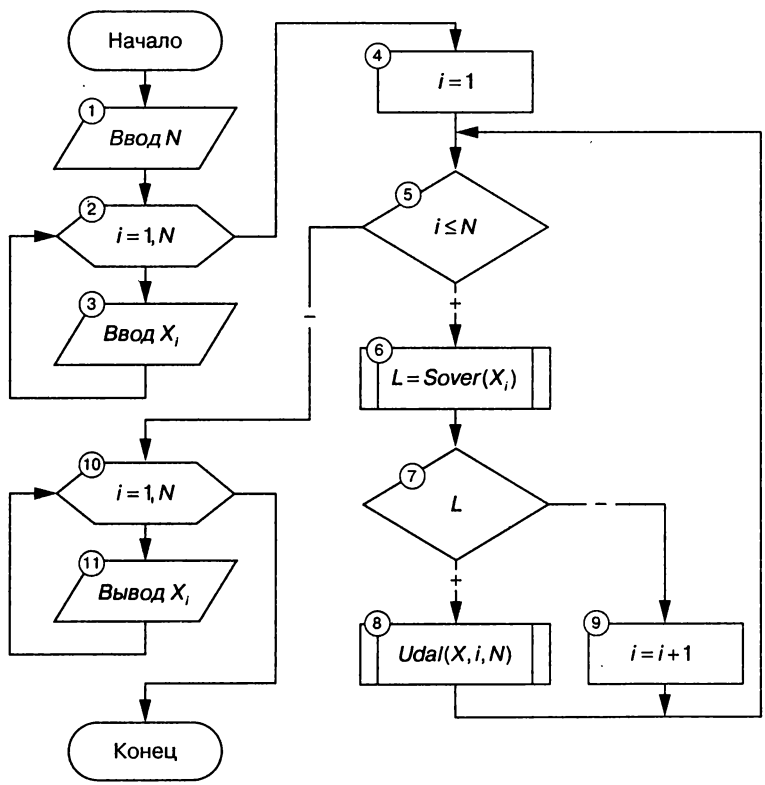


Рис. 6.3 ▾ Блок-схема решения примера 6.3

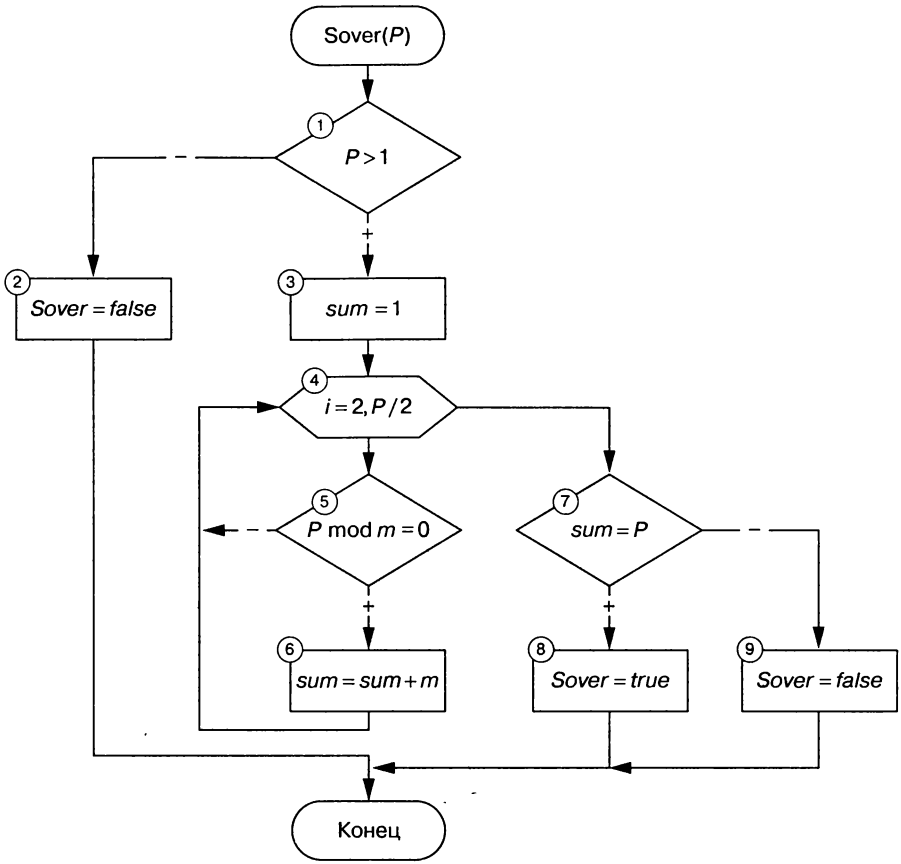


Рис. 6.4 ▼ Блок-схема функции `Sover`

Заголовок функции `Sover` имеет следующий вид:

```
Function Sover(P:longint):boolean;
```

Функция возвращает значение `true`, если число `p` является совершенным, и `false` в противном случае. Рассмотрим блок-схему функции (рис. 6.4).

Для определения, является ли число совершенным, надо найти сумму всех делителей числа `P`. Если `P < 2` (блок 1), то число не является совершенным – `Sover=false` (блок 2). Для проверки остальных чисел будем использовать следующий алгоритм. В начале сумма (переменная `sum`) равна 1 («единица»

является делителем любого числа) (блок 3), затем последовательно проверяем, делится ли число P нацело на 2, 3,...P/2 (блоки 4–6)[1]. Числа, на которые число P делится (проверка осуществляется в блоке 5), добавляем к сумме (блок 6). После нахождения суммы сравниваем ее с числом P (блок 7). Если ее значение равно P (sum=P), то число совершенное, и функция возвращает true (блок 8), в противном случае – false (блок 9).

Заголовок процедуры удаления элемента из массива имеет следующий вид:

```
Procedure Udal(var x:massiv; i:word; var N:word);
```

В процедуре Udal используются следующие формальные параметры:

- массив x. Из него мы будем удалять элемент. Он описан как параметр-переменная, потому что в главную программу будет возвращаться измененный массив. Перед описанием процедуры следует описать тип данных massiv (например, massiv=array [1..200] of longint);
- i – номер удаляемого элемента;
- переменная n – размер массива x. Она описана как параметр-переменная, потому что после удаления элемента из массива его размер уменьшается на 1.

Задача удаления элемента из массива рассматривалась в главе 5, поэтому блок-схему процедуры Udal мы приводим без комментариев (рис. 6.5).

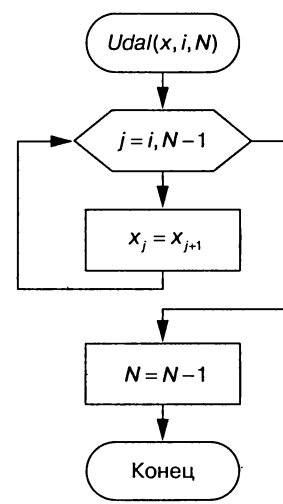


Рис. 6.5 ▾ Блок-схема процедуры Udal

Ниже представлен листинг программы с комментариями:

```
{ Тип данных massiv будет использоваться при
описании процедуры удаления i-го элемента. }
type
  massiv=array [1..1000] of longint;
{ Описание функции, проверяющей, является ли число
совершенным. }
Function Sover(P:longint):boolean;
var sum,m:longint;
begin
  if P>1 then
{ Нахождение суммы общих делителей. }
  Begin
{ В начале сумма равна 1, так как любое число
делится на 1. }
```

[1] Число не может делиться нацело больше чем на половину самого себя.

```
    sum:=1;
{ Делителем числа может быть любое число от 2 до половины самого себя. }
    for m:=2 to p div 2 do
{ Если число m - делитель, то добавляем его к сумме. }
    if p mod m =0 then sum:=sum+m;
{ Если сумма делителей равна самому числу P, то функция возвращает
значение true,}
    if sum=P then Sover:=true
{ иначе - false. }
    else Sover:=false
  end
{ Числа, меньшие или равные 1, не являются совершенными, поэтому функция
возвращает значение false. }
  else Sover:=false
end;
{ Описание процедуры удаления i-го элемента из массива x. }
Procedure Udal(var x:massiv; i:word; var N:word);
var j:word;
begin
{ Сдвиг массива, начиная с i-го, влево на один элемент. }
  for j:=i to N-1 do
  x[j]:=x[j+1];
{ После удаления элемента его размер становится на один меньше. }
  N:=N-1;
end;
var x:massiv;
i,n:word;
l:boolean;
begin
  write('N=');readln(N);
  writeln(' Массив X');
  for i:=1 to N do
  read(x[i]);
{ Просмотр массива начинаем с первого элемента. }
  i:=1;
{ Проверяем, не достигнут ли конец массива. }
  while(i<=N) do
  begin
{ Обращение к функции Sover, которая проверяет, является ли элемент массива
x[i] совершенным; результат записывается в логическую переменную L. }
    L:=Sover(x[i]);
{ Если число совершенное, то удаляем его из массива. При этом текущим
остается элемент с номером i, но после удаления там хранится другой
элемент. }
```

```
    if L then Udal(x,i,N)
{ Если число не является совершенным, то переходим к следующему элементу
массива. }
    else i:=i+1
  end;
  writeln(' Преобразованный массив X');
  for i:=1 to N do
  write(x[i],' ');
end.
```

Пример 6.4. В матрице натуральных чисел $A(N,M)$ найти строки, в которых находится максимальное из простых чисел. Элементы в них упорядочить по возрастанию. Если в матрице нет простых чисел, оставить ее без изменений.

Перед решением задачи отметим некоторые ее особенности[1]. В матрице может не обнаружиться простых чисел, максимальных значений может быть несколько, и при этом некоторые из них будут находиться в одной строке.

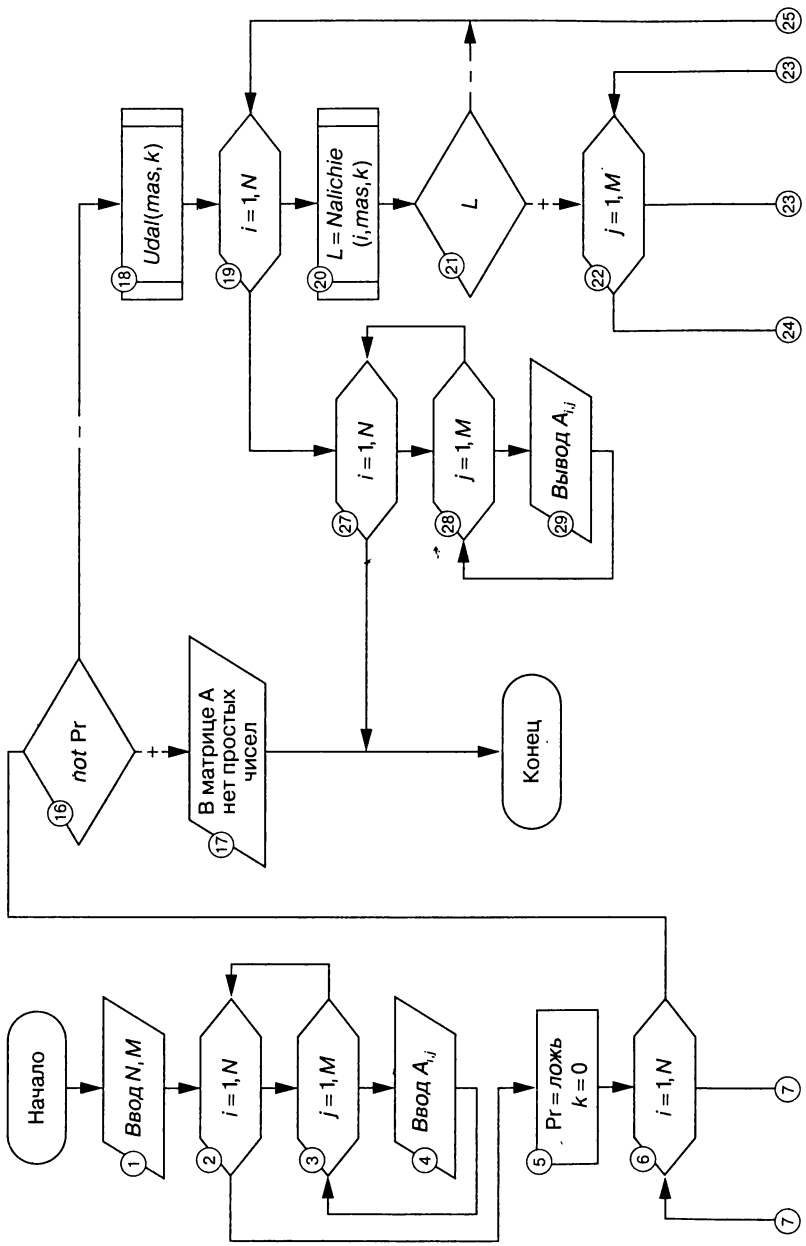
Блок-схема решения задачи представлена на рис. 6.6.

После ввода матрицы предполагаем, что простых чисел нет. В логическую переменную `Pr` записываем значение `false`, как только встретится простое число. В переменную `Pr` следует записать `true`. Количество максимальных значений среди простых чисел равно `0`, `k = 0` (блок 5).

Для того чтобы проверить, является ли простым числом каждый элемент матрицы, обращаемся к функции `Prostoe`, которая возвращает логическое значение `true`, если число простое, и `false` – в противном случае. Результат обращения к функции записываем в логическую переменную `L` (блок 8). Если число простое (блок 9), проверяем, первое ли это простое число в матрице (блок 10). Если это так, то переписываем его в переменную `max`. В переменную `k` записываем число `1` (количество максимумов равно 1). Номер строки, в которой находится максимум, записываем в `mas[k]`[2]. В переменную `Pr` записываем `true`, в матрице есть простые числа (блок 11). Если это не первое простое число, сравниваем `A[i,j]` с переменной `max`. Если `A[i,j] > max` (блок 12), то в переменную `max` запишем `A[i,j]`, в переменную `k` – 1 (у нас есть один максимум), а в `mas[k]` – `i`, то есть номер строки, в которой находится максимальный элемент (блок 13). Если `A[i,j] = max` (блок 14), то встретилось число, равное переменной `max`. В этом случае значение `k` увеличиваем на 1 и в `mas[k]` записываем

[1] Авторы рекомендуют читателям внимательно изучить этот пример, так как в нем сконцентрированы практически все основные моменты, рассмотренные нами до сих пор.

[2] В массиве `mas` будут храниться номера строк, в которых находится максимум.


Начало
1 Ввод N, M
2 i = 1, N
3 j = 1, M
4 Ввод A_ij
5 Pr = ложь
k = 0
6 i = 1, N
7
7
16 not Pr
17 В матрице A нет простых чисел
Конец
18 Udal(mas, k)
19 i = 1, N
20 L = Nalichie (i, mas, k)
21 L
22 j = 1, M
23
23
24
25
27 i = 1, N
28 j = 1, M
29 Вывод A_ij

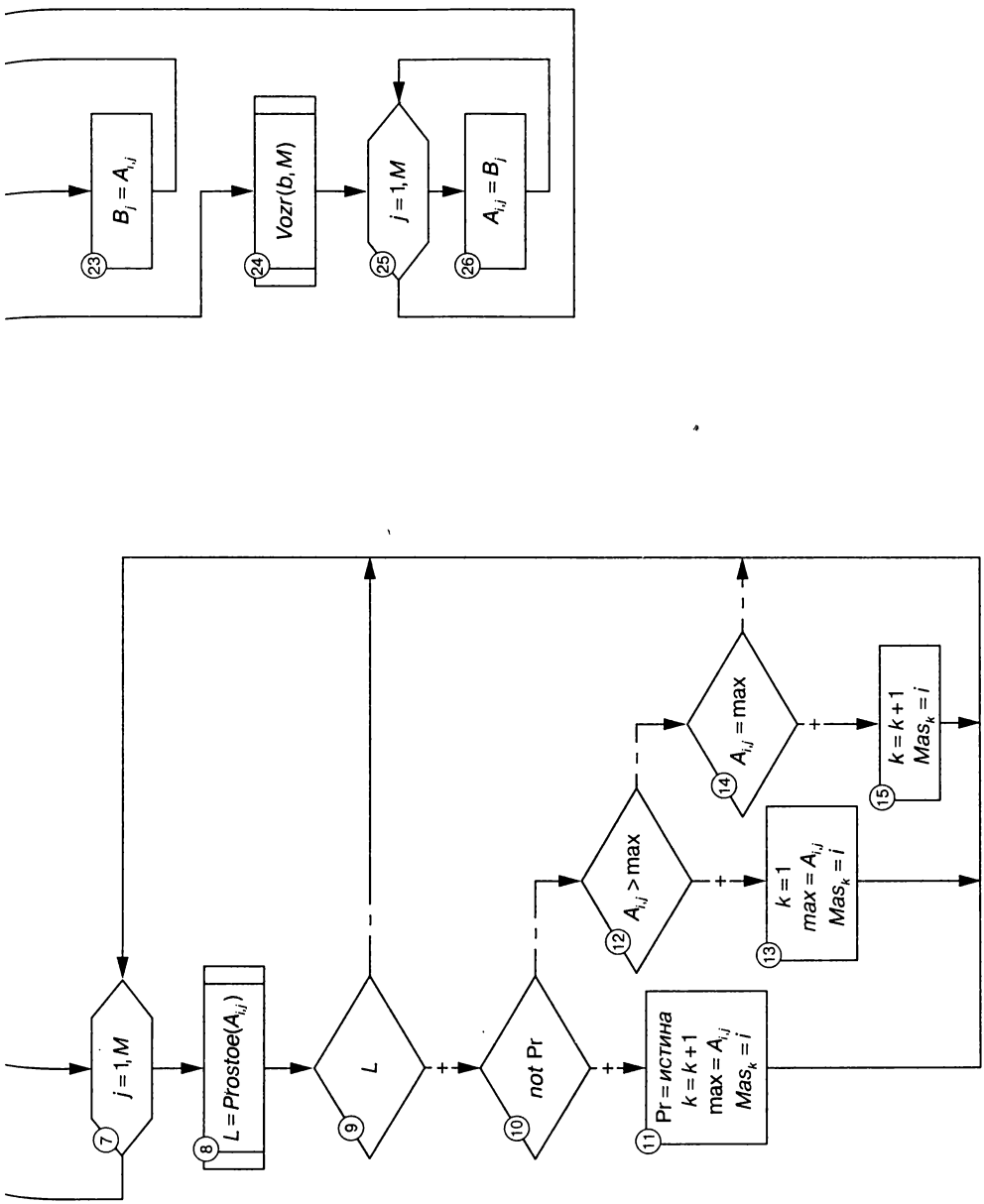


Рис. 6.6 ▾ Блок-схема решения примера 6.4

номер строки, в которой находится элемент, равный max. В результате двойного цикла обработки всех элементов матрицы (блоки 6–15) в переменной max будет храниться максимальное из простых чисел, в переменной k – количество максимумов, в массиве mas из k элементов будут содержаться номера строк, в которых находятся максимальные значения среди простых чисел матрицы. В переменной Pr хранится значение true, если в матрице есть простые числа; в противном случае там будет значение false.

Если в матрице нет простых чисел (блок 16), выводим соответствующее сообщение (блок 17), в противном случае с помощью процедуры Udal (блок 18) удаляем из массива mas элементы, встречающиеся более одного раза[1]. Затем просматриваем все строки матрицы (цикл начинается блоком 19). Если номер этой строки присутствует в массиве mas (обращение к функции Nalichie осуществляется в блоке 20, проверка логической переменной – в блоке 21), то переписываем текущую строку матрицы в массив b (блоки 22–23) и обращаемся к процедуре упорядочивания массива по возрастанию – Vozr (блок 24). Упорядоченный массив b переписываем в i-ю строку матрицы A (блоки 25–26). На последнем этапе выводим на экран матрицу A после преобразования (блоки 27–29).

При решении задачи будут использоваться три подпрограммы:

1. Функция Prostoe, которая проверяет, является ли число P типа word простым. Она возвращает значение true, если число P простое, и false в противном случае. Заголовок функции имеет вид:

   ```
   Function Prostoe (P:word):Boolean;
   ```

   Задача о простом числе обсуждалась в главе 3; блок-схема функции Prostoe приведена без комментариев на рис 6.7.
2. Процедура Udal, которая из массива чисел x удаляет значения, встречающиеся более одного раза. У процедуры два параметра: массив x и его размер N, оба – параметры-переменные. Заголовок процедуры имеет вид:

   ```
   Procedure Udal(var x:massiv; var N:word);
   ```

   Перед описанием процедуры следует описать тип данных massiv (например, massiv = array [1..200] of word). Блок-схема процедуры Udal представлена на рис. 6.8.

   Удаление повторяющихся элементов происходит следующим образом. Просматриваются все элементы, начиная с первого (i = 1, 2, ..., n),

[1] В массиве mas есть повторяющиеся элементы, если некоторые максимальные элементы находятся в одной строке.

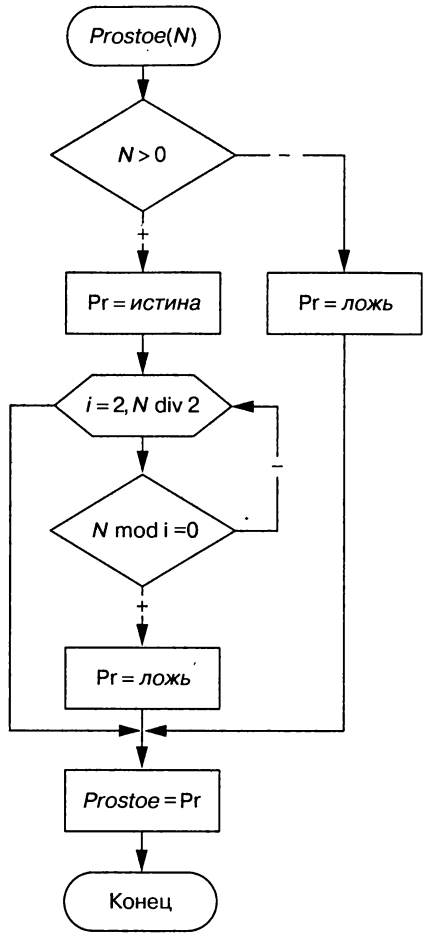


Рис. 6.7 ▾ Блок-схема функции `Prostoe`

`i`-й элемент сравнивается со всеми последующими (`j = i + 1, i + 2, ..., n`). Если $x_i = x_j$, то встретился повторяющийся элемент, и мы удаляем из массива элемент с номером `j` (блоки 6–8). Алгоритм удаления полностью аналогичен описанному в примере 6.3.

3. Функция `Nalichie` возвращает значение `true`, если число `a` присутствует в массиве `b`, и `false` – в противном случае. Блок-схема функции представлена на рис. 6.9.

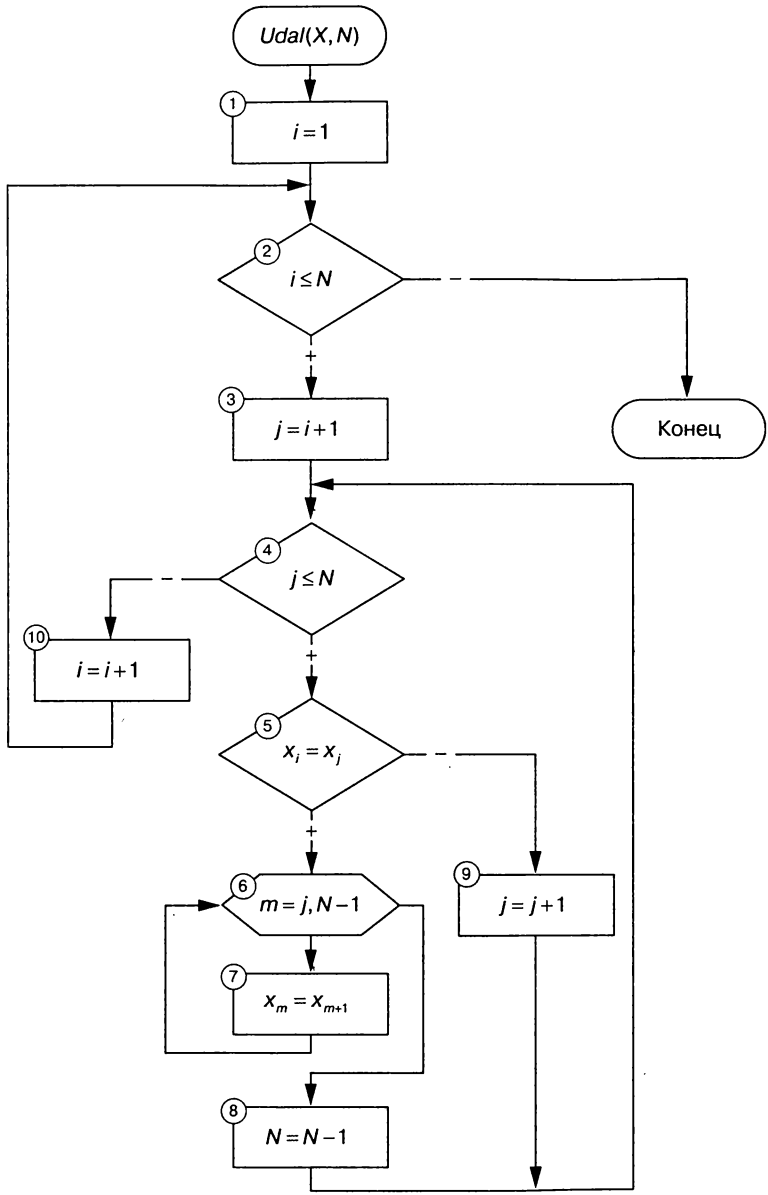


Рис. 6.8 ▼ Блок-схема процедуры `Udal`

Заголовок процедуры имеет вид:

```
Function Nalichie(a:word; x:massiv; N:word);
```

4. Процедура `Vozr` упорядочения массива x по возрастанию. Алгоритмы упорядочения рассматривались в главе 4. Блок-схема процедуры `Vozr` представлена на рис. 6.10.
   У процедуры `Vozr` два параметра: массив x (параметр-переменная) и его размер `N` (параметр-значение). Заголовок процедуры имеет вид:

```
Procedure Vozr(var x:massiv; N:word);
```

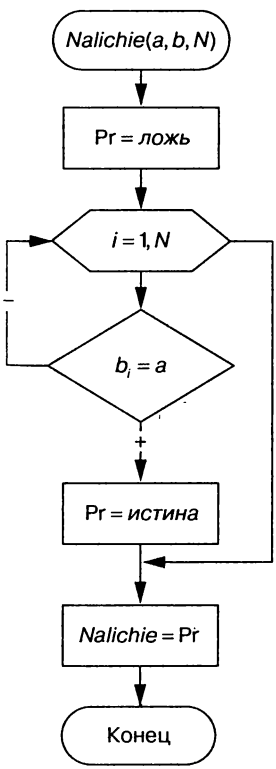


Рис. 6.9 ▾ Блок-схема функции `Nalichie`

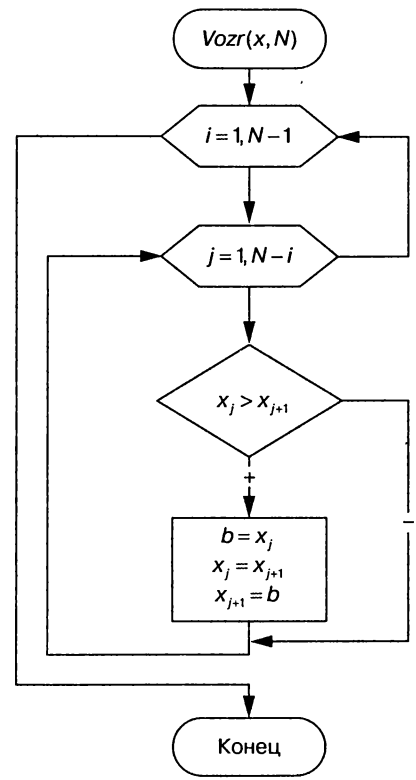


Рис. 6.10 ▾ Блок-схема процедуры `Vozr`

Ниже приведен листинг всей программы с комментариями:

```
{ Тип данных massiv будет использоваться при описании процедур. }
type
massiv=array[1..200] of word;
{ Функция prostoe проверяет, является ли число N простым (true) или нет
(false). }
function prostoe(N:word):boolean;
var pr:boolean;
i:word;
begin
  if N>0 then
  begin
{ Предполагаем, что число N - простое (pr=true). }
     pr:=true;
{ Проверяем, делится ли число N на какое либо из чисел от 2 до N/2. }
     for i:=2 to n div 2 do
{ Если встречается число i, на которое делится N, то }
     if N mod i =0 then
     begin
{ число N не является простым (pr=false), поэтому }
        pr:=false;
{ выходим из цикла. }
        break;
     end
  end
  else pr:=false;
{ Имени функции присваиваем значение переменной pr. }
  prostoe:=pr;
end;
{ Процедура udal удаляет из массива x элементы, которые встречаются более
одного раза. X, N являются параметрами-переменными, так как эти значения
возвращаются в головную программу при вызове процедуры udal. }
procedure udal(var x:massiv; var n:word);
var i,j,m:word;
begin
  i:=1;
{ Просматриваем все элементы, начиная с первого: i=1,2,…,N;i-й элемент
сравниваем с последующими: j=i+1,i+2,…,n. }
  while(i<=n) do
  begin
     j:=i+1;
     while(j<=N) do
{ Если x[i] равно x[j], то встретился повторяющийся элемент. }
     if x[i]=x[j] then
     begin
```

```
{ Удаляем его (x[j]) из массива. }
       for m:=j to N-1 do
       x[m]:=x[m+1];
{ После удаления элемента количество элементов уменьшаем на 1, при этом не
переходим к следующему элементу, так как после удаления под j-м номером
находится уже другой элемент. }
       N:=N-1;
    End
{ Если x[i] не равно x[j], переходим к следующему элементу. }
    else j:=j+1;
    i:=i+1;
  end;
end;
{ Функция возвращает значение true, если число a встречается в массиве b;
в противном случае возвращается значение false. }
function nalichie(a:word;b:massiv;n:word):boolean;
var
  pr:boolean;
  i:word;
begin
{ Предполагаем, что в массиве b не встречается значение a, pr=false. }
  pr:=false;
{ Перебираем все элементы массива. }
  for i:=1 to N do
{ Если очередной элемент массива b равен значению a, то в pr записываем
true }
  if b[i]=a then
  begin
     pr:=true;
{ и выходим из цикла. }
     break
  end;
{ Имени функции присваиваем значение переменной pr. }
  nalichie:=pr
end;
{ Процедура vozr упорядочивает массив x по возрастанию. }
procedure vozr(var x:massiv; n:word);
{ X является параметром-переменной, именно массив и возвращается в
головную программу при вызове процедуры vozr. }
var i, j, b :word;
begin
  for i:=1 to N-1 do
  for j:=1 to N-i do
  if x[j]>x[j+1] then
  begin
     b:=x[j];
```

```
    x[j]:=x[j+1];
    x[j+1]:=b;
  end
end;
var
  N, M, i, j, k, max :word;
  A:array[1..20,1..20] of word;
  pr, L: boolean;
  mas, b:massiv;
begin
{ Вводим элементы матрицы A. }
  write('N=');readln(N);
  write('M=');readln(M);
writeln(' Matrica A');
  for i:=1 to N do
  for j:=1 to M do
    read(A[i,j]);
{ Предполагаем, что в матрице нет простых чисел. }
  Pr:=false;
{ Количество элементов, равных максимальному, - 0. }
  k:=0;
{ Перебираем все элементы в матрице. }
  for i:=1 to N do
  for j:=1 to M do
  begin
{ Обращаемся к функции, которая проверяет, является ли число A[i,j]
простым. }
    L:=Prostoe(A[i,j]);
{ Если число простое и }
    if L then
{ если простое число встретилось первый раз, }
       if not Pr then
       begin
{ записывем в pr значение true и }
         Pr:=true;
{ увеличиваем количество максимумов на 1. Можно было просто написать k:=1. }
         k:=k+1;
{ Это число записываем в переменную max (это первое простое число) и
предполагаем, что оно максимальное. }
         max:=A[i,j];
{ В mas[k] записываем номер строки, в которой хранится число A[i,j]. }
         mas[k]:=i
       end
       else
{ Если A[i,j] не первое простое число, то сравниваем max и текущее простое
значение матрицы A. }
```

```
          if A[i,j]>max then
{ Если A[i,j]>max, то }
          Begin
{ количество максимумов равно 1, так как встретился наибольший в данный
момент элемент. }
             k:=1;
{ В переменную max записываем A[i,j], }
             max:=A[i,j];
{ в mas[k] записываем номер строки, где хранится число A[i,j]. }
             mas[k]:=i
          end
          else
{ Если A[i,j]=max (встретился элемент, равный максимуму), то }
if A[i,j]=max then
             begin
{ количество максимумов увеличиваем на 1. }
                k:=k+1;
{ В mas[k] записываем номер строки, в которой хранится число A[i,j]. }
                mas[k]:=i
             end
  end;
{ Если в pr осталось значение false,то выводим сообщение, что в матрице
нет простых чисел, }
  if not Pr then writeln('В матрице A нет простых чисел', )
  else
  begin
{ иначе удаляем из массива mas номера строк, в которых хранятся максимумы,
повторяющиеся элементы. }
     Udal(mas,k);
{ Перебираем все строки матрицы. }
     for i:=1 to N do
     begin
        L:=Nalichie(i,mas,k);
{ Если номер строки присутствует в массиве mas, }
        if L then
        begin
{ то переписываем строку в массив b. }
          for j:=1 to M do
          b[j]:=A[i,j];
{ Упорядочиваем массив b по возрастанию. }
          Vozr(b,M);
{ Упорядоченный массив записываем на место i-й строки матрицы A. }
          for j:=1 to M do
          A[i,j]:=b[j];
        end
     end;
```

```
    writeln('Преобразованная матрица A');
    for i:=1 to N do
    begin
       for j:=1 to M do
       write(A[i,j],' ');
       writeln;
    end
  end
end.
```

6.4. Особенности работы с подпрограммами в Турбо Паскале версии 7.0

В Турбо Паскале версии 7.0 появились такие способы передачи данных в подпрограмму, как открытые массивы и параметры-константы.

6.4.1. Открытые массивы

В версии Турбо Паскаль 7.0 в качестве формальных параметров процедур могут использоваться *открытые массивы*, которые описываются с помощью служебного слова `array` и названия типа [10, 11]. Тип элементов массива должен быть скалярным.

Пример 6.5. Вычислить сумму и произведение элементов массивов `b`, `c`, `d`.

```
{ Процедура вычисления суммы и произведения массива. }
{ В процедуре три параметра. }
{ Входной параметр-значение - открытый массив a. }
{ Выходные параметры-переменные: сумма (sum) и произведение (proiz). }
procedure sum_proiz(a:array of real; var sum,proiz:real);
var
  i:integer;
begin
  sum:=0;
  proiz:=1;
  for i:=0 to high (a) do
  begin
     sum:=sum+a[i];
     proiz:=proiz*a[i]
  end
end;
var
  b:array [1..5] of real;
  c:array [6..14] of real;
```

```
  d:array [0..3] of real;
  s,p:real; k:integer;
begin
  for k:=1 to 5 do readln (b[k]);
  for k:=6 to 14 do readln (c[k]);
  for k:=0 to 3 do readln (d[k]);
{ Вызов sum_proiz для вычисления суммы и произведения массива b. }
  sum_proiz (b,s,p);
  writeln ('массив b');
  for k:=1 to 5 do write(b[k]:1:4,' ');
  writeln;
writeln ('s=',s:1:5,' p=',p:1:5);
{ Вызов sum_proiz для вычисления суммы и произведения массива c. }
sum_proiz (c,s,p);
  writeln ('массив c');
  for k:=6 to 14 do write(c[k]:1:4,' ');
  writeln;
  writeln ('s=',s:1:5,'p=',p:1:5);
{ Вызов sum_proiz для вычисления суммы и произведения массива d. }
  sum_proiz (d,s,p);
  writeln ('массив d');
  for k:=0 to 3 do write(d[k]:1:4,' ');
  writeln;
  writeln ('s=',s:1:5,' p=',p:1:5);
end.
```

В процедуре sum_proiz формальный параметр a описан как открытый массив, у которого не задан размер, однако указан тип его элементов. Это позволяет обращаться к процедуре с фактическими параметрами различной длины. Функция high, введенная в Турбо Паскаль [10, 11], возвращает верхнее возможное значение индекса массива, тогда как нижнее значение всегда равно 0. Однако в вызывающей программе нижняя и верхняя границы индекса могут быть любыми. (Пересчет индексов выполняется транслятором без нашего участия.)

Механизм открытых массивов работает только для одномерных массивов.

6.4.2. Параметры-константы

Параметр-константа указывается в заголовке подпрограммы подобно параметру-значению, однако перед его именем записывается зарезервированное слово const, действие которого распространяется до ближайшей точки с запятой. Параметр-константа передается в подпрограмму как параметр-переменная, но компилятор блокирует любые присваивания параметру-константе нового значения в теле подпрограммы.

В Турбо Паскале можно использовать параметры-переменные и параметры-константы без указания типа, и тогда фактический параметр может быть переменной любого типа, а ответственность за правильность использования того или иного параметра возлагается на программиста.

6.5. Процедурные типы

Для передачи имен функций и процедур в качестве фактических параметров обращения к другим процедурам и функциям фирмой Borland были разработаны *процедурные типы*. Для их объявления используется заголовок подпрограммы, в котором пропущено ее имя. Например:

```
Type
Fun1=function (x,y:real):real;
Fun2=function (x:real):real;
Proc1=procedure (x,y:real; var c,z:real);
```

У передаваемой в процедуру функции должен присутствовать описатель `far`, *указывающий, что процедура будет компилироваться в расчете на дальнюю модель памяти [10, 11].*

В Турбо Паскале есть возможность вызвать функцию и не использовать значение, которое она возвращает, то есть обратиться к ней как к процедуре. Для этого необходимо включить расширенный синтаксис языка. Данный режим действует по умолчанию. Для его включения в тексте программы надо указать директиву компилятора `{$X+}` (знак «+» – включение расширенного синтаксиса, «–» – выключение). Кроме того, в оболочке Паскаля команда **Extended Syntax** диалогового окна **Options/Compiler** может включать/выключать расширенный синтаксис.

Пример 6.6. Рассмотрим механизм передачи подпрограмм в качестве параметра на примере следующей задачи. Программа выводит на экран таблицу значений функций $f(x)$ и $g(x)$. Вычисление и вывод значений осуществляются с помощью функции `VivodFunc`. Ее параметры:

- интервал (`a,b`), на котором будет проводиться расчет;
- количество интервалов (`n`), в узлах которых будет рассчитываться функция;
- имя функции.

```
program test;
{ Описание процедурного типа func. }
type
  func=function(x:real):real;
{ Функция vivod ничего не возвращает в качестве результата. }
function VivodFunc (a,b:real;N:word;ff:func):integer;
var
  x,y,hx:real;
f:text;
begin
{ Шаг изменения переменной x. }
  hx:=(b-a)/N;
  x:=a;
  while(x<=b) do
  begin
     y:=ff(x);
     writeln('x=',x:5:2,' y=',y:7:2);
     x:=x+hx;
  end;
end;
{ Определение функций f и g с описателем far. }
function f(x:real):real;far;
begin
  f:=exp(sin(x))*cos(x);
end;
function g(x:real):real;far;
begin
  g:=exp(cos(x))*sin(x);
end;
begin
{ Вызов Vivodfunc с функцией f в качестве параметра. }
  VivodFunc(0,1,7,f);
  writeln;
{ Вызов Vivodfunc с функцией g в качестве параметра. }
  VivodFunc(0,2,8,g);
end.
```

Нельзя использовать стандартные процедуры и функции из системных библиотек Турбо Паскаля в качестве фактических параметров процедурного типа.

В Турбо Паскале есть возможность использования в качестве формальных параметров *массивов функций.*

Пример 6.7. Рассмотрим модификацию примера 6.6. Программа выводит на экран таблицу значений нескольких функций с помощью функции `VivodFunc`, которой в качестве параметров передают:

- ➤ интервал (`a,b`) – на нем будет проводиться расчет;
- ➤ количество интервалов (`n`), в узлах которых будет рассчитываться функция;
- ➤ массив функций `ff` – в них необходимо вычислить значения;
- ➤ количество функций (`m`) в массиве `ff`.

В тексте программы прокомментированы моменты, отличающие ее от той, которая рассматривается в примере 6.6.

```
program test;
type
  func=function(x:real):real;
{ В процедуру VivodFunc передается }
{ входной параметр ff - открытый массив функций. }
function VivodFunc(a,b:real;N:word;ff:array of func; m:word):integer;
var
  x,y,hx:real;
  i:integer;
begin
  hx:=(b-a)/N;
{ Цикл по всем функциям. }
  for i:=0 to m-1 do
  begin
     x:=a;
     while(x<=b) do
     begin
{ Вычисление значения i-й функции в точке x. }
        y:=ff[i](x);
        writeln('x=',x:5:2,' y=',y:7:2);
        x:=x+hx;
     end;
     writeln;
  end;
end;
function f(x:real):real;far;
begin
  f:=exp(sin(x))*cos(x);
end;
function g(x:real):real;far;
begin
  g:=exp(cos(x))*sin(x);
end;
```

```
{ Описание массива восьми функций fff. }
var fff:array[1..8] of func;
begin
  clrscr;
{ Запись реальных функций в массив fff. }
  fff[1]:=f;
  fff[2]:=g;
{Вызов процедуры.}
  VivodFunc(0,1,7,fff,2);
end.
```

6.6. Рекурсивные подпрограммы

Использование *рекурсивных подпрограмм* – одно из преимуществ языка Паскаль. Под *рекурсией* в программировании понимается вызов подпрограммы из ее тела [9]. Классическими рекурсивными алгоритмами могут быть возведение числа в целую положительную степень, вычисление факториала. В рекурсивных алгоритмах подпрограмма вызывает саму себя до выполнения какого-то условия. Ниже приведены примеры использования рекурсивных функций: вычисления факториала и возведения в степень.

```
Function stepen(a:real;n:word):real;
Begin
{ Если степень числа а равна 0, то результат - число 1. }
  If n=0 then stepen:=1
  Else
{ Иначе для возведения в степень число а надо умножить на а в степени n-1. }
  Stepen:=a*stepen(a,n-1)
End;
Function factorial(n:word):real;
Begin
{ Если N=0 или 1, то факториал равен 1. }
  If n<=1 then factorial:=1
  Else
{ Иначе факториал числа N рассчитывается так N*factorial(N-1). }
  factorial:=N*factorial(N-1)
End;
Var a:real;
  N:integer;
Begin
  writeln('Возведение числа а в степень N');
  write('a=');read(a);
  write('N=');read(N);
```

```
{ Если степень числа a положительная, то просто обращаемся к функции
stepen(a,n). }
  if N>=0 then writeln(stepen(a,n))
{ Если степень числа отрицательная, то вычисляем a в степени n, как
1|a^|n|. }
     else writeln(1/stepen(a,abs(n)));
writeln(' Вычисление N!');
write('N=');read(N);
write(N,'!=',factorial(N))
End.
```

Пример 6.8. Задано число *N*. Сложить все цифры числа N, затем все цифры найденной суммы и повторять эти действия до тех пор, пока не получим цифру, называемую цифровым корнем числа [6]. Разработать функцию для нахождения цифрового корня натурального числа. Задача выделения цифр (разрядов) в числе рассматривалась в главе 3 (см. пример 3.15).

Блок-схема рекуррентной функции вычисления цифрового корня числа N приведена на рис. 6.11.

Текст функции вычисления цифрового корня приведен ниже.

```
function tsifr_koren(N:word):word;
var
  s:word;
begin
{ Если N<=0, то мы получили цифровой корень. }
  if N<=9 then
     tsifr_koren:=N
  else
{ Суммируем все разряды числа. }
     Begin
{ Сумму положим равной нулю. }
        s:=0;
        repeat
{ Очередным разрядом числа будет остаток от деления N на 10. }
           s:=s+N mod 10;
{ Делим число N на 10. }
           N:=N div 10;
{ Если получился 0, то мы просуммировали все разряды числа, иначе
продолжаем выделять разряды числа. }
        until N=0;
{ Нашли сумму разрядов числа и рекуррентно вызываем функцию
tsifr_koren(s). }
     tsifr_koren:=tsifr_koren(s);
  end
end;
```

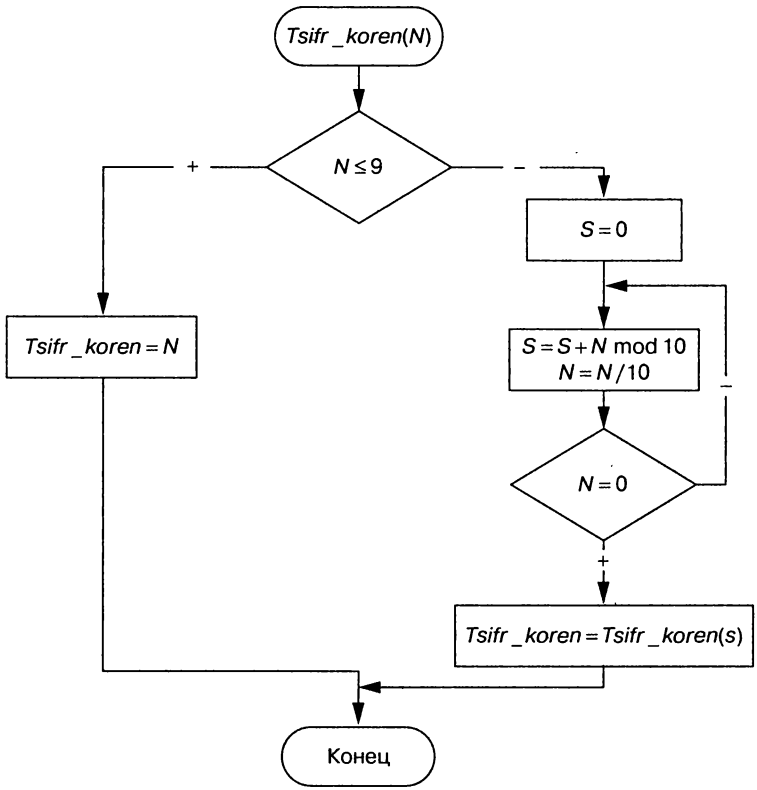


Рис. 6.11 ▾ Блок схема функции вычисления цифрового корня

6.7. Упражнения по теме «Подпрограммы»

Составить программы, используя подпрограммы. Во всех заданиях подпрограммы составить таким образом, чтобы ввод-вывод данных осуществлялся в основной программе.

1. В матрице целых чисел $A(M,N)$ обнулить столбцы, в которых нет простых и совершенных чисел.
2. Упорядочить по убыванию строки матрицы целых чисел $C(M,N)$, где находится 5 простых чисел.
3. В матрице целых чисел $C(M,N)$ упорядочить по возрастанию столбцы, в которых находится наименьшее из совершенных чисел матрицы. Если совершенных чисел нет, оставить матрицу без изменений.

4. В каждой строке матрицы упорядочить по убыванию элементы, расположенные между максимальным и минимальным элементом строки.
5. В матрице целых чисел $C(M,M)$ найти максимальное из совершенных чисел, расположенных на ее диагоналях.
6. В массиве обнулить элементы, расположенные после предпоследнего простого числа.
7. Поменять местами предпоследнее совершенное и третье простое число массива целых чисел $Y(N)$.
8. В массиве целых чисел упорядочить по возрастанию элементы, расположенные между предпоследним совершенным числом и минимальным значением массива.
9. Если массив упорядочен, удалить из него наибольшее из совершенных чисел (учесть, что таких чисел может быть несколько).
10. Задана матрица целых чисел. Поменять местами строки, в которых расположено минимальное простое и максимальное совершенное число.
11. Проверить, является ли матрица $A(N,N)$ обратной к $B(N,N)$.
12. Преобразовать массив таким образом, чтобы сначала располагались простые числа, затем все остальные.
13. Упорядочить предпоследний столбец матрицы целых чисел $G(N,M)$, в котором сумма совершенных чисел меньше максимального элемента.
14. Найти сумму двух наибольших простых чисел матрицы.
15. В массиве целых чисел найти среднее арифметическое простых чисел, расположенных между максимальным и минимальным элементами массива.

Глава 7 Работа с файлами в языке Турбо Паскаль

В данной главе читатель познакомится с возможностями Турбо Паскаля, используемыми для обработки файлов. В программе существует три класса файлов – типизированные, бестиповые и текстовые, – которые позволяют считывать большие объемы данных непосредственно с диска, не вводя их с клавиатуры.

Текстовыми называют файлы, состоящие из любых символов. Они организуются по строкам, каждая из которых заканчивается символом «конец строки». Конец самого файла обозначается символом «конец файла». При записи информации в текстовый файл, просмотреть который можно с помощью любого текстового редактора, все данные преобразуются в символьный тип и хранятся в этом виде.

Файлы, состоящие из компонентов одного типа, число которых заранее не определено и может быть любым, называются *типизированными*. Они заканчиваются символом «конец файла», хранятся в двоичном виде и не просматриваются с помощью текстовых редакторов.

В *бестиповых* файлах информация считывается и записывается блоками определенного размера. В них могут храниться данные любого вида и структуры.

7.1. Описание файловых переменных

Текстовый файл описывается[1] с помощью служебного слова `text`.

```
var f:text;
```

Типизированные файлы могут описываться следующим образом:

```
var имя:file of тип;
```

Предварительно можно определить новый тип данных:

```
type имя=file of тип;
```

Бестиповый файл описывается с помощью служебного слова `file`:

```
var имя:file;
```

Например:

```
type
  massiv=array[1..25]of real;
type
  ff=file of real;
var
  a:text;
  b:ff;
  c:file of integer;
  d:file of massiv;
{ В файле d элементом является массив из 25 вещественных чисел. }
```

7.2. Обработка типизированных файлов

Ниже рассмотрены процедуры и функции, которые используются для работы как с типизированными, так и с текстовыми файлами.

7.2.1. Процедура assign

Для начала работы с файлами необходимо связать файловую переменную в программе с файлом на диске. Для этого используется процедура `assign(f,s)`, где `f` – имя файловой переменной, а `s` – полное имя файла на диске (файл должен находиться в текущем каталоге при условии, что к нему специально не указывается путь).

[1] Вообще говоря, описывается файловая переменная, которая в теле программы с помощью процедуры `Assign` связывается с файлом на диске (см. раздел 7.2.1).

Например:

```
var
  f:file of real;
begin
  assign (f,'d:\tp\tmp\abc.dat');
```

7.2.2. Процедуры reset, rewrite

После установления связи между файловой переменной и именем файла на диске нужно открыть файл, воспользовавшись процедурами `reset` или `rewrite`.

Когда будет выполнена процедура `reset(f)`, где `f` – имя файловой переменной, файл будет открыт для чтения и станет доступен его первый элемент. Далее можно выполнять чтение и запись информации из файла.

Файл можно открыть для записи и очистить при помощи процедуры `rewrite(f)`, где `f` – имя файловой переменной. Она открывает и очищает файл (то есть удаляет из него информацию), после чего его можно использовать для записи.

7.2.3. Процедура close

Процедура `close(f)`, где `f` – имя файловой переменной, закрывает файл, который ранее был открыт процедурами `rewrite`, `reset`. Именно ее следует использовать при закрытии файла, в который была записана информация. Дело в том, что `write` не обращается непосредственно к диску, а пишет информацию в специальный участок памяти, называемый *буфером файла*. После того как буфер заполнится, вся информация из него вносится в файл. При выполнении операции `close` сначала происходит запись буфера файла на диск, и только потом файл закрывается. Если его не закрыть вручную, то это произойдет автоматически при завершении работы программы, однако пропадет информация, хранимая в буфере файла.

После записи информации в файл его необходимо закрывать с помощью процедуры `close`.

7.2.4. Процедура rename

Переименование файла, связанного с файловой переменной `f`, осуществляется в то время, когда он закрыт, при помощи процедуры `rename(f,s)`, где `f` – файловая переменная, `s` – новое имя файла (строковая переменная).

7.2.5. Процедура erase

Удаление файла, связанного с переменной f, выполняется посредством процедуры erase(f), в которой f также является именем файловой переменной. Для корректного выполнения этой операции файл должен быть закрыт.

7.2.6. Функция eof

Функция eof (f) (end of file), где f – имя файловой переменной, принимает значение «истина» (true), если достигнут конец файла, иначе – «ложь» (false).

7.2.7. Процедуры write, read

Для чтения информации из файла, связанного с файловой переменной f, можно воспользоваться стандартными операторами чтения следующей структуры:

```
read(f,x1,x2,x3,...,xn);
read(f,x);
```

Операторы последовательно считывают компоненты из файла в указанные переменные.

Процедура read *не проверяет, достигнут ли конец файла. За этим нужно следить с помощью функции* eof.

Для записи в файл можно применять стандартные операторы записи следующей структуры:

```
write(f,x1,x2,...,xn);
write(f,x);
```

Операторы последовательно записывают в файл значения переменных.

Типы файла и переменных должны совпадать.

Для того чтобы *создать файл*, необходимо выполнить следующие действия:

1. Описать файловую переменную.
2. Связать ее с физическим файлом (assign).
3. Открыть файл для записи (rewrite).
4. Внести необходимую информацию в файл (write).
5. *Обязательно* закрыть файл (close).

Для выполнения *считывания информации* из файла надо:

1. Описать файловую переменную.
2. Связать ее с физическим файлом.

3. Открыть файл для чтения.
4. Считать необходимую информацию (на этом этапе нужно проверять, достигнут ли конец файла).
5. Закрыть файл.

Рассмотрим несколько примеров.

Пример 7.1. Записать n действительных чисел в файл.

```
program abc;
var
  f:file of real;
  a:real;
  i,n :integer;
begin
{ Связываем файловую переменную с файлом на диске. }
  assign (f,'d:\tp\abc.dat');
{ Открываем пустой файл для записи. }
  rewrite(f);
{ Определяем количество элементов в файле. }
  read(n);
{ В цикле вводим очередной элемент и записываем его в файл. }
  for i:=1 to n do
  begin
     write('a=');
     read(a);
     write(f,a)
  end;
{ Закрываем файл. Здесь это обязательно. }
  close (f);
end.
```

Пример 7.2. На диске D в каталоге TP находится файл вещественных чисел. Необходимо распечатать его содержимое и вычислить количество компонентов файла.

```
program bca;
var
  f1:file of real;
  a:real;
  n:integer;
begin
{ Связываем файловую переменную с файлом на диске. }
  assign(f1,'D:\TP\abc1.dat');
{ Открываем файл. }
  reset(f1);
{ В переменной n будет накапливаться количество элементов в файле. }
  n:=0;
```

```
{ Вход в цикл осуществляется, если не достигнут конец файла. }
  while not eof (f1) do
  begin
{ Считываем очередной элемент из файла. }
     read(f1,a);
{ Увеличиваем количество элементов в файле на один. }
     n:=n+1;
{ Выводим очередной элемент на экран. }
     writeln(n, '-й элемент файла равен ', a:10:6)
  end;
  writeln;
  writeln ('в файле ',n,' элементов');
{ Закрываем файл. Здесь это необязательно. }
  close(f1);
end.
```

7.3. Последовательный и прямой доступ к файлам

Файл – последовательная структура данных. После его открытия доступен первый компонент. Можно последовательно считывать или записывать один компонент файла за другим. Допустим, необходимо считать пятнадцатый, а затем первый элементы. С помощью последовательного доступа к файлу это можно сделать следующим образом:

```
reset(f);
for i:=1 to 15 do
  read(f,b);
reset(f);
read(f,a);
```

Как видно, такое чтение компонента из файла, а затем повторное открытие файла – не самый удачный способ. Гораздо удобнее использовать встроенные процедуры и функции Турбо Паскаля для прямого доступа к компонентам файла, что означает возможность определять позицию интересующего нас компонента внутри файла и указывать непосредственно на него. При прямом доступе к файлу его компоненты нумеруются от 0 до n–1, где n – число компонентов в файле. Самый первый компонент имеет номер 0.

7.3.1. Функция filesize

Функция `filesize(f)`, где `f` – файловая переменная, возвращает значение типа `longint`, то есть число реальных компонентов в открытом файле `f`. Для пустого файла она вернет 0.

7.3.2. Функция filepos

Функция `filepos(f)` возвращает значение типа `longint` – текущую позицию в файле `f`, который должен быть открыт. Если файл только что открылся, то `filepos(f) = 0`. После прочтения последнего компонента из файла значение `filepos(f)` совпадает со значением `filesize(f)`; что указывает на достижение конца файла. Последнее можно проверить еще и так:

```
if filepos(f)=filesize(f) then
  writeln('достигнут конец файла');
```

Пример 7.3. Вычислить количество компонентов в файле вещественных чисел, вывести содержимое файла на экран.

```
var
  f:file of real;
  a:real;
  i:word;
begin
  assign(f,'abc.dat');
  reset(f);
  writeln('в файле',filesize(f), 'чисел');
  for i:=1 to filesize(f) do
  begin
     read(f,a);
     write(a, ' ')
  end;
  close(f)
end.
```

7.3.3. Процедура seek

Процедура `seek(f,n)` устанавливает указатель в открытом файле `f` на компонент с номером `n` (нумерация компонентов идет от 0). Затем значение компонента может быть считано.

Пример 7.4. На диске E в каталоге ABC есть файл целых чисел a2.int. Необходимо поменять местами его максимальный и минимальный элементы.

Ниже приведены два варианта этой программы. В первом после считывания компонентов файла в массив происходит поиск минимального и максимального элементов массива и их индексов. Затем максимальное и минимальное значения перезаписываются в файл.

Вторая программа работает по другому алгоритму. Организован один цикл, в котором очередное значение считывается в переменную. Здесь же осуществляется поиск минимального и максимального элементов среди компонентов

файла и их индексов. Затем происходит перезапись в файл максимального и минимального значений.

Вариант 1:

```
program var_1;
var
  f:file of integer;
  i, max, nmax, min, nmin :integer;
  a:array[0..200] of integer;
begin
  assign(f,'e:\abc\a2.int');
  reset(f);
{ Считываем компоненты файла в массив a. }
  for i:=0 to filesize(f)-1 do read(f,a[i]);
{ Начальное присваивание максимального и }
{ минимального элементов массива и их индексов. }
  max:=a[0]; nmax:=0;
  min:=a[0]; nmin:=0;
{ Основной цикл для поиска максимального и }
{ минимального элементов массива и их индексов. }
  for i:=1 to filesize(f)-1 do
  begin
     if a[i]>max then
     begin
        max:=a[i];
        nmax:=i
     end;
     if a[i]<min then
     begin
        min:=a[i];
        nmin:=i
     end;
  end;
{ Перезапись максимального и минимального значений в файл. }
  seek(f,nmax);
  write(f,min);
  seek(f,nmin);
  write(max);
  close(f)
end.
```

Вариант 2:

```
var
  f:file of integer;
  a,i,max,nmax,min,nmin:integer;
```

```
begin
  assign(f, 'e:\abc\a12.int');
  reset(f);
  for i:=0 to filesize(f)-1 do
  begin
     read(f,a);
     if i=0 then
{ Начальное присваивание максимального и }
{ минимального значений и его индекса. }
     begin
        max:=a; nmax:=i;
        min:=a; nmin:=i;
     end
     else
     begin
{ Сравнение текущего значения с максимальным (минимальным). }
     if max<a then
     begin
        max:=a;
        nmax:=i
     end;
     if min>a then
     begin
        min:=a;
        nmin:=i
     end
  end
end;
{ Перезапись максимального и минимального значений в файл. }
seek(f,nmax);
write(f,min);
seek(f,nmin);
write(f,max);
end.
```

7.3.4. Процедура truncate

Процедура `truncate(f)`, где `f` – имя файловой переменной, отсекает часть открытого файла, начиная с текущего компонента, и подтягивает на его место конец файла.

Пример 7.5. Задан файл вещественных чисел abc.dat. Удалить из него максимальный и минимальный элементы.

Рассмотрим две программы, решающие эту задачу. Алгоритм первой состоит в следующем: считываем компоненты файла в массив, в котором находим

максимальный и минимальный элементы и их номера. Открываем файл для записи и вносим в него все элементы, за исключением максимального и минимального.

```
program var3;
  var
  f:file of real;
  max,min:real;
  j, i,nmax,nmin:integer;
  a:array [1..300] of real;
begin
  assign(f,'abc.dat');
  reset(f);
{ Запоминаем в переменной j количество компонентов в файле. }
  j:=filesize(f);
{ Считываем компоненты файла в массив a. }
  for i:=1 to filesize(f) do read(f,a[i]);
  close(f);
{ Открываем файл для записи. }
  rewrite(f);
{ Начальное присваивание максимального и }
{ минимального элементов массива и их индексов. }
  max:=a[1];min:=a[1];
  nmax:=1;nmin:=1;
{ Основной цикл для поиска максимального и }
{ минимального элементов массива и их индексов. }
  for i:=2 to filesize(f) do
  begin
     if a[i]>max then
     begin
        max:=a[i];
        nmax:=i
     end;
     if a[i]<min then
     begin
        min:=a[i];
        nmin:=i;
     end;
  end;
{ Перезапись элементов массива в файл, }
{ за исключением элементов с номерами nmax и nmin. }
  for i:=1 to filesize(f) do
     if (i<>nmax)and (i<>nmin) then
        write(f,a[i]);
  close(f)
end.
```

Вторая программа работает следующим образом. Находим максимальный и минимальный элементы и их номера среди компонентов файла (описание программы var2). Если nmin > nmax, то меняем содержимое переменных nmin и nmax. Далее элементы, лежащие между минимальным и максимальным (формально между элементами с номерами nmin и nmax), сдвигаем на один порядок влево. Тем самым мы убираем элемент с номером nmin. После этого все компоненты, лежащие после элемента с номером nmax, сдвинем на два порядка влево. Этим мы сотрем максимальный элемент. Затем два последних компонента в файле необходимо удалить.

```
program var4;
var
  f:file of real;
  a:real;
  max,min:real;
  i,nmax,nmin:integer;
begin
  assign(f,'abc.dat'); reset (f);
{ Поиск максимального и минимального элементов в файле и их индексов. }
  for i:=0 to filesize(f)-1 do
  begin
     read(f,a);
     if i=0 then
     begin
        max:=a;
        nmax:=i;
        min:=a;
        nmin:=i
     end
     else
     begin
        if a>max then
        begin
           max:=a;
           nmax:=i;
        end;
        if a<min then
        begin
           min:=a;
           nmin:=i;
        end
     end
  end;
```

```
{ Сравниваем nmin и nmax. }
  if nmax<nmin then
  begin
     i:=nmax;
     nmax:=nmin;
     nmin:=i
  end;
{ Сдвигаем элементы, лежащие между компонентами }
{ с номерами nmin и nmax, на один влево. }
  for i:=nmin to nmax-2 do
  begin
     seek(f,i+1);
     read(f,a);
     seek(f,i);
     write(f,a)
  end;
{ Сдвигаем элементы, лежащие после компонента }
{ с номером nmax, на два влево. }
  for i:=nmax to filesize(f)-3do
  begin
     seek(f,i+1);
     read(f,a);
     seek(f,i-1);
     write(f,a);
  end;
{ Отрезаем последние два компонента. }
  truncate(f);
  close(f);
end.
```

7.4. Обработка ошибок ввода-вывода

Компилятор Турбо Паскаля позволяет генерировать код программы в двух режимах: с проверкой корректности ввода–вывода и без нее. В оболочке этот режим включается при последовательном выполнении команд **Options/Compiler/I/O checking**.

В программу может быть включен ключ режима компиляции. Кроме того, предусмотрен перевод контроля ошибок ввода–вывода из одного состояния в другое:

- `{$I+}` – режим проверки ошибок ввода–вывода включен;
- `{$I-}` – режим проверки ошибок ввода–вывода отключен.

По умолчанию, как правило, действует режим {$I+}. Можно многократно включать и выключать режимы, создавая области с контролем ввода и без него. Все ключи компиляции описаны в приложении.

При включенном режиме проверки ошибка ввода–вывода будет фатальной – программа прервется, выдав номер ошибки [9–11].

Если убрать режим проверки, то при возникновении ошибки ввода–вывода программа не будет останавливаться, а продолжит работу со следующего оператора. Результат операции ввода–вывода будет не определен.

Для опроса кода ошибки лучше пользоваться специальной функцией Турбо Паскаля IOResult, однако необходимо помнить, что опросить ее можно только один раз после каждой операции ввода или вывода: данная функция обнуляет свое значение при каждом вызове. IOResult возвращает целое число, соответствующее коду последней ошибки ввода–вывода. Если IOResult = 0, то при вводе–выводе ошибок не было, иначе IOResult возвращает код ошибки. Некоторые коды ошибок приведены в табл. 7.1.

Таблица 7.1 ▾ Описание кодов возврата ошибок ввода–вывода

Код ошибки	Описание
2	Файл не найден
3	Путь не найден
4	Слишком много открытых файлов
100	Ошибка чтения с диска
101	Ошибка записи на диск
102	Файл не связан
103	Файл не открыт
104	Файл не открыт для ввода
105	Файл не открыт для вывода
106	Неправильный числовой формат

Рассмотрим несколько практических примеров обработки ошибок ввода–вывода [9]:

1. При открытии проверить, существует ли заданный файл и возможно ли чтение данных из него.

```
assign (f, 'abc.dat');
{$I-}
reset(f);
{$I+}
if IOResult<>0 then
```

```
 writeln ('файл не найден или не читается')
else
begin
 read(f,...);
 …
 close(f);
end;
```

2. Проверить, является ли вводимое с клавиатуры число целым.

```
var i:integer;
begin
{$I-}
repeat
 write('введите целое число i');
 readln(i);
until (IOResult=0);
{$I+}
{ Этот цикл повторяется до тех пор, пока не будет введено целое число. }
end.
```

7.5. Обработка бестиповых файлов

В Турбо Паскале есть особый обобщенный тип файла, который можно назвать *бестиповым*. В этом файле можно хранить данные любого типа. Описание бестиповой файловой переменной происходит следующим образом:

```
var имя:file;
```

После описания такой файловой переменной она обычным способом с помощью процедуры `Assign` должна быть связана с реальным файлом на диске. При открытии бестиповых файлов следует использовать расширенный синтаксис процедур `reset` и `rewrite`.

```
Reset(var f: File; BuferSize:word);
Rewrite(var f: File; BuferSize:word);
```

`BufSize` определяет размер блока передачи данных – количество байт, считываемых или записываемых в файл данных за одно обращение к нему. Если этот параметр отсутствует, то используется значение, устанавливаемое по умолчанию (128) при открытии файла. Для большей гибкости рекомендуем устанавливать размер блока 1 байт.

Для записи данных в бестиповый файл используется процедура `BlockWrite`:

```
BlockWrite(var f:file; var X; Count:word; var WriteCount:word);
```

где f – имя файловой переменной;

X – имя переменной, из которой данные записываются в файл;

Count – количество блоков размером BufSize, записываемых в файл;
WriteCount - количество блоков размером BufSize, записанных в файл.

BlockWrite записывает в файл, связанный с файловой переменной f, Count блоков[1] из переменной X. При корректной записи в файл возвращаемое процедурой BlockWrite значение WriteCount должно совпадать с Count.

При чтении данных из бестипового файла используется процедура BlockRead:

```
BlockRead(var f:file; var Y; Count:word; var ReadCount:word);
```

где f – имя файловой переменной;
Y – имя переменной, в которую считываются данные из файла;
Count – количество блоков размером BufSize, считываемых из файла;
ReadCount - количество блоков размером BufSize, считанных из файла.

BlockRead считывает из файла, связанного с файловой переменной f, Count блоков в переменную Y. При корректном чтении из файла возвращаемое процедурой BlockRead значение ReadCount должно совпадать с Count.

Параметры ReadCount и WriteCount в процедурах BlockRead и BlockWrite соответственно не являются обязательными.

Рассмотрим работу с бестиповыми файлами на примере следующей задачи.

Пример 7.6. В файле Pr_7_6.dat хранятся матрицы вещественных чисел $A(N,M)$, $B(P,L)$ и их размеры. Умножить, если это возможно, A на B, результат дописать в файл Pr_7_6.dat.

Сначала напишем программу создания файла. Для этого определимся со структурой файла. Пусть в нем хранятся два значения типа word N и M, затем матрица вещественных чисел $A(N,M)$, потом P и L типа word и матрица $B(P,L)$.

Ниже приведен листинг программы создания бестипового файла с комментариями:

```
var
  f:file;
  i,j,n,m,l,p:word;
  a,b:array[1..20,1..20] of real;
begin
  Assign(f,'Pr_7_6.dat');
{ Открываем файл для записи, размер блока передачи данных 1 байт. }
  rewrite(f,1);
  Write('N='); readln(N);
  write('M='); readln(M);
```

[1] Размер блока определяется при выполнении процедур reset, rewrite.

```
{ Записываем в файл f целое число N, а точнее 2 блока (sizeof(word)=2) по
одному байту из переменной N. }
  BlockWrite(f,N,sizeof(word));
{ Записываем в файл f целое число M, а точнее 2 блока (sizeof(word)=2) по
одному байту из переменной M. }
  BlockWrite(f,M,sizeof(word));
writeln('Matrica A');
  for i:=1 to N do
  for j:=1 to M do
  begin
{ Вводим очередной элемент матрицы. }
     read(A[i,j]);
{ Записываем в файл f вещественное число A[i,j], а точнее 6 блоков
(sizeof(real)=6) по одному байту из переменной A[i,j]. }
     BlockWrite(f,A[i,j],sizeof(real));
  end;
  write('L='); readln(L);
  write('P='); readln(P);
{ Записываем в файл 2 блока по одному байту из переменной L. }
  BlockWrite(f,L,sizeof(word));
{ Записываем в файл 2 блока по одному байту из переменной P. }
  BlockWrite(f,P,sizeof(word));
writeln('Matrica B');
  for i:=1 to L do
  for j:=1 to P do
  begin
     read(B[i,j]);
{ Записываем в файл 6 блоков по одному байту из переменной B[i,j]. }
     BlockWrite(f,B[i,j],sizeof(real));
  end;
{ Закрываем файл. }
  close(f);
end.
```

Теперь напишем программу, которая из бестипового файла считает матрицы *A*, *B*, их размеры и вычислит матрицу *C* как произведение *A* на B^1, после чего матрицу *C* допишем в бестиповый файл.

```
var
  f:file;
  k,i,j,n,m,l,p:word;
  a,b,c:array[1..20,1..20] of real;
begin
{ Связываем файловую переменную с файлом на диске. }
  Assign(f,'Pr_7_6.dat');
```

[1] Программа умножения матриц рассматривалась в главе 5.

```
{ Открываем файл, и устанавливаем размер блока в 1 байт. }
  reset(f,1);
{ Считываем в переменную N из файла f 2 байта ( sizeof(word)=2 блоков по
одному байту). }
  BlockRead(f,N,sizeof(word));
{ Считываем в переменную M из файла f 2 байта ( sizeof(word)=2 блоков по
одному байту). }
  BlockRead(f,M,sizeof(word));
  for i:=1 to N do
  for j:=1 to M do
{ Считываем в переменную A[i,j] из файла f 6 байт ( sizeof(real)=6 блоков
по одному байту). }
     BlockRead(f,A[i,j],sizeof(real));
{ Считываем в переменную L из файла f 2 байта. }
     BlockRead(f,L,sizeof(word));
{ Считываем в переменную P из файла f 2 байта. }
  BlockRead(f,P,sizeof(word));
  for i:=1 to L do
  for j:=1 to P do
{ Считываем в переменную B[i,j] из файла f 6 байт. }
     BlockRead(f,B[i,j],sizeof(real));
{ Вывод матрицы A на экран. }
  writeln('Matrica A');
  for i:=1 to N do
  begin
     for j:=1 to M do
        write(A[i,j]:1:2,' ');
     writeln
  end;
{ Вывод матрицы B на экран. }
  writeln('Matrica B');
  for i:=1 to L do
  begin
     for j:=1 to P do
        write(B[i,j]:1:2,' ');
     writeln
  end;
{ Проверяем, возможно ли умножение матриц, если да, }
  if m=l then
  begin
{ то умножаем матрицу A на B. }
     for i:=1 to N do
     for j:=1 to P do
        begin
        c[i,j]:=0;
        for k:=1 to M do
```

```
            c[i,j]:=c[i,j]+a[i,k]*b[k,j]
          end;
{ Вывод матрицы C. }
     writeln('Matrica C=A*B');
     for i:=1 to N do
     begin
     for j:=1 to P do
        write(C[i,j]:1:2,' ');
     writeln
     end;
{ Записываем в файл f целое число N, а точнее 2 блока по одному байту из
переменной N. }
     BlockWrite(f,N,sizeof(word));
{ Записываем в файл f целое число P, а точнее 2 блока по одному байту из
переменной P. }
     BlockWrite(f,P,sizeof(word));
     for i:=1 to N do
        for j:=1 to P do
{ Записываем в файл f вещественное число C[i,j], а точнее 6 блоков по
одному байту из переменной C[i,j]. }
           BlockWrite(f,c[i,j],sizeof(real));
     end
  else
  writeln('Умножение невозможно');
{ Закрываем файл. }
  close(f);
end.
```

7.6. Работа с текстовыми файлами

При работе с текстовыми файлами действие процедур `reset`, `rewrite`, `close`, `rename`, `erase` и функции `eof` аналогично их действию при работе с компонентными (типизированными) файлами.

Процедуры `seek`, `truncate` *и функции* `filepos` *не работают с текстовыми файлами.*

При работе с текстовыми файлами можно пользоваться процедурой `append`.

Процедура `append(f)`, где `f` – имя файловой переменной, служит для специального открытия файлов для записи. Она применима только к уже физически существующим файлам, открывает и готовит их для добавления информации в конец файла.

Запись в текстовый файл и чтение осуществляются с помощью процедур write, writeln, read, readln следующей структуры:

```
read(f, x1, x2, x3,..., xn);
read(f, x);
readln(f, x1, x2, x3,..., xn);
readln(f, x);
write(f, x1, x2, x3,..., xn);
write(f, x);
writeln(f, x1, x2, x3,..., xn);
writeln(f, x);
```

В этих операторах смысл переменных f, x1, x2, x3, ..., xn такой же, как и при работе с типизированными файлами.

Однако имеется ряд особенностей при работе операторов write, writeln, read, readln с текстовыми файлами. Имена переменных могут быть целого, вещественного, символьного и строкового типа. Перед записью данных в текстовый файл с помощью процедуры write происходит их преобразование в тип string. Действие оператора writeln отличается тем, что он записывает в текстовый файл символ конца строки.

При чтении данных из текстового файла с помощью процедур read и readln происходит преобразование из строкового в нужный тип данных. Если преобразование невозможно, то генерируется код ошибки, значение которого можно узнать, обратившись к функции IOResult.

При работе с текстовым файлом необходимо помнить специальные правила чтения значений переменных:

- когда вводятся числовые значения, два числа считаются разделенными, если между ними есть хотя бы один пробел, символ табуляции или символ конца строки;
- при вводе строк начало текущей строки идет сразу за последним введенным до этого символом. Вводится количество символов, равное объявленной длине строки. Если при чтении встретился символ «конец строки», то работа с этой строкой заканчивается. Сам символ конца строки является разделителем и в переменную никогда не считывается;
- процедура readln считывает значения текущей строки файла, курсор переводится в новую строку файла, и дальнейший ввод осуществляется с нее.

Пример 7.7. Написать программу сложения двух матриц, результирующую матрицу вывести на экран дисплея и в текстовый файл.

```
program abc;
var
  f:text;
  a,b,c:array[1..5,1..5] of real;
  i,j,k:integer;
begin
  assign(f, 'abcd.txt');
{ Открываем пустой текстовый файл. }
  rewrite (f);
  writeln('Введите матрицу A');
  for i:=1 to 5 do
     for j:=1 to 5 do
        read(a[i,j]);
  writeln('Введите матрицу B');
  for i:=1 to 5 do
     for j:=1 to 5 do
        read(b[i,j]);
  writeln(' Matrica C=A*B');
  writeln(f,' Matrica C=A*B');
  for i:=1 to 5 do
  begin
     for j:=1 to 5 do
     begin
        c[i,j]:=c[i,j]+a[i,j]+b[i,j];
        write (c[i,j]:8:3,' ');
{ Выводим очередной элемент в текстовый файл. }
        write (f,c[i,j]:8:3,' ');
     end;
     writeln;
{ Переводим курсор в файле на новую строку. }
     writeln(f);
  end;
  close(f);
end.
```

В качестве текстовых файлов могут использоваться физические устройства: клавиатура, дисплей, последовательные и параллельные порты. Они имеют фиксированные имена и во многом схожи с файлами. В табл. 7.2 приведены имена основных устройств MS DOS.

Физические файлы-устройства организуются как текстовые файлы. Для нормальной работы их надо связывать с текстовым логическим файлом и осуществлять операции ввода–вывода с этим файлом. Имена физических устройств следует записывать без точек, двоеточий, запятых и т.д. Например: `assign(f,'prn'); rewrite(f);` но не так – `assign(f, 'prn.'); rewrite(f)`. В этом случае речь идет о файле с именем prn, а не о принтере.

Таблица 7.2 ▾ Имена основных устройств MS DOS

Имя	Название устройства	Назначение
Con	Клавиатура и экран	Ввод из Con – это чтение с клавиатуры; вывод в Con – запись на экран
Prn	Принтер	Вывод на принтер
lpt1	Параллельные порты	Через эти имена файлов происходит вывод данных на принтер или другие устройства
lpt2	Параллельные порты	
lpt3	Параллельные порты	
com1	Последовательные порты	Подключение к портам
com2	Последовательные порты	
Nul	Фиктивное устройство	Это «бездонный» файл, принимающий что угодно, но всегда пустой

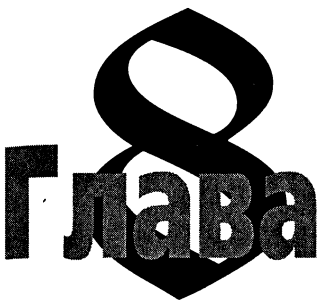
Рассмотрим пример построчного вывода матрицы на принтер:

```
Var
  f:text;
  i,j:integer;
  A:array[1..20,1..20] of real;
Begin
{ Связываем файловую переменную с принтером. }
  assign(f,'prn');
{ Открываем вывод на принтер. }
  rewrite(f);
  writeln (f,'Матрица A');
  for i:=1 to 5 do
  begin
     for j:=1 to 5 do
        write(f,a[i,j]:10:5,' ');
     writeln(f);
  end;
{ Закрываем файл, с помощью которого осуществлялся вывод на принтер. }
  close(f);
end.
```

7.7. Упражнения по теме «Работа с файлами в языке Турбо Паскаль»

Во всех заданиях составить две программы. Первая должна сформировать типизированный файл исходных данных. Вторая – считать данные из этого файла, выполнить соответствующие вычисления и записать их результаты в текстовый файл.

1. Из компонентов исходного файла целых чисел сформировать массивы четных и нечетных чисел. Определить наибольший четный компонент файла и наименьший нечетный.
2. На основе исходного файла целых чисел создать массив удвоенных нечетных чисел. Упорядочить его по возрастанию элементов.
3. Сформировать массив положительных чисел, делящихся на пять без остатка, используя элементы исходного файла целых чисел. Упорядочить массив по убыванию элементов.
4. Из компонентов исходного файла целых чисел сформировать массивы отрицательных и простых чисел. Вычислить количество нулевых компонентов файла.
5. Взяв за основу файл целых чисел, создать массив, элементы которого не являются простыми числами и расположены до минимального элемента.
6. Из компонентов исходного файла целых чисел сформировать массив, записав в него только ненулевые компоненты, находящиеся после максимального элемента.
7. Создать массив из элементов исходного файла вещественных чисел, внеся в него числа, превосходящие среднее значение положительных компонентов файла.
8. Из компонентов исходного файла сформировать массив, записав в него числа, расположенные в файле до минимального элемента и после максимального.
9. Массив создать из компонентов исходного файла. Внести в него числа, расположенные в файле между минимальным и максимальным элементами.
10. Из компонентов исходного файла сформировать массив, в котором вначале расположить четные, а затем нечетные числа. Определить номера наибольшего нечетного и наименьшего четного компонентов.
11. Из файла целых чисел удалить минимальное среди совершенных чисел и наибольшее простое число.
12. Из файла вещественных чисел удалить элементы, большие среднего арифметического.
13. Из файла целых чисел удалить второе, третье и пятое простое число.
14. В файле целых чисел поменять местами максимальное среди отрицательных и наибольшее среди простых.
15. Из файла целых чисел переписать все простые, бóльшие среднего арифметического в новый файл.



Обработка строк в языке Турбо Паскаль

В примерах задач, рассмотренных нами ранее, происходила обработка, в основном, числовой информации. Но в памяти компьютера могут обрабатываться не только числа, но и символы. Символ занимает в памяти один байт и описывается в тексте программы служебным словом char. Значение символьной переменной – это символ, заключенный в апострофы, например 'S', 'Ф', '@', '7'.

```
var
x,y:char;
begin
  read(x); { Ввод символьной переменной. }
  y:= 'B'; { Присвоение значения символьной переменной. }
  if x=y then
     writeln(x)  { Печать символьной переменной. }
  else
     writeln(y);
end.
```

Символы, как и любые однотипные данные, могут быть объединены в массивы. В описании можно задать строку символов или таблицу, состоящую из строк символов:

```
A:array [1..20] of char; { Строка из 20 символов. }
B: array [1..10,1..15] of char;{ 10 строк по 15 символов каждая. }
```

Обработка символьных массивов практически не отличается от обработки числовых.

Пример 8.1. Слово задано как массив символов. Получить новое слово, состоящее из символов исходного, записанных в обратном порядке.

Поскольку слово задано таким образом, это означает, что каждый символ имеет свой порядковый номер в массиве. Следовательно, решение задачи сводится к тому, что необходимо создать новый массив, каждый элемент которого будет формироваться так, как показано в таблице 8.1.

Таблица 8.1 ▼ Процесс записи элементов массива в обратном порядке

Формирование массива *B*	Формула пересчета индексов
$B_1 = A_n$	$B_i = A_{n-i+1}$
$B_2 = A_{n-1}$	
$B_3 = A_{n-2}$	
...	
$B_{n-2} = A_3$	
$B_{n-1} = A_2$	
$B_n = A_1$	

Текст программы можно записать следующим образом:

```
var a,b:array [1..10] of char;
  c:char;
  i,n:byte;
begin
  write('Введите длину слова');
  readln(n);
  for i:=1 to n do
     read(a[i]);{ Ввод слова. }
  for i:=1 to n do
     b[i]:=a[n-i+1];{ Формирование нового слова. }
  for i:=1 to n do
     write(b[i]);{ Вывод полученного слова. }
end.
```

Кроме того, символы можно объединять в множества. *Множество* – это типичный представитель группы структурных типов. Оно задает интервал значений, который является множеством всех подмножеств базового типа. Любой конечный скалярный тип, состоящий не более чем из 256 элементов, может быть базовым типом.

Константы множественного типа записывают с помощью квадратных скобок и списка элементов. Например:

```
abc=['A','B','C'];{ Множество из трех букв. }
cifr=[0..9]; { Множество цифр. }
p=[];
```

Переменная abc может принимать значения из набора: {('A','B','C'), ('A','B'), ('A','C'), ('B','C'), ('A'), ('A'), ('B'), ('C'), ()}.

Отсутствие списка элементов в квадратных скобках обозначает пустое множество.

Описывают множества, используя ключевое слово set:

```
Var
M: set of Char;
```

Присвоить значение переменной множественного типа можно, например, так:

```
M:=['a'..'y']
```

Для того чтобы определить, принадлежит ли элемент множеству, используют зарезервированное слово in:

```
if 's' in M then …
```

Над множествами в языке Паскаль можно выполнять операции объединения (+), пересечения (*), разности (–), а так же операции отношений (=, < >, <, >, <= , >=). Например, используя операцию объединения, к множеству M можно добавить символ 'z', представив его как множество, состоящее из одного элемента:

```
M:=M+['z'];
```

Ту же операцию выполняет процедура Include(M, 'z'). А процедура Exclude предназначена для исключения элемента из множества. Например, Exclude(M, 's') исключит символ 's' из множества M.

Однако удобнее всего обрабатывать символьную информацию, представленную в виде строки[1]. Фактически строка представляет собой массив из n+1 символов:

```
string[n]=array [0..n] of char;
```

Нулевой символ предназначен для указания используемого количества символов строки – там хранится символ, код которого равен длине строки.

[1] Определение и описание строк дано в главе 2.

8.1. Операции над строками

Для строк определены операции конкатенации (+) и сравнения (=, < >, < , >, <=, >=). Например:

```
Var
  s1,s2,s3,s:string[80];
begin
  s1:='turbo';
  s2:='pascal';
  s3:='7.0';
{ В переменной s хранится строка символов "turbo pascal 7.0". }
  s:=s1+' '+s2+' '+s3;
end;
```

Сравнение строк производится слева направо до первого несовпадающего символа: длиннее считается та строка, в которой первый несовпадающий символ имеет больший номер в таблице кодов (буквы в таблице кодов расположены по возрастанию): `'abd'>'abc'`.

Если строки имеют различную длину, но в общей части символы совпадают, считается, что более короткая строка меньше: `'abcd'<'abcdef'`.

Обращение к элементу строки аналогично обращению к элементу массива: `s[6]`, `s[8]`, `s[10]`.

При вводе строковых переменных оператором `read` необходимо ввести количество символов, соответствующее описанной длине строки. При этом удобнее пользоваться оператором `readln`, который осуществляет ввод переменной `s` до конца строки. Обратите внимание на то, что при вводе текста пробел – это такой же символ, как и любой другой.

8.2. Процедуры и функции обработки строк

Над строками в Паскале определены следующие процедуры и функции.

Функция `pos(s,st)` – функция поиска. Она определяет, с какой позиции строка `s` входит в строку `st`. Если вхождение имеет место, то результатом работы функции будет номер символа в строке `st`, с которого начинается строка `s`. Если вхождения нет, то результат – ноль.

Процедура `delete(st,p,n)` удаляет из строки `st` (тип `string`) `n` (тип `integer`) символов начиная с позиции `p` (тип `integer`). Например:

```
st:='turbo pascal 7.0';
delete(st,6,11);
```

Процедура insert(s,subs,n) вставляет в строку s (тип string) подстроку subs (тип string) начиная с позиции n (тип integer). Например:

```
s1:='pascal';
s:='turbo 7.0';
insert(s,s1,7);
```

Функция length(s) возвращает текущую длину (тип integer) строки s (тип string).

Функция copy(s,n,l) возвращает подстроку длиной l (тип integer) начиная с позиции n (тип integer) строки s (тип string).

Процедура str(x[:f] [:n],s) преобразует числовое значение x (тип real) в строку s (возможно задание формата; в этом случае f – число позиций в числе, n – количество позиций после точки).

Процедура val(s,x,err) превращает строковое значение s (тип string) в числовую переменную x (тип real), err (тип integer) возвращает номер позиции, в которой произошла ошибка преобразования, или 0, если ошибки не было.

Пример 8.2. Решим задачу из примера 8.1, определив слово как строку символов.

Пусть a – строка, определяющая исходное слово, а b – обращенная строка. Для решения данной задачи воспользуемся функцией length(a) и определим длину заданного слова. Далее вспомним формулу пересчета индексов, выведенную в примере 8.1, и воспользуемся функцией copy. Передавая в эту функцию исходную строку a, будем «вырезать» из нее последовательно по одному символу с конца и записывать их в переменную b. Программа описанного алгоритма может иметь следующий вид:

```
var a,b:string[20];
  i,n:integer;
  c:char;
begin
  readln(a);  { Ввод заданного слова. }
  n:=length(a);  { Определение длины слова. }
{ Запись в переменную b пустой строки для последующей операции
объединения. }
  b:='';
  for i:=1 to n do
{ Последовательное накапливание «перевернутой» строки }
{ при помощи операции объединения строк и функции, выделяющей из исходной }
{ строки по одному элементу с конца. }
     b:=b+copy(a,n-i+1,1);
  writeln(b);
end.
```

Пример 8.3. Дано слово. Определить, является ли оно палиндромом[1]. Предлагаем читателю самостоятельно разобраться в тексте программы, представленной ниже.

```
var str:string[30];
  k,i,n:integer;
  Fl:boolean;
begin
  readln(str);
  n:=length(str);
  Fl:=true;
  for i:=1 to n div 2 do
     if str[i]<>str[n-i+1] then
        FL:=false;
  if Fl then
     writeln('Полиндром')
  else
     writeln('Не полиндром');
end.
```

Пример 8.4. Дана строка символов. Группа символов между пробелами считается словом. Подсчитать количество слов в строке.

Из условия понятно, что задачу можно решить, перебирая строку посимвольно. Сравнивая каждый символ с пробелом, можно подсчитать их количество. Количество слов будет на одно больше. Например, строка «МАМА МЫЛА РАМУ» содержит два пробела и три слова. Программу можно записать так:

```
var st:string[30];
  i,k:integer;
begin
  readln(st);
  for i:=1 to length(st) do
     if St[i]=' ' then k:=k+1;
  writeln(k+1);
end.
```

Пример 8.5. Дана строка символов. Группа символов между пробелами считается словом. Определить самое длинное слово в строке и количество слов такой же длины.

Решение этой задачи предусматривает подсчет количества символов между пробелами. Зная значение этого параметра, можно вычислить его наибольшее значение, используя стандартный алгоритм определения максимального

[1] *Палиндромы* – это слова, которые читаются одинаково в обоих направлениях, то есть симметричны относительно своей середины. Например, «потоп» или «боб».

элемента в последовательности чисел. Алгоритм предусматривает использование двух циклов. Первый необходим для просмотра элементов всей строки. Второй – непосредственно для расчета количества символов в слове. Ниже приведен текст программы с подробными комментариями.

```
var str:string[30];
  kol,k,max,i,n:integer;
begin
  readln(str); { Ввод строки символов. }
  n:=length(str);{ Определение длины строки. }
  max:=0; { Максимальная длина слова. }
  kol:=1; { Количество слов максимальной длины. }
  i:=1; { Определение параметра цикла. }
  while i<=n do { Цикл для просмотра всей строки. }
  begin
     k:=0; { Длина слова. Присвоение начального значения. }
     repeat { Цикл для определения длины слова. }
        k:=k+1;{ Вычисление длины слова. }
        i:=i+1;{ Переход к следующему символу. }
{ Выйти из цикла, если текущий символ - пробел или длина строки исчерпана. }
     until (str[i]=' ') or (i>n);
{ Если длина слова превышает предполагаемый максимум, то }
{ записать ее значение. }
     if k>max then
        max:=k
{ Если длина слова совпадает по значению с максимальной, }
{ то увеличить значение счетчика, вычисляющего количество }
{ слов максимальной длины. }
     else if k=max then
     kol:=kol+1;
     i:=i+1; { Перейти к символу, следующему за пробелом. }
  end;
  writeln(max,' ',kol);
end.
```

Пример 8.6. Дана строка символов. Группа символов между пробелами считается словом. Определить количество слов, являющихся записью десятичного числа.

Для решения этой задачи так же необходимо просматривать всю строку и вычислять длину каждого слова. Полученное значение позволит применить функцию `copy` и выделить это слово из строки. Описанная выше процедура `val` преобразует его в вещественное число и передаст в основную программу номер позиции, в которой произошла ошибка преобразования, или 0, если ошибки не было.

```
var str:string[30];
  s:string;
  k,err,i,n,kol:integer;
  Fl:boolean;
  x:real;
begin
  readln(str);
  n:=length(str);
  i:=1;
  while i<=n do
  begin
     k:=0;
     Fl:=true;
{ Вычисление длины слова. }
     while (str[i]<>' ') and (i<=n) do
     begin
        k:=k+1;
        i:=i+1;
     end;
{ Выделение слова из строки. }
     s:=copy(str,i-k,k);
{ Преобразование полученного слова в вещественное число. }
     val(s,x,err);
{ Если число преобразовано успешно, нарастить счетчик. }
     if err=0 then kol:=kol+1;
     i:=i+1;
  end;
  writeln(kol);
end.
```

Пример 8.7. Дана строка символов. Удалить из нее все пробелы.

Для того чтобы удалить символ пробела из строки, необходимо выяснить, под каким номером в строке он находится. Сделать это можно, используя функцию pos. Надо отметить, что данная функция найдет первое вхождение заданной подстроки в исходную строку. Переменная, которой будет присвоен результат работы этой функции, примет значение, с которого можно начинать удаление одного символа. Эта операция выполняется при помощи процедуры delete. После удаления исходная строка изменится, а длина ее уменьшится. Теперь необходимо снова повторить поиск пробела в измененной строке, и, если он будет найден, удалить его. Действия повторяются до тех пор, пока не будет достигнут конец строки. Процесс окончится раньше, если результат работы функции pos примет нулевое значение.

```
var str:string[30];
  s:string;
  k,p,i,n,kol:integer;
```

```
  Fl:boolean;
  x:real;
begin
  readln(str);
  n:=length(str);
  i:=1;
  while i<=n do
  begin
     p:=pos(' ',str); { Поиск пробела в строке. }
     if p<>0 then
     begin
        delete(str,p,1); { Удаление пробела из строки. }
        n:=n-1
     end
     else break; { Пробелов больше нет. }
        i:=i+1;
  end;
  writeln(str);
end.
```

Пример 8.8. Текст из 10 строк хранится в файле. Определить, сколько в тексте знаков препинания.

Алгоритм решения этой задачи очень простой. Нужно просмотреть все строки текста, сравнивая каждый символ со знаками препинания и увеличивая некоторую переменную k на единицу, если равенство выполняется. Для того чтобы проверяемое условие не выглядело слишком громоздко, представим знаки препинания в виде множества P. Тогда достаточно будет проверить, принадлежит ли некоторый символ заданному множеству. Чтобы было удобно обращаться к отдельному символу, опишем весь текст как массив str из десяти строк, каждая из которых не содержит, например, более 20 символов. Тогда для обращения к некоторому символу текста достаточно будет указать номер строки i, в которой он находится, и номер его позиции j в этой строке. Считывание текста из файла организуем построчно.

```
var
  f:text;
  str:array[1..10] of string[20];{ Описание массива строк. }
  i,j,m,n,k:integer;
  P:set of char; { Описание множества символов. }
begin
  P:=['.','!','?',';',',',':']; { Множество разделителей. }
  assign(f,'prim_8_8.txt');
  reset(f);
  n:=1;
{ Построчное считывание текста из файла. }
```

```
  while not(eof(f)) do
  begin
     readln(f,str[n]);
     writeln(str[n]);
     n:=n+1; { Количество строк в файле. }
  end;
  close(f);
  k:=0;
  for i:=1 to n-1 do
  begin
     m:=length(str[i]); { Длина текущей строки. }
     for j:=1 to m do
{ Проверка на вхождение символа в множество разделителей. }
        if str[i][j] in P then
           k:=k+1;
  end;
  writeln(k);
end.
```

8.3. Упражнения по теме «Обработка строк»

Дана строка до точки, группа символов в которой между пробелами считается словом:

1. Подсчитать количество знаков препинания в строке.
2. Удалить из строки все запятые.
3. Приведено некоторое число. Вставить его после каждого пробела.
4. Найти сумму чисел, встречающихся в строке.
5. Удалить из строки все числа.
6. Удалить из строки слово, имеющее наибольшую длину.
7. Определить, содержит ли строка простые числа.
8. Вставить в строку пробел после каждого знака препинания.
9. Сформировать строку, состоящую из слов исходной строки, записанных наоборот.
10. Найти, сколько в строке самых коротких слов.

Текст из 20 строк хранится в файле. Выполнить следующие операции:

1. Определить количество слов в нечетных строках текста.
2. Определить количество знаков препинания в четных строках текста.
3. Определить количество предложений, учитывая, что предложение заканчивается точкой, вопросительным или восклицательным знаком.
4. Вычислить, сколько в тексте слов-палиндромов.
5. Вывести на печать самое длинное предложение.

Работа с записями

Данная глава посвящена обработке таблиц с помощью специального типа данных Турбо Паскаль – записей. Предлагаемые примеры позволят читателю разобраться с принципами обработки таблиц в программе.

9.1. Общие сведения о записях

Запись – это структурированный тип данных, состоящий из определенного числа компонентов, которые называются *полями записи.* Они могут иметь разный тип. Описание переменной типа *запись* начинается ключевым словом record, за которым следует список полей с указанием их типов. Заканчивается описание служебным словом end. Например:

```
Var
  Name: record
     N1:byte;
     N2:string;
     N3:real;
  End;
```

или:

```
Type
  abc=record
     a,b:string[15];
     c:real;
     d:real;
     ef:array [1..20] of byte;
```

```
  end;
var
  skl: abc;
```

Для того чтобы обратиться к полю записи, необходимо указать имя переменной и через точку – имя поля.

Например:

```
skl.c:=skl.d+17;
```

Обращение к полям имеет громоздкий вид, и, если работать с полями записи, нерационально постоянно указывать имя переменной и поля. Удобнее воспользоваться специально предназначенным для этого оператором:

```
with < переменная типа «запись»> do <операторы>;
```

Внутри оператора `with` с полями записи можно работать, как с обычными переменными (то есть без указания составного имени). Например:

```
with skl do
  c:=d+17;
with skl do
  begin
     c:=c+1;
     d:=d+0.1;
  end;
```

В Паскале существует возможность задавать тип записи, содержащий произвольное число вариантов структуры. Такие записи называются *записями с вариантами*. Например:

```
Type
  complex=record
     re,im:real;
  end;
  urav=record
     a,b,c:real;
     case pr: byte of
        0:(x1,x2:real);
        1:(x:real);
        2:(y1,y2:complex);
  end;
var
  abc:urav;
```

Обработка отдельных полей определенных типов осуществляется так же, как и переменных данного типа.

Записи часто используют при работе с таблицами, где каждая запись – это одна строка таблицы. Следовательно, для обработки всей таблицы, необходимо использовать массивы записей. Например:

```
Type
  ved=record
     fio:string[35];
     date:integer;
     zarhlata:real;
  end;
var
  a:array [1..30] of ved;
```

Для обращения к некоторому полю i-го элемента таблицы необходимо обратиться к этому полю в i-м элементе массива a. Например, при обращении к полю fio в пятой строке таблицы достаточно указать соответствующий элемент массива – a[5].fio.

9.2. Примеры обработки таблиц с использованием записей

Рассмотрим несколько примеров задач обработки таблиц.

Пример 9.1. О каждом студенте известна следующая информация:

- фамилия, инициалы;
- год рождения;
- группа;
- отметка по математике;
- отметка по истории;
- отметка по ВТ;
- отметка по статистике.

Сформировать таблицу, записав в нее всю известную информацию о каждом студенте и его средний балл. Подсчитать средний балл по каждому предмету, вывести таблицу на экран дисплея в алфавитном порядке.

```
Type       { Описание записи о каждом студенте. }
  tablica=record
     name:string[15];
     group:string[8];
     god:integer;
     vt,history,stat,math:byte;
```

```
    sr_bal:real;
  end;
var
  i,j,n:integer; a:tablica;
  mas:array [1..30] of tablica;{ Таблица - массив записей. }
{ Переменные для хранения средних значений по предметам. }
  s_vt, s_history, s_stat, s_math:real;
begin
{ Ввод количества n записей. }
  write('n='); readln(n);
{ Ввод элементов в массив записей. }
  for i:=1 to n do
    with mas[i] do
    begin
      writeln('i=',i:4);
      writeln('FIO');
      readln(name);
      write('Group');
      readln(group);
      write('Year');
      readln(god);
      write('Otsenki');
      readln(vt,history,stat,math);
      sr_bal:=(vt+history+stat+math)/4;
    end;
{ Вычисление среднего балла по каждому предмету. }
  s_vt:=0; s_history:=0; s_stat:=0; s_math:=0;
  for i:=1 to n do
  begin
    s_vt:=s_vt+mas[i].vt;
    s_history:=s_history+mas[i].history;
    s_stat:=s_stat+mas[i].stat;
    s_math:=s_math+mas[i].math;
  end;
  s_vt:=s_vt/n;
  s_history:=s_history/n;
  s_stat:=s_stat/n;
  s_math:=s_math/n;
{ Упорядочение массива записей в алфавитном порядке фамилий. }
  for i:=1 to n-1 do
    for j:=1 to n-i do
      if mas[j].name>mas[j+1].name then
      begin
        a:=mas[j];
        mas[j]:=mas[j+1];
        mas[j+1]:=a;
```

```
        end;
{ Вывод результатов. }
  clrscr;
  write(' ':4,'FIO ',' ':4);
  write(' ':2,'GROUP ',' ':2);
  write(' ':5,'OTSENKI',' ':4);
  writeln('Sr. Bal');
  for i:=1 to n do
     with mas[i] do
     begin
        write(name:15);
        write(' ',group:8);
        write(' ',god:4);
        writeln(' ',vt:3,' ',history:3,' ',stat:3,' ',math:3,
        ' ',sr_bal:5:2);
     end;
  writeln(' ','Sr. Bal:',' ':17,s_vt:3:1,' ',s_history:3:1,' ',
  s_stat:3:1,' ',s_math:3:1);
end.
```

Пример 9.2. В файле записей steel.dat хранится информация о химическом составе различных сталей, представленная в табл. 9.1.

Таблица 9.1 ▾ Химический состав различных сталей

Марка стали	Содержание элементов						
	C	Cr	Ni	Mn	Si	P	Ti
X17T	1	17		1	1	0,02	
30XCA	0,3	1		1	1	0,01	
...	...	...	...	...	...	...	...

Вывести на экран содержимое файла. Компоненты файла расположить в порядке возрастания концентрации хрома (Cr). На их основе сформировать два файла, в один из которых поместить информацию о сталях, включающих хром (хромированные стали), а во второй – данные о сталях, содержащих титан (Ti) и меньше 0,3% углерода (C). Информацию о сталях, которые вошли в оба созданных файла, удалить из файла steel.dat. Вывести на экран данные о сталях, в состав которых входит марганец (Mn). Ниже приведен текст программы с комментариями.

```
{ Тип steel - запись для хранения информации о типах стали. }
type steel=record
  name:string[10];
     c,cr,ni,mn,si,p,ti:real;
  end;
```

```
var
  b:steel;
{ f - исходный файл; f1,f2 - выходные файлы. }
  f,f1,f2:file of steel;
  a:array [1..50] of steel;
  i,j,k,n:integer;
  fl:boolean;
  s: array [1..50] of string;
begin
  assign(f,'steel.dat'); reset(f);
  assign(f1,'cr.dat');
  assign(f,'ti.dat');
  i:=1;
{ Считывание содержимого файла в массив a. }
  while not eof(f) do
  begin
     read(f,a[i]);
     i:=i+1;
  end;
  close(f);
  rewrite(f1);
  rewrite(f2);
  fl:=false;
  k:=0;
  for n:=1 to i-1 do
  begin
     if a[n].cr>0 then
     begin
        write(f1,a[n]);
        fl:=true;
     end;
     if fl then
        if (a[n].ti>0) and (a[n].c<0.3) then
        begin
           fl:=false;
           write(f2,a[n]);
{ В массиве s список сталей, попавших в оба файла. }
           s[k+1]:=a[n].name;
           k:=k+1;
        end;
     if not fl then
     if (a[n].ti>0) and (a[n].c<0.3) then
        write(f2,a[n]);
  end;
  close(f1); close(f2);
{ Упорядочение таблицы сталей в порядке возрастания концентрации хрома. }
```

```
  for n:=1 to i-2 do
     for j:=1 to i-2-n do
        if a[j].cr>a[j+1].cr then
        begin
           b:=a[j];
           a[j]:=a[j+1];
           a[j+1]:=b;
           end;
{ Вывод упорядоченного массива сталей. }
  for n:=1 to i-1 do
     with a[n] do
     begin
        write(name:10);
        write(c:6:2);
        write(cr:6:2);
        write(ni:6:2);
        write(mn:6:2);
        write(si:6:2);
        write(p:6:2);
        write(ti:6:2);
     end;
  rewrite(f);
  for n:=1 to i-1 do
     with a[n] do
     begin
        fl:=false;
{ Цикл по j проверяет, встретилась ли текущая сталь в массиве s. }
        for j:=1 to k do
           if name=s[j] then fl:=true;
{ Если текущей стали нет в массиве s, то записать ее в исходный файл f. }
        if not fl then write(f,a[n]);
     end;
  close(f);
{ Печать списка сталей, в которых есть марганец. }
  for n:=1 to i-1 do
     with a[n] do
     begin
        if mn>0 then
        begin
           write(name:10);
           write(c:6:2);
           write(cr:6:2);
           write(ni:6:2);
           write(mn:6:2);
           write(si:6:2);
           write(p:6:2);
```

```
            write(ti:6:2);
          end;
      end;
end.
```

Пример 9.3. В текстовом файле stud.txt хранится информация о результатах рейтинговых испытаний некоторого вуза (табл. 9.2). Испытание считается не пройденным, если абитуриент набрал менее 25 баллов. Зачисляется в вуз абитуриент, средний балл которого по результатам трех туров не менее 25. Необходимо, воспользовавшись информацией, хранящейся в текстовом файле, создать файл записей, сформировав поле «Средний балл». Удалить из файла фамилии всех абитуриентов, не преодолевших барьер в 25 баллов. Информацию об абитуриентах, зачисленных в вуз, вывести на экран в порядке убывания среднего балла.

Таблица 9.2 ▼ Результаты рейтинговых испытаний

ФИО абитуриента	1-й тур	2-й тур	3-й тур	Средний балл
Иванов И. И.	26	16	34	
Козлов С. С.	21	18	35	
Петров Ф. Ф.	18	27	35	
Сидоров П. П.	12	21	20	
...	...	...	...	

Читателю несложно будет разобраться в тексте программы, приведенном ниже, так как алгоритмы, используемые при решении данной задачи (удаление и сортировка) были подробно описаны в главе 5, а основные принципы работы с файлами – в главе 7. Отличие состоит лишь в том, что теперь в роли компонента файла выступает не элемент массива, а более сложная конструкция – запись.

```
Program prim9_3;
Type
  stud=record
     Fio:string[15];
     Tur_1:byte;
     Tur_2:byte;
     Tur_3:byte;
     Ball:byte;
  end;
Var
  F:text; { Ранее созданный файл, в котором хранится текстовая информация. }
{ Файл записей, который необходимо создать на базе текстового файла. }
  F1:file of stud;
```

```
  p:stud; { Переменная типа запись. }
Procedure CreateArchive;
{ Создание файла записей. }
Var
  t1,t2,t3:byte;
Begin
  Reset(F); { Открыть текстовый файл для считывания информации. }
  Rewrite(F1); { Открыть файл записей для занесения информации. }
  While not eof (F) do { Пока не достигнут конец файла, }
  With P do
  Begin
{ считать текущую запись, }
     Readln(F,Fio,Tur_1,Tur_2,Tur_3);
{ если результат испытания по одному из туров менее 25 баллов, }
{ обнулить вспомогательную переменную. }
     if Tur_1 < 25 then T1:=0
     else t1:=Tur_1;
     if Tur_2 < 25 then T2:=0
     else t2:=Tur_2;
     if Tur_3 < 25 then T3:=0
     else t3:=tur_3;
{ Вычислить среднее значение по итогам трех туров, }
{ результат занести в соответствующее поле записи. }
     Ball:=trunc((T1+T2+T3)/3);
     Write(F1,P);
  end;
  Close(F); { Закрыть текстовый файл. }
  Close(F1); { Закрыть файл записей. }
  Writeln('Файл записей создан. Для продолжения нажмите ENTER!');
  readln;
end;{ Конец процедуры CreateArchive. }
Procedure WriteArchive;
{ Вывод информации на экран. }
Var
  i:integer;
Begin
  Reset(F1); { Открыть файл записей для считывания информации. }
  Writeln('Результаты рейтинговых испытаний');
  Writeln('-------------------------------------------------------');
  Writeln(':ФИО абитуриента:1-й тур:2-й тур:3-й тур:Средний балл:');
  Writeln('-------------------------------------------------------');
  While not eof(F1) do { Пока не достигнут конец файла, }
  Begin
     Read(F1,P); { считать запись и вывести ее на экран. }
     With P do
     Writeln(':',Fio,':',Tur_1:7,':',Tur_2:7,':',Tur_3:7,':',Ball:12,':');
  end;
```

```
  Writeln('-----------------------------------------------------');
  Close(F1); { Закрыть файл записей. }
  writeln('Для продолжения нажмите ENTER!');
  readln;
End;{ Конец процедуры WriteArchive. }
Procedure Sr_Ball;
{ Удаление записей со средним баллом, равным нулю. }
Var
  n,k,i,j:integer;
  Sr:byte;
Begin
  Reset(F1);
  i:=0;
  n:=filesize(f1)-1;{ Определить количество компонентов в файле записей. }
  k:=0;
  while i<=n do     { Пока не достигнут конец файла, }
  Begin
     seek(f1,i);    { установить указатель на текущую запись. }
     Read(F1,P);
     with p do
        if Ball=0 then { Если средний балл равен нулю, то }
        begin { удалить запись сдвигом последующих компонентов }
           for j:=i to n-1 do   { от текущей записи и до конца файла. }
           begin
{ Установить указатель на следующую запись. }
              seek(f1,j+1);
{ Прочитать следующую запись. }
              read(f1,p);
{ Установить указатель на текущую запись. }
              seek(f1,j);
{ Переписать следующую запись на текущую. }
              write(f1,p);
           end;
           seek(f1,n); { Установить указатель на последнюю запись и }
{ удалить ее. }
           truncate(f1);
           n:=n-1; { Уменьшить количество записей на единицу.}
           k:=k+1; { Увеличить счетчик удаленных записей. }
        end
{ Если средний балл отличен от нуля, перейти к следующей записи. }
        else i:=i+1;
  end;
  Close(F1); { Закрыть файл. }
```

```
  writeln('Произошло удаление записей для абитуриентов со средним баллом,
меньшим 25');
  writeln('Удалено ',k,' записей');
  writeln('Осталось ',n,' записей');
  readln;
  writeln('Для просмотра результатов удаления нажмите ENTER!');
end; { Конец процедуры Sr_Ball. }
Procedure U_Ball;
{ Упорядочить информацию по убыванию среднего балла. }
Var
  s,k,i:integer;
  r,b:stud;
begin
  Reset(F1);
  for k:=0 to filesize(f1)-2 do
  begin
     for i:=0 to filesize(f1)-k-2 do
     begin
{ Прочитать текущую и следующую записи. }
        seek(f1,i);
        read(f1,p);
        seek(f1,i+1);
        read(f1,r);
{ Проверить условие упорядочивания. }
        if p.ball<r.ball then
        begin
{ Поменять местами записи. }
           b:=p;
           seek(f1,i);
           write(f1,r);
           seek(f1,i+1);
           write(f1,b);
        end;
     end;
  end;
  writeln('Произведено упорядочивание записей по среднему баллу!');
 writeln('Для продолжения нажмите ENTER!');readln;
end; { Конец процедуры U_Ball. }
Begin { Основная программа. }
{ Связь файловых переменных с файлами на диске. }
  Assign (F1,'st.dat');
  Assign (F,'stud.txt');
  CreateArchive; { Вызов процедуры создания файла записей. }
```

```
  WriteArchive;  { Вызов процедуры вывода на печать информации. }
  Sr_Ball;     { Вызов процедуры удаления записей из файла. }
  WriteArchive;  { Вызов процедуры вывода на печать информации. }
  U_Ball;      { Вызов процедуры упорядочивания записей в файле. }
  WriteArchive;  { Вызов процедуры вывода на печать информации. }
end. { Конец основной программы. }
```

9.3. Упражнения по теме «Работа с записями»

1. В текстовом файле хранится информация о перенесенных инфекционных заболеваниях учащимися средней школы (табл. 9.3). Создать файл записей, добавив поле «Средняя заболеваемость за полугодие». Упорядочить файл в порядке возрастания заболеваемости.
2. Информация, приведенная в табл. 9.4, хранится в текстовом файле. Создать файл записей, добавив поле «Средний прирост населения». Упорядочить информацию, расположив названия областных центров в алфавитном порядке.
3. Текстовый файл содержит информацию о успеваемости студентов некоторого факультета за все время обучения (табл. 9.5). Создать файл записей, добавив поле «Средняя успеваемость». Удалить из файла информацию о студентах со средним баллом менее 3,5.
4. Данные занесены в текстовый файл так, как показано в табл. 9.6. Создать файл записей, добавив поле «Общая сумма выданных ссуд каждым банком». Не заносить в файл информацию о банках с общей суммой ссуд менее 100 тысяч. Упорядочить информацию в файле, расположив названия банков в алфавитном порядке.
5. Информация о продаже путевок некоторой туристической фирмой (табл. 9.7) хранится в текстовом файле. Создать файл записей, добавив поля «Продано путевок на сумму» и «Среднее количество проданных путевок». Вывести на экран информацию о странах, в которые продано больше всего путевок. Отсортировать файл в порядке убывания информации в поле «Продано путевок на сумму».
6. Известна информация о каждом клиенте сберкассы:
   - номер сберкнижки;
   - фамилия, инициалы;
   - год рождения;
   - сумма вклада;
   - приход;
   - расход.

Сформировать таблицу, записав в нее всю известную информацию о каждом клиенте. Если клиент снимал деньги со счета, то приход равен нулю и наоборот. Сформировать поле «Итого». Отсортировать таблицу в алфавитном порядке. Определить количество клиентов, которые внесли деньги на счет, и количество клиентов, снявших деньги.

7. Данные занесены в текстовый файл так, как показано в табл. 9.8. Создать файл записей, добавив поле «Итого» и «На сумму». Вывести на экран информацию о передовиках. Удалить из файла записи с наименьшей добычей.
8. Известна информация о сотрудниках некоторого предприятия:
   - фамилия, инициалы;
   - год рождения;
   - должность;
   - стаж;
   - оклад.

   Сформировать таблицу, записав в нее всю известную информацию о каждом сотруднике. Создать поле «Зарплата», добавляя 10% к окладу, если стаж работы более 10 лет, и 15%, если более 20. Отсортировать таблицу в алфавитном порядке. Определить количество пенсионеров (старше 60 лет) и количество молодых специалистов (моложе 25 лет).
9. О сотрудниках некоторого предприятия известно следующее:
   - фамилия, инициалы;
   - год рождения;
   - должность;
   - пол.

   Сформировать таблицу, записав в нее всю известную информацию о каждом сотруднике. Вывести на печать информацию о количестве пенсионеров – женщин старше 55 лет и мужчин старше 60 лет. Если их количество превышает 10 человек, удалить фамилии всех пенсионеров из таблицы.
10. О сотрудниках некоторого НИИ известна следующая информация:
    - фамилия, инициалы;
    - год рождения;
    - ученая степень.

    Сформировать таблицу, записав в нее всю известную информацию о каждом сотруднике. Вывести на печать информацию о количестве молодых ученых (моложе 35 лет), имеющих ученую степень. Удалить из таблицы фамилии всех сотрудников старше 40 лет, без ученой степени.

11. Известна информация о сотрудниках некоторого предприятия:
    - фамилия, инициалы;
    - должность;
    - оклад.

    Сформировать таблицу, записав в нее всю известную информацию о каждом сотруднике. Сформировать поле «Подоходный налог», учитывая, что он составляет 13% от оклада, и поле «К выплате». Определить количество сотрудников с минимальной зарплатой и количество с максимальной. Отсортировать таблицу по полю «Должность».
12. Ведомость о закупках чая хранится в текстовом файле в виде, представленном в табл. 9.9. Создать файл записей, заполнить поле «Сумма» и графу «Всего». Сформировать отдельные таблицы о закупках зеленого, черного, желтого и красного чая. Информацию в них упорядочить по убыванию суммы продаж.
13. Ведомость о закупках чая, хранящуюся в текстовом файле (см. табл. 9.9), вывести на печать, упорядочив по алфавиту названия фирм-производителей. Вывести на печать название фирмы с максимальным объемом продаж.
14. Вывести на печать ведомость о закупках чая, хранящуюся в текстовом файле (см. табл. 9.9). Упорядочить информацию, расположив в алфавитном порядке тип чая, а затем форму упаковки. Определить, в какой упаковке чай продается лучше.
15. Информацию (см. табл. 9.9), хранящуюся в текстовом файле, использовать для создания файлов записей, в которых будут храниться данные о закупках чая по каждой фирме-производителю. Информацию в файлах расположить в порядке возрастания количества упаковок проданного чая. Вывести на печать информацию из каждого файла, заполнив поле « Сумма» и графу «Всего».

Таблица 9.3 ▼ Статистика заболеваемости по школе

Болезнь	Количество больных					
	Январь	Февраль	Март	Апрель	Май	Июнь
Грипп	120	132	97	54	12	3
...	...	...	...	...	...	...

Таблица 9.4 ▾ Прирост населения в крупных городах СНГ

Город	Прирост населения в тыс. чел.				
	1996	1997	1998	1999	2000
Москва	50	54	60	59	63
Донецк	−14	−13	−10	−8	−5
...	...	...	...	...	...

Таблица 9.5 ▾ Средняя успеваемость

ФИО студента	Успеваемость (средний балл)					
	1 семестр	2 семестр	3 семестр	4 семестр	...	10 семестр
Иванов	5	4,8	3,7	4,5	...	4
...	...	...	...	...	...	...

Таблица 9.6 ▾ Ссуды, выданные банками

Название банка	Выданные ссуды, тыс. руб.				
	1999	2000	2001	2002	2003
Гаммабанк	20	35	56	70	120
...	...	...	...	...	...

Таблица 9.7 ▾ Продажа путевок

Страна	Цена путевки, $	Количество проданных путевок					
		Апрель	Май	Июнь	Июль	Август	Сентябрь
Греция	546	75	120	150	158	160	130
...	...	...	...	...	...	...	...

Таблица 9.8 ▾ Добыча угля

Номер бригады	Цена за тонну	Добыча угля в смену, т				
		Понедельник	Вторник	Среда	Четверг	Пятница
1	198	50	40,5	36,2	48	33,5
	...	...	...	...	...	...

Таблица 9.9 ▾ Закупка чая

Тип чая	Форма упаковки	Производитель	Цена, грн.	Количество	Сумма
Зеленый	Пачка	Riston	5,8	50	
Черный	Пакетики	Edwin	4,7	23	
Красный	Пакетики	Dilmah	6.9	56	
Желтый	Пакетики	Edwin	4,4	32	
Черный	Пачка	Ahmad	3,9	89	
Желтый	Жесть	Dilmah	20,1	23	
Черный	Фарфор	Edwin	25,4	76	
...	...				
			Всего:		

Глава 10 Динамические переменные и указатели

Данная глава посвящена описанию возможностей Турбо Паскаля при работе с динамическими переменными и указателями.

10.1. Работа с динамическими переменными и указателями

Все объявленные в программе статические переменные, которые мы рассматривали до этого момента, размещаются в одной непрерывной области оперативной памяти, которая называется *сегментом данных*. Его длина определяется архитектурой микропроцессора и составляет 65536 байт, что может вызвать известные затруднения при обработке больших массивов. Этим и обусловлено ограничение на размер массива 64 Кб.

С другой стороны, объема памяти даже персонального компьютера первого поколения XT(640 Кб) было достаточно для решения задач с большой размерностью. Для работы с подобными массивами можно воспользоваться так называемой *динамической памятью*, то есть оперативной памятью ПК, которая выделяется программе при ее работе за вычетом сегмента данных (64 Кб), стека (обычно 16 Кб) и собственно тела программы. Размер динамической памяти можно варьировать в широких пределах. По умолчанию он определяется всей доступной памятью ПК в режиме командной строки и составляет не менее 200–300 Кб.

Динамическое размещение данных осуществляется компилятором непосредственно в процессе выполнения программы. Причем заранее не известно количество размещаемых данных. Кроме того, к ним нельзя обращаться по именам, как к статическим переменным.

Оперативная память ПК представляет собой совокупность элементарных ячеек для хранения информации – байтов, каждый из которых имеет собственный номер. Эти номера называются *адресами.* Они позволяют обращаться к любому байту памяти.

Турбо Паскаль имеет гибкое средство управления памятью – указатели.

Указатель – переменная, которая в качестве своего значения содержит адрес байта памяти. Один указатель позволяет адресовать 64 Кб. Указатель занимает 2 байта.

Как правило, в Турбо Паскале указатель связывается с некоторым типом данных. В таком случае он называется *типизированным.* Для его объявления используется знак ^, который помещается перед соответствующим типом. Например:

```
type
  massiv=array [1..2500] of real;
var
  a:^integer; b,c:^real; d:^massiv;
```

В Турбо Паскале можно объявлять указатель, не связывая его с конкретным типом данных. Для этого служит стандартный тип `pointer`. Например:

```
var
  p,c,h: pointer;
```

Указатели такого рода здесь и далее будем называть *нетипизированными.* Поскольку такие указатели не связаны с конкретным типом, с их помощью удобно динамически размещать данные, структура и тип которых меняются в процессе работы программы.

Значениями указателей являются адреса переменных памяти, поэтому следует ожидать, что значение одного из них можно передавать другому. На самом деле это не совсем так. В Турбо Паскале эта операция проводится только среди указателей, связанных с одними и теми же типами данных. Например:

```
Var
  p1, p2:^integer; p3:^real; pp:pointer;
```

В этом случае присваивание `p1:=p2;` допустимо, в то время как `p1:=p3;` запрещено, поскольку `p1` и `p3` указывают на разные типы данных. Это ограничение не

распространяется на нетипизированные указатели, поэтому можно записать pp:=p3; p1:=pp; и достичь необходимого результата.

Вся динамическая память в Турбо Паскале представляет собой сплошной массив байтов, называемый *кучей*. Физически куча располагается за областью памяти, которую занимает тело программы.

Начало кучи хранится в стандартной переменной heaporg, конец – в переменной heapend. Текущая граница незанятой динамической памяти хранится в указателе heapprt.

Память под любую динамическую переменную выделяется процедурой new, параметром обращения к которой является типизированный указатель. В результате обращения последний принимает значение, соответствующее динамическому адресу, начиная с которого можно разместить данные. Например:

```
var
  i,j:^integer;
  r:^real;
begin
  new(i);
  new(R);
  new(j)
```

В результате выполнения первого оператора указатель i принимает значение, которое перед этим имел указатель кучи heapprt. Сам heapprt увеличивает свое значение на два, так как длина внутреннего представления типа integer, связанного с указателем i, составляет 2 байта. Оператор new(r) вызывает еще одно смещение указателя heapprt, но уже на 6 байт, потому что такова длина внутреннего представления типа real. Аналогичная процедура применяется и для переменной любого другого типа. После того как указатель стал определять конкретный физический байт памяти, по этому адресу можно разместить любое значение соответствующего типа, для чего сразу за указателем без каких-либо пробелов ставится знак ^. Например:

```
i^:=4+3;
j^:=17;
r^:=2 * pi.
```

Таким образом, значения, на которые указывает указатель, то есть собственно данные, размещенные в куче, обозначаются символом ^, который ставится сразу за указателем. Если за последним этот значок отсутствует, то имеется в

виду адрес, по которому размещаются данные. Динамически размещенные данные (но не их адрес!) можно использовать для констант и переменных соответствующего типа в любом месте, где это допустимо. Например:

```
r^:=sqr(r^)+sin(r^+i^)-2.3
```

Невозможен оператор

```
r:=sqr(r^)+i^;
```

так как указателю r нельзя присвоить значение вещественного типа.

Точно также недопустим оператор

```
r^:=sqr(r);
```

поскольку значением указателя r является адрес и его (в отличие от того значения, которое размещено по данному адресу) нельзя возводить в квадрат. Ошибочным будет и присваивание r^:=i, так как вещественным данным, на которые указывает r^, нельзя присваивать значение указателя (адрес). Динамическую память можно не только забирать из кучи, но и возвращать обратно. Для этого используется процедура dispose(p), где p – указатель, который не изменяет значение указателя, а лишь возвращает в кучу память, ранее связанную с указателем.

При работе с указателями и динамической памятью необходимо самостоятельно следить за правильностью использования процедур `new`, `dispose` и работы с адресами и динамическими переменными, так как транслятор не контролирует эти ошибки. Ошибки этого класса могут привести к «зависанию» компьютера, а иногда и к более серьезным последствиям.

Другая возможность состоит в освобождении целого фрагмента кучи. С этой целью перед началом выделения динамической памяти текущее значение указателя `heapptr` запоминается в переменной-указателе с помощью процедуры `mark`. Теперь можно в любой момент освободить фрагмент кучи, начиная с того адреса, который запомнила процедура `mark`, и до конца динамической памяти. Для этого используется процедура `release`.

Процедура `mark` запоминает текущее указание кучи `heapptr` (обращение `mark(ptr)`, где `ptr` – указатель любого типа, в котором будет возвращено текущее значение `heapptr`). Процедура `release(ptr)`, где `ptr` – указатель любого типа, освобождает участок кучи от адреса, хранящегося в указателе до конца кучи.

10.2. Работа с динамическими массивами и матрицами с помощью процедур getmem и freemem

Для работы с указателями любого типа используются процедуры `getmem`, `freemem`. Процедура `getmem(p,size)`, где `p` – указатель, `size` – размер в байтах выделяемого фрагмента динамической памяти (`size` типа `word`), резервирует за указателем фрагмент динамической памяти требуемого размера.

Процедура `fremem(p,size)`, где `p` – указатель, `size` – размер в байтах освобождаемого фрагмента динамической памяти (`size` типа `word`), возвращает в кучу фрагмент динамической памяти, который был зарезервирован за указателем. При применении процедуры к уже освобожденному участку памяти возникает ошибка.

После рассмотрения основных принципов и процедур работы с указателями возникает вопрос: а зачем это нужно? В основном для того, чтобы работать с так называемыми динамическими массивами. Последние представляют собой массивы переменной длины, память под которые может выделяться (и изменяться) в процессе выполнения программы как при каждом новом запуске программы, так и в разных ее частях. Обращение к `i`-му элементу динамического массива `x` имеет вид `x^[i]`.

Пример 10.1. Рассмотрим процесс функционирования динамических массивов на примере решения следующей задачи: найти максимальный и минимальный элементы массива `x(n)`.

Напомним читателю процесс решения задачи традиционным способом.

```
Program din_mas1;
Var
  x:array [1..150] of real;
  i,n:integer; max,min:real;
begin
  writeln('введите размер массива');
  readln (n);
  for i:=1 to N do
  begin
     write('x[',i,']='); readln(x[i]);
  end;
```

```
  max:=x[1]; min:=x[1];
  for i:=2 to N do
  begin
     if x[i] > max then max:=x[i];
     if x[i] < min then min:=x[i];
  end;
  writeln ('максимум=',max:1:4);
  writeln ('минимум=',min:1:4);
end.
```

Теперь рассмотрим процесс решения задачи с использованием указателей. Распределение памяти проводим с помощью процедур `new-dispose` или `getmem-freemem`.

```
Program din_mas2;
type massiw= array[1..150]of real;
  var x:^massiw;
  i,n:integer;max,min:real;
begin
{ Выделяем память под динамический массив из 150 вещественных чисел. }
  new(x);
  writeln('введите размер массива'); readln(n);
  for i:=1 to N do
  begin
     write('x(',i,')='); readln(x^[i]);
  end;
  max:=x^[1];min:=x^[1];
  for i:=2 to N do
  begin
     if x^[i] > max then max:=x^[i];
     if x^[i] < min then min:=x^[i];
  end;
  writeln('максимум=',max:1:4,' минимум=',min:1:4);
{ Освобождаем память. }
  dispose(x);
end.

Program din_mas3;
type
  massiw=array[1..150]of real;
var x:^massiw;
  i,n:integer;max,min:real;
begin
  writeln('введите размер массива'); readln(n);
{ Выделяем память под n элементов массива. }
  getmem(x,n*sizeof(real));
```

```
  for i:=1 to N do
  begin
     write('x(',i,')='); readln(x^[i]);
  end;
  max:=x^[1];min:=x^[1];
  for i:=2 to N do
  begin
     if x^[i] > max then max:=x^[i];
     if x^[i] < min then min:=x^[i];
  end;
  writeln('максимум=',max:1:4,' минимум=',min:1:4);
{ Освобождаем память. }
  freemem(x,n*sizeof(real));
end.
```

При работе с динамическими переменными необходимо соблюдать следующий порядок работы:

1. Описать указатели.
2. Распределить память.
3. Обработать динамический массив.
4. Освободить память.

Понятие динамического массива можно распространить и на матрицы. *Динамическая матрица* представляет собой массив указателей, каждый из которых адресует одну строку (или один столбец).

Рассмотрим описание динамической матрицы. Пусть есть типы данных `massiv` и указатель на него `din_massiv`.

```
type massiv=array [1..1000] of real;
  din_massiv=^massiv;
```

Динамическая матрица `X` будет представлять собой массив указателей:

```
Var X: array[1..100] of din_massiv;
```

Работать с матрицей необходимо следующим образом:

1. Определить ее размеры (пусть `N` – число строк, `M` – число столбцов).
2. Выделить память под матрицу.

```
for i:=1 to N do
 getmem(X[i],M*sizeof(real));
```

Каждый элемент статического массива `X[i]` – указатель на динамический массив, состоящий из `M` элементов типа `real`. В статическом массиве `X` находится `N` указателей.

3. Для обращения к элементу динамической матрицы, расположенному в i-й строке и j-м столбце, следует использовать конструкцию языка Турбо Паскаль `X[i]^[j]`.
4. После завершения работы с матрицей необходимо освободить память.

```
for i:=1 to N do
  freemem(b[i],M*sizeof(real));
```

Рассмотрим работу с динамической матрицей на следующем примере.

Пример 10.2. В каждой строке матрицы вещественных чисел $B(N,M)$ упорядочить по возрастанию элементы, расположенные между максимальным и минимальным значением.

Алгоритмы упорядочивания рассматривались в главе 4, основные принципы работы с матрицами – в главе 5, поэтому в комментариях к тексту программы основное внимание уделено особенностям работы с динамическими матрицами.

```
{ Описываем тип данных massiv как массив 1000 вещественных чисел. }
type massiv=array [1..1000] of real;
{ Указатель на массив. }
 din_massiv=^massiv;
{ Тип данных matrica - статический массив указателей, каждый элемент
которого является адресом массива вещественных чисел. }
  matrica=array [1..100] of din_massiv;
var
  Nmax,Nmin,i,j,n,m,k:word;
{ Описана динамическая матрица b. }
  b:matrica;
  a,max,min:real;
begin
{ Вводим число строк N и число столбцов M. }
 write('N=');readln(N);
  write('M=');readln(M);
{ Выделяем память под матрицу вещественных чисел размером N на M. }
  for i:=1 to N do
  getmem(b[i],M*sizeof(real));
{ Вводим матрицу B. }
  writeln('Matrica B');
  for i:=1 to N do
     for j:=1 to M do
        read(b[i]^[j]);
{В каждой строке находим максимальный, минимальный элементы и их номера и
элементы, расположенные между ними, упорядочиваем методом «пузырька». }
  for i:=1 to N do
  begin
```

```
{ Поиск минимального и максимального элементов в i-й строке матрицы и их
номеров. }
     max:=b[i]^[1];
     Nmax:=1;
     min:=b[i]^[1];
     Nmin:=1;
     for j:=2 to M do
     begin
        if b[i]^[j]>max then
        begin
           max:=b[i]^[j];
           nmax:=j
        end;
        if b[i]^[j]<min then
        begin
           min:=b[i]^[j];
           nmin:=j
        end;
     end;
{ Если минимальный элемент расположен позже максимального, nmin и nmax
меняем местами. }
     if nmax<nmin then
     begin
        j:=nmax;
        nmax:=nmin;
        nmin:=j;
     end;
{ В i-й строке упорядочиваем элементы, расположенные между nmin и nmax,
методом «пузырька». }
     j:=1;
     while nmax-1-j>=nmin+1 do
     begin
        for k:=nmin+1 to nmax-1 -j do
        if b[i]^[k]>b[i]^[k+1] then
        begin
           a:=b[i]^[k];
           b[i]^[k]:=b[i]^[k+1];
           b[i]^[k+1]:=a;
        end;
        j:=j+1;
     end;
  end;
{ Выводим преобразованную матрицу. }
  writeln('Упорядоченная матрица B');
  for i:=1 to N do
  begin
```

```
    for j:=1 to M do
      write(b[i]^[j]:6:2, ' ');
    writeln
  end;
{ Освобождаем память. }
  for i:=1 to N do
    freemem(b[i],M*sizeof(real));
end.
```

Динамическая матрица может быть достаточно большой, ведь каждая строка может достигать почти 64 Кб[1]. Фактически ее размер ограничен только объемом свободной памяти в DOS-секции[2]. В следующем параграфе описан прием построения одномерного динамического массива объемом более 64 Кб.

10.3. Массивы больше 64 Кб в Турбо Паскале

В куче нельзя выделить память больше 64 Кб [1]. Сегмент данных не может превышать этого размера. Дело в том, что адресуемое пространство памяти в операционной системе MS DOS организовано сегментами – последовательными блоками памяти по 64 Кб каждый. Однако в Турбо С можно выделить блок памяти более 64 Кб (функции `farcalloc` и `farmalloc`), хотя и тот и другой язык являются разработкой фирмы Borland. Возникает вопрос: почему нельзя выделить блок памяти более 64 Кб средствами Турбо Паскаля? А если можно, то как обращаться к элементам массива, лежащими за пределами 64 Кб?

Синтаксис языка Паскаль не позволяет выделить более 65521 байт. Как кажется авторам, это ограничение можно обойти. Заметим, что адрес, с точки зрения MS DOS, состоит из сегмента и смещения:

```
адрес = сегмент * 16 + смещение;
сегмент = адрес div 16;
смещение = адрес mod 16;
```

Один сегмент может адресовать 64 Кб.

В Турбо Паскале существуют две функции типа `word`, которые возвращают сегментную часть адреса (`seg(p)`) и смещение (`ofs(p)`).

Аргумент `p` в обращении может быть любого типа. Существует функция `ptr(seg,ofs: word)`, которая возвращает значения указателя (тип `pointer`), а фактически по значению сегмента и смещения вычисляет значение адреса.

Есть функция MS DOS с номером 48H, которая позволяет выделять память параграфами (параграф равен 16 байт).

---

[1] Точнее, 65521 байт.

[2] Турбо Паскаль – это DOS-приложение.

Входные данные функции 48H: в регистре BX – количество выделяемой памяти в параграфах.

Выходные данные функции 48H: в регистре AX – адрес сегмента, начиная с которого выделено указанное количество памяти.

Если невозможно выделить указанное число параграфов, то в регистре AX возвращается число 8. Когда при выделении памяти происходит сбой, в регистре AX возвращается число 7.

На базе этой функции была написана процедура far_getmem (var p:pointer; size:longint), которая позволяет выделить всю доступную свободную оперативную память.

```
procedure far_getmem(var p:pointer;size:longint);
var
  r:registers; newsize,dosseg:word;
begin
  newsize:=(size+15) div 16;
  r.bx:=newsize;
  r.ah:=$48;
  msdos(r);
  if r.ax=$07 then
  begin
     writeln('Сбойные блоки управления памятью');
     halt(1);
  end
  else
     if r.ax =$08 then
     begin
        writeln('нельзя выделить ',size,' байт');
        halt(1);
     end
     else
        p:=ptr(r.ax,0);
end;
```

Необходимо помнить, что данная процедура выделяет память средствами MS DOS за пределами Паскаль-программы. Если пользоваться установками Турбо Паскаля по умолчанию, то ваша программа захватит под свою кучу всю доступную память, и процедуре far_getmem ничего не останется.

Если использовать эту идею на практике, то необходимо в директиве {$M..} явно указать максимум кучи для вашей программы.

При использовании только статических переменных Турбо Паскаля можно максимум кучи установить равным нулю. Однако некоторые стандартные процедуры и функции языка используют память кучи: так, при работе с графикой процедура initgraph в динамической памяти размещает графический драйвер.

Поэтому максимум кучи в директиве {$M..} необходимо установить исходя из требований вашей задачи. В среде Турбо Паскаль 7.0 вам доступно около 300 Кб памяти для процедуры far_getmem. При запуске из командной строки процедура far_getmem позволяет выделить память размером около 600 Кб.

Функция MS DOS 49H дает возможность освободить память, выделенную с помощью функции 48H.

Входные данные функции 49H: в регистре ES – адрес сегмента, начиная с которого освобождается память.

Выходные данные функции 49H: в регистре AX возвращает код ошибки;

AX = 7 – сбойные блоки управления памяти;

AX = 8 – память не была выделена процедурой 48H.

На базе этой функции была написана процедура far_freemem, которая освобождает память, выделенную процедурой far_getmem.

```
procedure far_freemem(var p:pointer;size:longint);
var
  r:registers;
  segment,dosseg:word;
begin
  if ofs(p^)<0 then
  begin
     p:=nil;
     system.freemem(p,1);
  end;
  segment:=seg(p^);
  r.es:=segment;
  r.ah:=$49;
  msdos(r);
  if r.ax=$07 then
  begin
     writeln('Сбойные блоки управления памятью');
     halt(1);
  end
  else
     if r.ax =$08 then
     begin
        writeln('Память не была выделена процедурой getmem');
        halt(1);
     end;
end;
```

Процедура far_maxavail позволяет вычислить максимальный объем памяти, который может быть выделен far_getmem.

```
Function far_MaxAvail:LongInt;
var
  NewSize :Word;
```

```
Begin
  Asm
  mov ah, 048h
  mov bx, 0FFFFh
  int 21h
  mov NewSize, bx
End;
  far_MaxAvail:=LongInt(NewSize)*16;
End;
```

Рассмотрим механизм выделения и освобождения памяти с помощью рассмотренных ранее процедур на примере:

```
{$M 16384,0,16384}
uses
  dos_mem;
type
  massiv = array [1..100] of real;
var
  a: ^massiv;
  i,n:longint;
begin
  far_getmem(pointer(a), 20000 * Sizeof(real));
{ Выделяем память для массива из 20000 вещественных чисел, но обращение
вида a^[i] правильно адресует элементы массива, находящимся в первых 64 Кб
памяти. }
  far_freemem(pointer(a), 20000 * Sizeof(real))
end.
```

Однако нужно не только выделить большую область памяти, но и получить возможность обратиться к определенному ее участку (элементу динамического массива). Стандартными средствами Турбо Паскаля можно адресовать 64 Кб, но с помощью конструкции `a^[i]` нельзя будет обратиться к элементу массива, лежащему за пределами 64 Кб.

Возникла проблема, как обратиться к элементу массива (конкретному участку памяти). Решить ее можно с помощью функции `far_adres`, которая вычисляет адрес `i`-го элемента массива.

```
Function far_ADRES(P:pointer;n:longint;size:byte):pointer;
var
  longintadr:longint; newseg,newofs:word;
Begin
  longintadr:=longint(seg(p^)) shl 4 + (n-1)*size;
  newseg:=longintadr div 16;
  newofs:=longintadr mod 16;
  far_adres:=ptr(newseg,newofs)
end;
```

Зная адрес, требуется только преобразовать значение, хранящееся там, к нужному типу данных: тип (far_adres(p, i, Sizeof(тип))^).

Задан файл вещественных чисел объемом более 64 Кб. Необходимо считать элементы в один динамический массив. Структура программы выглядит так:

```
{$M,16384,0,16384}
{ Здесь должны быть тексты подпрограммы работы с массивами более 64K. }
var
  a:pointer;
  i,n:integer;
  f:file of real;
begin
  assign(f,'abc.dat');
  reset(f);
  if (filesize(f)*6)<=far_maxavail then
  begin
{ Выделение памяти под большой массив с помощью процедуры far_getmem. }
     far_getmem(f,filesize(f)*Sizeof(real);
{ Обращение к i-му элементу массива real(far_adres(a,i,Sizeof(real))^). }
     for i:=0 to filesize(f)-1 do
        read(f,real(far_adres(a,i,Sizeof(real))^));
{ Элементы считаны в массив. }
{ Далее следует обработка массива. }

{ Освобождение памяти. }
     far_freemem(f,filesize(f) * 6)
  end
end.
```

Процедуры, описанные в этой главе, делают доступной всю основную память ПК. Если выделять под кучу всю память, доступную в режиме DOS, то размер массива, определяемый с помощью кучи, может достигать 600 Кб, что полностью снимает ограничение 64 Кб в Турбо Паскале.

10.4. Задания по теме «Динамические переменные и указатели»

Выполнить задания из глав 4 и 5 с использованием динамических массивов и матриц.

Глава 11

Модули в Турбо Паскале

В этой главе описаны стандартные модули Турбо Паскаля: CRT, DOS, PRINTER. В разделе 11.1 приводятся общие сведения о стандартных модулях языка, раздел 11.2 посвящен модулю CRT, в котором собраны основные процедуры и функции для работы с экраном дисплея, клавиатурой и звуком. В разделе 11.3 описан модуль PRINTER. В разделе 11.4 описаны некоторые процедуры и функции модуля DOS, используемые при написании программ обработки файлов. Последний теоретический раздел 11.5 посвящен созданию собственных модулей программиста.

11.1. Стандартные модули Турбо Паскаля

Возможности Турбо Паскаля расширяются благодаря использованию модулей, которые представляют собой набор констант, типов данных, переменных, процедур и функций. Язык располагает стандартными (встроенными) модулями SYSTEM, DOS, OVERLAY, GRAPH, CRT, PRINTER, TURBO3, GRAPH3. Два последних предназначены для поддержки совместимости программ, написанных в версии 3.0. Обычно модуль имеет расширение .tpu (Turbo Pascal Unit). Модули CRT, SYSTEM, DOS, OVERLAY, PRINTER объединены в библиотеку и хранятся в файле turbo.tpl (Turbo Pascal Library). Модуль GRAPH находится в файле graph.tpu, модули TURBO3 и GRAPH3 – в файлах graph3.tpu и turbo3.tpu соответственно.

Модуль SYSTEM поддерживает все стандартные процедуры и функции, обеспечивает ввод-вывод данных, обработку строк, динамическое распределение оперативной памяти и ряд других возможностей Турбо Паскаля. Он подключается к любой программе автоматически.

Модуль DOS содержит многочисленные процедуры и функции, многие из которых по своему действию эквивалентны командам DOS.

Поддержку системы оверлеев обеспечивает модуль OVERLAY.

Вывод информации на принтер позволяет организовать модуль PRINTER.

Модуль CRT содержит процедуры и функции, которые обеспечивают работу с клавиатурой, экраном дисплея в текстовом режиме и управление звуком.

Модуль GRAPH обеспечивает работу с экраном дисплея в графическом режиме.

Модули CRT, DOS, PRINTER описаны в этой главе. Графическим возможностям Паскаля посвящена глава 12.

Для того чтобы использовать модули в программах, их следует указывать в операторе uses, который должен размещаться до раздела описаний. Например:

```
uses crt, graph, printer;
```

После того как модуль будет подключен с помощью uses, все его константы, типы, переменные, процедуры и функции окажутся доступными в программе.

Перед тем как познакомиться с модулем CRT, рассмотрим еще две функции:

- chr(x), x – переменная, константа или выражение типа byte. Эта функция возвращает символ (тип char), соответствующий коду x;
- ord(x), x – символ (char). Эта функция возвращает значение типа byte, являющееся кодом символа x.

11.2. Использование модуля CRT

Процедуры и функции модуля CRT позволяют управлять работой клавиатуры, текстовым выводом на экран и звуком.

Ранее говорилось об операторах write и writeln, которые предназначены для вывода информации на экран дисплея. Теперь более подробно рассмотрим текстовые режимы работы дисплея и управление цветом фона и выводимых на экран символов.

11.2.1. Основные процедуры и функции модуля CRT. Работа с экраном дисплея

Процедура `textmode` предназначена для задания одного из возможных текстовых режимов работы адаптера:

```
procedure textmode (Mode:word);
```

`Mode` – код текстового режима. В качестве его значения могут использоваться следующие константы, определенные в модуле CRT:

- `bw40=0` (черно-белый, 40 символов в строке, 25 строк);
- `co40=1` (цветной, 40 символов в строке, 25 строк);
- `bw80=2` (черно-белый, 80 символов в строке, 25 строк);
- `co80=3` (цветной, 80 символов в строке, 25 строк);
- `Mono=7` (для MDA-адаптеров);
- `Font8x8=256` (цветной, 80 символов в строке, 43 строки для адаптеров EGA, 50 строк для адаптеров VGA).

Код режима, установленного с помощью процедуры `textmode`, запоминается в глобальной переменной `lastmode` модуля CRT и может использоваться для начального состояния экрана.

В модуле CRT есть глобальная переменная `textattr` (типа `byte`). В ней хранятся текущие цветовые атрибуты мерцания и цвета символов, а также для цвета фона. Структура этого байта следующая:

7	6	5	4	3	2	1	0

где:

- бит 7 – бит мерцания (1 – да, 0 – нет);
- биты 4–6 – цвет фона;
- биты 0–3 – цвет символов.

Исходя из этого для фона предусмотрено 8 цветов, для символов – 16.

Переменную `textattr` можно использовать для управления цветовым режимом вывода символов на экран с помощью формулы:

`textattr` = «цвет символов» + 16 * «цвет фона» [+128].

Добавление 128 приводит к мерцанию символов. Для задания цвета фона и цвета символа можно использовать мнемонические константы, определенные в модуле CRT:

```
black=0 (черный);
blue=1 (синий) ;
green=2 (зеленый);
cyan=3 (голубой);
red=4 (красный);
magenta=5 (пурпурный);
brown=6 (коричневый);
lightgray=7 (светло-серый);
darkgray=8 (темно-серый);
lightblue=9 (светло-синий);
lightgreen=10 (светло-зеленый);
lightcyan=11 (светло-голубой);
lightred=12 (светло-красный);
lightmagenta=13 (светло-пурпурный);
yellow=14 (желтый);
white=15 (белый);
blink=128 (мерцание).
```

Кроме того, для установления цвета фона и цвета выводимых символов в модуле CRT есть две процедуры: `textcolor` и `textbackground`.

```
procedure textcolor (color:byte);
```

определяет цвет выводимых символов.

```
procedure textbackground(color:byte);
```

определяет цвет фона.

В качестве переменной `color` могут использоваться рассмотренные ранее цветовые константы модуля CRT.

Процедура `window` определяет текстовое окно, которое в дальнейшем будет рассматриваться процедурами вывода как весь экран.

```
procedure window (x1, y1, x2, y2: byte);
```

где `x1`, `y1`; `x2`, `y2` – координаты левого верхнего и правого нижнего углов окна. Границы текущего окна запоминаются в двух глобальных переменных модуля CRT: `WindMin` типа `word` хранит `x1` и `y1` (`x1` – в младшем байте), а переменная того же типа `WindMax` – `x2` и `y2` (`x2` – в младшем байте).

Турбо Паскаль предоставляет функции для работы над отдельными байтами машинных слов:

- `Hi(x)` – старший байт `x`, `Lo(x)` – младший байт `x`, `x` – переменная типа `word` или `integer`;
- `Xmin:=Lo(WindMin);`
- `Xmax:=Lo(WindMax);`
- `Ymin:=Hi(WindMin);`
- `Ymax:=Hi(WindMax).`

Каждое новое использование функции `window` отменяет действие предыдущей процедуры `window` (по умолчанию окном является весь экран).

Очистка экрана (или текущего окна) осуществляется с помощью процедуры `clrscr`. После обращения к ней окно (экран) заполняется цветом фона, и курсор перемещается в левый верхний угол окна.

Процедура `gotoxy(x,y: byte)` переводит курсор к позиции экрана с координатами `(x,y)`, которые задаются относительно границ экрана (окна).

Функции `wherex` и `wherey` возвращают горизонтальную (`wherex`) и вертикальную (`wherey`) координаты курсора (типа `byte`).

Процедура `clreoln` стирает часть строки от текущего положения курсора до правой границы экрана (окна).

Процедура `delline` уничтожает всю текущую строку.

Процедура `insline` вставляет строку, после чего строка с курсором и все, что ниже ее, сдвигаются вниз на одну позицию, а последняя строка теряется. Положение курсора не изменяется.

Процедуры `lowvideo`, `normvideo`, `highvideo` устанавливают соответственно пониженную, нормальную или повышенную яркость символов.

11.2.2. Работа с клавиатурой

Управление клавиатурой реализовано в языке Турбо Паскаль всего двумя функциями: `keypressed` и `readkey`. Несмотря на свою простоту, эти две функции представляют собой довольно мощные средства для работы с клавиатурой.

`Keypressed` – функция логического типа. Она описывает состояние буфера клавиатуры и принимает значение `true`, если в буфере есть хотя бы один символ, и `false`, если он пуст.

В операционной системе реализуется так называемый асинхронный ввод с клавиатуры. По мере нажатия клавиш соответствующие коды помещаются в

буфер клавиатуры[1], откуда затем они могут быть считаны процедурой `read` (`readln`) или функцией `readkey` [9].

Следовательно, процедуры `read`, `readln` и функция `readkey` работают не непосредственно с клавиатурой, а с ее буфером.

При работе с подпрограммами `read`, `readln`, `readkey` и считывании с их помощью текстовых и символьных переменных необходимо помнить, что реально данные считываются из буфера. Коды нажатых клавиш будут помещаться в буфер. В таком случае возможно следующее:

- символ, возвращаемый из буфера, не совпадает с нажатой клавишей;
- переменная будет принимать значения из буфера, и при этом нет необходимости ввода данных.

Чтобы избавиться от этого эффекта, нужно очистить буфер клавиатуры перед вводом с экрана дисплея строковых (символьных) переменных. Как это сделать, рассмотрим после знакомства с функцией `readkey`, которая возвращает значение типа `char`. При обращении к этой функции анализируется буфер клавиатуры: если в нем есть хотя бы один символ, то первый символ буфера и возвращается в качестве результата. В противном случае функция ожидает нажатия на любую клавишу.

Возникает вопрос, что вернет `readkey` в случае нажатия комбинаций клавиш **Alt+символ**, **Shift+символ**, **Ctrl+символ**. Алфавитно-цифровые клавиши, нажатые одновременно с клавишей **Alt**, функциональные клавиши в любом регистре, клавиши ↑, →, ↓, ←, **Insert**, **Home**, **End**, **Delete**, **Page Up**, **Page Down** возвращают сразу два символа: первый – с кодом 0, второй – с цифровым кодом, характеризующим нажатую клавишу. Эти символы называются расширенным кодом. Часть кодов [8] возвращаемых функцией `readkey` значений приведена в табл. 11.1.

Таблица 11.1 ▼ Таблица кодов функциональных клавиш

Клавиша	Нормальное нажатие	+Shift	+Ctrl	+Alt
A–Z	65–90	97–123	1–26	
↑	0 72	56		8
→	0 77	54	0 116	6
↓	0 80	50		2
←	0 75	52	0 115	4
Ins	0 82	48		
Home	0 71	55	0 119	

[1] Буфер клавиатуры – участок оперативной памяти, в котором может храниться до 127 символов, вводимых с клавиатуры.

Таблица 11.1 ▾ Таблица кодов функциональных клавиш (окончание)

Клавиша	Нормальное нажатие	+Shift	+Ctrl	+Alt
End	0 79	49	0 117	1
Delete	0 83	46		
Page Up	0 73	57	0 132	9
Page Down	0 81	51	0 118	3
F1–F10	0 59–0 68	0 84–0 93	0 94–0 103	0 104–0 113
F11	0 133	0 135	0 137	0 139
F12	0 134	0 136	0 138	0 140

Рассмотрим ряд примеров использования модуля CRT.

Пример 11.1. Считать символ с клавиатуры; если это клавиша ↑, то переместить курсор на строку вверх, если ↓, то, соответственно, вниз, если →, то переместить курсор на символ вправо, если ←, то, следовательно, влево.

```
uses crt;
var
  c:char;
begin
  gotoxy(2,2);
  c:=readkey;
{ Проверяем, нажата ли клавиша с кодом 0. }
  if c = #0 then
{ Если нажата, то проверяем, какой второй символ. }
     case ord(readkey) of
{ Если это символ с кодом 72, то двигаем на одну строку вверх. }
     72: gotoxy (wherex, wherey-1);
{ Если это символ с кодом 80, то двигаем на одну строку вниз. }
     80: gotoxy (wherex, wherey+1);
{ Если это символ с кодом 75, то двигаем на один символ влево. }
     75: gotoxy (wherex-1, wherey);
{ Если это символ с кодом 77, то двигаем на один символ вправо. }
     77: gotoxy (wherex+1, wherey);
  end
  else writeln(c) ;
     readln
end.
```

Приведем пример процедуры очистки буфера клавиатуры [9]:

```
procedure clrbuf;
var
  c:char;
begin
{ Пока буфер занят, считывать его первый символ в переменную ch. }
```

```
  while keypressed do
     c:=readkey
end;
```

Рассмотрим еще одну часто встречающуюся операцию:

```
{ Процедура wait - ожидание нажатия клавиши. }
procedure wait;
begin
  repeat until keypressed
end;
```

11.2.3. Основы программирования звука

В модуле CRT есть три процедуры, с помощью которых можно запрограммировать любую последовательность звуков.

Процедура `sound(f: word)`, где `f` – выражение типа `word`, определяющее частоту звука в герцах, заставляет звучать динамик с заданной частотой. После обращения к ней включается динамик, управление возвращается в основную программу. Звук будет включен впредь до вызова процедуры `nosound`, которая завершает работу динамика.

Процедура `delay (t: word)` обеспечивает задержку (приостановку) выполнения программы на *t* миллисекунд. Так написано в Справке Турбо Паскаля. Необходимо помнить, что Турбо Паскаль был создан в 1993 году, а следовательно, ориентирован на компьютеры того поколения. Современные ПК работают значительно быстрее. Поэтому данную функцию следует понимать так: задержка на *t* единиц времени, где *t* зависит от быстродействия ПК, и его надо подбирать для каждого компьютера. В примерах книги параметр процедуры `delay` подобран для процессора Celeron с тактовой частотой 2,4 ГГц. Вы можете его изменить для своего компьютера. Кроме того, можно заменить процедуру `delay` следующим образом [13].

```
Procedure Delay(x:longint);
{ Пауза на X тиков, тик - 1/18 секунды. }
var
  l:longint;
begin
  l := MemL[Seg0040:$6c];
  while MemL[Seg0040:$6c] < l+x do; { Задержка на X тиков. }
end;
```

При генерации любой мелодии эти процедуры вызываются по схеме `sound-delay-nosound`.

В приведенной ниже программе [9] вычисляется частота по заданной ноте и октаве:

```
var
  hz: word; okt; integer; nota: byte;
begin
  readln(okt,nota);
  hz:=round(440*exp(ln(2)*okt-(10-nota)/12));
end.
```

Здесь `okt` – номер октавы (–3, –2, ..., 4), (–3 – низкая, 4 – высокая), `nota` – номер ноты в октаве (до – 1, до-диез – 2, ..., си – 12).

Рассмотрим пример восходящей и нисходящей хроматической гаммы.

Пример 11.2.

```
program soundone;
uses crt;
var
  i, k: integer;
  hz: word;
begin
{ Восходящая хроматическая гамма. }
  for i:=-3 to 4 do
     for k:=1 to 12 do
     begin
        hz:=round(440*exp(ln(2)*i-(10-k)/12));
        sound(hz);
        delay(5000);
        nosound
     end;
  delay(10000);
{ Нисходящая хроматическая гамма. }
  for i:=4 downto - 3 do
     for k:=1 to 12 do
     begin
        hz:=round(440*exp(ln(2)*(i-(10-k)/12)));
        sound(hz);
        delay(5000);
        nosound
     end
end.
```

11.2.4. Вывод псевдографики и спецсимволов

Оператор `write` (`writeln`) позволяет выводить любой символ (даже если его нет на клавиатуре) на экран. Для этого в операторе `write` необходимо набрать символ # и код требуемого символа, например `writeln(#194#191)`. Для рисования таблиц могут понадобиться символы псевдографики, в табл. 11.2 приведены их коды.

Таблица 11.2 ▾ Коды символов псевдографики

Символ	Код
┌	218
┐	191
┘	217
└	192
├	195
┤	180
┬	194
┴	193
─	196
│	179

Для того чтобы отобразить символ на экране дисплея, можно поступить следующим образом: нажать клавишу ***Alt*** *и, не отпуская ее, набрать код символа, затем отпустить* ***Alt****, после чего на экране дисплея появится необходимый символ.*

Ниже приведена программа, которая выводит массивы x и y в том виде, как они представлены в табл. 11.3.

Таблица 11.3 ▾ Вывод программой массивов x и y

x	y
12	34
45	176
23	78
...	...
2	909

```
uses crt;
var
  x, y: array [1..10] of integer;
  i: integer;
begin
  for i:=1 to 10 do
  begin
     write('x(',i,')='); readln(x[i])
  end;
  for i:=1 to 10 do
```

```
  begin
     write('y(',i,')='); readln(y[i])
  end;
  clrscr;
  write(#218);
  for i:=1 to 5 do
     write(#196);
  write(#194);
  for i:=1 to 5 do
     write(#196);
  writeln(#191); write(#179);
  write(' x ',#179,' y ');
  writeln(#179); write(#195);
  for i:=1 to 5 do
     write(#196);
  write(#197);
  for i:=1 to 5 do
     write(#196);
  writeln(#180);
  for i:=1 to 10 do
  begin
     write(#179); write(' ',x[i]:3,' ');
     write(#179); write(' ',y[i]:3,' ');
     writeln(#179)
  end;
  write(#192);
  for i:=1 to 5 do
     write(#196);
  write(#193);
  for i:=1 to 5 do
     write(#196);
  writeln(#217);
  writeln('нажмите любую клавишу для окончания ');
  repeat until keypressed;
end.
```

11.3. Использование модуля PRINTER

После подключения модуля PRINTER в предложении `uses` можно выводить информацию не только на экран, но и на принтер. Для этого первым параметром в операторе `write` (`writeln`) необходимо указать `lst`, например:

```
writeln ('x=', x:1:3, ' y=', y:1:3); { Вывод на экран. }
writeln (lst, 'x=', x:1:3, ' y=',y:1:3); { Вывод на принтер. }
```

11.4. Использование модуля DOS

В модуле DOS реализовано большое количество процедур работы с операционной системой. Рассмотрим некоторые из них.

11.4.1. Работа с датой и временем

Для опроса и установки даты и времени в модуле DOS существует четыре процедуры (табл. 11.4).

Таблица 11.4 ▼ Процедуры работы с датой и временем

Объявление процедуры	Описание процедуры
`SetDate(Year, Month, Day : Word)`	Устанавливает новую системную дату в операционной системе: `Year` (год) – значение должно находиться в диапазоне от 1980 до 2099; `Month` (месяц) – значение находится в пределах от 1 до 12; `Day` (число) – значение находится в пределах 1–31. Если введенная дата недопустима, то системная дата не изменяется
`GetDate(Var Year, Month, Day, DayOfWeek: Word)`	Возвращает системную дату: `Year` – текущий год, `Month` – месяц, `Day` – число, `DayOfWeek` – день недели (0 – воскресенье, 1 – понедельник и т.д.)
`SetDate(Hour, Minute, Second, Sec100 : Word)`	Устанавливает новое системное время в операционной системе: `Hour` (часы) – значение должно находиться в диапазоне от 0 до 23; `Minute` (минуты), `Second` (секунды) – значения должны находиться в диапазоне 0 до 59; `Sec100` (сотые доли секунды) – значения должны находиться в пределах от 0 до 99
`GetDate(Var Hour, Minute, Second, Sec100 : Word)`	Возвращает системное время

Пример 11.3. Используя процедуры работы с датой, определить день недели даты, введенной с клавиатуры.

Для этой программы понадобится логическая функция, которая проверяет корректность любой даты. Входными параметрами процедуры будут переменные типа `word`: `Y` (год), `M` (месяц) и `D` (день). Если дата корректна, функция возвращает `true`, иначе `false`.

Заголовок процедуры может иметь вид:

```
function Correctly(Y,M,D:word):boolean;
```

Ниже приведен текст всей программы с комментариями.

```
{ Подключение модулей CRT и DOS. }
uses CRT,DOS;
va
  Year1,Month1,Day1,Year,Month,Day,DayofWeek:word;
```

```
{ Типизированная константа массив строк, где хранятся названия дней недели
на русском языке. }
Const
  days:array[0..6] of
  string[11]=('Воскресенье','Понедельник','Вторник','Среда','Четверг',
'Пятница','Суббота');
{ Логическая функция проверки корректности даты. }
function Correctly(Y,M,D:word):boolean
begin
{ Если год до 1980 или после 2099 (см. табл. 11.4), }
  if (Y<1980) and (Y>2099) then
{ то дата некорректна. }
     Correctly:=false
  Else
{ Если номер месяца > 12, }
     if M>12 then
{ то дата некорректна. }
        Correctly:=false
     Else
{ Если год високосный и в феврале число больше 29, }
        if (Y mod 4 = 0) and (M=2) and (D>29) then
{ то дата некорректна. }
           Correctly:=false
        Else
{ Если год не високосный и в феврале число больше 28, }
           if (Y mod 4 <> 0) and (M=2) and (D>28) then
{ то дата некорректна. }
              Correctly:=false
{ Если в январе, марте, мае, июле, августе, октябре, декабре больше 31
дня, }
              Else if ((M=1) or (M=3) or (M=5) or (M=7) or (M=8) or (M=10)
or (M=12)) and (D>31) then
{ то дата некорректна. }
           Correctly:=false
           Else
{ Если в апреле, июне, сентябре или ноябре больше 30 дней, }
           if ((M=4) or (M=6) or (M=9) or (M=11)) and (D>30) then
{ то дата некорректна. }
           Correctly:=false
           Else
{ Если номер месяца или дня равен 0, }
           if (M=0) or (D=0) then
{ то дата некорректна. }
           Correctly:=false
{ Во всех остальных случаях дата корректна. }
           else Correctly:=True
end;
```

```
begin
  ClrScr;
{ Ввод года, месяца и числа, день недели которого надо определить. }
  writeln('ВВОД ДАТЫ');
  writeln('Введите число');
  readln(Day);
  writeln('Введите месяц');
  readln(Month);
  writeln('Введите год');
  readln(Year);
{ Проверка корректности даты. }
  if Correctly(Year,Month,Day) then
{ Если дата корректна, }
  begin
     writeln('Дата корректна');
{ Сохраняем системную дату в переменных Year1, Month1, Day1 и DayofWeek
(день недели). }
     GetDate(Year1,Month1,Day1,DayofWeek);
{ Устанавливаем введенную дату как системную. }
     SetDate(Year,Month,Day);
{ Обращаемся к функции определения системной даты, которая в переменную
DayOfWeek вернет номер дня недели введенной с клавиатуры даты. }
     GetDate(Year,Month,Day,DayofWeek);
{ Восстанавливаем корректную системную дату из переменных Year1, Month1,
Day1. }
     SetDate(Year1,Month1,Day1);
{ Вывод сообщения о дне недели. Подумайте, что проверяет оператор if. }
     if DayofWeek=0 then
        writeln(Day,'.',Month,'.',Year,' было ',
        days[DayofWeek],'!!!')
     else
        if (DayofWeek=6) or (DayofWeek=5) then
        writeln(Day,'.',Month,'.',Year,' была ',
        days[DayofWeek],'!!!')
     else
        writeln(Day,'.',Month,'.',Year,' был ',
        days[DayofWeek],'!!!')
  end
  else
     writeln('Дата некорректна')
end.
```

11.4.2. Процедуры и функции работы с дисками, файлами и каталогами

В модуле DOS есть две функции анализа дисков:

```
DiskFree(Drive:Word):longint;
DiskSize(Drive:Word):longint;
```

Функция DiskFree возвращает объем свободного пространства, а функция DiskSize – вместимость на диске, определяемом параметром Drive, который может принимать следующие значения: 0 – текущий диск, 1 – дисковод A, 2 – дисковод B, 3 – диск C и т.д.

В модуле DOS есть несколько функций работы с файлами[1]. Рассмотрим некоторые из них.

Процедуры FindFirst и FindNext

Процедура FindFirst предназначена для поиска первого подходящего файла на диске. Ее заголовок имеет вид:

```
FindFirst(Path:String; Attr:word; var FR:SeachRec);
```

Path – строка, в которой хранятся параметры поиска, например для поиска всех файлов с расширением .pas из каталога C:\BP\BIN нужно указать строку C:\BP\BIN*.PAS.

Attr – атрибут искомых файлов. В качестве него можно использовать одну из определенных в модуле DOS констант (табл. 11.5).

Таблица 11.5 ▾ Константы атрибутов, определенные в модуле DOS

Константа	Описание
ReadOnly=$01	Файл с атрибутом «Только для чтения»
Hidden=$02	Файл с атрибутом «Скрытый»
SysFile=$04	Файл с атрибутом «Системный»
VolumeID=$08	Метка диска
Directory=$10	Каталог
Archive=$20	Архивный файл
AnyFile=$3F	Любой файл

Атрибуты между собой можно складывать. Например, чтобы найти файлы с атрибутами «Скрытый» (Hidden) и «Только для чтения» (ReadOnly) в качестве Attr следует указать ReadOnly+Hidden. Константа AnyFile представляет собой сумму всех других констант, к ней не следует добавлять другие константы, а вот в вычитании из нее констант есть определенный смысл. Например, если вы хотите исключить из поиска имена каталогов, то в качестве Attr можно указать AnyFile-Directory.

Последний параметр возвращает параметры первого найденного файла, для работы с параметрами найденных файлов в модуле DOS введен тип данных SearchRec:

[1] Следует учесть, что файловую систему вашего компьютера Турбо Паскаль «видит» как файловую систему MS DOS: длина имени ограничена 8 символами (еще 3 символа – расширение).

```
Type SearchRec = Record
  Fill: Array [1..21] Of Byte;
{ Параметр зарезервирован операционной системой и не должен изменяться. }
  Attr: Byte; { Содержит атрибуты файла. }
  Time: Longint; { Содержит дату и время создания файла. }
  Size: Longint; { Содержит размер файла в байтах. }
  Name: String[12]; { Содержит имя файла. }
  End;
```

Дата и время хранятся во внутреннем формате операционной системы в виде переменной типа Longint. Для работы с датой и временем в модуле DOS введен тип данных DataTime.

```
Type
  DataTime=Record
  Year, Month, Day, Hour, Minute, Second : Word;
  End;
```

Для преобразования даты из внутреннего формата операционной системы к типу данных DataTime в модуле DOS предусмотрена процедура

```
UnPackTime(Var DT: DataTime; Vat T:Longint)
```

Обратное преобразование осуществляется процедурой

```
PackTime(Vat T:Longint; DT: DataTime).
```

Процедура FindFirst осуществляет поиск первого файла и возвращает информацию о нем в параметре FR. Кроме этого FindFirst подготавливает специальную системную запись в памяти, которая будет использоваться процедурой поиска следующего файла. Поэтому, для того чтобы найти следующий файл, удовлетворяющий условиям поиска, необходимо сразу после процедуры FindFirst вызвать процедуру поиска следующего файла FindNext. У нее один параметр-переменная FR типа SearchRec, в котором будет возвращаться информация о следующем файле, удовлетворяющем условиям поиска.

Если поиск с помощью функций FindFirst, FindNext окончился неудачей, то системная константа DosError типа Integer возвращает значение кода ошибки операционной системы. При успешном поиске DosError=0.

Рассмотрим использование процедур FindFirst, FindNext на примере следующей задачи.

Пример 11.4. Вывести на экран список всех файлов с расширением .pas, которые находятся в каталоге C:\BP\BIN.

```
uses CRT,DOS;
{ Критерий поиска хранится в константе S. }
const
  S:string='C:\BP\BIN\*.pas';
var
  FR:SearchRec;
  Attr:Word;
begin
  ClrScr;
{ Определяем атрибуты файла, который мы будем искать, это файл, но не
каталог, AnyFile-Directory. }
  Attr:=AnyFile-Directory;
{ Осуществляем первый поиск любого файла с расширением .pas в каталоге
C:\BP\BIN. Результат поиска будет хранится в переменной FR. }
  FindFirst(S,Attr,FR);
{ Если поиск был успешен, то }
  if DosError=0 then
  begin
{ входим в цикл repeat .. until. }
     Repeat
{ Печатаем имя очередного найденного файла. }
        writeln(FR.Name);
{ Поиск очередного файла. }
        FindNext(FR)
{ Выход из цикла, если последний поиск был неудачен. }
     until DosError<>0;
  end
{ Если поиск первого файла был неудачен, то }
  Else
{ вывод соответствующего сообщения. }
     writeln('Таких файлов нет')
end.
```

Функция FSearch

Для поиска файлов можно также воспользоваться функцией `FSearch`[1]:

```
FSearch(Path:PathStr; DirList:String):PathStr;
```

`Path` определяет имя файла, который будем искать. В строке `DirList` хранится список каталогов, разделенных точкой с запятой, в которых осуществляется поиск. Функция возвращает имя файла из параметра `Path` или пустую строку, если файл не найден.

[1] Тип Pathstr определен в модуле DOS как string [79].

Процедура FSplit

Процедура FSplit служит для разбиения полного имени файла на части. Обращение к ней имеет вид[1]:

```
FSplit(Y:PathStr; var Disk:DirStr; var Name:NameStr; var Ext:ExtStr);
```

где Y – полное имя файла на диске;
DirStr – путь к файлу;
NameStr – имя файла;
ExtStr – расширение имени файла.

Процедура из полного имени файла Y формирует путь к файлу DirStr, имя файла NameStr и его расширение ExtStr.

Процедуры GetFAttr и SetFAttr

Для установки атрибутов файлу следует пользоваться процедурой

```
SetFAttr( var f; Attr :word);
```

Файлу, связанному с файловой переменной f, присваивается атрибут, определяемый параметром Attr. Для корректного присваивания атрибутов файлу файловая переменная должна быть связана с реальным файлом на диске с помощью процедуры Assign.

Процедура

```
GetFAttr( var f; var Attr :word)
```

возвращает атрибуты файла, связанного с файловой переменной f в переменной Attr.

При работе с процедурами GetFAttr *и* SetFAttr *файл не должен быть открыт.*

11.5. Создание собственных модулей

Модуль – это автономная программная единица, включающая в себя различные компоненты: константы, переменные, типы, процедуры и функции. До этого момента мы рассматривали, как пользоваться уже существующими программными модулями. Теперь рассмотрим как создавать личный модуль.

Модуль имеет следующую структуру:

- UNIT – имя модуля;
- INTERFACE – интерфейсная часть;

[1] В модуле DOS определены следующие типы данных: DirStr=string[67], NameStr=string[8], ExtStr=string[4].

- IMPLEMENTATION – исполняемая часть;
- BEGIN – инициирующая часть;
- END.

Заголовок модуля состоит из служебного слова UNIT и следующего за ним имени. Причем имя модуля должно совпадать с именем файла, в котором он хранится. Модуль EDIT должен храниться в файле edit.pas.

Интерфейсная часть начинается служебным словом INTERFACE, за которым находятся объявления всех глобальных объектов модуля: типов, констант, переменных и подпрограмм. Эти объекты будут доступны всем модулям и программам, вызывающим данный модуль.

Исполняемая часть начинается служебным словом IMPLEMENTATION и содержит описания подпрограмм, объявленных в интерфейсной части. Здесь же могут объявляться локальные объекты, которые используются только в интерфейсной части и остаются недоступными программам и модулям, вызывающим данный модуль.

В *инициирующей части* размещаются исполняемые операторы, содержащие некоторый фрагмент программы. Эти программы исполняются до передачи управления основной программе и обычно используются для подготовки ее работы. Инициирующая часть может отсутствовать вместе с начинающим ее словом begin.

После написания модуля его следует откомпилировать, предварительно отметив пункт **Destination Disk** в меню **Compile**. В результате будет создан файл с расширением .tpu. Затем этот модуль можно использовать так же, как любой стандартный.

Пример модуля изображения графиков приведен в следующей главе.

11.6. Задания по теме «Модули»

При выполнении заданий по обработке файлов, кроме подпрограмм, описанных в этой главе, следует пользоваться процедурами и функциями по обработке файлов, представленными в главе 7.

1. Написать программу движения курсора по экрану, предусмотреть обработку клавиш ↑, →, ↓, ←, **Home**, **End** и обработку клавиш на границах экрана. Выход из программы осуществляет при нажатии комбинации клавиш **Ctrl+X**.
2. Найти самой большой файл с расширением .pas в заданном каталоге.
3. Удалить два самых больших файла с расширением .bak в каталоге C:\BP\BIN.

4. Найти самый маленький файл в корневом каталоге диска C:.
5. Переименовать самый маленький файл с расширением .bak из каталога C:\BP\BIN в файл с расширением ~ba.
6. Вывести на экран все файлы из текущего каталога, созданные в этом месяце.
7. В самый большой файл с расширением .txt текущего каталога дописать вашу фамилию.
8. Вывести на экран самый большой файл из текущего каталога, созданный в этом году.
9. Удалить из текущего каталога все файлы с атрибутом «Только для чтения», созданные в прошлом году.
10. Переименовать все файлы с расширением .txt из текущего каталога, созданные в среду.
11. Вычислить, сколько файлов в каталоге C:\BP\BIN имеют размер менее 1 Мб.
12. Распечатать на принтере содержимое файлов с расширением .pas из каталога C:\BP, созданных в ваш день рождения.
13. Проверьте, правда ли, что 28 июня, 5 июля, 13 сентября и 13 декабря попадают на один день недели. Если да, то определите, какой это будет день недели в году.
14. Выведите на экран високосные года нынешнего столетия, когда День победы попадает на вторник.
15. Будут ли года в этом столетии, когда 29 февраля попадет на воскресенье.

Глава 12 Графические средства Турбо Паскаля

Начиная с версии 4.0 в состав Турбо Паскаля включен графический модуль GRAPH, который содержит в общей сложности более 50 процедур и функций, предоставляющих программисту различные возможности управления графическим экраном.

Стандартный режим работы дисплея ПК под управлением DOS – текстовый, поэтому любая программа, использующая графические средства, должна переключить экран в графический режим. После завершения графической части программы можно вернуть дисплей ПК в текстовый режим.

12.1. Краткая характеристика графических режимов

Настройка графических процедур на работу с конкретным адаптером достигается за счет подключения нужного графического драйвера. *Драйвер* – специальная программа, управляющая различными устройствами компьютера. Графический драйвер управляет дисплейным адаптером в графическом режиме. Графические драйверы разработаны фирмой Borland практически для всех типов адаптеров. Они располагаются на диске в виде файлов с расширением .bgi (от английского Borland Graphics Interface), например cga.bgi – драйвер для CGA-адаптера (дисплея), egavga.bgi – драйвер для адаптеров EGA и VGA. Графические возможности конкретного адаптера определяются расширением

экрана, то есть общим количеством точек, а также количеством цветов, которыми может отображаться конкретная точка. Кроме того, многие адаптеры могут работать с несколькими графическими страницами – областями оперативной памяти, используемыми для создания «карты» экрана, то есть содержащими информацию о светимости (цвете) каждой точки. Рассмотрим основные графические режимы под управлением DOS.

Адаптер CGA (Color Graphics Adapter – цветной графический адаптер) имеет пять графических режимов. Четыре из них соответствуют низкой разрешающей способности экрана (320 точек по горизонтали и 200 точек по вертикали, то есть 320×200) и отличаются только набором допустимых цветов – палитрой. Каждая палитра состоит из трех цветов, а с учетом несветящегося цвета – из четырех. Пятый графический режим соответствует разрешающей способности экрана 640×200, палитра в этом случае состоит из единственного цвета, а с учетом несветящегося символа в этом режиме доступны два цвета. В режиме CGA адаптер может находиться в одном из следующих режимов:

- режим 1 – разрешающая способность 320×200, палитра 0 (светло-зеленый, розовый, желтый);
- режим 2 – разрешающая способность 320×200, палитра 1 (светло-бирюзовый, малиновый, белый);
- режим 3 – разрешающая способность 320×200, палитра 2 (зеленый, красный, коричневый);
- режим 4 – разрешающая способность 320×200, палитра 3 (бирюзовый, фиолетовый, светло-серый);
- режим 5 – разрешающая способность 640×200, но каждая точка в этом случае может либо светиться каким-то заранее определенным и одинаковым для всех точек цветом, либо не светиться вовсе, то есть палитра данного режима содержит два цвета.

В графическом режиме адаптер CGA использует одну страницу.

Адаптер EGA (Enhanced Graphics Adapter – усовершенствованный графический адаптер) поддерживает все режимы адаптера CGA. Кроме того, в нем возможны режимы низкого (640×200, 16 цветов, 4 страницы) и высокого разрешения (640×350, 16 цветов, 2 страницы).

Адаптер MCGA (Multi-Color Graphics Adapter – многоцветный графический адаптер) совместим с CGA и имеет еще один режим (640×480, 2 цвета, 1 страница).

Адаптер VGA (Video Graphics Array – графический видеомассив) поддерживает режимы адаптеров CGA и EGA и дополняет их режимом высокого разрешения (640×480, 16 цветов, 1 страница).

Адаптер SVGA (SuperVGA) с разрешением 1024×768 использует 256 цветов.

Адаптер фирмы Hercules (адаптер HGC) имеет разрешение 720×348 и два цвета.

12.2. Управление графическими режимами

Процедура `InitGraph` переводит экран в графический режим. Заголовок процедуры:

```
InitGraph(var grdr, grmd:integer, path:string);
```

Параметр `grdr` определяет тип графического адаптера. В качестве него могут использоваться константы, определенные в модуле GRAPH:

- `DETECT=0` – режим автоматического определения;
- `CGA=1`;
- `MCGA=2`;
- `EGA=3`;
- `EGA64=4`;
- `EGAMONO=5`;
- `IBM8514=6`;
- `HERCMONO=7`;
- `ATT400=8`;
- `VGA=9`;
- `PC3270=10`.

Если параметру `grdr` присвоить значение `detect`, то система переходит в режим самоопределения. При возможном переключении в графический режим система переходит в него с максимальным разрешением.

Параметр `grmd` – номер (значение) режима, допустимого при данном адаптере. Рассмотрим допустимые режимы для адаптеров CGA, EGA и VGA.

Адаптер CGA допускает пять графических режимов:

- `CGAC0=0` – 320×200, 4 цвета, палитра 0;
- `CGAC1=1` – 320×200, 4 цвета, палитра 1;
- `CGAC2=2` – 320×200, 4 цвета, палитра 2;
- `CGAC3=3` – 320×200, 4 цвета, палитра 3;
- `CGAHi=4` – 640×200, 2 цвета.

Адаптер EGA поддерживает два режима:

- `EGALo=0` – 640×200, 16 цветов, 4 страницы;
- `EGAHi=1` – 640×350, 16 цветов, 2 страницы.

Адаптер VGA поддерживает три режима:

- VGALo=0 – 640×200, 16 цветов, 4 страницы;
- VGAMed=1 – 640×350, 16 цветов, 2 страницы;
- VGAHi=2 – 640×480, 16 цветов, 1 страница.

Третий параметр pathstr – строковая константа или переменная, которая указывает путь к каталогу, где находится драйвер (файл с расширением .bgi).

Если все параметры указаны верно (и, кроме того, подключен модуль GRAPH), то экран дисплея переключится в графический режим.

Процедура InitGraph возвращает два значения: grdr, grmd. Если значение grdr определено как detect, то InitGraph вернет конкретные значения grdr и grmd. Только после того как система перейдет в графический режим, можно пользоваться всеми функциями и процедурами модуля GRAPH (рисование линий, окружностей и т.д.). И еще одно замечание по поводу параметра pathstr. Система сначала ищет графический драйвер в текущем каталоге, а затем в том, который указан в pathstr. Следовательно, если поместить драйвер в текущий каталог, то проблема его поиска отпадает.

Процедура без параметров CloseGraph завершает работу в графическом режиме и переводит компьютер в текстовый режим.

В случае неудачного завершения инициализации процедура InitGraph возвращает ошибку в параметре grdr. В случае ошибки параметр grdr отрицательный и может принимать значения, указанные в табл. 12.1.

Таблица 12.1 ▼ Значение ошибок в параметре grdr

Значение	Причина ошибки
–2	Нет графического адаптера
–3	Не найден файл драйвера
–4	Ошибка в файле (его коде)
–5	Не хватает памяти для загрузки драйвера
–10	Невозможный режим для выбранного драйвера
–15	Нет такого драйвера

В модуле GRAPH есть функция GraphResult, которая тоже возвращает код результата (тип integer) последнего вызова одной из графических функций. Если GraphResult меньше 0, то произошла ошибка (GraphResult возвращает 0, если ошибки не было). Для быстрой выдачи простого сообщения о типе ошибки графической системы используется функция, преобразующая результат вызова GraphResult в сообщение, которое можно вывести на экран процедурой Write или OutTextXY. Эта функция объявлена так:

```
GraphErrorMsg(ErrorCode):String
```

Константы ошибок выглядят следующим образом:

- grok = 0 – нет ошибок;
- grInitGraph = –1 – не инициирован графический режим;
- grNotDetected = –2 – не определен тип драйвера;
- grFileNotFind = –3 – не найден графический драйвер;
- grInvalidDriver = –4 – неправильный тип драйвера;
- grNoLoadMem = –5 – нет памяти для размещения драйвера;
- grNoScanMem = –6 – нет памяти для просмотра областей;
- grNoFloodMem = –7 – нет памяти для закраски областей;
- grFontNotFound = –8 – не найден файл со шрифтом;
- grNoFontMem = –9 – нет памяти для размещения шрифта;
- grInvalidMode = –10 – неправильный графический режим;
- grError = –11 – общая ошибка;
- grIOError = –12 – ошибка ввода-вывода;
- grInvalidFont = –13 – неправильный формат шрифта;
- grInvalidFontNum = –14 – неправильный номер шрифта.

Процедура RestoreCRTMode служит для кратковременного переключения в текстовый режим, при этом память, выделенная для размещения графического драйвера, не освобождается.

Функция GetGraphMode возвращает значение типа integer – код режима работы графического адаптера.

Процедура SetGraphMode переключает адаптер в другой режим работы. Обращение к процедуре:

```
SetGraphMode(Mode),
```

где Mode – переменная типа integer, в которой содержится код устанавливаемого режима.

Представляет интерес процедура определения типа графического адаптера и его режима, доступная до переключения экрана в графический режим. Обращение к процедуре DetectGraph имеет вид:

```
Detectgraph (Drv, Md),
```

где Drv – переменная типа integer, возвращающая тип адаптера; Md – переменная типа integer, возвращающая максимально возможный режим для данного адаптера.

Процедура GetModeRange возвращает диапазон возможных режимов работы заданного графического адаптера. Обращение к процедуре имеет вид:

```
GetModeRange(Drv,Min,Max),
```

Drv –переменная типа integer, в которой передается в процедуру тип графического адаптера. Min, Max – переменные типа integer, в которые возвращаются нижнее и верхнее из возможных значений режима. Если задано неправильное значение параметра Drv, то процедура в переменных Min и Max вернет значение –1.

Следующая программа демонстрирует возможности переключения между текстовым и графическим режимами, а также анализ корректности инициализации последнего.

```
Uses Crt, Graph;
var GrDr,GrMd,Error:integer;
Begin
  GrDr:=Detect;
{ Переход в графический режим. }
  InitGraph(GrDr,GrMd,'c:\bp\bin');
{ В переменную Error записывается код ошибки после оператора InitGraph
(ошибка инициализации графики). }
  Error:=GraphResult;
  if Error<>grOk then
  Begin
{ Если была ошибка, выводится стандартное сообщение о ней }
     Writeln(GraphErrorMSG(error));
{ и прекращается выполнение программы. }
     Halt;
  End;
{ Если ошибки инициализации не было, то оператор будет выполняться. }
  OutText ('this is graph mode');
{ Вывод текста в графическом режиме. }
  Delay(2000);   { Задержка на две секунды. }
  RestoreCRTMode;   { Переключение в текстовый режим. }
  Writeln('а это текстовый режим');
  Delay(3000);
  SetGraphMode(GetGraphMode);   { Возврат в графический режим. }
  OutText ('This is graph mode');
  repeat until keypressed; { Ожидание нажатия клавиши. }
  CloseGraph;
End.
```

12.3. Некоторые графические процедуры и функции

После перехода дисплея в графический режим система координат определена так, как показано на рис. 12.1.

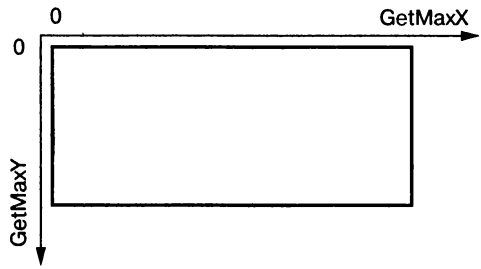


Рис. 12.1 ▾ Система координат в графическом режиме

В отличие от текстового режима, единицей информации в графическом режиме является точка. Система координат в графическом режиме расположена следующим образом: начало координат (точка с координатами 0,0) находится в левом верхнем углу, ось *OX* направлена слева направо (от 0 до `GetMaxX`), ось *OY* направлена сверху вниз (от 0 до `GetMaxY`). Функции `GetMaxX` и `GetMaxY` возвращают значения типа `word`, содержащие максимальные координаты экрана в текущем режиме работы по горизонтали и по вертикали соответственно.

Функции `GetX` и `GetY` возвращают значения типа `integer`, содержащие текущие координаты графического курсора по горизонтали и вертикали соответственно.

Процедура `MoveTo` устанавливает новые текущие координаты графического курсора. Обращение к ней имеет вид:

```
MoveTo(X,Y),
```

где `x`, `y` – переменные типа `integer` с новыми координатами графического курсора по горизонтали и вертикали.

Процедура `MoveRel` устанавливает новое положение графического курсора в относительных координатах. Обращение к ней имеет вид:

```
MoveRel(dx,dy),
```

где `dx`, `dy` – переменные типа `integer`, которые содержат приращение координат графического курсора по горизонтали и вертикали относительно его текущего положения.

Процедура `SetColor` устанавливает текущий цвет выводимых линий и символов. Обращение к ней имеет вид:

```
SetColor(color),
```

где `color` – переменная типа `word`, в которой задается цвет. Константы задания цвета в модуле GRAPH определены точно так же, как и в модуле CRT.

Процедура SetBkColor устанавливает цвет фона. Обращение к ней выглядит так:

```
SetBkColor(color),
```

где color – переменная типа word, определяющая цвет фона.

В отличие от текстового режима, в графическом режиме цвет фона может быть любым. Установка нового цвета фона немедленно изменяет цвет графического экрана. Нельзя создать изображение двух участков, которые имеют разный цвет фона. В работе с графикой в режиме адаптера CGA есть сложности со сменой цвета фона: в частности, данная процедура не рекомендуется по той причине, что она изменяет и цвет активных точек.

Процедура PutPixel выводит точку с заданным цветом. Обращение к ней таково:

```
PutPixel(x,y,color),
```

где x, y – константы или переменные типа integer, задающие координаты точки.

Процедура Line рисует линию с указанными координатами начала и конца. Обращение к процедуре имеет вид:

```
line(x1,y1,x2,y2),
```

где x1, y1, x2, y2 – переменные или константы типа integer, (x1,y1) – координаты начала линии, (x2,y2) – координаты конца линии. Линия рисуется текущим цветом.

Процедура SetViewPort устанавливает прямоугольное окно на графическом экране. Обращение к ней выглядит так:

```
SetViewPort(x1,y1,x2,y2,clip),
```

где x1, y1, x2, y2 – переменные типа integer, содержащие координаты левого верхнего (x1,y1) и правого нижнего (x2,y2) углов окна. Clip – переменная или выражение типа boolean, определяющая удаление не вмещающихся элементов изображения. Если Clip=True, то последние отсекаются; в противном случае границы окна игнорируются.

Процедура ClearDevice очищает графический экран цветом фона, графический курсор устанавливается в начало координат (левый верхний угол экрана).

Процедура ClearViewPort очищает графическое окно, а если оно было не определено – весь экран.

Для изображения окружности используется процедура Circle. Обращение к ней имеет вид:

```
Circle(x,y,r),
```

где x, y – выражения, константы или переменные типа integer, определяющие координаты центра окружности, а r – выражение типа word – радиус окружности.

Более точно параметр r определяет количество точек в горизонтальном направлении, в вертикальном же эта величина находится с учетом соотношения сторон экрана. Окружность всегда рисуется правильная, если не пользоваться процедурой SetAspectRatio, которая устанавливает масштабный коэффициент соотношения сторон графического экрана. Обращение к ней имеет вид:

```
SetAspectRatio(x,y),
```

где x, y – выражения типа word, которые устанавливают масштабные соотношения сторон.

Процедура GetAspectRatio возвращает два числа, позволяющие оценить соотношение сторон экрана. Обращение выглядит так:

```
GetAspectRatio(x,y),
```

где x, y – выражения типа word. Значения, возвращаемые в этих переменных, позволяют вычислить соотношения сторон графического экрана в точках. Коэффициенты x/y или y/x позволяют строить правильные геометрические фигуры (квадраты, окружности).

Процедура Rectangle предназначена для рисования прямоугольника. Обращение к ней таково:

```
Rectangle(x1,y1,x2,y2),
```

где x1, y1, x2, y2 – выражения типа integer, определяющие координаты верхнего левого и правого нижнего углов экрана.

Рассмотрим теперь несколько примеров, в которых используются описанные ранее процедуры.

Пример 12.1. Пример работы с процедурой SetViewPort [10].

```
program gr_one;
uses Crt,Graph;
Var x,y,l,x11,x12,y11,y12,x22,x21,r,k: integer;
begin
  x:=Detect;
  InitGraph(x,y,'C:\bp\Bgi');
  SetBkColor(White);
```

```
  SetColor(Blue);
  l:=GraphResult;
  if l<>grOk then
     writeln(GraphErrorMsg(l))
  else
  Begin
     x11:=GetMaxX div 60;
     x12:=GetMaxX div 3;
     y11:=GetMaxY div 4;
     y12:=2*y11;
     r:=(x12-x11)div 4;
     x21:=x12*2;
     x22:=x21+x12-x11;
     Rectangle(x11,y11,x12,y12);
     Rectangle(x21,y11,x22,y12);
     SetViewPort(x11,y11,x12,y12,ClipOn);
     for k:=1 to 4 do
          Circle(0,y11,r*k);
     SetViewPort(x21,y11,x22,y12,ClipOff);
     for k:=1 to 4 do
          Circle(0,y11,r*k);
     repeat until keypressed
  end;
end.
```

До нажатия клавиши на экране появится картинка, изображенная на рис. 12.2.

Пример 12.2. Пример использования `SetAspectRatio` [11].

```
uses Graph,Crt;
const r=50;
  dx=1000;
var l1,xa,ya:word;
  d,l,m,k:integer;
begin
  d:=Detect;
  InitGraph(d,m,'c:\bp\bgi');
  l:=GraphResult;
  if l<>GrOk then
  begin
     writeln(GraphErrorMsg(l));
     exit;
  end;
  SetBkColor(White);
  SetColor(Blue);
  GetAspectRatio(xa,ya);
  for k:=1 to 26 do
```

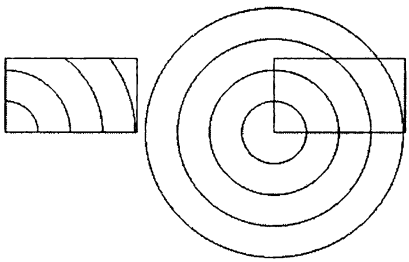


Рис. 12.2 ▾ Результат выполнения программы, приведенной в примере 12.1

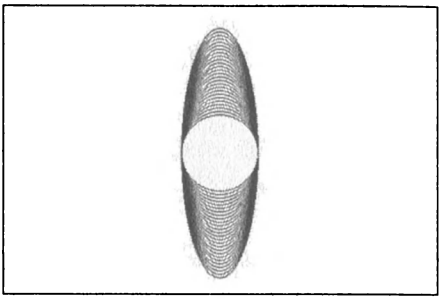


Рис. 12.3 ▾ Результат выполнения программы, приведенной в примере 12.2

```
  begin
     l1:=xa+k*dx;
     SetAspectRatio(l1,ya);
     circle(GetMaxX div 2, GetMaxY div 2,r);
  end;
  repeat until keypressed;
  CloseGraph;
end.
```

Результат работы программы представлен на рис. 12.3.

Рассмотрим еще несколько процедур и функций для работы с точками и линиями.

Функция `GetPixel` возвращает значение типа `word`, содержащее цвет точки с указанными координатами. Ее заголовок имеет вид:

```
Function GetPixel(x,y:integer):word,
```

где `x`, `y` – координаты точки.

Процедура `LineTo` вычерчивает линию от текущего положения графического курсора до положения, заданного его новыми координатами. Обращение к процедуре выглядит так:

```
LineTo(x,y),
```

где `x`, `y` – выражения типа `integer`, указывающие координаты конца линии.

Процедура `LineRel` вычерчивает линию от текущего местонахождения графического курсора до положения, заданного приращениями его координат. Обращение к процедуре имеет вид:

```
LineRel(dx,dy),
```

где dx, dy – выражения типа integer, задающие приращение координат для нового положения курсора. В процедурах рисования линий они определяют текущий цвет и стиль.

Процедура SetLineStyle устанавливает новый стиль вычерчиваемых линий. Вызвать ее можно следующим образом:

```
SetLineStyle(type,pattern,thick),
```

где type, pattern, thick – выражения, переменные или константы типа word, определяющие тип, образец и толщину линии.

Тип линии задан с помощью одной из следующих констант модуля GRAPH:

- SolidIn = 0 – сплошная линия;
- DottedIn = 1 – точечная линия;
- CenternIn = 2 – штрихпунктирная линия;
- DashedIn = 3 – пунктирная линия;
- UserbitIn = 4 – линия, определенная пользователем.

Параметр Pattern учитывается только для линий, определенных пользователем. Он представляет собой 16 бит. Если бит равен 1, то соответствующая точка у образца светящаяся, если 0 – несветящаяся.

Параметр Thick может принимать значение одной из двух констант модуля GRAPH:

- Normwidth = 1 – тонкая линия (1 точка);
- Thickwidth = 3 – толстая линия (3 точки).

Пример 12.3. Программа рисования звездного неба [11].

```
uses crt,Graph;
const n=15000;
var d,e,r,k:integer;
  l,x1,y1,x2,y2:integer;
  x,y:array [1..n] of integer;
begin
  d:=detect;
  InitGraph(d,r,'c:\bp\bgi');
  l:=GraphResult;
  if l<>GrOk then
  begin
     writeln(GraphErrorMsg(l));
     exit;
  end;
  x1:=0;
```

```
  y1:=0;
  x2:=GetmaxX-2;
  y2:=GetMaxY-2;
  Rectangle(x1,y1,x2,y2);
  SetViewPort(x1+1,y1+1,x2-1,y2-1,ClipOn);
  for k:=1 to n do
  begin
     x[k]:=random(x2-x1);
     y[k]:=random(y2-y1);
  end;
  for k:=1 to n do
     PutPixel(x[k],y[k],random(7));
  repeat until keypressed;
  CloseGraph
end.
```

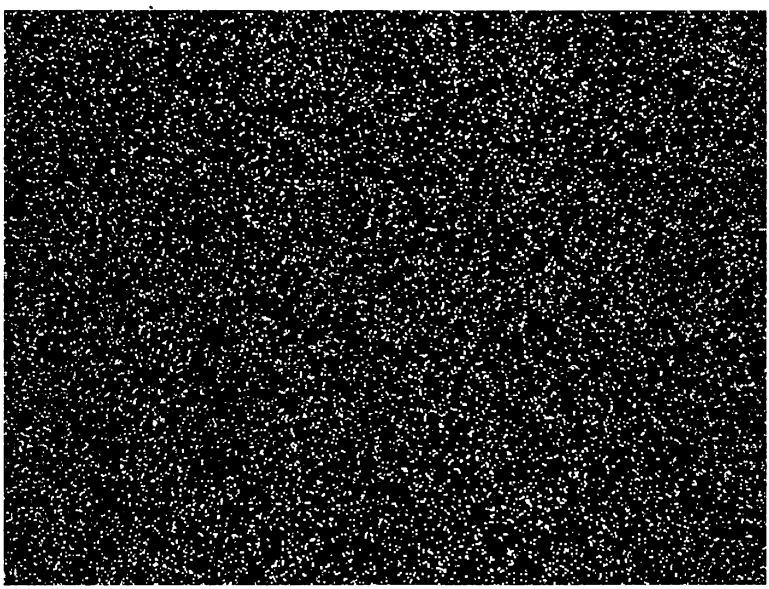


Рис. 12.4 ▾ Результат работы программы рисования звездного неба

Результат работы программы представлен на рис. 12.4.

Процедура `Arc` чертит дугу окружности. Обращение к ней имеет вид:

```
Arc(x,y,begA,endA,r),
```

где x, y – переменные или выражения типа integer, определяющие координаты центра. BegA, EndA, r – переменные или выражения типа word, определяющие начальный и конечный углы дуги и ее радиус. Углы отсчитываются против часовой стрелки в градусах от 0 до 359.

Процедура Ellipse вычерчивает эллиптическую дугу. Обращение к процедуре имеет вид:

```
Ellipse(x,y,BegA,EndA,rx,ry),
```

где x, y – переменные или выражения типа integer, содержащие координаты центра; BegA; EndA – значения типа word, определяющие начальный и конечный углы дуги; rx, ry – переменные типа word, в которых записаны данные горизонтального и вертикального радиуса эллипса в точках.

Процедура SetFillStyle устанавливает тип и цвет заполнения, с помощью которого можно покрывать какие-либо фрагменты изображения периодически повторяющимся узором. Обращение к процедуре имеет вид:

```
SetFillStyle(fill,color),
```

где fill, color – переменные типа word, определяющие тип и цвет заполнения, в качестве типа которого иногда используется одна из констант модуля GRAPH:

- Emptyfill = 0 – заполнение фоном;
- Solidfill = 1 – сплошное заполнение;
- Linefill = 2 – заполнение ——;
- Ltslashfill = 3 – заполнение ////////;
- Slashfill = 4 – заполнение утолщенными ////////;
- Bkslashfill = 5 – заполнение утолщенными \\\\\\\\;
- Ltbkslash = 6 –заполнение \\\\\\\;
- Hatchfill = 7 –заполнение ++++++++;
- Xhatchfill = 8 – заполнение XXXXXXXX;
- Centerleavefill = 9 – заполнение в прямоугольную клетку;
- Widedotfill = 10 – заполнение редкими точками;
- Closedotfill = 11 – заполнение частыми точками;
- Userfill = 12 – узор заполнения определяется пользователем.

В качестве Color можно взять одну из доступных констант. Если установлен тип заполнения Userfill, то для его определения необходимо использовать процедуру SetFillPattern [9–11].

Процедура Floodfill заполняет произвольную замкнутую фигуру, используя текущий стиль заполнения (узор и цвет). Обращение к процедуре имеет вид:

```
FloodFill(x,y,border),
```

где x, y – переменные типа integer, в которых хранятся координаты любой точки внутри замкнутой фигуры, border – цвет пограничной линии.

Процедура Bar заполняет прямоугольную область экрана. Указание на нее следующее:

```
Bar(x1,y1,x2,y2),
```

где x1, y1, x2, y2 – переменные или выражения типа integer, определяющие координаты закрашивающей области. Процедура закрашивает, но не обводит прямоугольники текущими образцами узора и цвета, которые задаются процедурой SetFillStyle.

Процедура Bar3D вычерчивает трехмерное изображение параллелепипеда и закрашивает его переднюю грань. Обращение к процедуре имеет вид:

```
Bar3D(x1,y1,x2,y2,Depth,Top),
```

где x1, y1, x2, y2 – выражения или переменные типа integer, определяющие координаты левого верхнего и правого нижнего углов передней грани, Top – выражение, переменная или константа логического типа. Если Top=True, то вычерчивается верхняя грань параллелепипеда, а в противном случае – нет. Depth – третье измерение («глубина») трехмерного изображения. Для задания этого параметра в модуле GRAPH определены следующие константы:

```
TopOn=True;
TopOff=False.
```

Процедура FillEllipse обводит линией и заполняет эллипс. Обращение к процедуре имеет вид:

```
FillEllipse(x,y,rx,ry),
```

где x, y – переменные типа integer, содержащие координаты центра; rx, ry – переменные типа word, определяющие горизонтальный и вертикальный радиусы эллипса в точках.

Эллипс обводится линией, заданной процедурами SetLineStyle и SetColor, и заполняется с использованием параметров, установленных процедурой SetFillStyle.

Процедура Sector вычерчивает и заполняет эллипсный сектор. Обращение к процедуре имеет вид:

```
Sector(x,y,BegA,EndA,rx,ry).
```

Здесь BegA, EndA – переменные типа word, определяющие соответственно начальный и конечный углы эллипсного сектора. Остальные параметры обращения аналогичны параметрам процедуры FillEllipse.

Процедура PieSlise вычерчивает и заполняет сектор окружности. Обращение к процедуре имеет вид:

```
PieSlise(x,y,BegA,EndA,r).
```

В отличие от процедуры Sector, указывается лишь один радиус r, остальные параметры аналогичны параметрам процедуры Sector.

12.4. Вывод текста в графическом режиме

Специально для графического режима разработаны процедуры, обеспечивающие вывод сообщений различными шрифтами в горизонтальном или вертикальном направлении с изменением размеров в стандартных шрифтах. В шрифтах, разработанных фирмой Borland, отсутствует кириллица, что исключает вывод русскоязычных сообщений. Однако пользователями разработаны шрифты, позволяющие выводить русскоязычные сообщения. Рассмотрим более подробно все стандартные средства модуля GRAPH для вывода текста.

Процедура OutText выводит текстовую строку начиная с текущего положения графического курсора. Обращение к процедуре имеет вид:

```
OutText(Txt),
```

где Txt – строковая переменная или константа. Строка выводится в соответствии с установленным стилем и выравниванием. Если текст выходит за границы экрана, то в случае стандартного шрифта он не выводится, а при использовании штриховых шрифтов отсекается.

Процедура OutTextXY выводит строку начиная с заданного места. Обращение к процедуре имеет вид:

```
OutTextXY(x,y,txt),
```

где x, y – переменные или константы типа integer (координаты точки вывода); txt – выводимая строка (переменная типа string).

А что делать, если необходимо вывести в графическом режиме переменную целого или вещественного типа (то есть численную переменную)? Следует предварительно перенести ее в строку символов с помощью функции Str. Обращение к функции Str имеет вид:

```
Str(x[:N[:M]],s),
```

где x – численная переменная, s – строковая переменная, n, m – формат представления переменной x (n, m – целые). Функция Str преобразует численное значение x в соответствии с форматом в строковое представление s.

Процедура SetTextStyle устанавливает стиль текстового вывода на экран. Обращение к процедуре имеет вид:

```
SetTextStyle(Font,Direct,Size),
```

где Font, Direct, Size – переменные или выражения типа word, определяющие код (номер) шрифта, его направления и размера. Для указания кода шрифта можно использовать следующие предварительно определенные константы:

```
DefaultFont = 0 - точечный шрифт 8x8.
```

Этот шрифт входит в модуль GRAPH и доступен в любой момент. Он используется по умолчанию. Для того чтобы корректно работать с русскими символами при выводе текста в графическом режиме, драйвер кириллицы должен поддерживать графический режим.

Все остальные шрифты векторные. Они отличаются богатыми изобразительными возможностями, главная же их особенность заключается в легкости изменения размеров без существенного ухудшения качества изображения. Каждый шрифт хранится в отдельном файле с расширением .chr. Чтобы шрифт был доступен, соответствующий файл должен находиться в текущем каталоге. Если в файле с расширением .chr есть символы кириллицы (файл русифицирован), тогда процедура OutText будет выводить сообщение на русском языке (это вообще отдельная, довольно сложная проблема):

- TriplexFont = 1 – утроенный шрифт Trip.chr;
- SmallFont = 2 – уменьшенный шрифт litt.chr;
- SansSerifFont = 3 – прямой шрифт;
- GothicFont = 4 – готический шрифт.

Эти константы определяют все шрифты для версий 4.0–6.0. В версии 7.0 набор шрифтов значительно расширен, однако для новых шрифтов не предусмотрены соответствующие мнемонические константы – надо просто указать номер шрифта. При обращении к SetTextStyle можно использовать следующие константы:

- 5 scri.chr – рукописный шрифт;
- 6 simp.chr – одноштриховый шрифт типа Courier;
- 7 tscr.chr – наклонный шрифт типа Times Italic;
- 8 lcom.chr – шрифт типа Times Roman;
- 9 euro.chr – шрифт типа 6 увеличенный;
- 10 bold.chr – крупный двухштриховой шрифт.

В качестве направления Direct выдачи текста можно использовать следующие константы модуля GRAPH:

- HorizDir = 0 – слева направо;
- VertDir = 1 – снизу вверх.

Размер выводимых символов (параметр Size) принимает значение от 1 до 10 (точечный шрифт от 1 до 32). Если значение параметра равно 0, то устанавливается размер 1, если больше 10 – 10.

Пример 12.4. Рассмотрим программу использования различных шрифтов [10, 11].

```
Uses Crt,graph;
Var i,d,m: integer;
Begin
  d:= detect;
  InitGraph(d,m,'c:\bp\bgi');
  For i:=1 to 10 do
  begin
     SetTextStyle(i,0,0);
     OutTextXY(20,20+(i-1)*18,'Example');
     delay(2000);
  end;
  Repeat until KeyPressed;
  CloseGraph;
End.
```

12.5. Сохранение и выдача изображений

Для работы с изображениями в Паскале предусмотрены следующие процедуры и функции.

Функция ImageSize возвращает размер памяти в байтах, необходимый для размещения фрагмента изображения. Заголовок процедуры:

```
Function ImageSize(X1,Y1,X2,Y2:Integer):word
```

Здесь X1, Y1, X2, Y2 – координаты верхнего левого и правого нижнего углов фрагмента изображения.

Процедура GetImage помещает в память копию прямоугольного фрагмента изображения. Заголовок процедуры имеет вид:

```
GetImage(X1,Y1,X2,Y2:Integer; var Buf)
```

Здесь X1, Y1, X2, Y2 – координаты верхнего левого и правого нижнего углов фрагмента изображения, Buf – переменная или участок кучи (см. главу 10), куда

будет помещена копия видеопамяти с фрагментом изображения. Но следует помнить, что Buf не может превышать 64 Кб (см. главу 10), однако для хранения всего графического экрана может потребоваться участок памяти большего размера. Ведь для хранения одной точки в режиме 256 цветов требуется 1 байт. Можно разделить изображение на несколько участков и хранить каждый из них, но это не совсем удобно. В главе 10 описаны подпрограммы FAR_IMAGESIZE, FAR_GETIMAGE, FAR_PUTIMAGE, позволяющие обойти это ограничение.

Процедура PutImage выводит в заданное место копию фрагмента изображения, ранее помещенную в память процедурой GetImage. Заголовок процедуры имеет вид:

```
Procedure PutImage(X,Y:Integer; var Buf; Mode:Word),
```

где X, Y – координаты левого верхнего угла того участка на экране, куда будет скопирован фрагмент изображения, хранящийся в переменной Buf; Mode – способ копирования изображения. Параметр Mode наиболее часто может принимать следующие значения:

- NormalPut = 0; в этом случае будет стираться часть экрана, а на ее место помещаться изображение, хранящееся в переменной Buf;
- XorPut = 1; данная операция, примененная к тому же месту экрана, откуда была получена копия, сотрет эту же часть экрана. Если данную операцию использовать дважды на одном и том же участке экрана, то в целом вид не изменится. Этот эффект был положен в основу алгоритма создания движущихся объектов. Для создания эффекта движения объект сохраняем в памяти с помощью процедуры getImage, а затем с помощью процедуры putImage перемещаем его по экрану;
- NotPut = 4; операция NotPut стирает часть экрана и на это место помещает содержимое переменной Buf в инверсном виде. Инверсия применяется к цифровому коду цвета точки. Для цвета Red (код 0100) получим цвет LightCyan (код 1011) и т.д.

12.6. Создание движущихся изображений

Создать эффект движения можно одним из двух способов.

Способ 1. Нарисовать объект, затем перерисовать его цветом фона – изображение исчезнет. После этого опять нарисовать его, сместив на несколько пикселей по экрану, перерисовать цветом фона и т.д. Создается эффект движения.

Способ 2. Можно выделить следующие этапы построения движущего изображения:

1. Вывести изображение на экран.
2. С помощью функции `ImageSize` вычислить размер движущегося фрагмента участка экрана.
3. Выделить необходимый участок оперативной памяти с помощью процедуры `GetMem`.
4. Записать в выделенную область памяти нужный фрагмент изображения с помощью процедуры `GetImage`.
5. Вывести изображение на экран с помощью процедуры `PutImage` с параметром `Mode`, равным `XorPut`, что приведет к исчезновению изображения.
6. Повторно вывести изображение на экран, сместив на несколько точек, с помощью `PutImage` с параметром `Mode`, равным `XorPut`.

Повторение пятого и шестого этапов нужное количество раз приведет к эффекту движения.

Пример 12.5. Изобразить движущийся шарик на экране дисплея.

Программа изображения движущегося шарика, используя первый способ создания движущегося изображения:

```
uses crt, graph;
var grdr,grmd,i,j,g:integer;
begin
{ Переводим экран в графический режим. }
  grdr:=detect;
  Initgraph(grdr,grmd,'C:\bp\bgi');
{ Определяем светло-серый цвет фона. }
  SetBkColor(lightGray);
{ Запоминаем цвет фона в переменной g. }
  g:=getbkcolor;
{ Начальная точка изображения шарика на экране. }
  i:=20;j:=20;
{ Если координата X не вышла за границу экрана, то выполняем необходимые
для движения шарика действия в цикле. }
  while i<GetmaxX do
  begin
{ Определяем цвет линий красным. }
     setcolor(red);
{ Определяем цвет заполнения: красный; тип заполнения: сплошной. }
     SetFillStyle(1,red);
{ Изображаем круг радиусом 5 точек, координаты центра круга i, j. }
     FillEllipse(i,j,5,5);
```

```
{ Пауза, изображение как бы замирает на экране. Напоминаем читателю, что
реальное время паузы зависит от быстродействия компьютера, следует
подобрать это значение, в зависимости от тактовой частоты ПК. }
     delay(3000);
{ Определяем цвет линий и заполнения совпадающий с цветом фона, тип
заполнения сплошной. }
     setcolor(g);
     SetFillStyle(1,g);
{ Изображаем цветом фона круг радиусом 5 точек, координаты центра круга i,
j, что приводит к исчезновению круга на экране. }
     FillEllipse(i,j,5,5);
{ Изменяем координаты круга. }
     i:=i+random(3);
     j:=j+random(2)
  end;
  readln;
end.
```

Программа изображения движущегося шарика с использованием второго способа создания движущегося изображения:

```
uses crt, graph;
var grdr,grmd,i,j,l:integer;
  p:pointer;
begin
  grdr:=detect;
  Initgraph(grdr,grmd,'C:\bp\bgi');
  SetBkColor(lightGray);
{ Вычисляем, сколько памяти необходимо для хранения шарика радиусом
5 точек. }
  l:=imageSize(15,15,25,25);
{ Выделяем память хранения шарика. }
  GetMem(p,l);
{ Рисуем красный шарик. }
  setcolor(red);
  SetFillStyle(1,red);
  FillEllipse(20,20,5,5);
{ Сохраняем изображение по адресу P. }
  i:=15;j:=15;
  GetImage(i,j,i+10,j+10,P^);
{ Пауза перед стиранием изображения. }
  delay(3000);
{ Стираем шарик. }
  PutImage(i,j,P^,xorPut);
  while i<GetmaxX do
```

```
  begin
{ Изменяем координаты круга. }
     i:=i+random(3);
     j:=j+random(2);
{ Выводим шарик на экран. }
     PutImage(i,j,P^,xorPut);
     delay(3000);
{ Стираем шарик. }
     PutImage(i,j,P^,xorPut);
  end;
  readln;
  FreeMem(p,l);
end.
```

Рекомендуем читателю попробовать самостоятельно написать программы с движением, используя методы, изложенные в этом разделе.

12.7. Построение графика функции на экране дисплея

Алгоритм построения графика непрерывной функции $y = f(x)$ на отрезке $[a,b]$ состоит в следующем: необходимо построить точки $(x_i, f(x_i))$[1] в декартовой системе координат и соединить их прямыми линиями. Чем больше точек будет изображено, тем более плавным станет построенный график.

При переносе этого алгоритма на экран дисплея следует учесть размеры экрана (ось X направлена слева направо, и ее координаты лежат в пределах от 0 до `GetMaxX`; ось OY направлена вниз, и ее координаты находятся в пределах от 0 до `GetMaxY`). Необходимо помнить, что значения координат X и Y должны быть целыми.

Далее следует все точки из «бумажной» системы координат (X изменяется в пределах от a до b, Y изменяется от минимального (`min`) до максимального (`max`)) пересчитать в «экранную» (в этой системе координат ось абсцисс обозначим буквой U, которая изменяется от 0 до `GetMaxX`, а ось ординат – буквой V ($V \in$ [0,`GetMaxY`]).

Для преобразования координаты X в координату U построим линейную функцию $cX + d$, которая переведет точки из интервала (a,b) в точки интервала (`X0`, `GetMaxX-XK`)[2]. В связи с тем, что точка a «бумажной» системы координат перейдет

[1] $hx = (b - a)/N$, $x_i = a + (I - 1)hx$, $y_i = f(x_i)$; $i = 1,N$; где N – количество точек на отрезке $[a,b]$.

[2] X0, XK, Y0, YK – отступы от левой, правой, нижней и верхней границ экрана.

в точку X0 «экранной», а точка b – в точку GetMaxX-XK, система линейных уравнений для нахождения коэффициентов c и d имеет вид:

$$\begin{cases} c \cdot a + d = \text{X0} \\ c \cdot b + d = \text{GetMaxX} - \text{XK} \end{cases}.$$

Решив ее, найдем коэффициенты c, d:

$$\begin{cases} c = \dfrac{\text{GetMaxX} - \text{XK} - \text{X0}}{b - a} \\ d = \text{X0} - c \cdot a \end{cases}.$$

Для преобразования координаты Y в координату V построим линейную функцию $V = gY + h$. Точка min «бумажной» системы координат перейдет в точку GetMaxY-Y0 «экранной», а точка max – в точку Y0. Для нахождения коэффициентов g и h необходимо решить систему линейных алгебраических уравнений:

$$\begin{cases} g \cdot \text{min} + h = \text{GetMaxY} - \text{YK} \\ g \cdot \text{max} + h = \text{Y0} \end{cases}.$$

Ее решение позволит найти коэффициенты g и h:

$$\begin{cases} g = \dfrac{\text{GetMaxY} - \text{YK} - \text{Y0}}{\text{min} - \text{max}} \\ h = \text{Y0} - g \cdot \text{max} \end{cases}.$$

Алгоритм построения графика на экране дисплея можно разделить на следующие этапы:

1. Определить число точек N, шаг изменения переменной X (hx = (b – a)/N).
2. Сформировать массивы X, Y, вычислить максимальное (max) и минимальное (min) значение Y.
3. Перевести экран в графический режим.
4. Найти коэффициенты c, d, g и h.
5. Создать массивы $U_i = cX_i + d$ и $V_i = gY_i + h$.
6. Соединить соседние точки (U_i, V_i, U_{i+1}, V_{i+1}) прямыми линиями с помощью функции line.
7. Изобразить систему координат, линий сетки и подписей.

При построении графиков k нескольких непрерывных функций вместо массивов Y и V рациональнее использовать матрицы размером kxn, элементы каждой строки которых являются ординатами соответствующего графика в «бумажной» (Y) и «экранной» (V) системе координат.

Блок-схема алгоритма изображения графиков k непрерывных функций на экране дисплея представлена на рис. 12.5. Блоки 2–3 реализуют первый этап алгоритма. В блоках 4–8 формируется массив X и матрица Y. Блоки 9–15 предназначены для определения максимального и минимального значения, отображаемого на графике. Блок 16 реализует перевод экрана в графический режим. В блоке 17 рассчитываются коэффициенты c, d, g и h перевода из «бумажной» системы координат в «экранную». В блоках 18–21 формируются массив значений U и матрица значений V в «экранной» системе координат. Блоки 22–24 предназначены для вывода графиков на экран дисплея. В блоке 25 представлен седьмой этап алгоритма (рис. 12.6).

Особенностью рисования линий сетки является то, что изображать их надо в «экранной» системе координат, а выводить реальные значения – из «бумажной» системы координат. Можно выделить следующие этапы алгоритма рисования линий сетки:

1. Определить количество линий сетки по оси *OX* (kvox) и по оси *OY* (kvoy) – блок 2.
2. Определить расстояние между линиями сетки в «экранной» системе координат (hu и hv) – блок 3.
3. Изобразить линии сетки, параллельные оси *OX* (блоки 6, 7) и *OY* (блоки 4, 5).
4. Вычислить значения, которые будут находиться под осью *OX* (a + (i – 1)hx), и вывести их на экран (блоки 9, 10). Найти данные, которые разместятся левее оси *OY* (max – (i – 1)hy), и вывести их на экран (блоки 11, 12).
5. Изобразить ось координат (если они попадают в интервал, отображаемый на графике) – блоки 13–16, – для чего необходимо провести прямую в точке round(d) – в «бумажной» системе координат это X = 0, – параллельную оси *OY*, и прямую в точке round(g) – Y = 0 в «бумажной» системе координат, параллельную оси *OX* в «экранной» системе координат.

Пример 12.6. Реализация рассмотренного алгоритма представлена в виде программы изображения четырех графиков функций $y = \sin(\alpha_i x)\cos(x)$[1] на интервале $[a,b]$.

[1] Заменив функцию f в программе, можно заставить рисовать ее графики любых непрерывных функций.

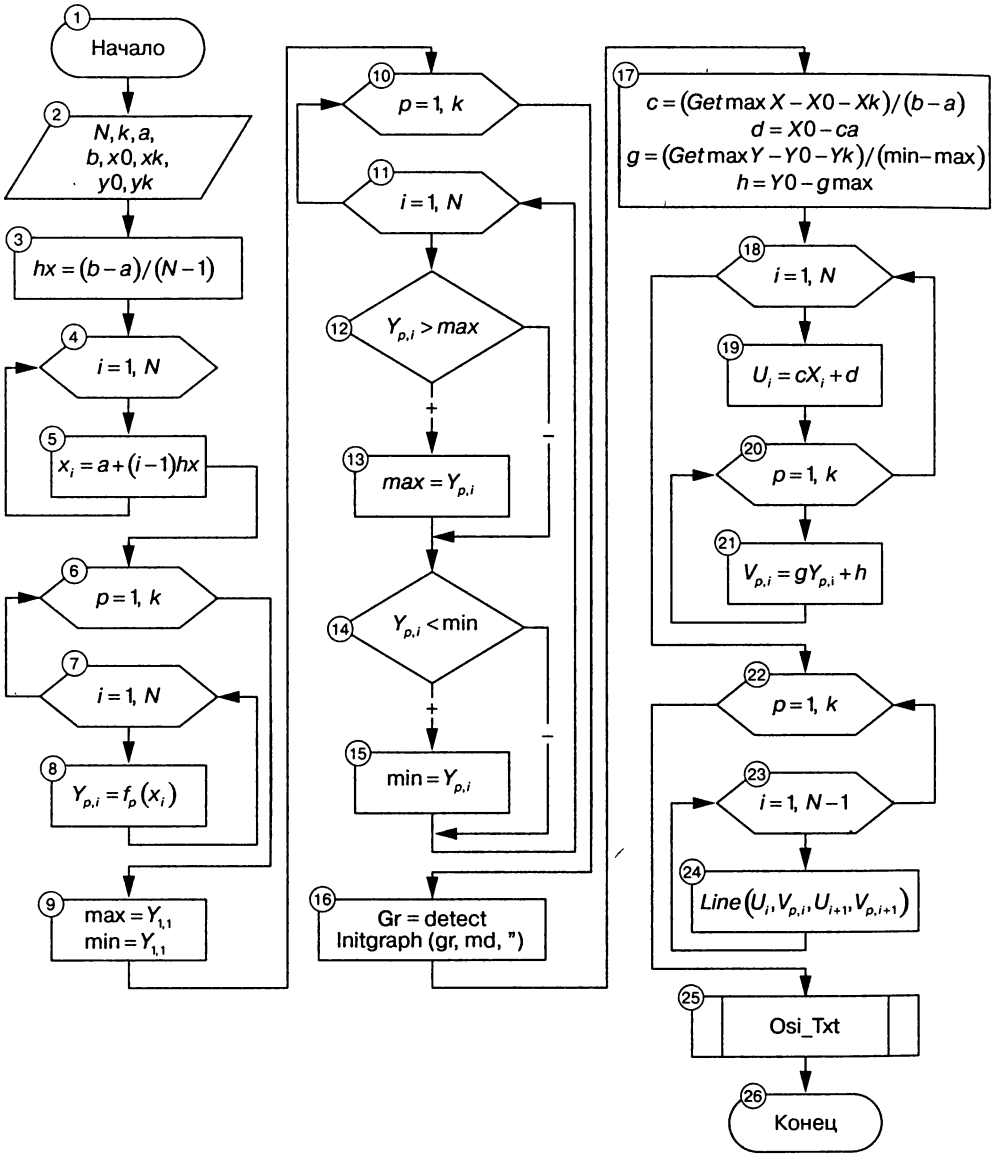


Рис. 12.5 ▾ Блок-схема алгоритма построения графиков нескольких функций

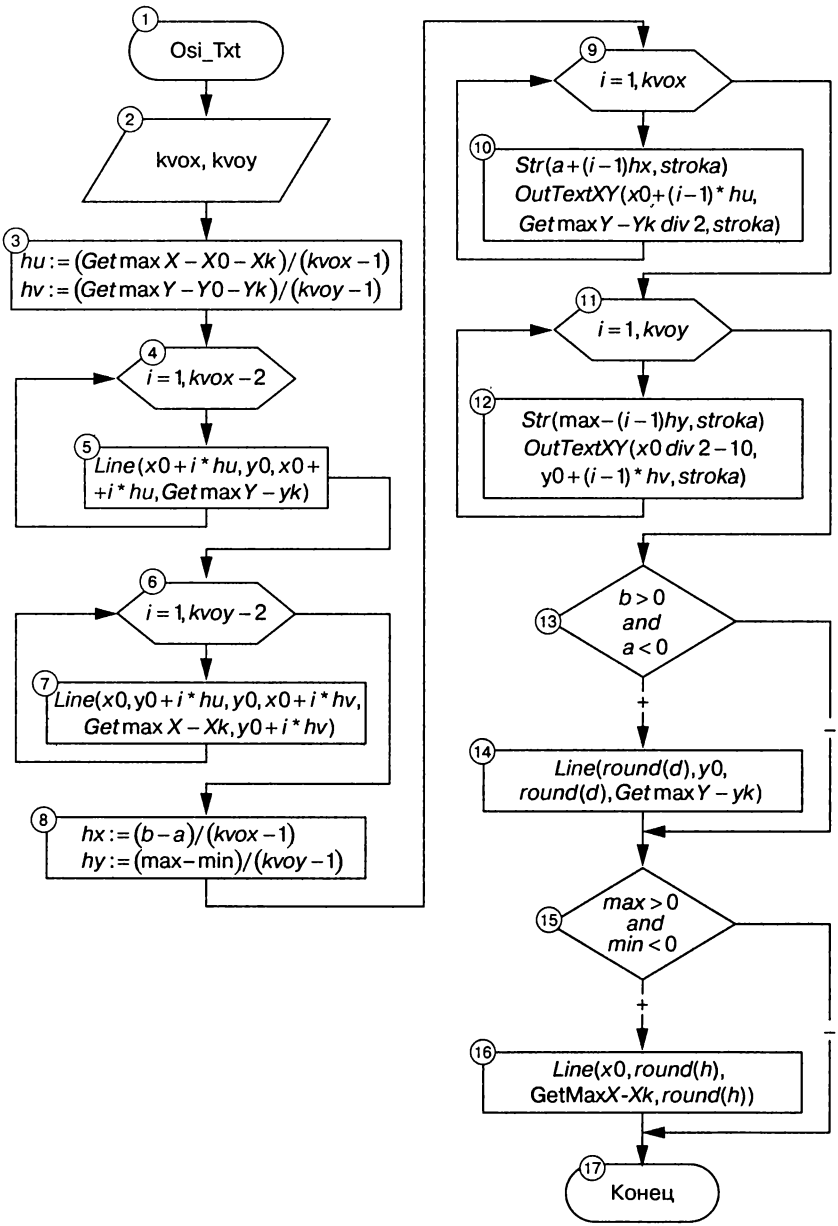


Рис. 12.6 ▼ Блок-схема алгоритма изображения линий сетки и осей координат

Приведем текст программы:

```
uses crt,graph;
type matr_real=array[1..10,1..200] of real;
  matr_int=array[1..10,1..200] of integer;
  mas_real=array[1..200] of real;
  mas_int=array[1..200] of integer;
function f(a,x:real):real;
begin
  f:=sin(a*x)*cos(x)
end;
var
  alf,x:mas_real;u:mas_int;
  y:matr_real;v:matr_int;
  hu,hv,kvoy,kvox,grdr,grmd,i,p,n,x0,y0,xk,yk:integer;
  max,min,a,b,hx,hy:real;
  c,d,g,h:real;
  Pattern:word;
  f1:text;
  stroka:string;
begin
  assign(f1,'graphik.txt');
{ В файле graphik.txt содержатся исходные данные программы. }
{ N - количество точек для изображения графиков. }
{ Интервал [a,b], на котором строится график. }
{ Alf(i) - коэффициенты функции. }
{ X0,xk,y0,yk - отступы от левой, правой, нижней, верхней границ экрана. }
{ Kvox, kvoy - количество линий сетки. }
  reset(f1);
  readln(f1,n);readln(f1,a,b);
  for i:=1 to 4 do
     read(f1,alf[i]);
  readln(f1,x0,xk,y0,yk);
  readln(f1,kvox,kvoy);
  close(f1);
{ Формирование массива X и матрицы Y, блоки 4-8 (см. рис. 12.6). }
  x[1]:=a;
  hx:=(b-a)/(n-1);
  for i:=1 to n do
     x[i]:=a+(i-1)*hx;
  for p:=1 to 4 do
     for i:=1 to n do
        y[p,i]:=f(alf[p],x[i]);
{ Поиск максимального и минимального значения Y, блоки 9-15 (см. рис.
12.6). }
  max:=y[1,1];
  min:=y[1,1];
```

```
  for p:=1 to 4 do
     for i:=1 to n do
     begin
        if y[p,i]>max then max:=y[p,i];
        if y[p,i]<min then min:=y[p,i];
     end;
 { Перевод экрана в графический режим. }
  grdr:=detect;
  InitGraph(grdr,grmd,'c:\bp\bgi');
 { Цвет фона в графическом режиме белый. }
  SetBkColor(15);
 { Цвет пера синий. }
  SetColor(1);
 { Изображение рамки вокруг графика. }
  Rectangle(x0,y0,GetmaxX-xk,GetmaxY-yk);
 { Вычисление коэффициентов пересчета. }
  c:=(GetMaxX-x0-xk)/(b-a);
  d:=x0-c*a;
  g:=(GetMaxY-y0-yk)/(min-max);
  h:=y0-g*max;
 { Формирование значений в "экранной" системе координат. }
  for i:=1 to n do
     u[i]:=trunc(c*x[i]+d);
  for p:=1 to 4 do
     for i:=1 to n do
        v[p,i]:=trunc(g*y[p,i]+h);
  for p:=1 to 4 do
  begin
 { Выбор цвета изображения очередного графика. }
     setcolor(p+2);
 { Выбор стиля линии для очередного графика. }
     SetlineStyle(p-1,pattern,1);
 { Изображение графика. }
     for i:=1 to n-1 do
        line(u[i],v[p,i],u[i+1],v[p,i+1]);
  end;
 { Определение тонких пунктирных линий для изображения линий сетки. }
  SetlineStyle(DashedLn,pattern,1);
 { Определение цвета для изображения линий сетки. }
  setcolor(1);
 { Определение расстояния между линиями сетки по оси OX в "экранной"
системе координат. }
  hu:=trunc((GetmaxX-x0-xk)/(kvox-1));
 { Определение расстояния между линиями сетки по оси OY в "экранной"
системе координат. }
  hv:=trunc((GetmaxY-y0-yk)/(kvoy-1));
 { Изображение линий сетки, параллельных оси OY. }
```

```
  for i:=1 to kvox-2 do
     line(x0+i*hu,y0,x0+i*hu,GetMaxY-yk);
{ Изображение линий сетки, параллельных оси OX. }
  for i:=1 to kvoy-2 do
     line(x0,y0+i*hv,GetMaxX-Xk,y0+i*hv);
{ Определение расстояния между линиями сетки по оси OX и оси OY в
"бумажной" системе координат. }
  hx:=(b-a)/(kvox-1);
  hy:=(max-min)/(kvoy-1);
  for i:=1 to kvox do
  begin
{ Вычисление очередного значения, выводимого под осью OX, и перевод его в
строковое значение. }
     Str(a+(i-1)*hx:1:1,stroka);
{ Вывод строкового значения на экран под осью OX. }
     OutTextXY(x0+(i-1)*hu-15,GetMaxY-Yk div 2 -15,stroka);
  end;
  for i:=1 to kvoy do
  begin
{ Вычисление очередного значения, выводимого левее оси OY, и перевод его в
строковое значение. }
     Str(max-(i-1)*hy:1:1,stroka);
{ Вывод строкового значения на экран левее оси OX. }
     OutTextXY(x0 div 2 - 10,y0+(i-1)*hv,stroka);
  end;
  OutTextXY(GetMaxX-Xk+50,Y0 div 2 -5,'A');
  for p:=1 to 4 do
  begin
{ Определение цвета и стиля выводимой легенды. }
     setcolor(p+2);
     SetlineStyle(p-1,pattern,1);
{ Вывод шаблона легенды. }
     Line(GetMaxX-Xk+10,(p+1)*Y0 div 2,GetMaxX-Xk+40,(p+1)*Y0 div 2);
{ Вывод коэффициента alf в качестве легенды. }
     Str(alf[p]:1:1,stroka);
     OutTextXY(GetMaxX-Xk+50,(p+1)*Y0 div 2 -5,stroka);
  end;
{ Определение цвета и стиля (сплошная линия) осей координат. }
  SetlineStyle(SolidLn,pattern,1);
  setcolor(1);
{ Изображение линий сетки в случае необходимости. }
  if (b>0) and (a<0) then line(round(d),y0,round(d),getmaxY-yk);
  if (max>0) and (min<0) then line(x0,round(h),GetMaxX-Xk,round(h));
{ Подпись над графиком. }
  OutTextXY(2* GetMaxX div 5, y0 div 2,'y=sin(Ax)cos(x)');
  repeat until keypressed;
end.
```

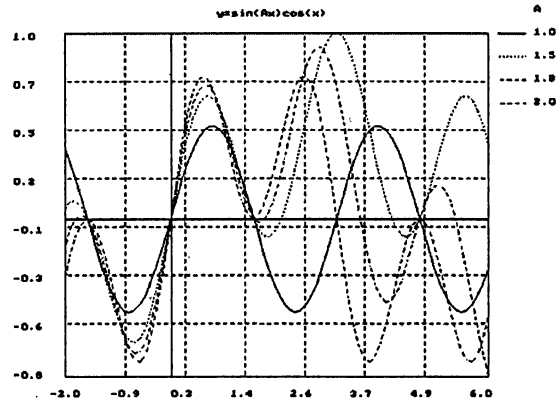


Рис. 12.7 ▼ График функций $y = \sin(\alpha,x)\cos(x)$

После запуска программы на выполнение на экране появится графическое окно, показанное на рис. 12.7.

Пример 12.7. Изобразить на экране дисплея графики разрывных функций $y = \frac{C}{x^2}$ при различных значениях C. Ниже приведен текст программы, где прокомментированы участки, относящиеся к особенностям изображения графика разрывной функции.

```
uses crt,graph;
type matr_real=array[1..10,1..200] of real;
  matr_int=array[1..10,1..200] of integer;
  mas_real=array[1..200] of real;
  mas_int=array[1..200] of integer;
function f(a,x:real):real;
begin
  f:=A/sqr(x)
end;
var
  c,x:mas_real;u:mas_int;
  y:matr_real;v:matr_int;
  hu,hv,kvoy,kvox,grdr,grmd,i,j,n,x0,y0,xk,yk:integer;
  max,min,a,b,hx:real;
  razr1,razr2,a1,b1,c1,d1:real;
  Pattern:word;
  f1:text;
  stroka:string;
```

```
begin
  assign(f1,'gr.txt');
{ В файле gr.txt содержатся исходные данные программы. }
{ N - количество точек для изображения графиков. }
{ Интервал [a,b], на котором строится график. }
{ Интервал разрыва [razr1,razr2], исключаемый из графика. }
{ График строится на двух интервалах [a,razr1] и [razr2,b]. }
{ C(i) - коэффициенты функции. }
{ X0,xk,y0,yk -отступы от левой, правой, нижней, верхней границ экрана. }
{ Kvox, kvoy - количество линий сетки. }
  reset(f1);
  readln(f1,n);readln(f1,a,b,razr1,razr2);
  for i:=1 to 4 do
     read(f1,c[i]);
  readln(f1,x0,xk,y0,yk);
  readln(f1,kvox,kvoy);
  close(f1);
{ В данном случае интервал массива x формируется из двух участков
[a,razr1] и [razr2,b]. }
  x[1]:=a;
{ Шаг изменения X на интервале [a,razr1]. }
  hx:=(razr1-a)/(n div 2 -1);
  for i:=2 to n div 2 do
     x[i]:=x[i-1]+hx;
{ Шаг изменения X на интервале [razr2,b]. }
  hx:=(b-razr2)/(n div 2 -1);
  x[n div 2+1]:=razr2;
  for i:=n div 2 + 2 to n do
     x[i]:=x[i-1]+hx;
  for j:=1 to 4 do
     for i:=1 to n do
        y[j,i]:=f(c[j],x[i]);
  max:=y[1,1];
  min:=y[1,1];
  for j:=1 to 4 do
     for i:=1 to n do
     begin
        if y[j,i]>max then max:=y[j,i];
        if y[j,i]<min then min:=y[j,i];
     end;
  grdr:=detect;
  InitGraph(grdr,grmd,'c:\bp\bgi');
  Rectangle(x0,y0,GetmaxX-xk,GetmaxY-yk);
  a1:=(GetMaxX-x0-xk)/(b-a);
  b1:=x0-a1*a;
  c1:=(GetMaxY-y0-yk)/(min-max);
```

```
  d1:=y0-c1*max;
  for i:=1 to n do
     u[i]:=trunc(a1*x[i]+b1);
  for j:=1 to 4 do
     for i:=1 to n do
        v[j,i]:=trunc(c1*y[j,i]+d1);
  SetlineStyle(SolidLn,pattern,3);
  for j:=1 to 4 do
  begin
     SetColor(j);
     for i:=1 to n-1 do
{ При построении графика исключаем соединительную линию
двух подинтервалов [a,razr1] и [razr2,b]. }
        if i<> n div 2 then line(u[i],v[j,i],u[i+1],v[j,i+1]);
  end;
  SetlineStyle(DashedLn,pattern,1);
  SetColor(White);
  hu:=trunc((GetmaxX-x0-xk)/kvox);
  hv:=trunc((GetmaxY-y0-yk)/kvoy);
  for i:=1 to kvox-1 do
     line(x0+i*hu,y0,x0+i*hu,GetMaxY-yk);
  hx:=(b-a)/kvox;
  for i:=1 to kvox+1 do
  begin
     Str(a+(i-1)*hx:1:1,stroka);
     OutTextXY(x0+(i-1)*hu-15,GetMaxY-Yk div 2 -15,stroka);
  end;
  for i:=1 to kvoy-1 do
     line(x0,y0+i*hv,GetMaxX-Xk,y0+i*hv);
  hx:=(max-min)/kvoy;
  for i:=1 to kvoy+1 do
  begin
     Str(max-(i-1)*hx:1:1,stroka);
     OutTextXY(x0 div 2 -10,y0+(i-1)*hv,stroka);
  end;
  OutTextXY(GetMaxX div 2, y0 div 2,'y=A/(x^2)');
  repeat until keypressed;
end.
```

Результат работы программы представлен на рис. 12.8. Если функция имеет несколько точек разрыва (например, tg(x), ctg(x)), необходимо вводить массивы интервалов разрыва [$razr1_i$,$razr2_i$], точки в которых не следует соединять между собой.

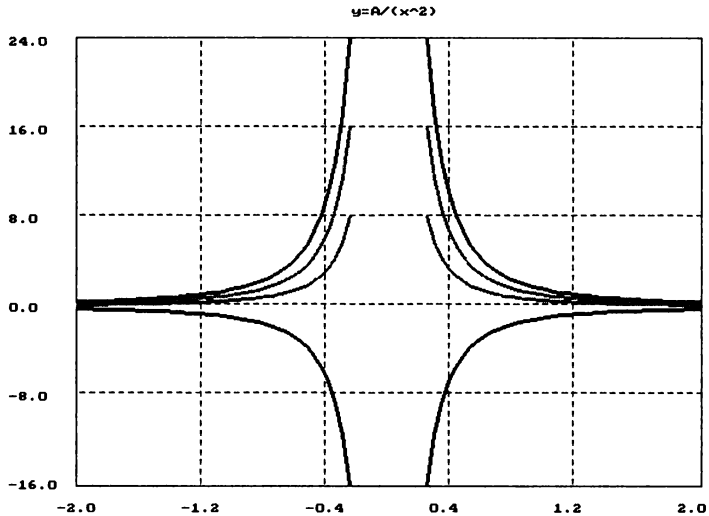


Рис. 12.8 ▾ Графики функций $y = \frac{C}{x^2}$

12.8. Модуль изображения графиков и поверхностей непрерывных функций

Ниже приведен модуль рисования графиков функций $y = f(x)$[1] и поверхностей двумерных функций $z = f(x, y)$ [2]. Он позволяет изображать график одной или нескольких функций в любой области экрана дисплея. В связи с тем что программа рисует графики различных непрерывных функций на любом участке экрана, вывод текстовой информации в подпрограммах сведен к минимуму. Читатель может самостоятельно расширить возможности подпрограмм.

```
{ Это большой модуль рисования графиков и поверхностей. }
unit GRAPHIC;
interface
uses graph,crt;
```

[1] В подпрограмме изображения графиков функций использован алгоритм, описанный в предыдущей главе.

```
type massiv=array [1..10000] of real;
  pmas=^massiv;
  dinmas=array [1..8] of pmas;
  func=function(x:real):real;
  func1=function(x,y:real):real;
PROCEDURE GRAFIK (a, b :real; n: integer; fun:ARRAY OF FUNC;
x1, y1, x2, y2: word; Colors: array of WORD; var Err:shortint);
{ Процедура рисования графиков восьми функций в одной системе координат. }
{ a,b - интервал, на котором рисуются графики, n - количество графиков (n<=8).
fun1 ,…,fun8 - имена изображаемых функций. }
{ x1, y1, x2, y2 - прямоугольник на экране,
внутри которого будут изображаться графики. }
{ Colors - массив цветов. }
{ Colors[0] - цвет фона. }
{ Colors[1] - цвет первого графика. }
{ Colors[2] - цвет второго графика. }
…
…
{ Colors[8] - цвет восьмого графика. }
{ Colors[9] - цвет рамки и подписей. }
{ Err - код ошибки. }
{ 0 - нормальное завершение программы.
-1 - n>8 or n<1,
-2 - x1,y1<0 or x2,y2>getmaxX,getmaxY, }
PROCEDURE GRAFIK1(n:integer;m:integer; x:pmas;y:dinmas;
x1,y1,x2,y2:word;Colors: array of WORD; var Err:shortint);
{ Процедура рисования восьми графиков функций по точкам в одной системе
координат. }
{ a,b - интервал, на котором рисуются графики. }
{ m - количество точек в массивах x, y. }
{ n - количество графиков (n <= 8). }
{ x- динамический массив x (указатель). }
{ y- массив указателей y. }
{ x1, y1, x2, y2 - прямоугольник на экране, внутри которого будут
изображаться графики. }
{ Colors - массив цветов. }
{ Colors[0] - цвет фона. }
{ Colors[1] - цвет первого графика. }
{ Colors[2] - цвет второго графика. }
…
{ Colors[8] - цвет восьмого графика. }
{ Colors[9] - цвет рамки и подписей. }
{ Err - код ошибки. }
{ 0 - нормальное завершение программы. }
{ -1 - n>8 or n<1. }
{ -2 - x1,y1<0 or x2,y2>getmaxX,getmaxY. }
```

```
Procedure surfase(xMin,xMax,yMin, yMax, Rad, Theta, Phi, D:real;
fun:func1);
{ Процедура рисования графика двумерной функции Z=F(X,Y). }
{ fun - функция f(x,y). }
{ xmin,xmax,ymin,ymax - границы изменения переменных x и y. }
{ d - расстояние до кривой. }
{ rad, theta, phi - три угла в радианах. }
implementation
PROCEDURE GRAFIK(a,b:real; n:integer; fun:ARRAY OF FUNC;
  x1,y1,x2,y2:word; Colors:array of WORD; var Err:shortint);
var
  x:pmas;
  y:dinmas;
  rx,ry,ymax,ymin,a1,b1,c1,d1:real;
  kol,y1n,y2n,x1n,x2n,k,dx,dy,i,j,nmax,nmin:integer;
  s:string[10];
begin
  if (n<1) or (n>8) then
  begin
     Err:=-1;
     exit
  end;
  if (x1<0) or (y1<0) or (x2>getmaxX) or (y2>GetmaxY) then
  begin
     Err:=-2;
     exit
  end;
{ Устанавливается цвет фона. }
  SetBkColor(Colors[0]);
{ Устанавливается цвет линий и текста. }
  SetColor(Colors[9]);
  dx:=x2-x1;
  dy:=y2-y1;
  x1n:=x1+trunc(dx/10);
  x2n:=x2-trunc(dx/20);
  y1n:=y1+trunc(dy/10);
  y2n:=y2-trunc(dy/10);
  kol:=x2n-x1n;
{ Выделение памяти под динамический массив X. }
  getmem(x,(kol+1)*6);
{ Формирование массива X. }
  x^[1]:=a;
  for i:=2 to kol+1 do
     x^[i]:=x^[i-1]+(b-a)/kol;
{ Выделение памяти под динамический массив Y. }
```

```
  for i:=1 to n do
     GetMem(y[i],(kol+1)*6);
{ Запись в массив Y значений функций. }
  for i:=1 to kol+1 do
  begin
     y[1]^[i]:=fun[0](x^[i]);
     if n>=2 then y[2]^[i]:=fun[1](x^[i]);
     if n>=3 then y[3]^[i]:=fun[2](x^[i]);
     if n>=4 then y[4]^[i]:=fun[3](x^[i]);
     if n>=5 then y[5]^[i]:=fun[4](x^[i]);
     if n>=6 then y[6]^[i]:=fun[5](x^[i]);
     if n>=7 then y[7]^[i]:=fun[6](x^[i]);
     if n>=8 then y[8]^[i]:=fun[7](x^[i])
  end;
{ Поиск максимума и минимума в массиве Y. }
  ymin:=y[1]^[1];
  ymax:=y[1]^[1];
  nmin:=1;
  nmax:=1;
  for j:=1 to n do
     for i:=1 to kol+1 do
     begin
        if y[j]^[i]>ymax then
        begin
           ymax:=y[j]^[i];
           nmax:=i;
        end;
        if y[j]^[i]<ymIN then
        begin
           ymin:=y[j]^[i];
           nmin:=i;
        end
     end;
{ Формирование коэффициентов пересчета в «экранную» систему координат. }
  a1:=(x2n-x1n)/(b-a);
  b1:=x1n-a1*a;
  c1:=(y1n-y2n)/(ymax-ymin);
  d1:=y2n-c1*ymin;
{ Рисование осей и подписи. }
  rx:=(x2n-x1n-1)/5;
  ry:=(y2n-y1n+1)/5;
  line(x1n-1,y2n+1,x2n+trunc(0.0375*dx),y2n+1);
  str(x^[1]:1:2,s);
  settextjustify(0,1);
  outtextxy(x1n-25,y2n+1+trunc(dy/20),s);
  str(x^[kol+1]:1:2,s);
```

```
  settextjustify(2,1);
  outtextxy(x1n+5*trunc(rx)+25,y2n+1+trunc(dy/20),s);
  for i:=1 to 5 do
  begin
     line(x1n+1+i*trunc(rx),y2n+1-trunc(dy/50),x1n+1+i*trunc(rx),y2n+1);
     line(x1n+1+i*trunc(rx),y2n+1,x1n+1+i*trunc(rx),y1n);
  end;
  line(x1n-1,y2n+1,x1n-1,y1n-trunc(0.075*dy));
  str(ymin:1:2, s);
  settextjustify(2,1);
  outtextxy(x1n-trunc(dx/200),y2n+1-trunc(dy/50)+10,s);
  str(ymax:1:2, s);
  settextjustify(2,1);
  outtextxy(x1n-trunc(dx/200),y1n+trunc(dy/50)-10,s);
  for i:=1 to 5 do
  begin
     line(x1n+1+trunc(dx/100),y2n-1-i*trunc(ry),x1n,y2n-1-i*trunc(ry));
     line(x2n,y2n-1-i*trunc(ry),x1n,y2n-1-i*trunc(ry));
  end;
  line(x2n+trunc(0.0375*dx),y2n+1,
  x2n+trunc(0.0375*dx)-trunc(dx/50),trunc(y2n+1+dy/200));
  line(x2n+trunc(0.0375*dx),y2n+1,
  x2n+trunc(0.0375*dx)-trunc(dx/50),trunc(y2n+1-dy/200));
  line(x1n-1,y1n-trunc(0.075*dy),x1n-1+trunc(dx/120),
  y1n-trunc(0.075*dy)+trunc(dy/50));
  line(x1n-1,y1n-trunc(0.075*dy),x1n-1-trunc(dx/120),
  y1n-trunc(0.075*dy)+trunc(dy/50));
{ Рисование графиков непрерывных функций. }
  for j:=1 to n do
  begin
     SetColor(Colors[j]);
     for i:=1 to kol do
        line(trunc(a1*x^[i]+b1),trunc(c1*y[j]^[i]+d1),
     trunc(a1*x^[i+1]+b1),trunc(c1*y[j]^[i+1]+d1))
  end;
{ Освобождение памяти. }
  FreeMem(x,(kol+1)*6);
  for i:=1 to n do
     FreeMem(y[i],(kol+1)*6)
end;
PROCEDURE GRAFIK1(n:integer;m:integer; x:pmas;y:dinmas;
  x1,y1,x2,y2:word;Colors: array of WORD; var Err:shortint);
var
  a,b:real;
  rx,ry,ymax,ymin,a1,b1,c1,d1:real;
  kol,y1n,y2n,x1n,x2n,k,dx,dy,i,j,nmax,nmin:integer;
```

```
  s:string[10];
begin
  a:=x^[1];
  b:=x^[m];
  if (n<1) or (n>8) then
  begin
     Err:=-1;
     exit
  end;
  if (x1<0) or (y1<0) or (x2>getmaxX) or (y2>GetmaxY) then
  begin
     Err:=-2;
     exit
  end;
  SetColor(Colors[9]);
  dx:=x2-x1;
  dy:=y2-y1;
  x1n:=x1+trunc(dx/10);
  x2n:=x2-trunc(dx/20);
  y1n:=y1+trunc(dy/10);
  y2n:=y2-trunc(dy/10);
  kol:=m;
  ymin:=y[1]^[1];
  ymax:=y[1]^[1];
  nmin:=1;
  nmax:=1;
  for j:=1 to n do
     for i:=1 to kol do
     begin
        if y[j]^[i]>ymax then
        begin
           ymax:=y[j]^[i];
           nmax:=i;
        end;
        if y[j]^[i]<ymin then
        begin
           ymin:=y[j]^[i];
           nmin:=i;
        end
     end;
  a1:=(x2n-x1n)/(b-a);
  b1:=x1n-a1*a;
  c1:=(y1n-y2n)/(ymax-ymin);
  d1:=y2n-c1*ymin;
  rx:=(x2n-x1n-1)/5;
  ry:=(y2n-y1n+1)/5;
```

```
  line(x1n-1,y2n+1,x2n+trunc(0.0375*dx),y2n+1);
  str(x^[1]:1:2,s);
  settextjustify(0,1);
  outtextxy(x1n-25,y2n+1+trunc(dy/20),s);
  str(x^[kol]:1:2,s);
  settextjustify(2,1);
  outtextxy(x1n+5*trunc(rx)+25,y2n+1+trunc(dy/20),s);
  for i:=1 to 5 do
     line(x1n+1+i*trunc(rx),y2n-1,x1n+1+i*trunc(rx),y1n);
  line(x1n-1,y2n- 1,x1n-1,y1n-trunc(0.075*dy));
  str(ymin:1:2, s);
  settextjustify(2,1);
  outtextxy(x1n-trunc(dx/200),y2n+1-trunc(dy/50),s);
  str(ymax:1:2, s);
  settextjustify(2,1);
  outtextxy(x1n-trunc(dx/200),y1n+trunc(dy/50),s);
  for i:=1 to 5 do
     line(x2n,y2n-1-i*trunc(ry),x1n,y2n-1-i*trunc(ry));
  line(x2n+trunc(0.0375*dx),y2n+1,
  x2n+trunc(0.0375*dx)-trunc(dx/50),trunc(y2n+1+dy/200));
  line(x2n+trunc(0.0375*dx),y2n+1,
  x2n+trunc(0.0375*dx)-trunc(dx/50),trunc(y2n+1-dy/200));
  line(x1n-1,y1n-trunc(0.075*dy),x1n-1+trunc(dx/120),
  y1n-trunc(0.075*dy)+trunc(dy/50));
  line(x1n-1,y1n-trunc(0.075*dy),x1n-1-trunc(dx/120),
  y1n-trunc(0.075*dy)+trunc(dy/50));
  for j:=1 to n do
  begin
     SetColor(Colors[j]);
     for i:=1 to kol-1 do
        line(trunc(a1*x^[i]+b1),trunc(c1*y[j]^[i]+d1),
     trunc(a1*x^[i+1]+b1),trunc(c1*y[j]^[i+1]+d1))
  end;
end;
Procedure surfase (xMin,xMax,yMin,yMax,Rad,Theta,Phi,D:real; fun:func1);
var
  x,y,dx,dy,Ax,Ay,Bx,By:real;
  dxMax,dxMin,dyMax,dyMin:real;
  xStep,yStep:real;
  i,j,xCount,yCount:integer;
  xNew,yNew,xOld,yOld:integer;
  Show:boolean;
const
  Big=9.999999E+10;
  Margin=0.1;
```

```
procedure FindEyeCoordinates(x,y: real; var dx,dy:real;
  Theta,Phi,Rad,D:real);
var
  z,xx,yy,zz:real;
begin
  z:=Fun(x,y);
  xx:=-x*sin(Theta)++y*cos(Theta);
  yy:=-x*cos(Theta)*cos(Phi)-y*sin(Theta)*cos(Phi)+z*sin(Phi);
  zz:=-x*cos(Theta)*sin(Phi)-y*sin(Theta)*sin(Phi)-z*cos(Phi)+Rad;
  dx:=D*xx/zz;
  dy:=D*yy/zz
end;
procedure FindScreenCoordinates(Ax,Ay,Bx,By,dx,dy:real;
var xNew,yNew:integer);
begin
  xNew:=trunc(Ax+Bx*dx);
  yNew:=GetMaxY-trunc(Ay+By*dy)
end;
procedure FindLimits(dx,dy:real;var dxMax,dxMin,dyMax,dyMin:real);
begin
  if dx>dxMax then dxMax:=dx;
  if dx<dxMin then dxMin:=dx;
  if dy>dyMax then dyMax:=dy;
  if dy<dyMin then dyMin:=dy;
end;
procedure FindWindow (dxMax,dxMin,dyMax,dyMin :real; var
Ax,Ay,Bx,By:real);
var
  xSize,ySize:real;
begin
  xSize:=dxMax-dxMin;
  ySize:=dyMax-dyMin;
  dxMin:=dxMin-Margin*xSize;
  dyMin:=dyMin-Margin*ySize;
  dxMax:=dxMax+Margin*xSize;
  dyMax:=dyMax+Margin*ySize;
  Bx:=GetMaxX/(dxMax-dxMin);
  By:=GetMaxY/(dyMax-dyMin);
  Ax:=-dxMin*Bx;
  Ay:=-dyMin*By
end;
begin
  xCount:=20;
  yCount:=20;
  line(0,0,GetMaxX,0);
  line(GetMaxX,GetMaxY,0,GetMaxY);
```

```
  line(0,0,0,GetMaxY);
  line(GetMaxX,0,GetMaxX,GetMaxY);
  xStep:=(xMax-xMin)/xCount;
  yStep:=(yMax-yMin)/yCount;
  dxMin:=Big;
  dxMax:=-Big;
  dyMin:=Big;
  dyMax:=-Big;
  for Show:=false to true do
  begin
     for i:=0 to xCount do
     begin
        x:=xMin+i*xStep;
        y:=yMin;
        FindEyeCoordinates(x,y,dx,dy,Theta,Phi,Rad,D);
        if Show then
        begin
           FindScreenCoordinates(Ax,Ay,Bx,By,dx,dy,xNew,yNew);
           xOld:=xNew;
           yOld:=yNew;
           MoveTo(xOld,yOld)
        End
        else FindLimits(dx,dy,dxMax,dxMin,dyMax,dyMin);
           for j:=0 to yCount do
           begin
              y:=yMin+j*yStep;
                 FindEyeCoordinates(x,y,dx,dy,Theta,Phi,Rad,D);
              if Show then
              begin
              FindScreenCoordinates(Ax,Ay,Bx,By,dx,dy,xNew,yNew);
              lineTo (xNew,yNew);
              xOld:=xNew;
           yOld:=yNew
              end
           else FindLimits(dx,dy,dxMax,dxMin,dyMax,dyMin)
           end
        end;
           if not Show then FindWindow(dxMax, dxMin, dyMax, dyMin, Ax, Ay,
Bx, By)
     end;
     for i:=0 to yCount do
     Begin
        y:=yMin+i*yStep;
        x:=xMin;
        FindEyeCoordinates(x,y,dx,dy,Theta,Phi,Rad,D);
        if show then
```

```
        begin
           FindScreenCoordinates(Ax,Ay,Bx,By,dx,dy,xNew,yNew);
           xOld:=xNew;
           yOld:=yNew;
           moveTo(xOld,yOld);
        end
        else
           FindLimits(dx,dx,dxMax,dxMin,dyMax,dyMin);
           for j:=0 to xCount do
           begin
              x:=xMin+j*xStep;
              FindEyeCoordinates(x,y,dx,dy,Theta,Phi,Rad,D);
              if Show then
              begin
                 FindScreenCoordinates(Ax,Ay,Bx,By,dx,dy,xNew,yNew);
                 LineTo(xNew,yNew);
                 xOld:=xNew;
                 yOld:=yNew
              end
              else
              FindLimits(dx,dy,dxMax,dxMin,dyMax,dyMin)
           end
        end;
        repeat until KeyPressed;
  end;
end.
```

Пример 12.8. Изобразить графики и поверхности, используя модуль GRAPHIC.

```
uses graphic,graph,crt;
function g1(x:real):real;far;
begin
  g1:=sin(x);
end;
function g2(x:real):real;FAR;
begin
  g2:=cos(X)
end;
function g3(x:real):real;FAR;
begin
  g3:=cos(X)+0.6;
end;
function g4(x:real):real;FAR;
begin
  g4:=cos(sin(X))
end;
```

```
function g(x,y:real):real;FAR;
begin
  g:=sin(x)/(x*x+y*y+0.3)
end;
var color:array[1..10] of word;
  err:shortint;
  grdr,grmd:integer;
  i:integer;
  f1:array [1..8] of func;
begin
  grdr:=detect;
  initgraph(grdr,grmd,'c:\bp\bgi');
  f1[1]:=g1;
  f1[2]:=g2;
  color[1]:=white;
  color[2]:=lightgray;
  color[3]:=darkgray;
  color[10]:=blue;
  grafik(-pi,pi,2,f1,40,20,getmaxx div 2-30,getmaxy,color,err);
  color[2]:=red;
  color[3]:=green;
  f1[1]:=g3; f1[2]:=g4;
  grafik(-2*pi,2*pi,2,f1,getmaxx div 2,20,getmaxx-20,getmaxy,color,err);
  delay(8000);
  readln;
  cleardevice;
  setbkcolor(white);
  setcolor(brown);
  surfase(-1,1,-1,1,-10,0.8,-0.5,5,g);
  readln;
end.
```

В результате выполнения этой программы на экране дисплея будут нарисованы следующие два изображения (рис. 12.9, 12.10).

12.9. Включение драйвера и шрифтов в тело программы

Во всех рассмотренных графических программах есть одна общая проблема. На компьютере, где будет запускаться программа, должен быть драйвер с расширением .bgi, а если в программе используются векторные шрифты, то они также должны быть файлами с расширением .chr. Но в Турбо Паскале есть возможность включения драйвера и шрифтов в тело программы.

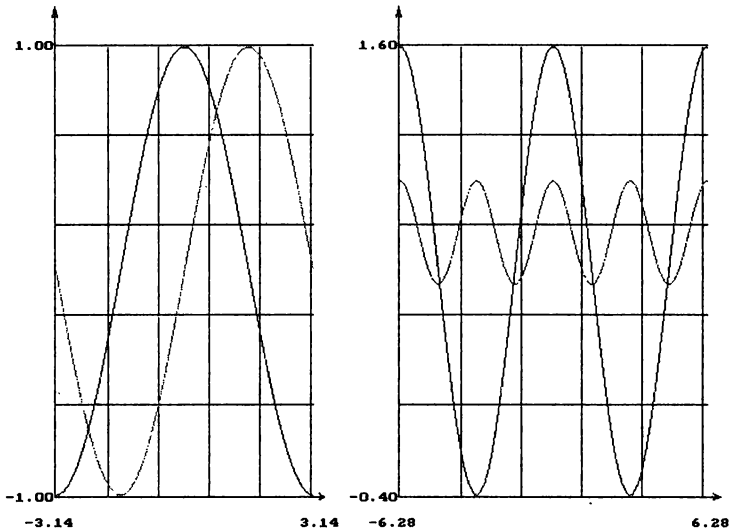


Рис. 12.9 ▼ График функций

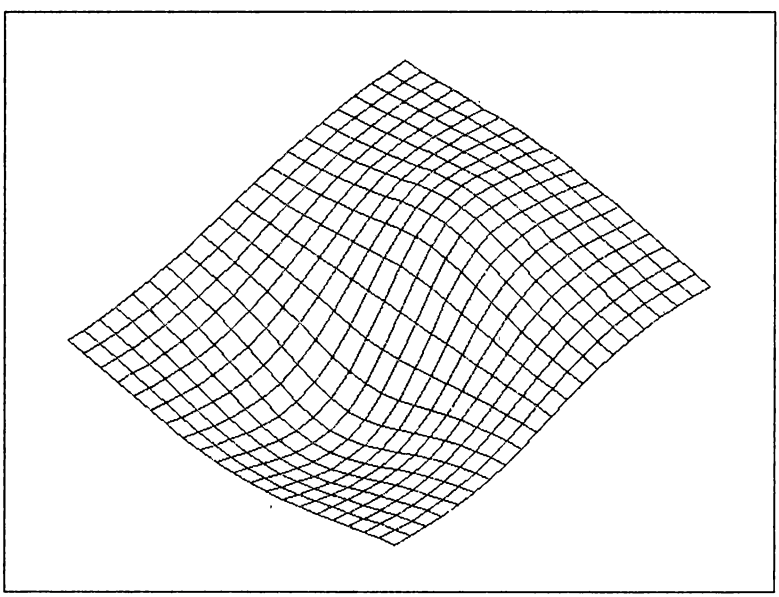


Рис. 12.10 ▼ График поверхности

12.9.1. Включение драйвера в тело программы

Для того чтобы включить драйвер в тело программы, его необходимо преобразовать в специальный файл с расширением .obj. Для этого в состав Турбо Паскаля входит утилита binobj.exe. Ее необходимо вызвать из командной строки. Обращение к ней имеет вид:

```
BINOBJ source[.BIN] destination[.OBJ] public_name
```

Source – имя файла, который будет преобразовываться в формат obj, если расширение опущено (подразумевается bin).

Destination – имя файла с расширением .obj, который будет создан в результате преобразования.

Public_name – имя, под которым драйвер будет использоваться в программе.

Например, для преобразования драйвера egavga.bgi в формат. obj в командной строке необходимо записать следующее:

```
BINOBJ egavga.bgi egavga drv_VGA
```

После того как получен файл с расширением .obj, надо включить специальную процедуру с текстом драйвера в программу. Посмотрим, как при этом изменится программа из примера 12.4.

```
Uses Crt,graph;
{ Объявляем внешнюю процедуру (external) drv_VGA, в которой хранится
драйвер в формате .obj. }
Procedure drv_VGA; external;
{ Подключение драйвера egavga в формате .obj. }
{ $L egavga.obj. }
Var i,d,m: integer;
Begin
{ Обращение к функции регистрации драйвера RegisterBGIDriver(@drv_VGA),
при не удачной регистрации функция возвращает отрицательное значение. }
  If RegisterBGIDriver(@drv_VGA) <0 then
     Writeln('Драйвер не зарегистрирован')
  Else
{ При удачной регистрации выполняется основная графическая часть программы
и при этом путь к файлу egavga.bgi указывать не надо, он включен в текст
программы (в exe файл). }
  begin
     d:= detect;
     InitGraph(d,m,' ');
     For i:=1 to 10 do
     begin
        SetTextStyle(i,0,0);
        OutTextXY(20,20+(i-1)*18,'Example');
```

```
      delay(2000);
    end;
    Repeat until KeyPressed;
  CloseGraph
  end
End.
```

12.9.2. Включение шрифтов в тело программы

Включение шрифтов осуществляется аналогичным образом. Сначала преобразовываем шрифт в формат .obj. Например:

```
BINOBJ c:\bp\bgi\goth.chr goth drv_goth
```

После этого в текст программы можно включить внешнюю процедуру drv_goth и зарегистрировать драйвер. Участок программы может иметь следующий вид:

```
Procedure drv_goth; external;
{ Подключение шрифта goth.chr в формате obj. }
{$L egavga.obj}
Var i,d,m: integer;
Begin
{ Обращение к функции регистрации шрифта RegisterBGIDriver(@drv_goth), при
не удачной регистрации функция возвращает отрицательное значение. }
  If RegisterBGIDriver(@drv_drv) <0 then
    Writeln('Драйвер не зарегистрирован')
  Else
    d:= detect;
    InitGraph(d,m,' ');
```

Регистрация драйверов должна проходить до переключения экрана в графический режим (до оператора `InitGraph`*).*

12.10. Упражнения по теме «Графические средства Турбо Паскаля»

Изобразить на экране дисплея графики указанных ниже функций (A принимает следующие значения: –1; 0,5; 1; 1,5). Построить координатные оси и выполнить соответствующие надписи на них:

- $Y = A\sin(\cos(x))$, $x \in [-3{,}2; 3{,}2]$;
- $Y = A2^x$, $x \in [-2{,}5]$;

- $Y = Ae^{\sin(x)}/x,\ x \in [-4,4]$;
- $Y = Ae^{-x^2},\ x \in [-3,3]$;
- $Y = A(3x^3 - 4x^2 + 5x) \cdot \sin(x),\ x \in [-10,10]$;
- $Y = A(2x^4 - 7x^3 + 13x^2 - 8x) \cdot \cos(x),\ x \in [-2,5]$;
- $Y = A(\sin(x) + x^2),\ x \in [-3,3]$;
- $Y = A(3x^3 + 7x) \cdot e^{\cos 5x},\ x \in [-7,7]$;
- $Y = A\cos(x)/(x^2),\ x \in [-4,4]$;
- $Y = Ae^x \cdot \cos(x),\ x \in [0,6]$.

Приложение 1

Отладка программ

Отладка программ – это процесс поиска и устранения ошибок. Все ошибки, встречающиеся в программе, можно разделить на следующие группы:

- синтаксические ошибки;
- ошибки времени выполнения;
- ошибки, возникшие на этапе разработки алгоритма (так называемые алгоритмические ошибки).

Синтаксические ошибки обнаруживаются и исправляются еще на этапе трансляции программы (об этом говорится в главе 1).

Ошибки времени выполнения (Run-time Error) обнаруживаются при первых запусках программы. При этом выполнение программы прерывается и курсор указывает на строку, которая вызвала данную ошибку.

Самыми сложными являются *алгоритмические ошибки.* Программа запускается, но выдает неправильный результат. В оболочке Турбо Паскаля есть мощные средства для выявления и исправления подобных ошибок: есть возможность останавливать выполнение программы на определенных участках (трассировка программы) и видеть значение переменных в процессе выполнения программы.

Для приостановки выполнения программы на определенных участках отладчик предоставляет следующие возможности:

- *выполнение программы до курсора.* Программа остановится в той строке, где находится курсор (если это возможно). Для этого необходимо выполнить команду **Go to cursor** (Выполнить до курсора) из пункта меню **Run** (Выполнить) или нажать клавишу **F4**;
- *построчное выполнение программы.* В этом режиме выполняется одна строка программы. Для его активизации необходимо выбрать команду **Step over** (На шаг) из пункта меню **Run** или нажать клавишу **F8**;
- *построчное выполнение программы с заходом в подпрограмму.* Главное отличие этого режима от предыдущего состоит в том, что если в текущей строке встретился оператор обращения к подпрограмме, то начинается построчное выполнение подпрограммы. Для выполнения программы в этом режиме необходимо вызвать команду **Trace into** (Трассировка до) из пункта меню **Run** или нажать клавишу **F7**. Выйти из режимов построчного выполнения программы или выполнения программы до курсора можно с помощью команды **Program Reset** (Перезапуск программы) из пункта меню **Run** (комбинация клавиш **Ctrl+F2**);
- *точки останова.* В этом режиме после запуска программы на выполнение она останавливается в определенных строках, которые помечены как точки останова. Для того чтобы пометить строку как точку останова, необходимо выполнить команду **Add BreakPoint** (Добавить точку останова) из меню **Debug** (Отладка), при этом точка останова подсвечивается красным цветом. Откроется диалоговое окно (рис. П1.1), в котором необходимо определить следующие параметры точки останова:
  - `Condition` – условие, при котором программа остановится в данной точке; если данное поле пустое, то останов в этой точке произойдет в любом случае;
  - `Pass Count` – параметр, определяющий, сколько раз точка останова должна быть пропущена перед тем, как выполнение программы прекратится;
  - `File Name` – имя файла программы;
  - `Line Number` – номер строки, в которой находится точка останова.

Для удаления или изменения параметров точек останова необходимо выполнить команду **BreakPoints** (Точка останова) из меню **Debug**. Появится диалоговое окно, в котором можно удалить точку останова (**Delete**), изменить ее параметры (**Edit**) или удалить все точки останова (**Clear all**).

Но одна приостановка выполнения программы в определенных точках не позволит найти ошибку. Для поиска алгоритмической ошибки полезно знать значения переменных в той или иной точке. Можно, конечно, для этого вставлять

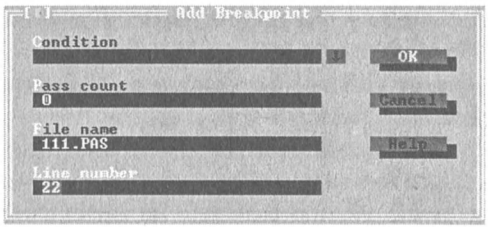


Рис. П1.1 ▾ Добавление точки останова

операторы вывода и просматривать значение переменной, используя комбинацию клавиш **Alt+F5** (см. главу 1) или открыть окно вывода результатов в оболочке, используя команды **Debug/OutPut** (Отладка/Вывод данных). Но оболочка позволяет просматривать значения переменных в специальном окне **Watch** (Просмотр). Его вызов осуществляется с помощью команды **Watch** из пункта меню **Debug**. Добавить переменные в это окно можно с помощью команды **Add watch** (Добавочный просмотр) пункта меню **Debug** (или комбинация клавиш **Ctrl+F7**). Если активным является окно **Watch**, то для добавления переменной достаточно нажать клавишу **Insert**.

Совместное использование приостановки программы и просмотра значений переменных в любой точке программы позволяет найти практически любую алгоритмическую ошибку.

Приложение 2

Ресурсы Internet, посвященные программированию на Паскале и алгоритмизации

1. www.borland.com – официальный сайт фирмы Borland.
2. www.borland.ru – русская версия сайта фирмы Borland.
3. www.freepascal.org – официальная страница свободно распространяемого кросс-платформенного компилятора языка Паскаль. Этот компилятор работает под управлением операционных систем DOS, Windows, Linux, FreeBSD, Solaris, NetBSD, BeOS, QNX, Amiga. На указанном сайте можно скачать последнюю версию компилятора и множество модулей к нему. Здесь же находится документация и много справочной информации, которая представлена на английском языке.
4. www.ru.freepascal.org – зеркало сайта www.freepascal.org в России.
5. www.fpc.by.ru – информация о программировании на языке Free Pascal на русском языке. Здесь также приведены тексты нескольких статей о

программировании на Free Pascal. Информации немного, но это единственный известный авторам сайт о Free Pascal на русском языке.

6. algolist.manual.ru – сайт, содержащий описание большого количества алгоритмов по таким темам, как математика, сортировка, структуры данных, поиск, строки, графические алгоритмы, защита информации, сжатие, кодирование. Тексты алгоритмов в основном приведены на языке C.
7. alglib.manual.ru (библиотека алгоритмов) – один из самых известных русскоязычных сайтов по алгоритмизации. Приведено большое количество блок-схем алгоритмов вычислительной математики. На сайте находится редактор блок-схем – бесплатно распространяемая программа, предназначенная для создания и редактирования блок-схем. В редакторе блок-схем можно нарисовать блок-схему, а потом редактор сгенерирует текст программы на Паскале.
8. www.pascal.dax.ru (Все о Паскале) – сайт, содержащий большое количество книг, статей и текстов программ на Паскале. Кроме того, имеются ссылки для скачивания различных дистрибутивов Паскаля.
9. pascal.sources.ru (Исходники Паскаля со всего мира) – на сайте находится большое количество текстов программ на Паскале. Можно скачать компиляторы всех версий Турбо Паскаля, за исключением четвертой. Много документации.
10. naphy.narod.ru/data/data_htm/tpascal/_index.htm – страничка, посвященная Турбо Паскалю на сайте информационно-технологического колледжа «Подиум». Здесь находятся: русская версия Borland Pascal 7.0 (переведены оболочки bp.exe, turbo.exe, библиотеки turbo.tpl, tpp.tpl и Turbo Vision, а также набор русских .chr-шрифтов), электронные учебники, HTML-версия учебного курса В. В. Фаронова по Турбо Паскалю.
11. faqs.org.ru/progr/pascal – русскоязычные разделы «Часто задаваемые вопросы» (FAQ) по Паскалю.
12. www.pascaler.wsnet.ru – обучающий курс по Турбо Паскалю.
13. http://www.borlpasc.narod.ru/schedule.htm – документация по Паскалю.

Вместо заключения

Перевернута последняя страница книги. Что теперь? Авторы надеются, что знакомство с языком Турбо Паскаль будет только первым этапом в изучении программирования. Если после прочтения книги у читателя возникло желание исправить программы в книге, оптимизировать приведенные алгоритмы, написать собственные программы и модули, то авторы сочтут свою задачу научить читателя основам программирования выполненной.

Следующим этапом в освоении программирования будет разработка своих алгоритмов и написание реально работающих программ под управлением Windows. Но для этого знаний программирования на Турбо Паскале явно недостаточно, следующим этапом может стать изучение среды визуального программирования под Windows – Delphi. В основе Delphi лежит язык программирования Object Pascal.

Возможно, в дальнейшем читатель предпочтет писать программы на других языках программирования: Visual Basic, Borland C++ Builder или Visual C++.

Используемая литература

1. Алексеев Е. Р., Муранова И. В., Палкин А. Е.. Массивы более 64 Кб в Турбо Паскале // Компьютеры + Программы. – 1996. – № 7. – С. 73–75.
2. Белецкий Ян. Турбо Паскаль с графикой для персональных компьютеров. – М.: Машиностроение, 1991.
3. Большая энциклопедия Кирилла и Мефодия. – М., 2003.
4. Бородич Ю. С., Вальвачев А. Н., Кузьмич А. И. Паскаль для персональных компьютеров. – Минск: Вышэйшая школа, 1991.
5. Бронштейн И. Н., Семендяев К. А. Справочник по математике для инженеров и учащихся втузов. – М.: Наука, 1981.
6. Гусева А. И. Учимся программировать: PASCAL 7.0. Задачи и методы их решения. – М.: Диалог МИФИ, 1998.
7. Джонс Ж., Харроу К. Решение задач в системе программирования Турбо Паскаль. – М.: Финансы и статистика, 1991.
8. Кнут Д. Э. Искусство программирования. Т. 3: Сортировка и поиск. – М.: Вильямс, 2000.
9. Поляков Д. Б., Круглов И. Ю. Программирование в среде Турбо Паскаль (версия 5.5). – М.: МАИ, РОСВУЗНАУКА, 1992.
10. Фаронов В. В. Турбо Паскаль 7.0. Начальный курс. – М.: Нолидж, 1997.
11. Фаронов В. В. Турбо Паскаль 7.0. Практика программирования. – М.: Нолидж, 1997.

Предметный указатель

О

П

P

C

Ф

Ц

Алексеев Евгений Ростиславович,
Чеснокова Оксана Витальевна

Турбо Паскаль 7.0

Главный редактор *Захаров И. М.*
editor-in-chief@ntpress.ru
Ответственный редактор *Захарова И. И.*
Верстка *Данилов Е. Р.*
Графика *Салимонов Р. В.*
Дизайн обложки *Клубничкин Д. Е.*

Издательство «НТ Пресс»,
129085, Москва, Звездный б-р, д. 21, стр. 1.

Издание осуществлено при техническом участии
ООО «Издательство АСТ»

Отпечатано с готовых диапозитивов
в ООО «Типография ИПО профсоюзов Профиздат».
109044, Москва, Крутицкий вал, 18.

Паскаль – самый мощный из простых, самый простой из мощных

Вы хотите научиться программированию, но не знаете, с чего начать? Вы умеете составлять алгоритмы, но не знаете, как писать программы? Вы хотите решать математически задачи, но вам мало калькулятора и Excel?

Во всем этом вам поможет Паскаль – хорошо структурированный язык, с помощью которого можно выполнить практически любой проект. Паскаль известен как лучший язык для начинающих, но его возможностей хватит и для серьезных проектов.

Прочитав эту книгу, вы сможете быстро:

- составлять алгоритмы решения математических задач;
- научиться самостоятельно писать самые различные программы;
- понять суть программирования и узнать его терминологию;
- отлаживать проекты, состоящие из нескольких программ и модулей.

Турбо Паскаль 7.0 – это интегрированная среда разработки, которая позволит вам:

- Быстро писать и выполнять любые программы
- Искать и исправлять всевозможные ошибки
- Решать сложные математические задачи
- Рисовать статические картинки и создавать анимацию
- Работать с текстовыми и двоичными файлами
- Использовать дополнительные модули
- Компилировать собственные программы в EXE-файл

ISBN 5-477-00012-0